CHAPTER

THREE

PRINCIPLES OF PIPELINING AND VECTOR PROCESSING

In this chapter, the structures of pipeline computers and vector processing principles are studied. It begins with the basic properties of pipelining, classifications of pipeline processors, and the required memory supports. Both instruction pipelines and arithmetic pipelines are studied in Section 3.2 with design examples. Pipeline design problems will be studied in Section 3.3, including instruction prefetch, branch control, interrupt handling, data buffering, busing structures, internal forwarding, register tagging, hazard detection and resolution, and reconfiguration control. Vector processing requirements and related optimization problems will be introduced with illustrative examples in Section 3.4. Various pipeline supercomputer systems, attached scientific processors, vectorization techniques, and performance evaluation of pipeline computers will be studied in Chapter 4.

3.1 PIPELINING: AN OVERLAPPED PARALLELISM

Pipelining offers an economical way to realize temporal parallelism in digital computers. The concept of pipeline processing in a computer is similar to assembly lines in an industrial plant. To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are streamed into the pipe and get executed in an overlapped fashion at the subtask level. The subdivision of labor in assembly lines has contributed to the success of mass production in modern industry. By the same token, pipeline processing has led to the tremendous improvement of system throughput in the modern digital computer. In this section, a sample design of a floating-point adder is used to illustrate the concept of linear pipelining. Basic properties and speedup of a linear-pipeline processor are characterized. Various types of pipeline processors are then classified according to pipelining levels and functional configurations. Finally, we introduce the reservation table as a design tool of general pipelines with either linear or nonlinear data-flow patterns.

3.1.1 Principles of Linear Pipelining

Assembly lines have been widely used in automated industrial plants in order to increase productivity. Their original form is a flow line (pipeline) of assembly stations where items are assembled continuously from separate parts along a moving conveyor belt. Ideally, all the assembly stations should have equal processing speed. Otherwise, the slowest station becomes the bottleneck of the entire pipe. This bottleneck problem plus the congestion caused by improper buffering may result in many idle stations waiting for new parts. The subdivision of the input tasks into a proper sequence of subtasks becomes a crucial factor in determining the performance of the pipeline.

In a uniform-delay pipeline, all tasks have equal processing time in all station facilities. The stations in an ideal assembly line can operate synchronously with full resource utilization. However, in reality, the successive stations have unequal delays. The optimal partition of the assembly line depends on a number of factors, including the quality (efficiency and capability) of the working units, the desired processing speed, and the cost effectivness of the entire assembly line.

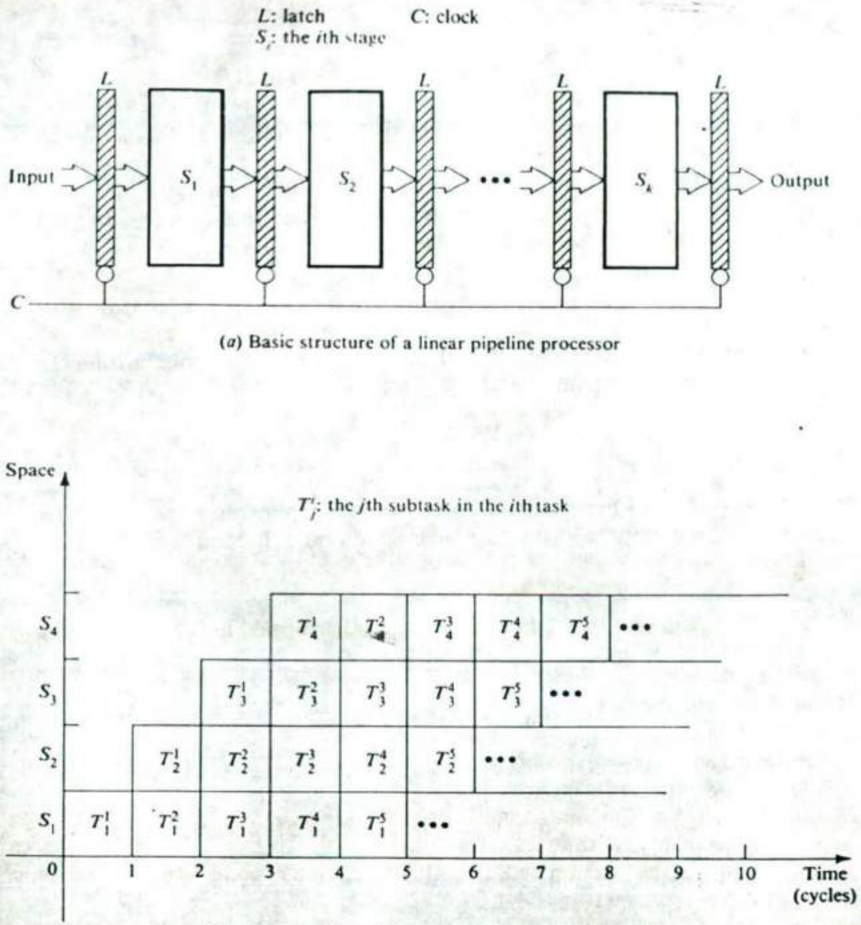
The precedence relation of a set of subtasks $\{T_1, T_2, \ldots, T_k\}$ for a given task T implies that some task T_j cannot start until some earlier task T_i $(i < j)$ finishes. The interdependencies of all subtasks form the *precedence graph*. With a linear precedence relation, task T_j cannot start until all earlier subtasks $\{T_i, \text{for all } i \leq j\}$ finish. A *linear pipeline* can process a succession of subtasks with a linear precedence graph.

A basic linear-pipeline processor is depicted in Figure 3.1*a*. The pipeline consists of a cascade of processing stages. The *stages* are pure combinational circuits performing arithmetic or logic operations over the data stream flowing through the pipe. The stages are separated by high-speed interface *latches*. The latches are fast registers for holding the intermediate results between the stages. Information flows between adjacent stages are under the control of a common clock applied to all the latches simultaneously.

Clock period The logic circuitry in each stage S_i has a time delay denoted by τ_i. Let τ_l be the time delay of each interface latch. The *clock period* of a linear pipeline is defined by

$$\tau = \max\{\tau_i\}_1^k + \tau_l = \tau_m + \tau_l \tag{3.1}$$

The reciprocal of the clock period is called the *frequency* $f = 1/\tau$ of a pipeline processor.



(*a*) Basic structure of a linear pipeline processor

(*b*) The space-time diagram depicting the overlapped operations

Figure 3.1 Linear pipeline processor for overlapped processing of multiple tasks.

One can draw a *space-time diagram* to illustrate the overlapped operations in a linear pipeline processor. The space-time diagram of a four-stage pipeline processor is demonstrated in Figure 3.1*b*. Once the pipe is filled up, it will output one result per clock period independent of the number of stages in the pipe. Ideally, a linear pipeline with k stages can process n tasks in $T_k = k + (n - 1)$ clock periods, where k cycles are used to fill up the pipeline or to complete execution of the first task and $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks. The same number of tasks (operand pairs) can be executed in a nonpipeline processor with an equivalent function in $T_1 = n \cdot k$ time delay.

Speedup We define the *speedup* of a k-stage linear-pipeline processor over an equivalent nonpipeline processor as

$$S_k = \frac{T_1}{T_k} = \frac{n \cdot k}{k + (n - 1)} \tag{3.2}$$

It should be noted that the maximum speedup is $S_k \to k$, for $n \gg k$. In other words, the maximum speedup that a linear pipeline can provide is k, where k is the number of stages in the pipe. This maximum speedup is never fully achievable because of data dependencies between instructions, interrupts, program branches, and other factors to be revealed in later sections. Many pipeline cycles may be wasted on a waiting state caused by out-of-sequence instruction executions.

To understand the operational principles of pipeline computation, we illustrate the design of a pipeline floating-point adder in Figure 3.2. This pipeline is linearly constructed with four functional stages. The inputs to this pipeline are two normalized floating-point numbers:

$$\begin{aligned} A &= a \times 2^p \\ B &= b \times 2^q \end{aligned} \tag{3.3}$$

where a and b are two fractions and p and q are their exponents, respectively. For simplicity, base 2 is assumed. Our purpose is to compute the sum

$$C = A + B = c \times 2^r = d \times 2^s \tag{3.4}$$

where $r = \max(p, q)$ and $0.5 \leq d < 1$. Operations performed in the four pipeline stages are specified below:

1. Compare the two exponents p and q to reveal the larger exponent $r = \max(p, q)$ and to determine their difference $t = |p - q|$.
2. Shift right the fraction associated with the smaller exponent by t bits to equalize the two exponents before fraction addition.
3. Add the preshifted fraction with the other fraction to produce the intermediate sum fraction c, where $0 \leq c < 1$.
4. Count the number of leading zeros, say u, in fraction c and shift left c by u bits to produce the normalized fraction sum $d = c \times 2^u$, with a leading bit 1. Update the large exponent s by subtracting $s = r - u$ to produce the output exponent.

The comparator, selector, shifters, adders, and counter in this pipeline can all be implemented with combinational logic circuits. Detailed logic design of these boxes can be found in the book by Hwang (1979). Suppose the time delays of the four stages are $\tau_1 = 60$ ns, $\tau_2 = 50$ ns, $\tau_3 = 90$ ns, and $\tau_4 = 80$ ns and the interface latch has a delay of $\tau_l = 10$ ns. The cycle time of this pipeline can be chosen to be at least $\tau = 90 + 10 = 100$ ns (Eq. 3.1). This means that the clock frequency of the pipeline can be set to $f = 1/\tau = 1/100 = 10$ MHz. If one uses a nonpipeline floating-point adder, the total time delay will be $\tau_1 + \tau_2 + \tau_3 + \tau_4 =$

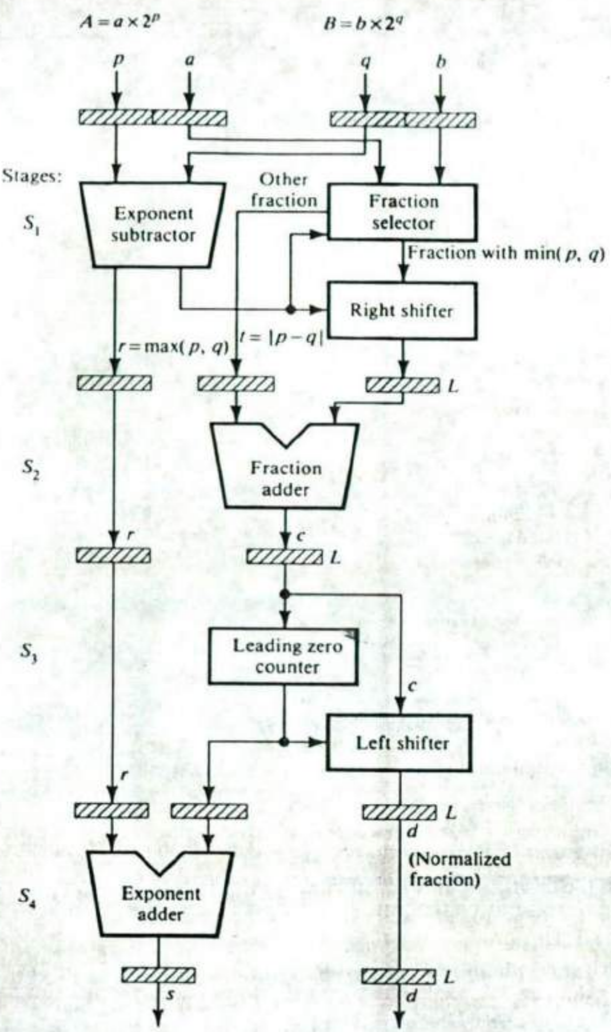


Figure 3.2 A pipelined floating-point adder with four processing stages.

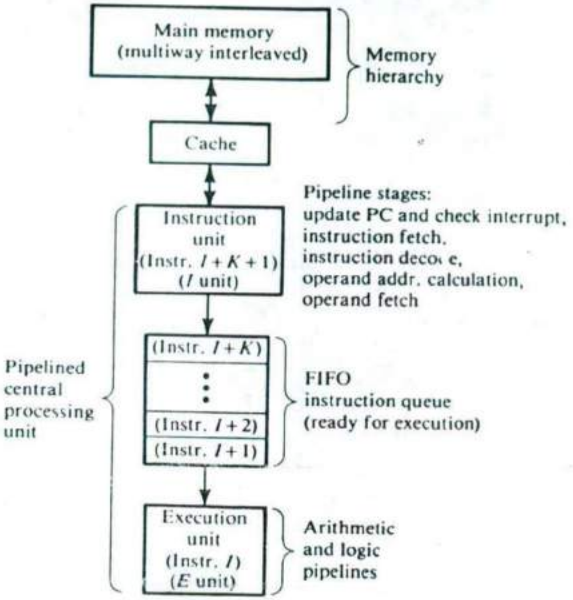


Figure 3.3 The pipelined structure of a typical central processing unit.

300 ns. In this case, the pipeline adder has a speedup of 300/100 = 3 over the nonpipeline adder design. If uniform delays can be achieved in all four stages, say 75 ns per stage (including the latch delay), then the maximum speedup of 300/75 = 4 can be achieved.

The central processing unit (CPU) of a modern digital computer can generally be partitioned into three sections: the *instruction unit*, the *instruction queue*, and the *execution unit*. From the operational point of view, all three units are pipelined, as illustrated in Figure 3.3. Programs and data reside in the main memory, which usually consists of interleaved memory modules. The cache is a faster storage of copies of programs and data which are ready for execution. The cache is used to close up the speed gap between main memory and the CPU.

The instruction unit consists of pipeline stages for instruction fetch, instruction decode, operand address calculation, and operand fetches (if needed). The instruction queue is a first-in, first-out (FIFO) storage area for decoded instructions and fetched operands. The execution unit may contain multiple functional pipelines for arithmetic logic functions. While the instruction unit is fetching instruction $I + K + 1$, the instruction queue holds instructions $I + 1, I + 2, \ldots, I + K$, and the execution unit executes instruction I. In this sense, the CPU is a good example of a linear pipeline. We will describe the detailed design of a pipeline CPU for instruction execution and arithmetic computations in Section 3.3.

After defining the clock period and speedup in Eqs. 3.1 and 3.2, we need to introduce two related measures of the performance of a linear pipeline processor. The product (area) of a *time interval* and a *stage space* in the space-time diagram (Figure 3.1*b*) is called a *time-space span*. A given time-space span can be in either a *busy* state or an *idle* state, but not both. We use this concept to measure the performance of a pipeline.

Efficiency The efficiency of a linear pipeline is measured by the percentage of busy time-space spans over the total time-space span, which equals the sum of all busy and idle time-space spans. Let n, k, τ be the number of tasks (instructions), the number of pipeline stages, and the clock period of a linear pipeline, respectively. The *pipeline efficiency* is defined by

$$\eta = \frac{n \cdot k \cdot \tau}{k \cdot [k\tau + (n-1)\tau]} = \frac{n}{k + (n-1)} \tag{3.5}$$

Note that $\eta \to 1$ as $n \to \infty$. This implies that the larger the number of tasks flowing through the pipeline, the better is its efficiency. Moreover, we realize that $\eta = S_k/k$ from Eqs. 3.2 and 3.3. This provides another view of the efficiency of a linear pipeline as the ratio of its actual speedup to the ideal speedup k. In the steady state of a pipeline, we have $n \gg k$, the efficiency η should approach 1. However, this ideal case may not hold all the time because of program branches and interrupts, data dependency, and other reasons to be discussed in Section 3.2.

Throughput The number of results (tasks) that can be completed by a pipeline per unit time is called its throughput. This rate reflects the computing power of a pipeline. In terms of the efficiency η and clock period τ of a linear pipeline, we define the *throughput* as follows:

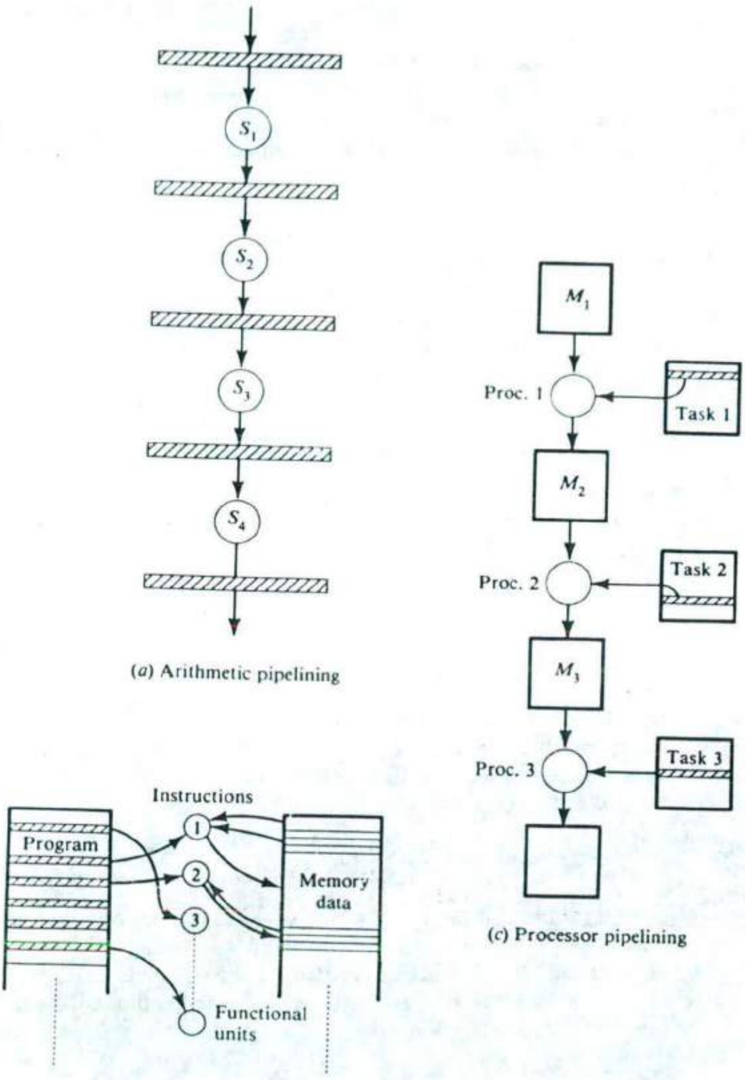
$$w = \frac{n}{k\tau + (n-1)\tau} = \frac{\eta}{\tau} \tag{3.6}$$

where n equals the total number of tasks being processed during an observation period $k\tau + (n-1)\tau$. In the ideal case, $w = 1/\tau = f$ when $\eta \to 1$. This means that the maximum throughput of a linear pipeline is equal to its frequency, which corresponds to one output result per clock period. We will further evaluate the performance of pipeline processors in Section 4.4.4.

3.1.2 Classification of Pipeline Processors

According to the levels of processing, Händler (1977) has proposed the following classification scheme for pipeline processors, as illustrated in Figure 3.4.

Arithmetic pipelining The arithmetic logic units of a computer can be segmentized for pipeline operations in various data formats (Figure 3.4*a*). Well-known arithmetic pipeline examples are the four-stage pipes used in Star-100, the eight-stage pipes used in the TI-ASC, the up to 14 pipeline stages used in the Cray-1, and the



(*a*) Arithmetic pipelining

(*c*) Processor pipelining

(*b*) Instruction pipelining

Figure 3.4 Händler classification of pipelined processors.

up to 26 stages per pipe in the Cyber-205. These arithmetic logic pipeline designs will be studied subsequently.

Instruction pipelining The execution of a stream of instructions can be pipelined by overlapping the execution of the current instruction with the fetch, decode, and operand fetch of subsequent instructions (Figure 3.4*b*). This technique is also known as *instruction lookahead*. Almost all high-performance computers are now equipped with instruction-execution pipelines.

Processor pipelining This refers to the pipeline processing of the same data stream by a cascade of processors (Figure 3.4*c*), each of which processes a specific task. The data stream passes the first processor with results stored in a memory block which is also accessible by the second processor. The second processor then passes the refined results to the third, and so on. The pipelining of multiple processors is not yet well accepted as a common practice.

According to pipeline configurations and control strategies, Ramamoorthy and Li (1977) have proposed the following three pipeline classification schemes:

Unifunction vs. multifunction pipelines A pipeline unit with a fixed and dedicated function, such as the floating-point adder in Figure 3.3, is called *unifunctional*. The Cray-1 has 12 unifunctional pipeline units for various scalar, vector, fixed-point, and floating-point operations. A *multifunction* pipe may perform different functions, either at different times or at the same time, by interconnecting different subsets of stages in the pipeline. The TI-ASC has four multifunction pipeline processors, each of which is reconfigurable for a variety of arithmetic logic operations at different times.

Static vs. dynamic pipelines A *static pipeline* may assume only one functional configuration at a time. Static pipelines can be either unifunctional or multifunctional. Pipelining is made possible in static pipes only if instructions of the same type are to be executed continuously. The function performed by a static pipeline should not change frequently. Otherwise, its performance may be very low. A *dynamic pipeline* processor permits several functional configurations to exist simultaneously. In this sense, a dynamic pipeline must be multifunctional. On the other hand, a unifunctional pipe must be static. The dynamic configuration needs much more elaborate control and sequencing mechanisms than those for static pipelines. Most existing computers are equipped with static pipes, either unifunctional or multifunctional.

Scalar vs. vector pipelines Depending on the instruction or data types, pipeline processors can be also classified as scalar pipelines and vector pipelines. A *scalar pipeline* processes a sequence of scalar operands under the control of a DO loop. Instructions in a small DO loop are often prefetched into the instruction buffer. The required scalar operands for repeated scalar instructions are moved into a data cache in order to continuously supply the pipeline with operands. The IBM

System/360 Model 91 is a typical example of a machine equipped with scalar pipelines. However, the Model 91 does not have a cache.

Vector pipelines are specially designed to handle vector instructions over vector operands. Computers having vector instructions are often called *vector processors*. The design of a vector pipeline is expanded from that of a scalar pipeline. The handling of vector operands in vector pipelines is under firmware and hardware controls (rather than under software control as in scalar pipelines). Pipeline vector processors to be studied in Chapter 4 include Texas Instruments' ASC, Control Data's STAR-100 and Cyber-205, Cray Research's Cray-1, Fujitsu's VP-200, AP-120B (FPS-164), IBM's 3838, and Datawest's MATP.

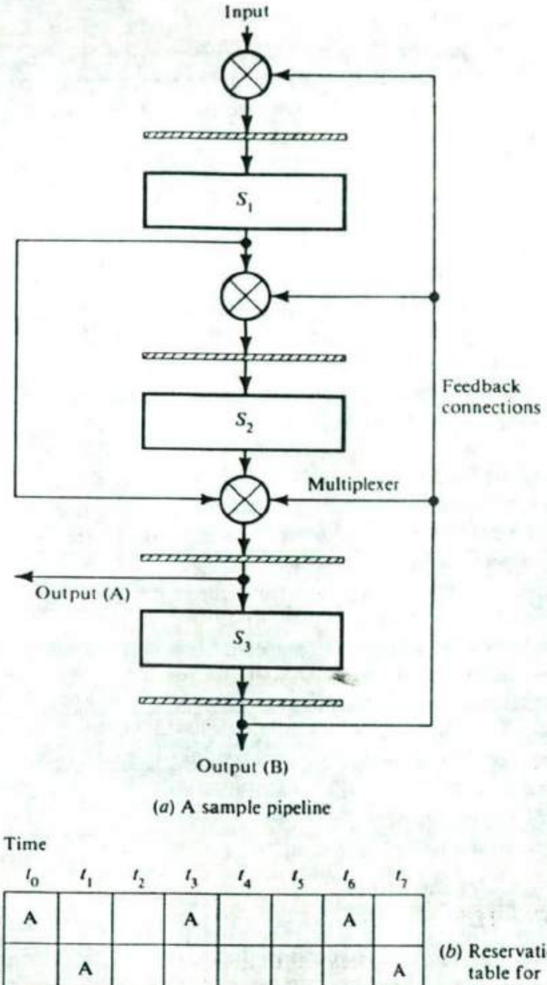
3.1.3 General Pipelines and Reservation Tables

What we have studied so far are linear pipelines without feedback connections. The inputs and outputs of such pipelines are totally independent. In some computations, like linear recurrence, the outputs of the pipeline are fed back as future inputs. In other words, the inputs may depend on previous outputs. Pipelines with feedback may have a nonlinear flow of data. The utilization history of the pipeline determines the present state of the pipeline. The timing of the feedback inputs becomes crucial to the nonlinear data flow. Improper use of the feedforward or feedback inputs may destroy the inherent advantages of pipelining. On the other hand, proper sequencing with nonlinear data flow may enhance the pipeline efficiency. In practice, many of the arithmetic pipeline processors allow nonlinear connections as a mechanism to implement recursion and multiple functions.

In this section, we characterize the interconnection structures and data-flow patterns in *general pipelines* with either feedforward or feedback connections, in addition to the cascaded connections in a linear pipeline. We use a two-dimensional chart known as the *reservation table* which is borrowed from the Gantt charts used in operation research to show how successive pipeline stages are utilized (or reserved) for a specific function evaluation in successive pipeline cycles. This reservation table was originally suggested by Davidson (1971). It is very similar to the space-time diagram introduced by Chen (1971) (Figure 3.1*b*).

Consider a sample pipeline that has a structure with both feedforward and feedback connections, as shown in Figure 3.5*a*. Assume that this pipeline is dual-functional, denoted as function *A* and function *B*. We will number the pipeline stages S_1, S_2, S_z from the input end to the output end. The one-way connections between adjacent stages form the original linear cascade of the pipeline. A *feedforward connection* connects a stage S_i to a stage S_j such that $j \geq i + 2$ and a *feedback connection* connects a stage S_i to a stage S_j such that $j \leq i$. In this sense, a "pure" linear pipeline is a pipeline without any feedback or feedforward connections. The crossed circles in Figure 3.5*a* refer to data multiplexers used for selecting among multiple connection paths in evaluating different functions.

The two reservation tables shown in Figure 3.5*b* and 3.5*c* correspond to the two functions of the sample pipeline. The rows correspond to pipeline stages and the columns to clock time units. The total number of clock units in the table is



(*a*) A sample pipeline

Time

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S_1	A			A			A	
S_2		A						A
S_3			A		A	A		

(*b*) Reservation table for function A

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
S_1	B				B		
S_2			B			B	
S_3		B		B			B

(*c*) Reservation table for function B

Figure 3.5 A sample pipeline and reservation tables for two of its functions.

called the *evaluation time* for the given function. A reservation table represents the flow of data through the pipeline for one complete evaluation of a given function.

A marked entry in the (i, j)th square of the table indicates that stage S_i will be used j time units after the initiation of the function evaluation. For a unifunctional pipeline, one can simply use an " × " to mark the table entries. For a multifunctional pipeline, different marks are used for different functions, such as the A's and B's in the two reservation tables for the sample pipeline. Different functions may have different evaluation times, such as 8 and 7 shown in Figure 3.5*b* and 3.5*c* for functions A and B, respectively.

The data-flow pattern in a static, unifunctional pipeline can be fully described by one reservation table. A multifunctional pipeline may use different reservation tables for different functions to be performed. On the other hand, a given reservation table does not uniquely correspond to one particular hardware pipeline. One· may find that several hardware pipelines with different interconnection structures can use the same reservation table.

Many interesting pipeline-utilization features can be revealed by the reservation table. It is possible to have multiple marks in a row or in a column. Multiple marks in a column correspond to the simultaneous usage of multiple pipeline stages. Multiple marks in a row correspond to the repeated usage (for marks in distant columns) or prolonged usage (for marks in adjacent columns) of a given stage. It is clear that a general pipeline may have multiple paths, parallel usage of multiple stages, and nonlinear flow of data.

In order to visualize the flow of data along selected data paths in a hardware pipeline for a complete function evaluation, we show in Figure 3.6 the snapshots of eight steps needed to evaluate function A in the sample pipeline. These snapshots are traced along the entries in reservation table A. Active stages in each time unit are shaded. The darkened connections are the data paths selected in case of multiple path choices. We will use reservation tables in subsequent sections to study various pipeline design problems.

3.1.4 Interleaved Memory Organizations

Pipeline or vector processors require effective access to linear arrays or sequential instructions, hence the memory must be designed to avoid access conflicts. There is a basic attribute to measure the effectiveness of a memory configuration, called the *memory bandwidth*, which is the average number of words accessed per second. The primary factors affecting the bandwidth are the processor architecture, the memory configuration, and the memory-module characteristics. The memory configuration is characterized by the number of memory modules and their addressing structure and bus width. The module characteristics include the memory-module size, access time, and cycle time. The memory bandwidth must match the demand of the processors as discussed in Chapter 1.

The *demand rate* of a processor architecture and its matching memory configuration is illustrated with an example: Consider a pipeline computer which operates with four independent 32-bit floating-point arithmetic pipelines. Each

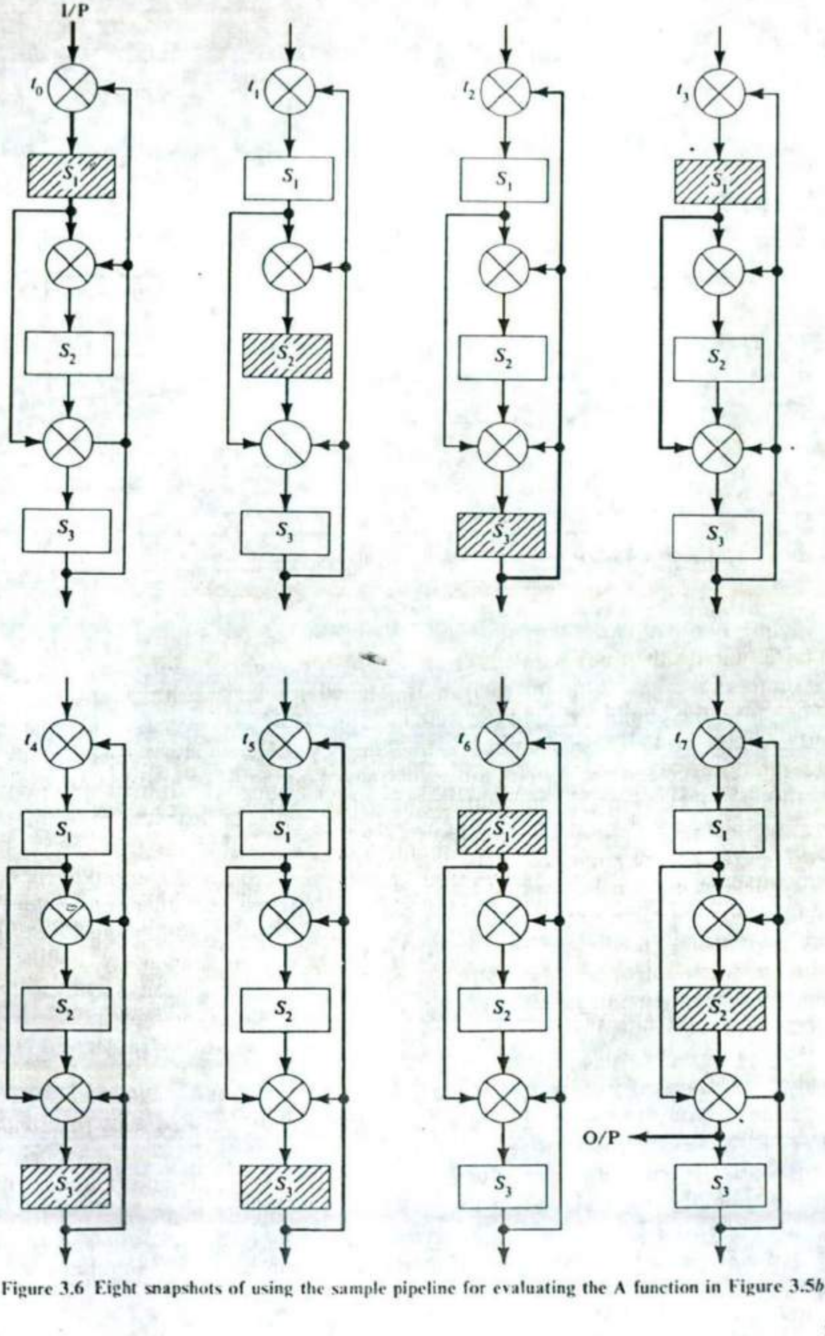


Figure 3.6 Eight snapshots of using the sample pipeline for evaluating the A function in Figure 3.5*b*.

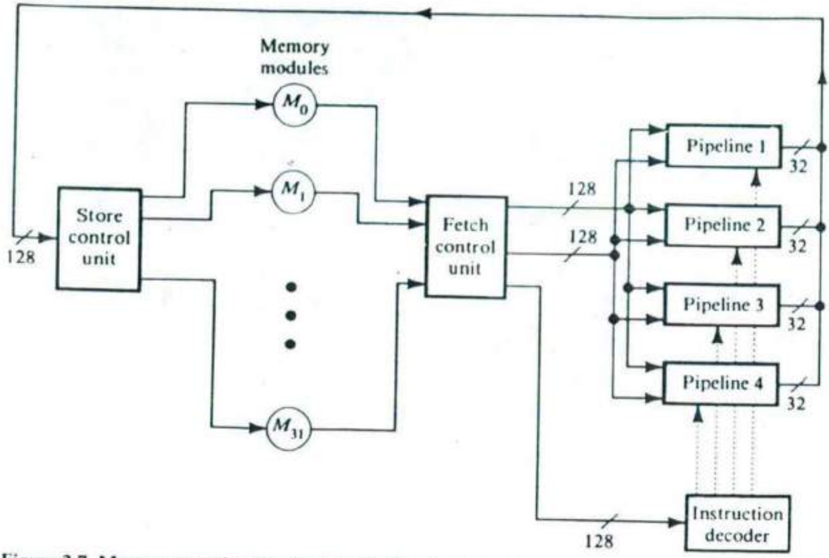
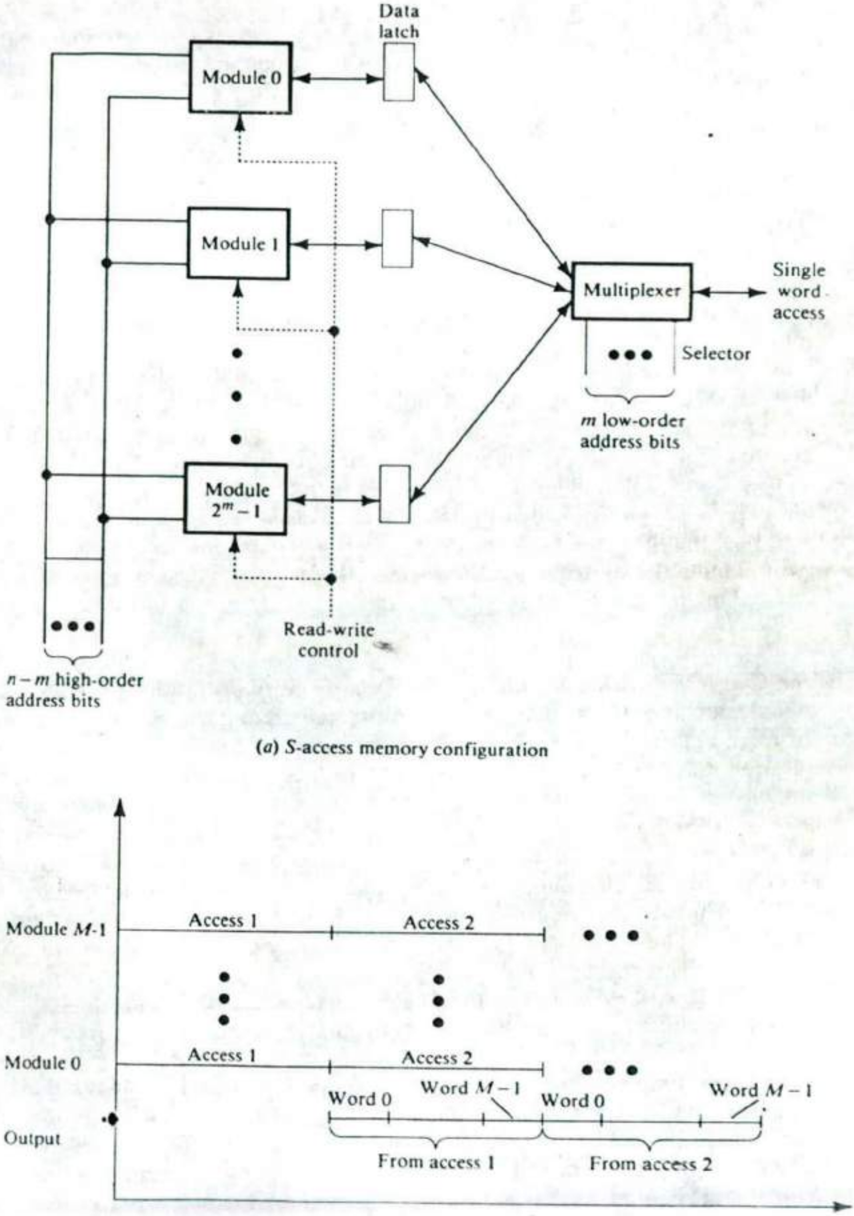


Figure 3.7 Memory organization of a four-pipeline vector processor.

pipeline requires two 32-bit operands every pipeline clock of 40 ns to produce one 32-bit result. Four parallel 32-bit results are produced for every 40 ns. We assume that one 32-bit instruction is fetched for each arithmetic operation. Therefore, the demand rate of the four pipelines is to fetch 3 × 4 × 32 bits every 40 ns and store 4 × 32 bits of result in the same 40 ns. Hence 512 bits of information need to be accessed every 40 ns. Since each set of operands or instructions consists of four 32-bit elements, the bus width to main memory for each access (fetch or store) can be made $4 \times 32 = 128$ bits (Figure 3.7). Since there are two operand fetches, one operand-store and one instruction fetch for each pipeline and in each 40-ns interval, the main memory should have four 128-bit wide unidirectional buses. If the memory cycle time is 1.28 μs, then 1.28 μs/40 ns or 32 memory modules are required to match the demand rate. Each memory configuration is controlled by a memory controller that defines the *storage scheme* for every memory reference. The storage scheme is the set of rules that determine the module number and the address of the element within each module.

The S access memory organization One of the simplest memory configurations for pipeline vector processors uses low-order interleaving and applies the higher $(n - m)$ bits of the address to all $M = 2^m$ memory modules simultaneously in one access. The single access returns M consecutive words of information from the M memory modules. Using the low-order m address bits, the information from a particular modules can be accessed. This configuration, which is shown in Figure 3.8*a*, is called *S access* because all modules are accessed simultaneously.



(*a*) *S*-access memory configuration

(*b*) Timing diagram for *S*-access configuration

Figure 3.8 The S-access interleaved memory configuration.

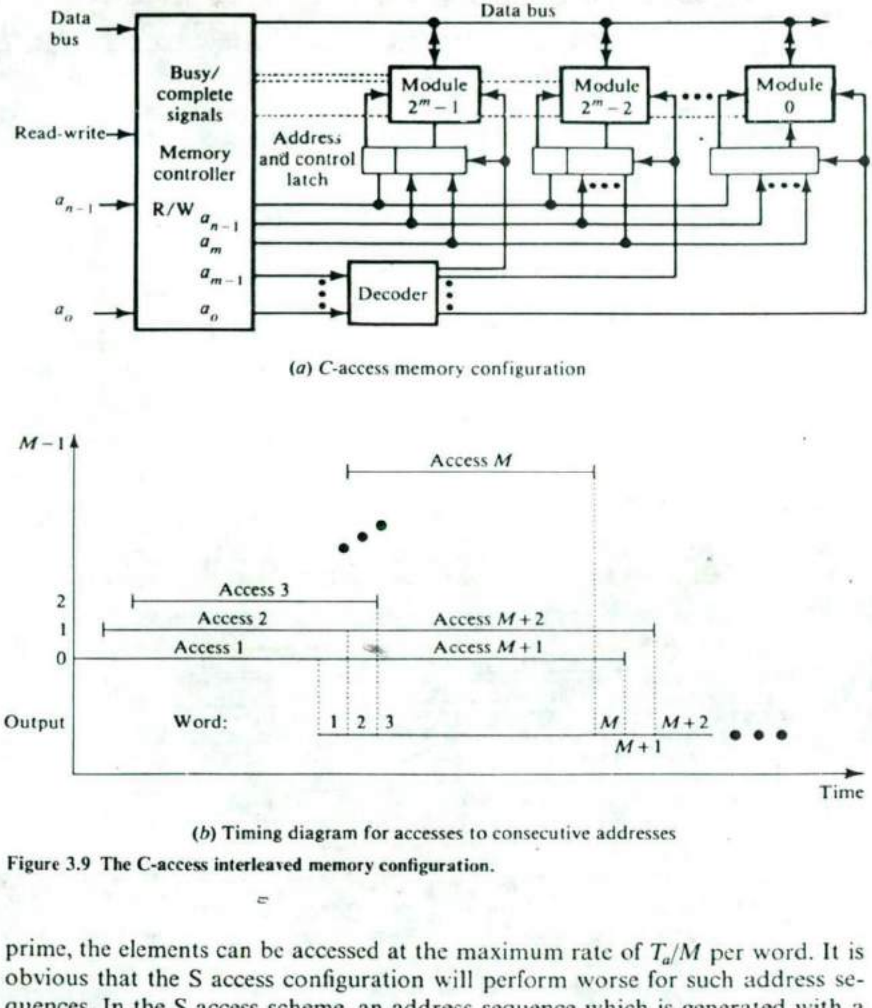
A data latch is associated with each module. For a fetch operation, the information from each module is gated into its latch, whereupon the multiplexer can be used to direct the desired data to the single-word bus. Figure 3.8*b* depicts the timing diagram for sample multiple-word read accesses using the S-access configuration. Notice that with a memory-access time T_a and a latch delay of τ, the time to access a single word is $T_a + \tau$. However, the total time it takes to access k consecutive words in sequence, starting in module i, is $T_a + k\tau$ if $i + k \leq M$, otherwise it is $2T_a + (i + k - M)\tau$. In both cases, $1 \leq k \leq M$. For effective access of long vectors, $M\tau \leq T_a$; otherwise, there would be a data overrun. S-access configuration is ideal for accessing a vector of data elements or for prefetching sequential instructions for a pipeline processor. It can also be used to access a block of information for a pipeline processor with a cache.

When nonsequentially addressed words are requested, the performance of the memory system deteriorates rapidly. To provide a partial remedy for nonsequential accesses, some concurrency can be introduced into the configuration by providing an address latch for each memory module so that the effective address cycle (hold time) t_a is much smaller than the memory cycle time t_c. Since the address is typically held on the address bus at least as long as data is held on the data bus, the data buses do not pose a limiting constraint on the performance. By providing the address latch, the group of M modules can be multiplexed on an internal memory address bus, called a *bank* or a *line* as to be studied in Chapter 7.

The C access memory organization When a memory operation is initiated in a module, it causes the bank to be active for t_a seconds and the module to be active for t_c seconds. If t_a is much less than t_c, the initiated module uses the bank for a duration much less than one memory cycle per access. Therefore, more than one module can share a bank, increasing the bank utilization and reducing the bank cost. This configuration is called *C access* because modules are accessed concurrently, as illustrated in Figure 3.9*a*. The low-order m bits are used to select the module and the remaining $n - m$ bits address the desired element within the module. The memory controller is used to buffer a request which both references a busy module and initiates service when the module completes its current cycle. Figure 3.9*b* shows an example timing diagram where K consecutive words are fetched in $T_a + k \cdot \tau$, assuming that the address cycle $t_a = \tau = T_a/M$.

The effectiveness of this memory configuration is revealed by its ability to access the elements of a vector. Consider a vector of s elements $V[0:s-1]$ in which every other element is accessed; that is, the *skip distance* is 2. Assuming that element $V[i]$ is stored in module $i \pmod M$ for $0 \leq i \leq s - 1$, the timing diagram in Figure 3.10*a* for $M = 8$ illustrates the performance. After the initial access, the access time for each sequential element is one per every 2τ seconds, where $\tau = T_a/M$. If the skip distance is increased to 3, the performance is one element per every τ seconds after the initial access. This is shown in Figure 3.10*b*.

In general, if an address sequence is generated with a skip distance d and there are M modules arranged in C access configuration such that M and d are relatively



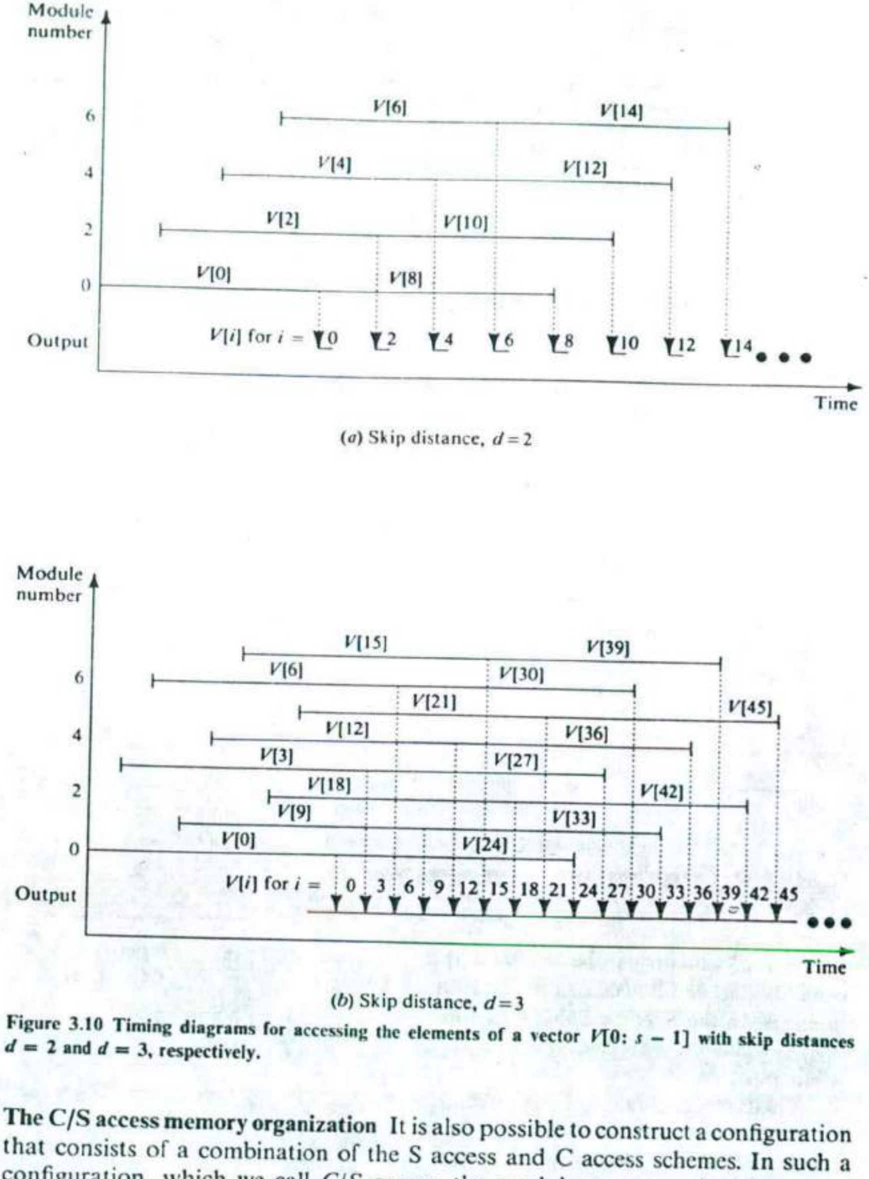
(*a*) *C*-access memory configuration

(*b*) Timing diagram for accesses to consecutive addresses

Figure 3.9 The C-access interleaved memory configuration.

prime, the elements can be accessed at the maximum rate of T_a/M per word. It is obvious that the S access configuration will perform worse for such address sequences. In the S access scheme, an address sequence which is generated with a skip distance of d has an *average data rate* of dT_a/M when $d \leq M$, and of T_a, when $d > M$.

The storage scheme for a vector can be extended to two and higher dimensional arrays. As an example, consider a two-dimensional array $A[0: R-1, 0: C-1]$. The elements can be mapped into a one-dimensional vector $[0: s-1]$, in two basic ways: *row-major* form or *column-major* form. In row-major form, the index element $A[i, j]$ in the vector V is given by the $iC + j$. Similarly, the index of $A[i, j]$ in column-major form is $jR + i$. The storage scheme for the two-dimensional array can then be derived from the storage schemes for V, as described earlier.



(a) Skip distance, $d = 2$

(b) Skip distance, $d = 3$

Figure 3.10 Timing diagrams for accessing the elements of a vector $V[0: s - 1]$ with skip distances $d = 2$ and $d = 3$, respectively.

The C/S access memory organization It is also possible to construct a configuration that consists of a combination of the S access and C access schemes. In such a configuration, which we call *C/S access*, the modules are organized in a two-dimensional array. If the S access is M-way interleaved and the C access is L-way interleaved, up to L different accesses to blocks of M conservative words can be in progress simultaneously. This scheme is effective for multiple pipeline processors.

Memory bandwidth estimation Various analytical models have been developed to evaluate the performance of interleaved memory configurations in a parallel-processing environment. All the models presented here assume some form of memory interleaving and evaluate the access-conflict problem. First, we summarize the memory bandwidth when a single processor is used. A sequence of memory requests from a process can be considered as an ordered set of memory-module numbers from 0 to $m - 1$.

Hellerman presented a model in which a single stream of independent instructions and data-memory requests is scanned in the order of their arrival until the first duplicate memory module is found. These first k-distinct requests are then accessed in parallel. The steady-state bandwidth can be taken to be the average length of an initial string of duplicate-free integers; that is, the distance between the first duplicates. The probability that k is the length of a string of distinct integers is $P(k) = k \cdot (m-1)!/[m^k \cdot (m-k)!]$ for an m-way interleaved memory system. The *average bandwidth* for the single processor and m memory modules is

$$B(1, m) = \sum_{k=1}^{m} k \cdot P(k) = \sum_{k=1}^{m} \frac{k^2 \cdot (m-1)!}{m^k \cdot (m-k)!} \tag{3.7}$$

$B(1, m)$ has a good numerical approximation of $m^{0.56}$ when $1 \le m \le 45$. Knuth and Rao showed a closed-form solution of Eq. 3.7, which shows that Hellerman's bandwidth is asymptotic to $\sqrt{m}$.

Burnett and Coffman improved the model by exploiting the principle of sequentiality of instructions. To model this effect, the instruction requests are separated from the data requests. The memory bandwidth can thus be increased considerably because of the locality of programs. This was modeled by introducing two parameters, α and β, where α is the probability of a request addressing the next module in sequence (modulo m) and $\beta = (1-\alpha)/(m-1)$. Assume that the memory requests at the start of a memory cycle can be represented as a sequence of m addresses $r_1, r_2, \ldots, r_m$ such that $0 \le r_i \le m-1$ for $1 \le i \le m$. Assume the following properties for the address sequence:

$$\text{Prob}[r_1 = k] = \frac{1}{m} \qquad \text{for } 0 \le k \le m-1$$

$$\text{Prob}[r_{i+1} = (r_i + 1) \bmod m] = \alpha \qquad \text{for } 1 \le i \le m$$

$$\text{Prob}[r_{i+1} \ne (r_i + 1) \bmod m] = \beta \qquad \text{for } 1 \le i \le m$$

The first property indicates that the first reference is made randomly to any memory module. The second property indicates the probability that the next reference is made to the next sequential module. The last property indicates the probability that the next reference is made to a nonsequential module.

Since the first request can be made to any module randomly, let us assume it is made to module 0. Assume that the first k requests are made to distinct modules for $1 \le k \le m$. Since the first request is to module 0, an arbitrary number j of the $k-1$ requests will be of the α type and $(k-j-1)$ requests will be of the β type. For example, suppose $m = 8$ and a sequence of eight distinct requests

$r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8$ reference modules 2, 5, 6, 7, 0, 3, 4, 5, respectively. Notice that the first seven requests are to distinct modules. Thus $\text{Prob}[r_1] = 1/m = \frac{1}{8}$; $\text{Prob}[r_2] = \text{Prob}[r_6] = \beta$; $\text{Prob}[r_3] = P[r_4] = P[r_5] = P[r_7] = \alpha$. Hence, the sequence has a probability $(1/m)\alpha^4\beta^2$. A generalization of this concept leads to an *expected bandwidth* of

$$B(1, m) = \sum_{k=1}^{m} \sum_{j=0}^{k-1} \alpha^j \beta^{k-j-1} C_m(j, k) \tag{3.8}$$

where $C_m(j, k)$ is the total number of sequences of length k with j and $(k - j - 1)$ requests of types α and β respectively. A combinational analysis shows that

$$C_m(j, k) = \sum (-1)^n \binom{j+n}{n} \binom{k-1}{j+n} (m - j - n - 1)_{k-j-1},$$

where $(m - j - 1)(m - j - 2) \cdots (m - k + 1)$ is denoted $(m - j - 1)_{k-j-1}$.

The bandwidth in Eq. 3.8 increases exponentially to m with α. As the program behavior exhibits more sequentiality, $B(1, m)$ increases exponentially to m. Such a behavior is more representative of instruction streams but does not adequately represent the overall program behavior, which must include the data references. The above model assumes a single processor with instruction lookahead capabilities. It has been called an *overlap processor model*. In general, there may be some *dependency* between any two addresses requested from a process. The bandwidth can further increase if consideration of data dependencies among program segments is included in the analysis.

3.2 INSTRUCTION AND ARITHMETIC PIPELINES

Before studying various pipeline design techniques and examining vector processing requirements, we need to understand how instructions can be overlapped, executed, and how repeated arithmetic computations can be done with pipelining. Instruction pipelining is illustrated with the designs in the IBM 360/91. Arithmetic pipelining will be studied in detail with four design examples for multiple-number addition, floating-point addition, multiplication, and division. Finally, multifunction-pipeline designs and array pipelining for matrix arithmetic will be introduced.

3.2.1 Design of Pipelined Instruction Units

Most of today's mainframes are equipped with pipelined central processors. We will study the instruction pipeline in the IBM System/360 Model 91 as a learning example. The IBM 360/91 incorporates a high degree of pipelining in both instruction preprocessing and instruction execution. It is a 32-bit machine specially designed for scientific computations in either fixed-point or floating-point data formats. Multiple pipeline functional units are built into the system to allow parallel arithmetic computations in either data format.

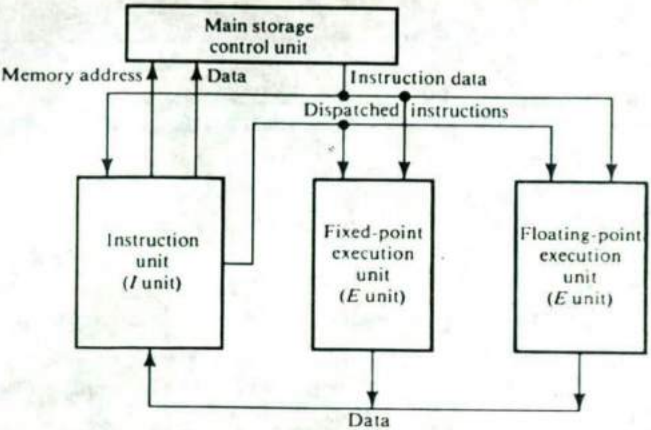
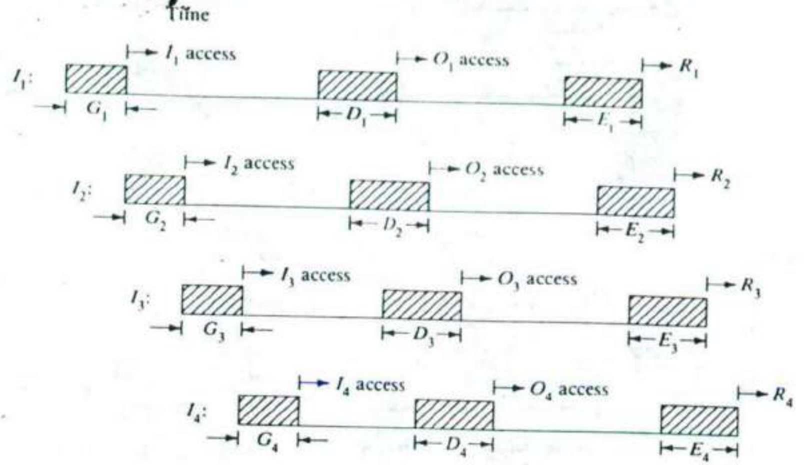

Main storage control unit
Memory address
Data
Instruction data
Dispatched instructions
Instruction unit (I unit)
Fixed-point execution unit (E unit)
Floating-point execution unit (E unit)
Data

Figure 3.11 The central processing unit (CPU) of IBM System 360/Model 91.

A block diagram of the CPU in the IBM 360/91 is depicted in Figure 3.11. It consists of four major parts: the *main storage control unit*, the *instruction unit*, the *fixed-point execution unit*, and the *floating-point execution unit*. The instruction unit (I unit) is pipelined with a clock period of 60 ns. This CPU is designed to issue instructions at a burst rate of one instruction per clock cycle, and the performance of the two execution units (E units) should support this rate. The storage control unit supervises information exchange between the CPU and the main-memory major functions of the I unit, including instruction fetch, decode, and delivery to the appropriate E unit, operand address calculation and operand fetch. The two E units are responsible for the fixed-point and floating-point arithmetic logic operations needed in the execution phase.

From memory access to instruction decode and execution, the CPU is fully pipelined across the four units shown in Figure 3.11. Concurrency among successive instructions in the Model 91 is illustrated in Figure 3.12. It is desirable to overlay separate instruction functions to the greatest possible degree. The shaded boxes correspond to circuit functions and the thin lines between them refer to delays caused by memory access. Obviously, memory accesses for fetching either instructions or operands take much longer time than the delays of functional circuitry. Following the delay caused by the initial filling of the pipeline, the execution results will begin emerging at the rate of one per 60 ns.

For the processing of a typical floating-point storage-to-register instruction, we show the functional segmentation of the pipeline in Figure 3.13 along with the clock-time divisions. The basic time cycle accommodates the pipelining of most hardware functions. However, the memory and many execution functions require a variable number of pipeline cycles. In general, these storage and execution functions require a large portion of time cycles, as revealed in Figure 3.13. After decoding, two parallel sequences of operation may be initiated: one for operand


Time
I_1:
$\vdash\!\!\to I_1$ access
G_1
$\vdash\!\!\to O_1$ access
D_1
$\vdash\!\!\to R_1$
E_1
I_2:
$\vdash\!\!\to I_2$ access
G_2
$\vdash\!\!\to O_2$ access
D_2
$\vdash\!\!\to R_2$
E_2
I_3:
$\vdash\!\!\to I_3$ access
G_3
$\vdash\!\!\to O_3$ access
D_3
$\vdash\!\!\to R_3$
E_3
I_4:
$\vdash\!\!\to I_4$ access
G_4
$\vdash\!\!\to O_4$ access
D_4
$\vdash\!\!\to R_4$
E_4

I_i: instruction i

O_i: operand access, i

G_i: generate I_i address

D_i: decode I_i and generate O_i address

E_i: execute I_i

R_i: result i

Figure 3.12 Concurrency among successive instruction fetch-decode-execute in the IBM 360/91.

access and the other for the setup of operands to be transmitted to an assigned execution station in the selected arthmetic unit. The effective memory access time must match the speeds of the pipeline stages.

Because of the time disparities between various instruction types, the Model 91 utilizes the organizational techniques of memory interleaving, parallel arithmetic functions, data buffering, and internal forwarding to overcome the speed gap problems. The depth of interleaving is a function of the memory cycle time, the CPU storage request rate, and the desired effective-access time. The Model 91 chooses a depth of 16 for interleaving 400 ns/cycle storage modules to satisfy an effective access time of 60 ns. We will examine pipeline arithmetic and data-buffering techniques in subsequent sections.

Concurrent arithmetic executions are facilitated in the Model 91 by using two separate units for fixed-point execution and floating-point execution. This permits instructions of the two classes to be executed in parallel. As long as no cross-unit data dependencies exist, the execution does not necessarily flow in the sequence in which the instructions are programmed. Within the floating-point E unit are an *add unit* and a *multiply/divide unit* which can operate in parallel. Furthermore, pipelining is practiced within arithmetic units, as will be described in Section

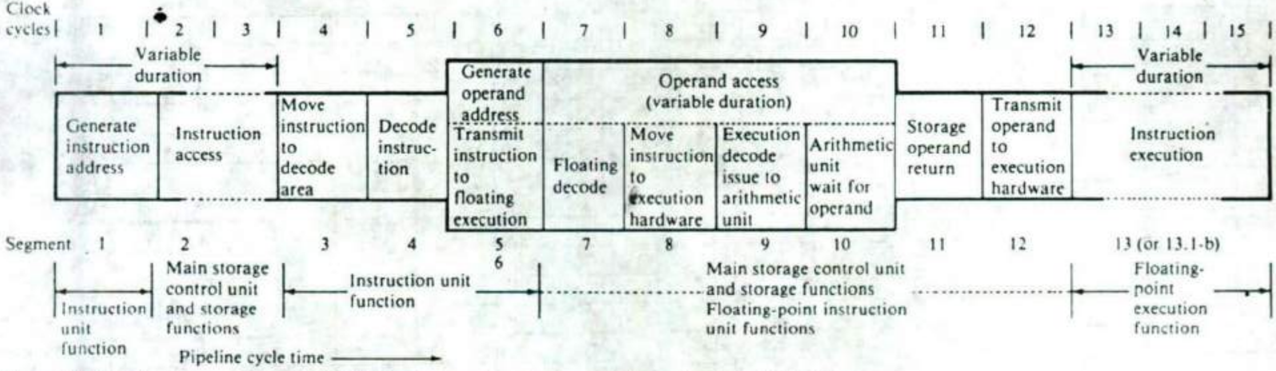

Clock cycles
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Variable duration
Generate instruction address
Instruction access
Move instruction to decode area
Decode instruction
Generate operand address
Transmit instruction to floating execution
Operand access (variable duration)
Floating decode
Move instruction to execution hardware
Execution decode issue to arithmetic unit
Arithmetic unit wait for operand
Storage operand return
Transmit operand to execution hardware
Variable duration
Instruction execution
Segment
1
2
3
4
5
6
7
8
9
10
11
12
13 (or 13.1-b)
Instruction unit function
Main storage control unit and storage functions
Instruction unit function
Main storage control unit and storage functions
Floating-point instruction unit functions
Floating-point execution function
Pipeline cycle time

Figure 3.13 Functional segmentation of a typical storage-to-register floating-point instruction in the IBM 360/91.

Segment / Stage ↓ \ Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	x														
2		x	x				x	x	x	x					
3				x											
4					x										
5						x									
6						x									
7							x								
8								x							
9									x						
10										x					
11											x				
12												x			
13													x	x	x
13.a													x		
13.b														x	

Operand access

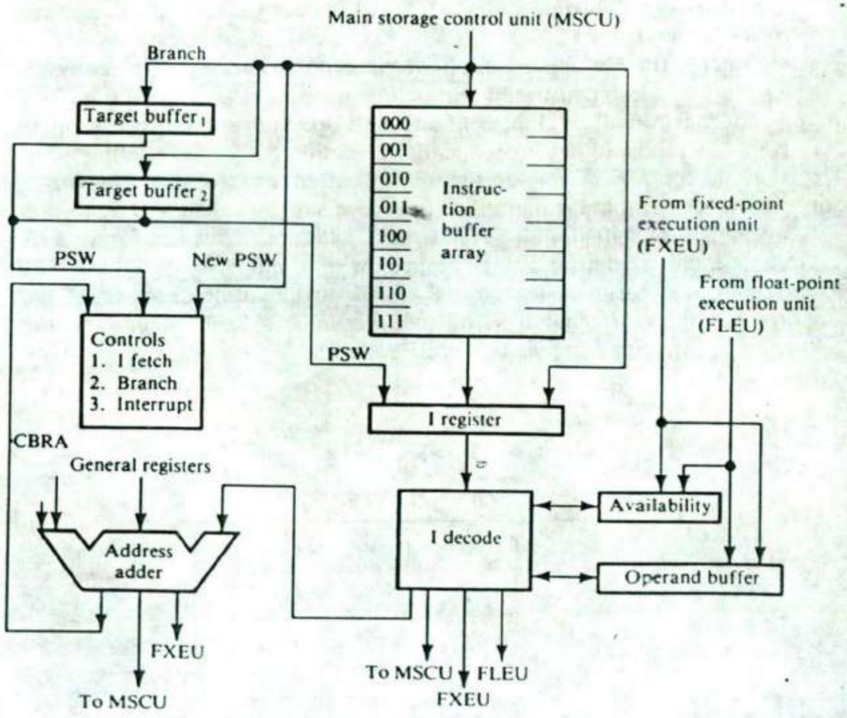
Multiplier

Adder

Figure 3.14 The reservation table for a typical Add or a Multiply function evaluated in the instruction pipeline of IBM 360/91.

3.2.2. Figure 3.14 shows a reservation table for the instruction pipeline in Figure 3.13. At stage 13, the path to follow depends on the instruction types, one using the floating-point adder and the other using the floating-point multiplier-divider. The adder requires two cycles and the multiplier requires six cycles.

The I unit in the Model 91 is specially designed (Figure 3.15) to support the above pipeline operations. A buffer is used to prefetch up to eight double words of instructions. A special controller is designed to handle instruction-fetch, branch, and interrupt conditions. There are two target buffers for branch handling. Sequential instruction-fetch branch and interrupt handling are all built-in hardware features. After decoding, the I unit will dispatch the instruction to the fixed-point E unit, the floating-point E unit, or back to the storage control unit. For memory reference instructions, the operand address is generated by an address adder. This adder is also used for branch-address generation, if needed. The performance of a pipeline processor relies heavily on the continuous supply of



CBRA: conditional branch recovery address

PSW: present state word

Figure 3.15 The instruction unit (I unit) in IBM 360/91 CPU.

instructions and data to the pipeline. When a branch or interrupt occurs, the pipeline will lose many cycles to handle the out-of-sequence operations. Techniques to overcome this difficulty include instruction prefetch, proper buffering, special branch handling, and optimized task scheduling. We will study these techniques in Section 3.3 and check their applications in real-life system designs in Chapter 4.

3.2.2 Arithmetic Pipeline Design Examples

Static and unifunction arithmetic pipelines are introduced in this section with design examples. We will study the pipeline design of Wallace trees for multiple-number addition, which can be applied to designing pipeline multipliers and dividers. Then we will review the arithmetic pipeline designs in the IBM 360/91 for high-speed floating-point addition, multiplication, and division. The method of *convergence division* will be introduced, since it has been widely applied in many commercial computers.

Traditionally, the multiplication of two fixed-point numbers is done by repeated add-shift operations, using an *arithmetic logic unit* (ALU) which has built-in add and shift functions. The number of add-shift operations required is proportional to the operand width. This sequential execution makes the multiplication a very slow process. By examining the multiplication array of two numbers in Figure 3.16, it is clear that the multiplication process is equivalent to the addition of multiple copies of shifted multiplicands, such as the six shown in Figure 3.16.

Multiple-number addition can be realized with a multilevel tree adder. The conventional *carry propagation adder* (CPA) adds two input numbers, say A and B, to produce one output number, called the sum $A + B$. A *carry-save adder* (CSA) receives three input numbers, say A, B, and D, and outputs two numbers,

$$\begin{array}{ccccccccccccl}
 & & & & & & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & =A \\
 & & & & & \times) & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & =B \\
\hline
 & & & & & & a_5b_0 & a_4b_0 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & =W_1 \\
 & & & & & a_5b_1 & a_4b_1 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & =W_2 \\
 & & & & a_5b_2 & a_4b_2 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & =W_3 \\
 & & & a_5b_3 & a_4b_3 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & =W_4 \\
 & & a_5b_4 & a_4b_4 & a_3b_4 & a_2b_4 & a_1b_4 & a_0b_4 & & & & & =W_5 \\
+) & a_5b_5 & a_4b_5 & a_3b_5 & a_2b_5 & a_1b_5 & a_0b_5 & & & & & & =W_6 \\
\hline
P_{11} & P_{10} & P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & =A\times B=P
\end{array}$$

Figure 3.16 The multiplication array of two 6-bit numbers ($A \times B = P$).

the *sum vector* S and the *carry vector* C. Mathematically, we have $A + B + D = S \oplus C$, where $+$ is arithmetic addition and $\oplus$ is bitwise exclusive-or operation.

$$\begin{array}{llcccccccc}
 & A & = & & 1 & 1 & 1 & 1 & 0 & 1 \\
 & B & = & & 0 & 1 & 0 & 1 & 1 & 0 \\
+) & D & = & & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
 & C & = & 1 & 1 & 0 & 1 & 1 & 1 & \\
\oplus) & S & = & & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
 & A + B + D & & & & & & & & \\
\text{or} & C \oplus S & = & 1 & 1 & 1 & 0 & 0 & 1 & 0
\end{array}$$

A carry-propagate adder can be implemented with a cascade of full adders with the carry-out of a lower stage connected to the carry-in of a higher stage. A carry-save adder can be implemented with a set of full adders with all the carry-in terminals serving as the input lines for the third input number D, and all the carry-out terminals serving as the output lines for the carry vector C. In other words, the carry lines of all full adders are not interconnected in a carry-save adder. For the present purpose, we can simply view a CPA as a two-to-one number converter and a CSA as a *three-to-two* number converter.

Now we are ready to show how to use a number of CSAs for multiple-number addition. This, in turn, serves the purpose of pipeline multiplication. This pipeline is designed to multiply two 6-bit numbers, as illustrated in Figure 3.17. There are five pipeline stages. The first stage is for the generation of all $6 \times 6 = 36$ immediate product terms $\{a^i b^j | 0 \leq i \leq 5 \text{ and } 0 \leq j \leq 5\}$, which form the six rows of shifted multiplicands $\{W^i | i = 1, 2, \ldots, 6\}$. The six numbers are then fed into two CSAs in the second stage. In total, four CSAs are interconnected to form a three-level *carry-save adder tree* (from stage two to stage four in the pipeline). This CSA tree merges six numbers into two numbers: the sum vector S and the carry vector C. The final stage is a CPA (carry lookahead may be embedded in it, if the operand length is long) which adds the two numbers S and C to produce the final output, the product $P = A \times B$.

If we restrict the CSA tree to adding only multiple single-bit numbers, we have the well-known bit-slice Wallace trees. In general, a v-level CSA tree can add up to $N(v)$ input numbers, where $N(v)$ is evaluated by the following recursive formula:

$$N(v) = \left\lfloor \frac{N(v) - 1}{2} \right\rfloor \times 3 + N(v - 1) \bmod 2 \qquad \text{with } N(1) = 3 \qquad (3.9)$$

For example, one needs a 10-level CSA tree to add 64 to 94 numbers in one pass through the tree. In other words, a pipeline with 10 stages on the CSA tree is needed to multiply two 64-bit fixed-point numbers in one pass. The floor notation $\lfloor x \rfloor$ refers to the largest integer not greater than x.

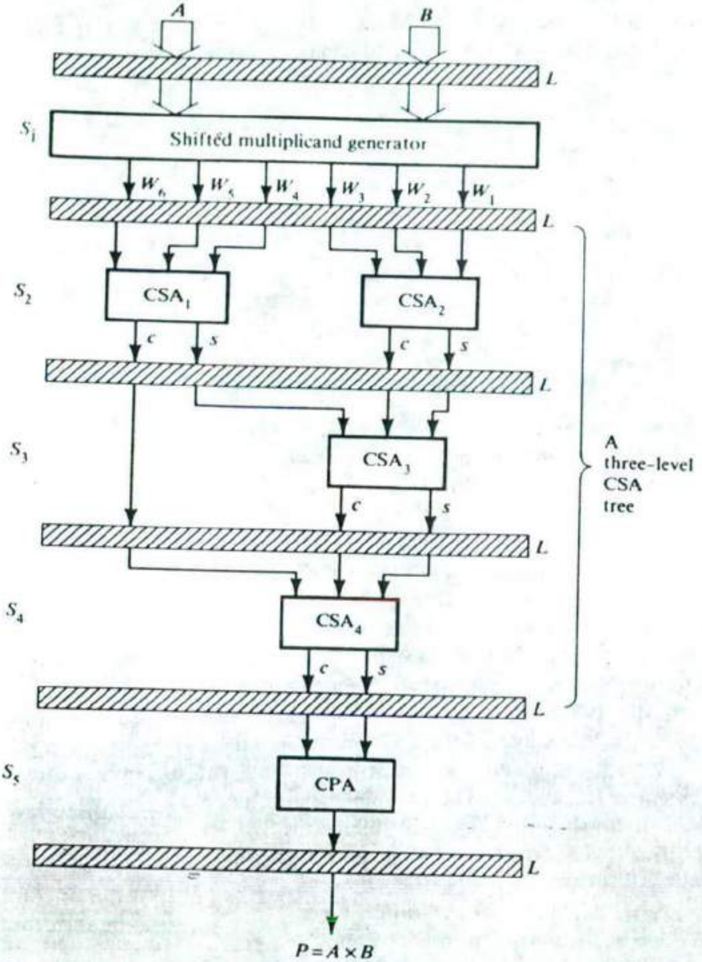


Figure 3.17 A pipelined multiplier built with a CSA tree.

The CSA-tree pipeline can be modified to allow multiple-pass usage by having feedback connections. The concept is illustraied in Figure 3.18. Two input ports of the CSA tree in Figure 3.17 are now connected with the feedback carry vector and sum vector. Suppose that the CPA expanded to require pipeline stages because of increased operand width. We can use this pipeline to merge four additional multiplicands per iteration. If one wishes to multiply two 32-bit numbers, only eight iterations would be needed in this CSA tree with feedback. A complete evaluation of the multiply function in this six-stage pipeline is represented by the reservation table in Figure 3.19. The total evaluation time of this function equals

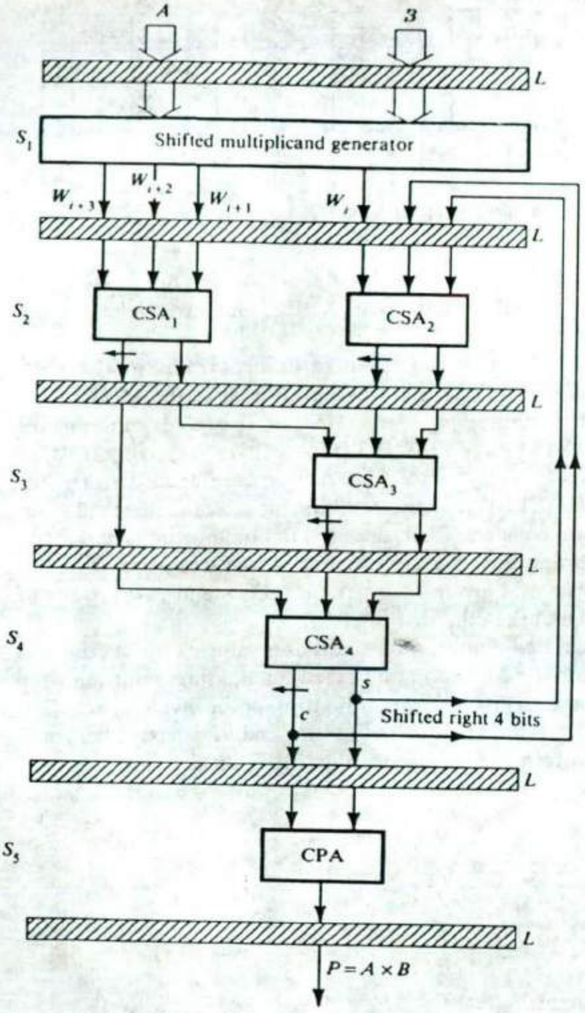


Figure 3.18 A pipelined multiplier using an interative CSA tree for multiple-shift multiplication.

26 clock periods, out of which 24 cycles are needed in the iterative CSA-tree hardware.

This iterative approach saves significantly in hardware compared to the single-pass approach. As a contrast, one-pass 32-input CSA-tree pipeline requires the use of 30 CSAs in eight pipeline stages. The increase in hardware is 26 additional CSAs (each 32-bits wide). The gain in total evaluation time is the saving of

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}	t_{26}
S_1	X			X			X			X			X			X			X			X					
S_2		X			X			X			X			X			X			X			X				
S_3			X			X			X			X			X			X			X			X			
S_4				X			X			X			X			X			X			X			X		
S_5																										X	X

Figure 3.19 The reservation table for the pipelined multiplier in Figure 3.18.

$26 - 11 = 15$ clock periods, where $11 = 1 + 8 + 2$ corresponds to one cycle for the input stage, eight cycles for the one-pass CSA tree, and two cycles for the CPA stages.

We are now ready to present floating-point arithmetic units in the IBM 360/91. A block diagram of the floating-point execution unit in the Model 91 is given in Figure 3.20. *The floating-point instruction unit* (FLIU) communicates with the main storage and the I unit in the CPU (Figure 3.7) in receiving instructions and data retrieval. Successively arrived instructions are queued in the instruction stack. A data buffer (FLB) is used to hold the block of data fetched from the memory. High-speed data registers (FLR) are used to hold operands and intermediate results. A common data bus is used to connect the three components in this E unit. This bus consists of a group of instruction lines and two groups of data lines, one for the FLB and the other for the FLR.

The Add unit can execute all floating-point add and subtract instructions in two machine cycles (120 ns). The M/D unit can execute a floating-point multiply instruction in six machine cycles (360 ns), and a floating-point divide instruction in 18 cycles (1.8 μs). A floating-point operand in the IBM 360/91 can be either short (32 bits) or long (64 bits) with the following two floating point data formats:

Short floating-point data format

Sign	Exponent	Fraction
0	1–7	8–31 (24 bits)

Long floating-point data format

Sign	Exponent	Fraction
0	1–7	8–63 (56 bits)

The Add unit allows three pairs of numbers to be loaded into the three reservation stations. The M/D unit has two reservation stations for two pairs of numbers. The pipeline floating-point adder in the Add unit has physically segmented into two stages. Logically, this pipeline adder can be separated into three algorithmic

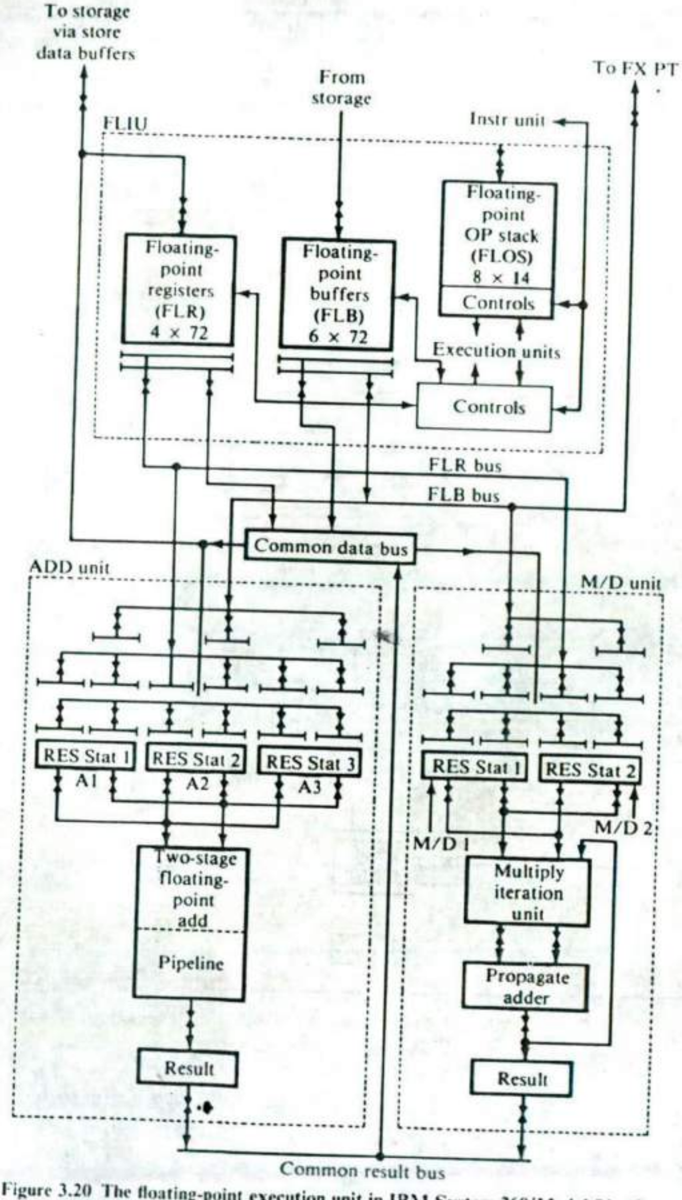


Figure 3.20 The floating-point execution unit in IBM System 360/Model 91. (Courtesy of International Business Machines Corp.)

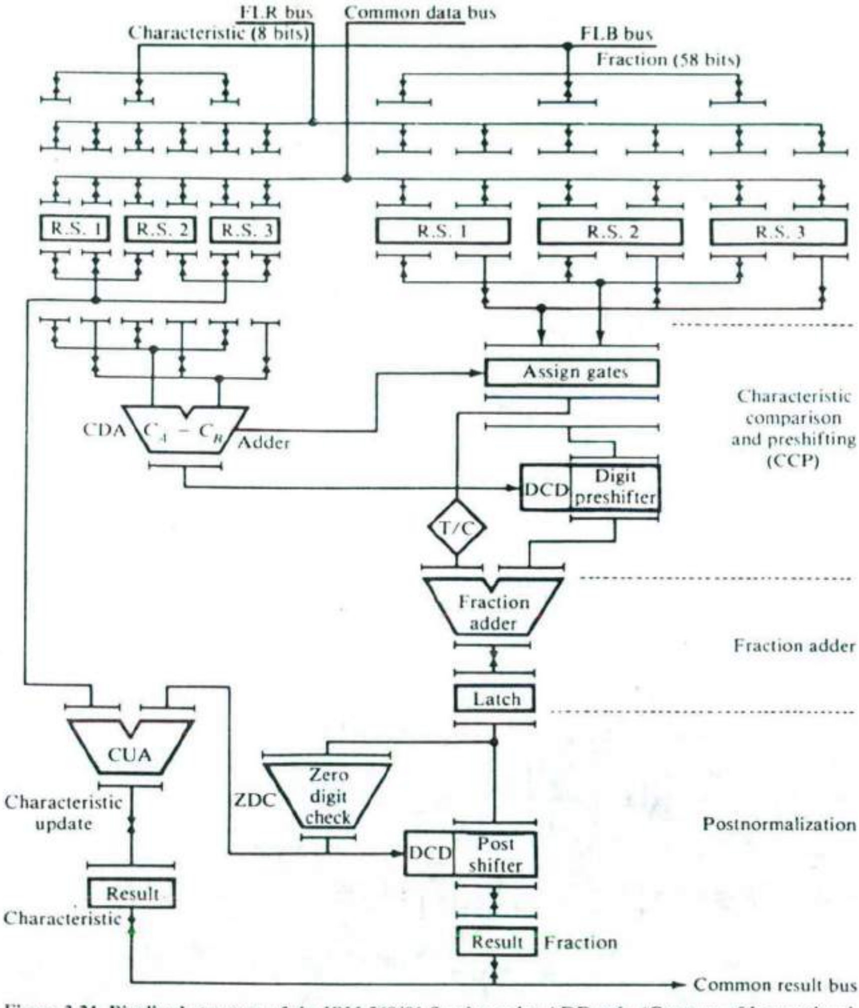


Figure 3.21 Pipelined structure of the IBM 360/91 floating-point ADD unit. (Courtesy of International Business Machines Corp.)

stages, as depicted in Figure 3.21. The functions of these three sections are very similar to what we have discussed in Figure 3.2. Exponent arithmetic and fraction addition and subtraction can be done in parallel. The fraction adder is 56 bits wide and the two exponent adders are each 7 bits wide. Both normalized and unnormalized instructions can be executed, as listed in Table 3.1. The two-cycle speed for double-precision (long-format) floating-point addition matches the instruction-issuing rate of the CPU.

Table 3.1 Typical floating-point instructions in IBM System/360 Model 91

Floating-point instruction	Processing unit	Pipeline cycles needed
Load (S/L)	FLIU	1
Load and test (S/L)	FLIU	1
Store (S/L)	FLIU	1
Load complement (S/L)	Add unit	2
Load positive (S/L)	Add unit	2
Load negative (S/L)	Add unit	2
Add normalized (S/L)	Add unit	2
Add unnormalized (S/L)	Add unit	2
Subtract normalized (S/L)	Add unit	2
Subtract unnormalized (S/L)	Add unit	2
Compare (S/L)	Add unit	2
Halve (S/L)	Add unit	2
Multiply	M/D unit	6
Divide	M/D unit	18

Note: S = short data format; L = long data format; FLIU = floating-point instruction unit; M/D = multiply/divide. Each pipeline cycle is 60 ns.

Floating-point multiply and divide share the same hardware M/D unit in the Model 91. Multiplier recoding techniques are used to speed up the multiplication process. Six multiplicand multiples are generated after the recoding. The complete pipeline structure of the M/D unit is shown in Figure 3.22. The hardware resources can be separated into two parts: the *iterative hardware* for multiple multiplicand addition through a CSA tree, as shown within the dashed-line box, and the *peripheral hardware* for input reservation, prenormalization, multiplier recoding, exponent arithmetic, carry propagation, and output storage. A quadratic convergence division method is applied to generate $Q = N/D$ through dual sequences of the multiplication of N and D by a series of converging factors until the denominator converges to unity. The resulting numerator becomes the desired quotient. Therefore, the aformentioned iterative-multiply hardware can do the job without additional facilities.

The convergence division method has been implemented in many models of the IBM 360/370 and in the CDC 6600/7600 systems. The method is briefly described below. We want to compute the ratio (quotient) $Q = N/D$, where N is the numerator (dividend) and D is the denominator (divisor). Consider normalized binary arithmetic in which $0.5 \leq N < D < 1$ to avoid overflow. Let R_i for $i = 1, 2, \ldots,$ be the successive *converging factors*. One can select

$$R_i = 1 + \delta^{2^{i-1}} \qquad \text{for } i = 1, 2, \ldots, k$$

where $\delta = 1 - D$ and $0 < \delta \leq 0.5$.

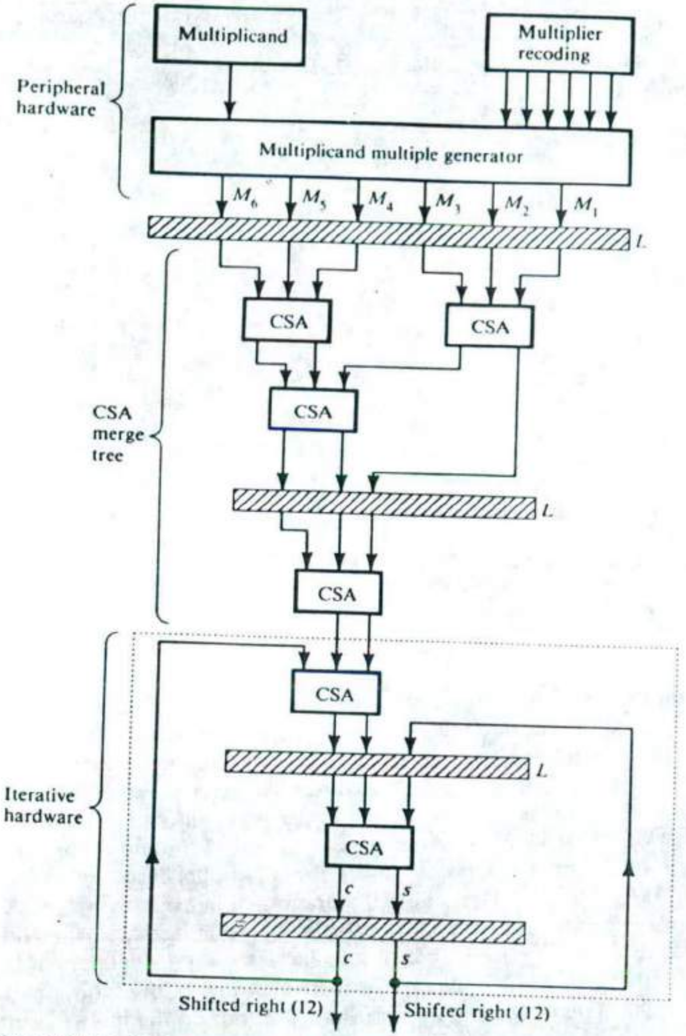


Figure 3.22 Pipelined structure of the IBM 360/91 floating-point MULTIPLY/DIVIDE unit. (Courtesy of International Business Machines Corp.)

To evaluate the quotient Q, we multiply both N and D by R_i, starting from $i = 1$ until a certain stage, say k. Mathematically, we have

$$Q = \frac{N}{D} = \frac{N \times R_1 \times R_2 \times \cdots \times R_k}{D \times R_1 \times R_2 \times \cdots \times R_k}$$
$$= \frac{N \times (1+\delta) \times (1+\delta^2) \times \cdots \times (1+\delta^{2^{k-1}})}{(1-\delta) \times (1+\delta) \times (1+\delta^2) \times \cdots \times (1+\delta^{2^{k-1}})} \tag{3.10}$$

where $D = 1 - \delta$ is being substituted. Denote $D \times R_1 \times R_2 \times \cdots \times R_i = D_i$ and $N \times R_1 \times R_2 \times \cdots \times R_i = N_i$ for $i = 1, 2, \ldots, k$. We have

$$\begin{aligned} D_i &= (1-\delta)(1+\delta)(1+\delta^2)\cdots(1+\delta^{2^{i-1}}) \\ &= (1-\delta^2)(1+\delta^2)(1+\delta^4)\cdots(1+\delta^{2^{i-1}}) \\ &= 1 - \delta^{2^i} \end{aligned}$$

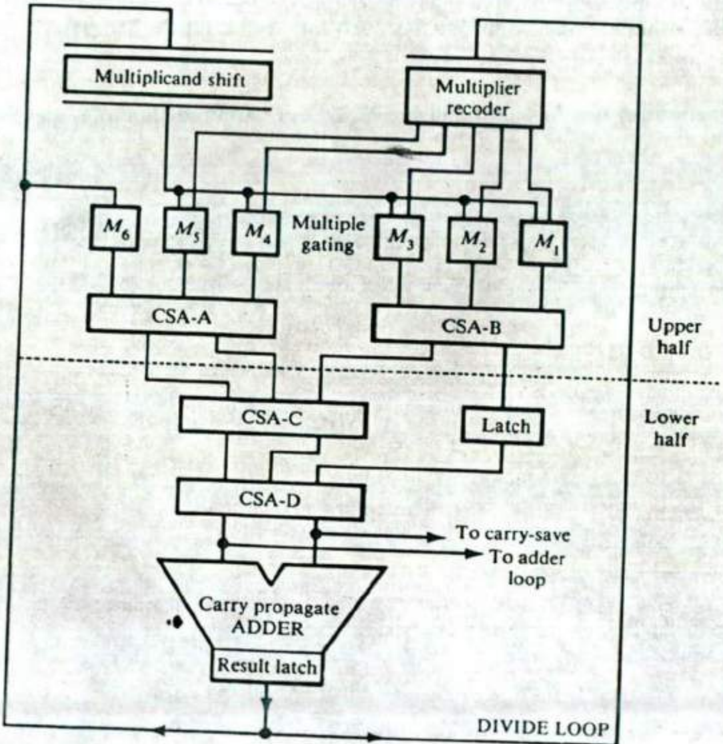


Figure 3.23 Convergence divide loop using the iterative hardware in IBM 360/91 floating-point MULTIPLY/DIVIDE unit. (Courtesy of International Business Machines Corp.)

It is clear that $0.5 \leq D < D_1 < D_2 < \cdots < D_k \to 1$, because of the fact $0.5 > \delta > \delta^2 > \delta^4 > \cdots > \delta^{2^k} > 0$. When the number of iterations k is sufficiently large, $\delta^{2^k} \to 0$ and thus $D_k \to 1$. We end up with

$$N_k = N \times (1 + \delta) \times (1 + \delta^2) \times \cdots \times (1 + \delta^{2^{k-1}}) \qquad (3.11)$$

which equals the desired quotient $Q = N/D = N_k$. The smaller the fraction δ, the faster will be the convergence process.

The multiply hardware in the M/D unit can be used iteratively to carry out the above convergence division of two 56-bit fractions in the Model 91. Figure 3.23 shows the *divide loop* when utilizing the iterative hardware in Figure 3.18 for convergence division. A time chart is given in Figure 3.24 to show the two overlapped sequences of multiplications (Eq. 3.9) carried out simultaneously by the upper half and the lower half of the divide loop. Five iterations are needed ($k = 5$) to converge the numerator into the desired quotient, the factor $\delta^{2^5} = \delta^{32}$ becoming small enough to be considered as zero within the limit of machine precision. In the M/D unit, 12 bits are being shifter per iteration by the *multiplier recoding* logic. The theory of multiplier recoding using redundant number representation can be found in Hwang's book on Computer Arithmetic (1979).

Concurrency in arithmetic operations has been exploited in the IBM 360/91 in four areas:

1. Concurrent operations of the Add unit and the M/D unit within the floating-point E unit

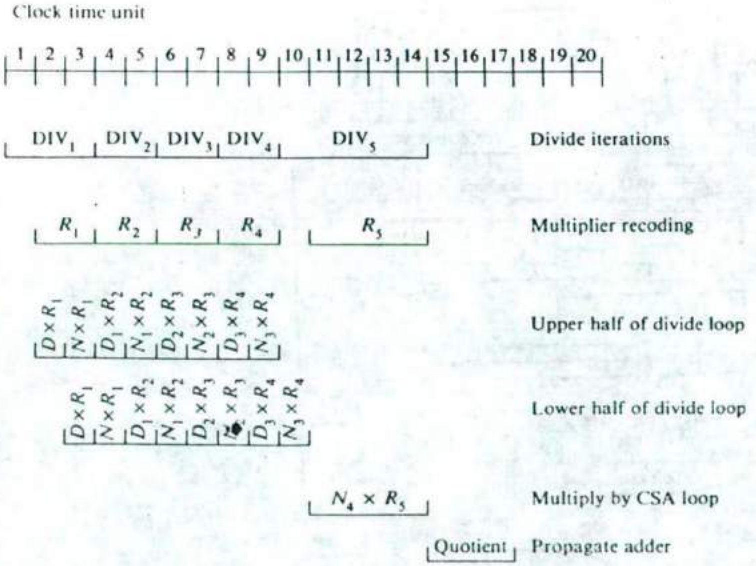


Figure 3.24 Timing chart showing the overlapped execution in the divide loop shown in Figure 3.23.

2. Pipelined executions in the Add unit and the M/D unit
3. Concurrent execution within the iterative multiply hardware
4. Concurrent operations of the fixed-point E unit and floating-point E unit

The hardware examines multiple instructions and optimizes the program execution by allowing simultaneous execution of multiple independent instructions.

3.2.3 Multifunction and Array Pipelines

In this section, we study static and multifunction-arithmetic pipelines and introduce the concept of *array pipelining*, with an example in matrix arithmetic. By definition, a multifunction pipeline can perform different functions at different times upon program control or firmware control. We present a four-function pipeline proposed by Kamal, et al (1974). This pipeline can perform *multiply*, *divide*, *squaring*, and *sqrt* (*sq*uare *root*) operations. Two types of building cells are used in this four-function pipeline construction. The two cell types are specified in Figure 3.25 by boolean equations. The *A* cell is a controlled 1-bit adder-subtractor with bypass signal lines. The *K* cells are for function selection and boundary carry control.

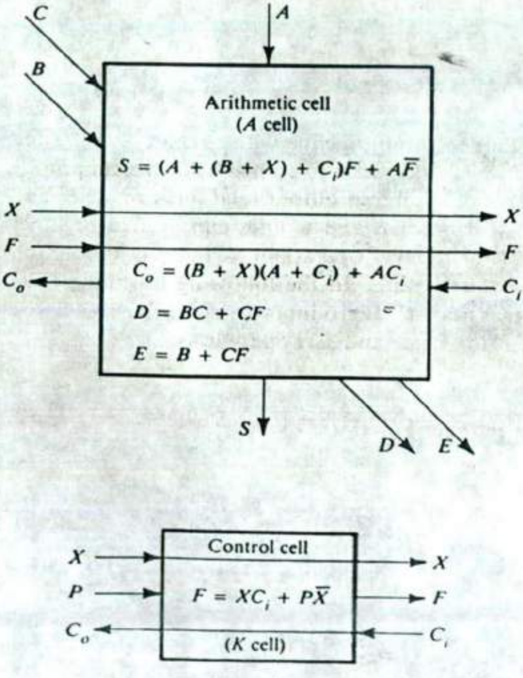


Figure 3.25 Building blocks used in the construction of the four-function arithmetic pipeline in Figure 3.26.

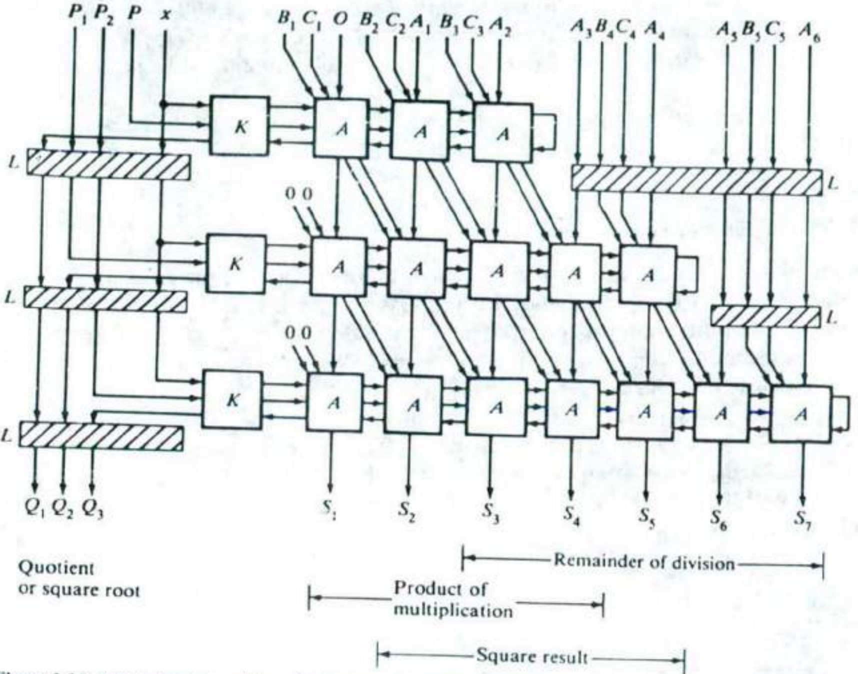


Figure 3.26 A four-function arithmetic pipeline. (Courtesy of *IEEE Trans. Computers*, Kamal et al. 1974.)

The schematic design of the four-function pipeline with A cells, K cells, and interface latches is shown in Figure 3.26. Arithmetic computations being implemented are specified below in terms of input and output relationships. A, B, and P lines are for operand inputs. All pairs of B and C lines can be tied together (with the same input value) except for the *sqrt* operation. K and X are function control signals. S and Q lines are for outputs. In the following arithmetic I/O equations, all unspecified input lines assume a zero input value (unless otherwise noted). The control signal $X = 1$ is for *divide* and *sqrt* operations.

Multiply operation

$$(A_1 A_2 A_3 A_4) + (B_1 B_2 B_3) \times (P_1 P_2) = (S_1 S_2 S_3 S_4) \qquad (3.12)$$

Divide operation

$$(A_1 A_2 A_3 A_4) - (B_1 B_2 B_3) = \underbrace{(Q_1 Q_2 Q_3)}_{\text{Quotient}} \text{ plus } \underbrace{(S_3 S_4 S_5)}_{\text{remainder}} \qquad (3.13)$$

Squaring operation

$$(P_1 P_2)^2 = (S_2 S_3 S_4 S_5) \qquad (3.14)$$

Square rooting operation

$$\sqrt{(A_1A_2\cdots A_6)} = \underbrace{(Q_1Q_2Q_3)}_{\text{Root}} \text{ plus } \underbrace{(S_3S_4\cdots S_7)}_{\text{Remainder}} \qquad (3.15)$$

provided $B_2C_2 = 01$ and $B_iC_i = 10$ for $i > 3$.

Each stage of the pipeline is essentially a ripple-carry adder-subtractor. Results for multiply, squaring, and for the remainders of *divide* and *sqrt* are generated at the bottom stage of the pipeline. Quotient bits and root bits are generated at the left end of each pipeline stage in a sequential manner.

We have only shown three pipeline stages in Figure 3.26. One can add additional stages at the bottom of the pipeline. The number of A cells used in the kth stage equals $2k + 1$. Of course, more pipeline stages imply higher precision in input-output number representations. This four-function pipeline must operate in a static manner, one function at a time. In this sense, it is not a dynamic pipeline. So far, dynamic pipelines have never been implemented in commercial computers.

Texas Instruments' Advanced Scientific Computer (ASC) was the first vector processor that was installed with multifunction pipelines in its arithmetic processors. We will review the system architecture of ASC in Chapter 4. Only the pipeline arithmetic units of ASC are studied in this section. The ASC arithmetic pipeline consists of eight stages, as illustrated in Figure 3.27. All the interconnection routes among the eight stages are shown. This pipeline can perform either fixed-point or floating-point arithmetic functions and many logical-shifting operations over scalar and vector operands of lengths 16, 32, or 64 bits. The basic pipeline clock period is 60 ns. One to four such pipelines can be installed in the ASC system. The maximum speed is about 0.5 to 1.5 megaflops for scalar processing and 3 to 10 megaflops for vector processing. The results of previous executions can be routed back as future inputs, such as those needed in vector dot product operations.

Different arithmetic logic instructions are allowed to use different connecting paths through the pipeline. Figure 3.28 shows four interconnection patterns of the ASC pipeline for the evaluation of the functions: *fixed-point add*, *floating-point add*, *fixed-point multiply*, and *floating-point vector dot product*. The receiver and output stages are used by all instructions. Simple instructions like *load*, *store*, *and*, and *or* only use these two stages. The multiply stage performs 32 × 32 multiplication. The multiply stage produces two 64-bit results, called *pseudo sum* and *pseudo carry*, which are sent to the accumulate stage or the 64-bit add stage to produce the desired product. The accumulator can also feed its output back to itself when double-precision multiply or divide operations are demanded. This feedback is also used to implement fixed-point vector dot product instruction.

The exponent subtract stage determines the exponent difference and sends this shift count to the align stage to align fractions for *floating-point add* or *subtract* instructions. All right-shift operations are also implemented in this align stage. The normalize stage does the floating-point normalization, all left-shift operations, and conversions between fixed-point and floating-point operands. With many functions, the ASC pipeline is still a static one, performing only one

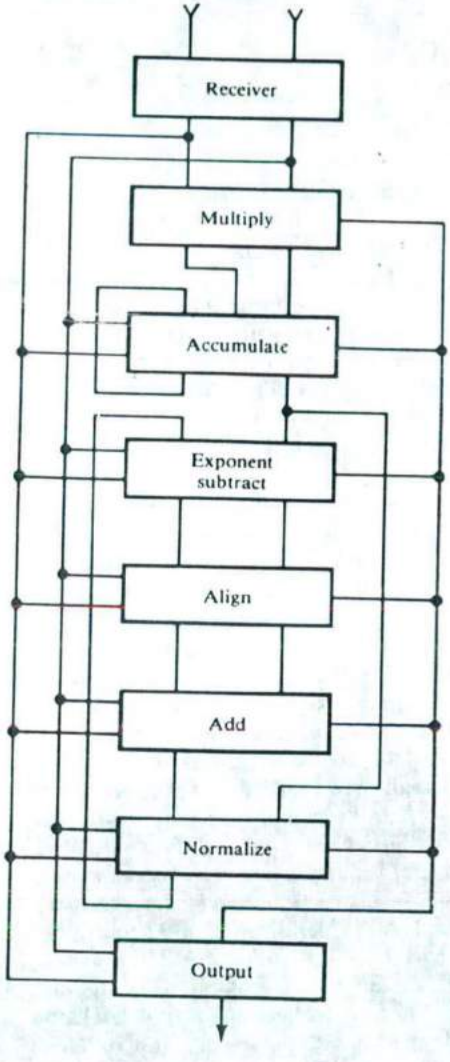


Figure 3.27 All possible interstage connections in the TI-ASC arithmetic pipeline. (Courtesy of Stephenson 1973.)

function at a time. Reconfiguration is needed when the pipeline switches its function from one to another. Multifunction pipelines offer better resource utilization and higher application flexibility. However, their control is much more complicated than their unifunction counterparts. Most of today's pipeline computers choose to use unifunction pipes because of cost effectiveness.

Array pipelines are two-dimensional pipelines with multiple data-flow streams for high-level arithmetic computations, such as matrix multiplication, inversion,

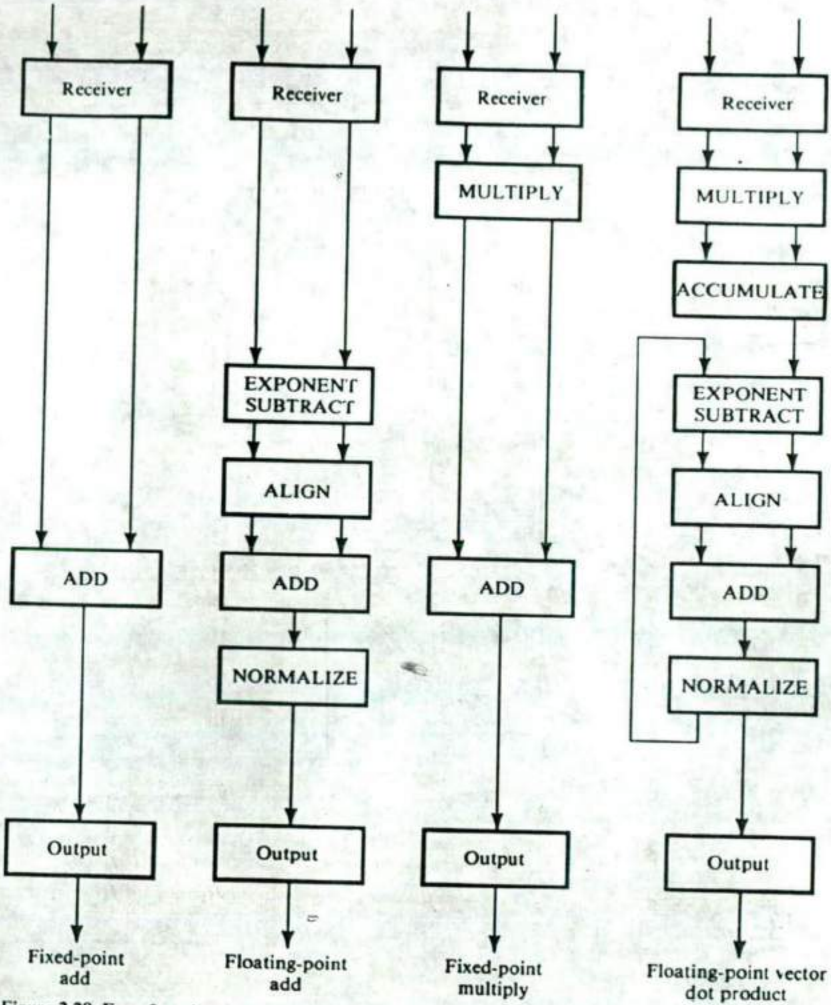


Figure 3.28 Four functional configurations in the TI-ASC arithmetic pipeline.

and L-U decomposition. The pipeline is usually constructed with a cellular array of arithmetic units. The cellular array is usually regularly structured and suitable for VLSI implementation. Presented below is only an introductory sample design of an array pipeline. This array is pipelined in three data-flow directions for the repeated multiplication of pairs of compatible matrices. The basic building blocks in the array are the *M cells*. Each *M* cell performs an additive inner-product operation as illustrated in Figure 3.29.

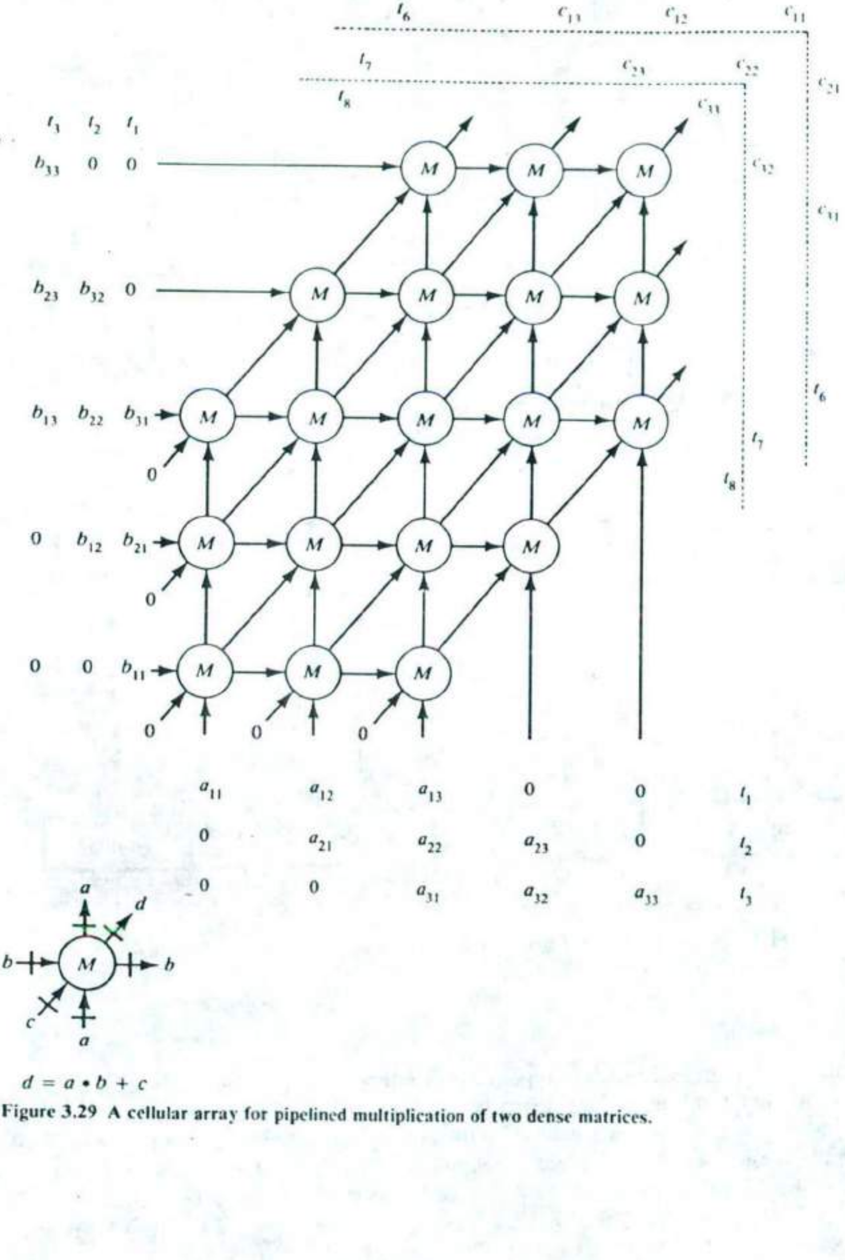


Figure 3.29 A cellular array for pipelined multiplication of two dense matrices.

Each M cell has the three input operands a, b, and c and the three outputs $a' = a$, $b' = b$, and $d = a \times b + c$. Fast latches (registers) are used at all input-output terminals and all interconnecting paths in the array pipeline. All latches are synchronously controlled by the same clock. Adjacency between cells is defined in three orientations: horizontal, vertical, and diagonal (45°) directions. The array shown in Figure 3.29 performs the multiplication of two 3×3 dense matrices $A \cdot B = C$.

$$A \cdot B = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = C \quad (3.16)$$

The input matrices are fed into the array in the horizontal and vertical directions. Three clock periods are needed for inputing the matrix entries: one row at a time for the A matrix and one column at a time for the B matrix. Dummy zero inputs are marked at unused input lines. "Don't care" conditions at the output lines are left blank. In general, to multiply two $(n \times n)$ matrices requires $3n^2 - 4n + 2$ M cells. It takes $3n - 1$ clock periods to complete the multiply process. When the matrix size becomes too large, the global array approach will pose a serious problem for monolithic chip implementation because of density and I/O packaging constraints. For a practical design of array pipelines, a block-partitioning approach will be introduced in Chapter 10 for VLSI matrix arithmetic. VLSI array-pipeline structures will be treated there with the potential real-time applications.

3.3 PRINCIPLES OF DESIGNING PIPELINE PROCESSORS

Key design problems of pipeline processors are studied in this section. We begin with a review of various instruction-prefetch and branch-control strategies for designing pipelined instruction units. Data-buffering and busing structures are presented for smoothing pipelined operations to avoid congestion. We will study internal data-forwarding and register-tagging techniques by examining instruction-dependence relations. The detection and resolution of logic hazards in pipelines will be described. Principles of job sequencing in a pipeline will be studied with reservation tables to avoid collisions in utilizing pipeline resources. Finally, we will consider the problems of designing dynamic pipelines and the necessary system supports for pipeline reconfigurations.

3.3.1 Instruction Prefetch and Branch Handling

From the viewpoint of overlapped instruction execution sequencing for pipelined processing, the instruction mixes in typical computer programs can be classified into four types, as shown in Table 3.2. The arithmetic load operations constitute 60 percent of a typical computer program. These are mainly data manipulation

Table 3.2 Typical instruction mix and pipeline cycle allocation

Instruction type and mix rate / Segment function				Conditional branch type	
	Arithmetic/load type, 60%	Store type, 15%	Branch type, 5%	Yes, 12%	No, 8%
Instruction fetch	6	6	6	6	6
Decode	2	2	2	2	2
Condition test				1	1
Operand address calculation	2	2	2	2	2
Operaad fetch(es)	6-12				
Arithmetic logic execution	4-8				
Store result		6			
Update PC and flags	1	1	1	1	1
Total pipeline cycles	21-31	17	11	12	12

operations which require one or two operand fetches. The execution of different arithmetic operations requires a different number of pipeline cycles. The store-type operation does not require a fetch operand, but memory access is needed to store the data. The branch-type operation corresponds to an unconditional jump. There are two possible paths for a conditional branch operation. The *yes* path requires the calculation of the new address being branched to, whereas the *no* path proceeds to the next sequential instruction in the program. The arithmetic-load and store instructions do not alter the sequential execution order of the program. The branch instructions (25 percent in typical programs) may alter the program counter (PC) in order to jump to a program location other than the next instruction. Different types of instructions require different cycle allocations. The branch types of instructions will cause some damaging effects on the pipeline performance.

Some functions, like interrupt and branch, produce damaging effects on the performance of pipeline computers. When instruction I is being executed, the occurrence of an interrupt postpones the execution of instruction $I + 1$ until the interrupting request has been serviced. Generally, there are two types of interrupts. *Precise interrupts* are caused by illegal operation codes found in instructions, which can be detected during the decoding stage. The other type, *imprecise interrupts*, is caused by defaults from storage, address, and execution functions.

Since decoding is usually the first stage of an instruction pipeline, an interrupt on instruction I prohibits instruction $I + 1$ from entering the pipeline. However, those instructions preceding instruction I that have not yet emerged from the pipeline continue to run until the pipeline is drained. Then the interrupt routine is serviced. An imprecise interrupt occurs usually when the instruction is halfway through the pipeline and subsequent instructions are already admitted into the pipeline. When an interrupt of this kind occurs, no new instructions are allowed to enter the pipeline, but all the incompleted instructions inside the pipeline, whether they precede or follow the interrupted instruction, will be completed before the processing unit is switched to service the interrupt.

In the Star-100 system, the pipelines are dedicated to vector-oriented arithmetic operations. In order to handle interrupts during the execution of a vector instruction, special interrupt buffer areas are needed to hold addresses, delimiters, field lengths, etc., that are needed to restart the vector instructions after an interrupt. This demands a capable recovery mechanism for handling unpredictable and imprecise interrupts.

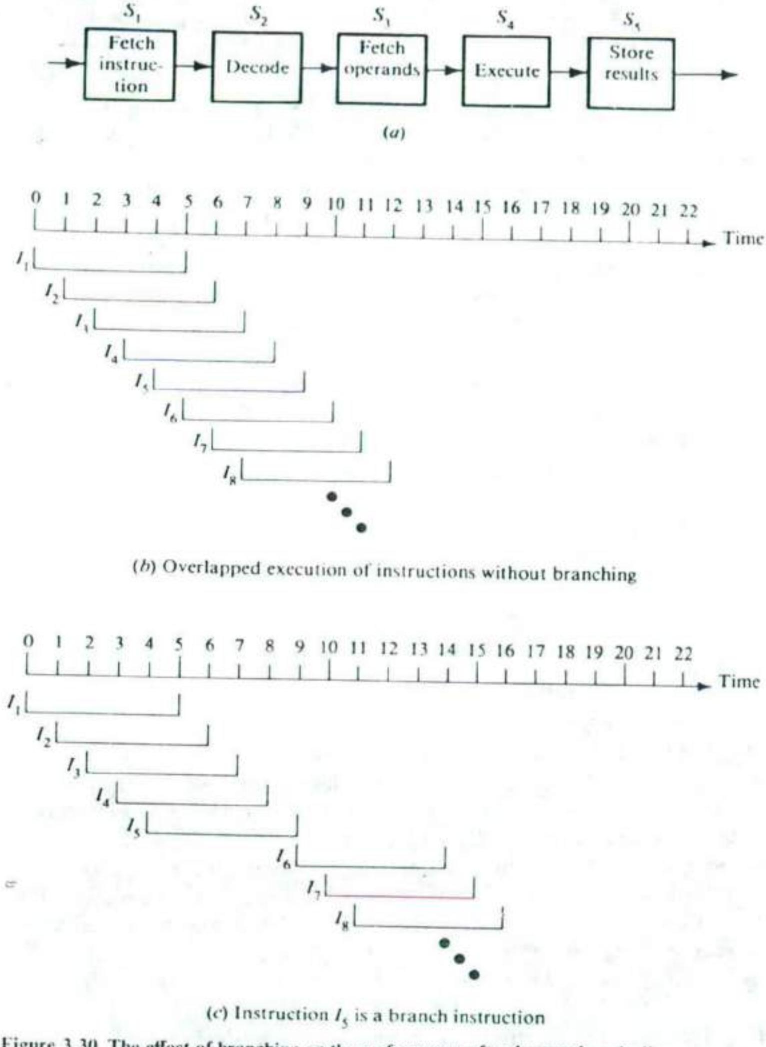
For the Cray-1 computer, the interrupt system is built around an *exchange package*. To change tasks, it is necessary to save the current processor state and to load a new processor state. The Cray-1 does this semiautomatically when an interrupt occurs or when a program encounters an *exit* instruction. Under such circumstances, the Cray-1 saves the eight scalar registers, the eight address registers, the program counter, and the monitor flags. These are packed into 16 words and swapped with a block whose address is specified by a hardware exchange address register. However, the exchange package does not contain all the hardware state information, so software *interrupt handlers* must save the rest of the states. "The rest" includes 512 words of vector registers, 128 words of intermediate registers, a vector mask, and a real-time clock.

The effect of branching on pipeline performance is described below by a linear instruction pipeline consisting of five segments: *instruction fetch, decode, operand fetch, execute*, and *store results*. Possible memory conflicts between overlapped fetches are ignored and a sufficiently large cache memory (instruction-data buffers) is used in the following analysis.

As illustrated in Figure 3.30, the instruction pipeline executes a stream of instructions continuously in an overlapped fashion if branch-type instructions do not appear. Under such circumstances, once the pipeline is filled up with sequential instructions (nonbranch type), the pipeline completes the execution of one instruction per a fixed latency (usually one or two clock periods).

On the other hand, a branch instruction entering the pipeline may be halfway down the pipe (such as a "successful" conditional branch instruction) before a branch decision is made. This will cause the program counter to be loaded with the new address to which the program should be directed, making all prefetched instructions (either in the cache memory or already in the pipeline) useless. The next instruction cannot be initiated until the completion of the current branch-instruction cycle. This causes extra time delays in order to drain the pipeline, as depicted in Figure 3.26*c*. The overlapped action is suspended and the pipeline must be drained at the end of the branch cycle. The continuous flow of instructions into the pipeline is thus temporarily interrupted because of the presence of a branch instruction.

In general, the higher the percentage of branch-type instructions in a program, the slower a program will run on a pipeline processor. This certainly does not merit the concept of pipelining. An analytical estimation of the effect of branching on an n-segment instruction pipeline is given below. The instruction cycle is assumed to



(*b*) Overlapped execution of instructions without branching

(*c*) Instruction I_5 is a branch instruction

Figure 3.30 The effect of branching on the performance of an instruction pipeline.

include *n* pipeline cycles. For example, one instruction cycle is equal to five pipeline clock periods in Figure 3.30. Clearly, if a branch instruction does not occur, the performance would be one instruction per each pipeline cycle. Let *p* be the probability of a conditional branch instruction in a typical program (20 percent by Table 3.2) and *q* be the probability that a branch is successful ($\frac{12}{20}$ = 60 percent by Table 3.2). Suppose that there are *m* instructions waiting to be executed through the pipeline. The number of instructions that cause successful branches equals $m \cdot p \cdot q$. Since $(n - 1)/n$ extra time delay is needed for each successful branch instruction, the total instruction cycles required to process these m instructions equal $(1/n)(n + m - 1) + (m \cdot p \cdot q)(n - 1)/n$. As m becomes very large, the performance of the instruction pipeline is measured by the average number of instructions executed per instruction cycle:

$$\lim_{m \to \infty} \frac{m}{(n + m - 1)\, n + m \cdot p \cdot q \cdot (n - 1)/n} = \frac{n}{1 + pq(n - 1)} \tag{3.17}$$

When $p = 0$ (no branch instructions encountered), the above measure reduces to n instructions per n pipeline clocks, which is ideal. In reality, the above ratio is always less than n. For example, with $n = 5$, $p = 20$ percent, and $q = 60$ percent, we have the performance of 3.24 instructions per instruction cycle (or 5 pipeline cycles), which is less than the ideal execution rate of 5 instructions per 5 pipeline cycles. In other words, an average of 35.2 percent cycles may be wasted because of branching. In order to cope with the damaging effects of branch instructions, various mechanisms have been developed in pipeline computers.

We have studied instruction prefetch in Section 3.2.1, where the I unit in the IBM 360/91 was described (Figure 3.15). Formally, a prefetching strategy can be stated as follows: Instruction words ahead of the one currently being decoded are fetched from the memory before the instruction-decoding unit requests them. The prefetch of instructions is modeled in Figure 3.31. The memory is assumed to be interleaved and can accept requests at one per cycle. All requests require T cycles to return from memory.

There are two prefetch buffers of sizes s and t instruction words. The s-size buffer holds instructions fetched during the sequential part of a run. When a *branch* is successful, the entire buffer is invalidated. The other buffer holds instructions fetched from the target of a conditional branch. When a conditional branch is resolved and determined to be unsuccessful, the contents of this buffer are invalidated. The decoder requests instruction words at a maximum rate of one per r cycles. If the instruction requested by the decoder is available in the sequential buffer for sequential instructions, or is in the target buffer if a conditional branch has just been resolved and is successful, it enters the decoder with zero delay. Otherwise, the decoder is idle until the instruction returns from memory.

Except for *jump* instructions, all decoded instructions enter the execution pipeline, where E units are required to complete execution. If the decoded instruction is an unconditional branch, the instruction word at the target of the *jump* is requested immediately by the decoder and decoding ceases until the target instruction returns from the memory. The pipeline will see the full memory latency time T, since there was no opportunity for target prefetching.

If the decoded instruction is a *conditional branch*, sequential prefetching is suspended during the E cycles it is being executed. The instruction simultaneously enters the execution pipeline, but no more instructions are decoded until the branch is resolved at the end of E units. Instructions are prefetched from the target memory

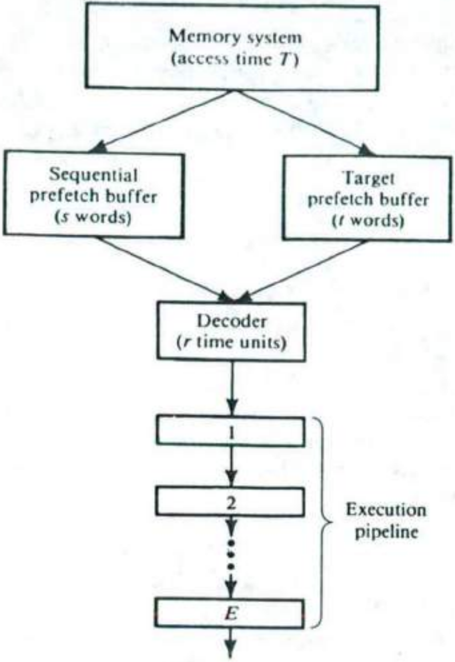


Figure 3.31 An instruction pipeline with both sequential and target prefetch buffers.

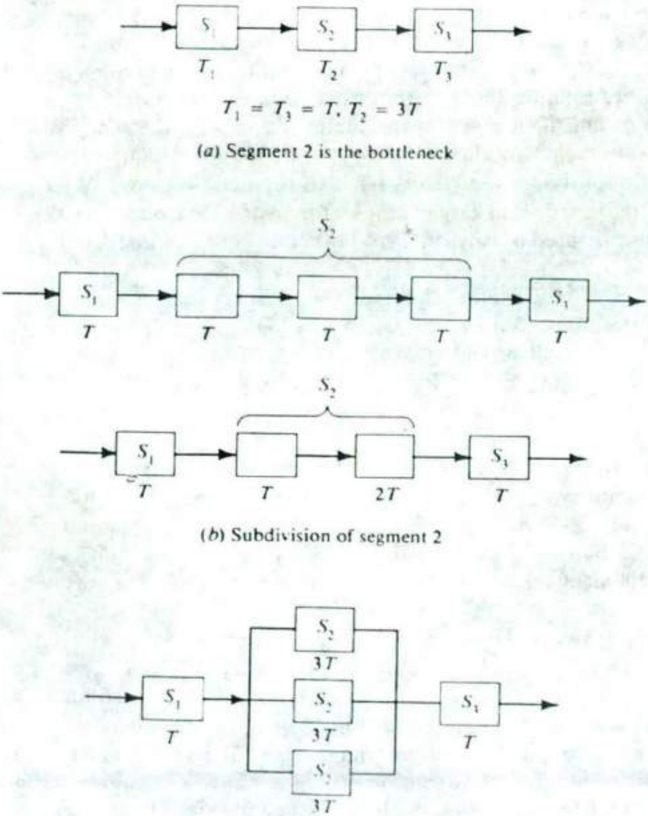
address of the conditional branch instruction. Requests for t target instructions are issued at the rate of one per cycle. Once the branch is resolved, target prefetching becomes unnecessary.

If the branch is successful, the target instruction stream becomes the sequential stream, and instructions are requested every r time units from this stream. Execution of this new stream begins when the target of the branch returns from memory, or whenever E units have elapsed, whichever is later. If the branch is unsuccessful, instruction requests are initiated every r units of time following the branch resolution and continue until the next branch or jump is decoded.

Instruction prefetching reduces the damaging effects of branching. In the IBM 360/91, a loop mode and back-eight test are designed with the help of a branch-target buffer. The idea is to keep a short loop of eight instruction double words or less completely in the branch-target buffer so that no additional memory accesses are needed until the loop mode is removed by the final branching out. This replacement of the condition mode by the local loop mode is established once a successful branch results and the back-eight test is satisfied. The load lookahead mechanism in the ASC system follows a similar approach. Another approach is to prefetch into the instruction buffer one (by guess) or even both instruction sequences forked at a conditional branch instruction. After the test result becomes available, one of the two prefetched instruction sequences will be executed and the other discarded. This branch-target prefetch approach may increase the utilization of the pipeline CPU and thus increase the total system throughput.

3.3.2 Data Buffering and Busing Structures

The processing speeds of pipeline segments are usually unequal. Consider the example pipeline in Figure 3.32*a*, with three segments having delays T_1, T_2, and T_3, respectively. If $T_1 = T_3 = T$ and $T_2 = 3T$, obviously segment S_2 is the bottleneck. The throughput of the pipeline is inversely proportional to the bottleneck. Therefore, it is desirable to remove the bottleneck which causes the unnecessary congestion. One obvious method is to subdivide the bottleneck. Figure 3.32*b* shows two different subdivisions of segment S_2. The throughput is increased



(*a*) Segment 2 is the bottleneck

(*b*) Subdivision of segment 2

(*c*) Replication of segment 2

Figure 3.32 Subdivision or replication to alleviate the bottleneck in a pipeline.

in either case. However, if the bottleneck is not subdivisible, using duplicates of the bottleneck in parallel is another way to smooth congestions, as depicted in Figure 3.32*c*. The control and synchronization of tasks in parallel segments are much more complex than those for cascaded segments.

Data and instruction buffers Another method to smooth the traffic flow in a pipeline is to use buffers to close up the speed gap between the memory accesses for either instructions or operands and the arithmetic logic executions in the functional pipes. The instruction or operand buffers provide a continuous supply of instructions or operands to the appropriate pipeline units. Buffering can avoid unnecessary idling of the processing stages caused by memory-access conflicts or by unexpected branching or interrupts. Sometimes the entire loop's instructions can be stored in the buffer to avoid repeated fetch of the same instruction loop, if the buffer size is sufficiently large. The amount of buffering is usually very large in pipeline computers.

The use of instruction buffers and various data buffers in the IBM System/360 Model 91 is shown in Figure 3.33. Three buffer types are used for various instruction and data types. Instructions are first fetched to the instruction-fetch buffers (64 bits each) before sending them to the instruction unit (Figure 3.15). After decoding, fixed-point and floating-point instructions and data are sent to their dedicated buffers, as labeled in Figure 3.33. The store-address and data buffers are used for continuously storing results back to the main memory. We have already explained the function of target buffers for instruction prefetches. The storage-conflict buffers are used only when memory-access conflicts are taking place.

In the STAR-100 system, a 64-word (of 128 bits each) buffer is used to temporarily hold the input data stream until operands are properly aligned. In addition, there is an instruction buffer which provides for the storage of thirty-two 64-bit instructions. Eight 64-bit words in the instruction buffer will be filled up by one memory fetch. The buffer supplies a continuous stream of instructions to be executed, despite memory-access conflicts.

In the TI-ASC system, two eight-word buffers are utilized to balance the stream of instructions from the memory to the execution unit. A *memory buffer unit* has three double buffers, X, Y, and Z. Two buffers (X and Y) are used to hold the input operands and the third (Z buffer) is used for the output results. These buffers greatly alleviate the problem of mismatched bandwidths between the memory and the arithmetic pipelines.

In the Floating-Point Systems AP-120B, there are two blocks of registers serving as operand buffers for the pipeline multiplier and adder. In the Cray-1 system, eight 64-bit scalar registers and sixty-four 64-bit data buffers are used for scalar operands. Eight 64-word vector registers are used as operand buffers for vector operations. There are also four instruction buffers in the Cray-1, each consisting of sixty-four 16-bit registers. With four instruction buffers, substantial program segments can be prefetched to allow on-line arithmetic logic operations through the functional pipes.

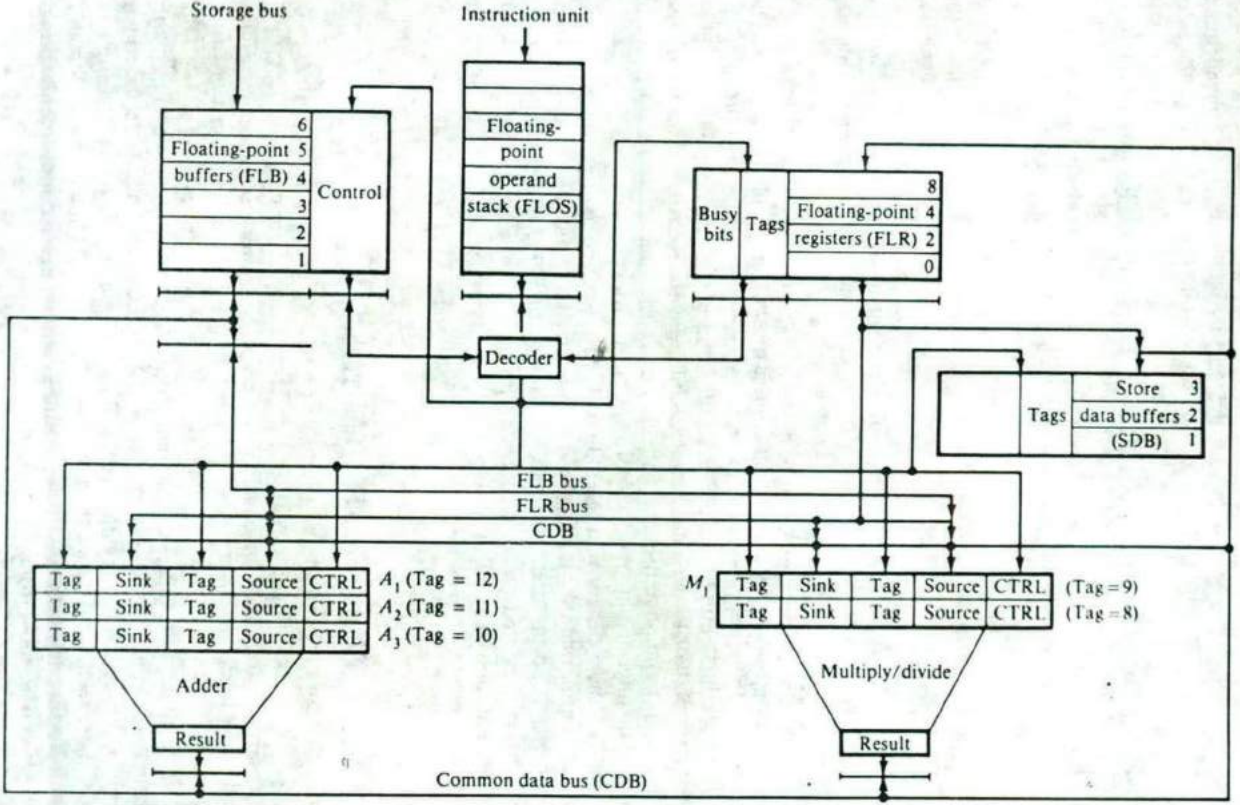


Figure 3.33 Data buffers, transfer paths, reservation stations, and common data bus (CDB) in the IBM System/360 Model 91 floating-point execution unit. (Courtesy of International Business Machines Corp.)

Busing structures Ideally, the subfunction being executed by one stage should be independent of the other subfunctions being executed by the remaining stages; otherwise, some processes in the pipeline must be halted until the dependency is removed. For example, when one instruction waiting to be executed is first to be modified by a future instruction, the execution of this instruction must be suspended until the dependency is released. Another example is the conflicting use of some registers or memory locations by different segments of a pipeline. These problems cause additional time delays. An efficient internal busing structure is desired to route results to the requesting stations with minimum time delays.

In the TI-ASC system, once instruction dependency is recognized, only independent instructions are distributed over the arithmetic units. Update capability is incorporated into the processor by transferring the contents of the Z buffer to the X buffer or the Y buffer. With such a busing structure, time delays due to dependency are significantly reduced. In the STAR-100 system, direct routes are established from the output transmit segment to the input receive segment. Thus, no registers are required to store the intermediate results, which causes a significant saving of data-forwarding delays.

In the AP-120B or FPS-164 attached processors, the busing structures are even more sophisticated. Seven data buses provide multiple data paths. The output of the floating-point adder in the AP-120B can be directly routed back to the input of the floating-point adder, to the input of the floating-point multiplier, to the data pad, or to the data memory. Similar busing is provided for the output of the floating-point multiplier. This eliminates the time delay to store and to retrieve the intermediate results to or from the registers.

In the Cray-1 system, multiple data paths are also used to interconnect various functional units and the register and memory files. Although efficient busing structures can reduce the damaging effects of instruction interdependencies, a great burden is still exerted on the compiler to produce codes exposing parallelism. If independent and dependent instructions are intermixed appropriately, more concurrent processing can take place in a multiple-pipe computer.

3.3.3 Internal Forwarding and Register Tagging

Two techniques are introduced in this section for enhancing the performance of computers with multiple execution pipelines. *Internal forwarding* refers to a "short-circuit" technique for replacing unnecessary memory accesses by register-to-register transfers in a sequence of fetch-arithmetic-store operations. *Register tagging* refers to the use of tagged registers, buffers, and reservation stations for exploiting concurrent activities among multiple arithmetic units. We will explain how these techniques have been applied in the IBM System/360 Model 91, which has multiple execution units with common data buffers and data paths. The application of these techniques is not limited to floating-point arithmetic or the System/360 architecture. It may be used in almost any computer that has multiple functional pipelines and accumulators.

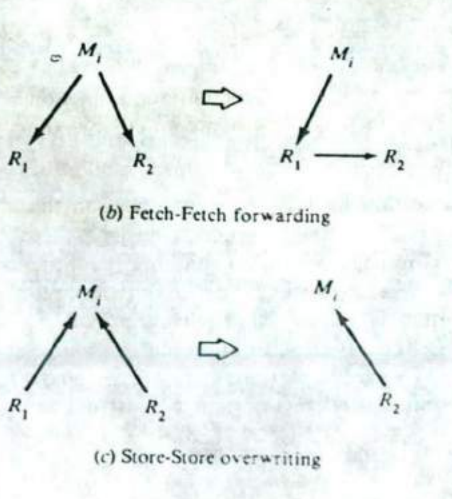
It is well understood that memory access is much slower than register-to-register operations. The computer performance can be greatly enhanced if one can eliminate unnecessary memory accesses and combine some transitive or multiple fetch-store operations with faster register operations. This concept of internal data forwarding can be explored in three directions, as illustrated in Figure 3.34. We use the symbols M_i and R_j to represent the ith word in the memory and the jth register in the CPU. We use arrows $\leftarrow$ to specify data-moving operations such as fetch, store, and register-to-register transfer. The contents of M_i and R_j are represented by (M_i) and (R_j), respectively.

Store-fetch forwarding The following sequence of the two operations store-then-fetch can be replaced by two parallel operations, one store and one register transfer, as shown in Figure 3.34*a*:

$$\left.\begin{array}{ll} M_i \leftarrow (R_1) & \text{(store)} \\ R_2 \leftarrow (M_i) & \text{(fetch)} \end{array}\right\}\text{Two memory accesses}$$

being replaced by

$$\left.\begin{array}{ll} M_i \leftarrow (R_1) & \text{(store)} \\ R_2 \leftarrow (R_1) & \text{(register transfer)} \end{array}\right\}\text{Only one memory access}$$



(*a*) Store-Fetch forwarding

(*b*) Fetch-Fetch forwarding

(*c*) Store-Store overwriting

Figure 3.34 Internal forwarding examples (thick arrows for slow memory accesses and thin arrows for fast register transfers).

Fetch-fetch forwarding The following two fetch operations can be replaced by one fetch and one register transfer, as shown in Figure 3.34*b*. Again one memory access has been eliminated:

$$\left.\begin{array}{ll} R_1 \leftarrow (M_i) & \text{(fetch)} \\ R_2 \leftarrow (M_i) & \text{(fetch)} \end{array}\right\} \text{Two memory accesses}$$

being replaced by

$$\left.\begin{array}{ll} R_1 \leftarrow (M_i) & \text{(fetch)} \\ R_2 \leftarrow (R_1) & \text{(register transfer)} \end{array}\right\} \text{One memory access}$$

Store-store overwriting The following two memory updates (stores) of the same word (Figure 3.34*c*) can be combined into one, since the second store overwrites the first:

$$\left.\begin{array}{ll} M_i \leftarrow (R_1) & \text{(store)} \\ M_i \leftarrow (R_2) & \text{(store)} \end{array}\right\} \text{Two memory accesses}$$

being replaced by

$$M_i \leftarrow (R_2) \quad \text{(store) One memory access}$$

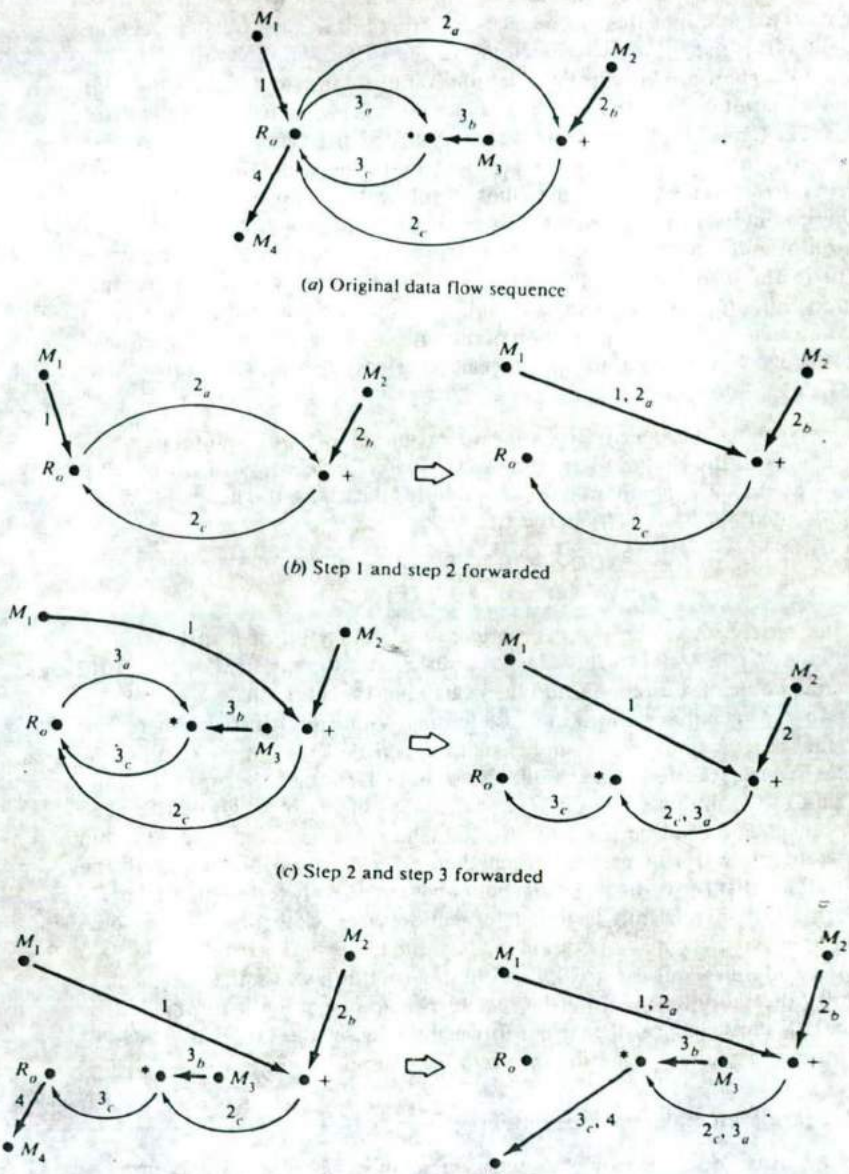
The following example shows how to apply internal forwarding to simplify a sequence of arithmetic and memory-access operations. Figure 3.35 depicts these simplification steps, in which adjacent steps are combined to minimize memory references. Nodes in the graph correspond to the memory cells, registers, an adder, or a multiplier.

Example 3.1 The inner loop of a certain program is completed to perform the following operations in a sequence:

1. $R_0 \leftarrow (M_1)$ (fetch)
2. $R_0 \leftarrow (R_0) + (M_2)$ (add)
3. $R_0 \leftarrow (R_0) * (M_3)$ (multiply)
4. $M_4 \leftarrow R_0$ (store)

After the internal forwarding, we end up handling a compound function (macroinstruction) $M_4 \leftarrow [(M_1) + (M_2)] * (M_3)$, as represented by the simplified data-flow graph in Figure 3.35*d*.

Both internal forwarding and resource tagging have been practiced in the IBM Model 91 floating-point execution unit. The data registers, transfer paths, floating-point adder and multiply-divide units, reservation stations, and the *common data bus* (CDB) in the Model 91 were shown in Figure 3.33. The three reservation stations for the adder are denoted as A_1, A_2, A_3. The two reservation stations in the multiply-divide unit are M_1 and M_2. Each station has the source and sink registers and their tag and control fields. The stations can hold operands for the next execution while the functional unit is busy executing current instruction



(*a*) Original data flow sequence

(*b*) Step 1 and step 2 forwarded

(*c*) Step 2 and step 3 forwarded

(*d*) Step 3 and step 4 forwarded

Figure 3.35 Internal data forwarding in Example 3.1 (memory accesses: thick arrows; register transfers: thin arrows).

Three *store data buffers* (SDB) and four *floating-point registers* (FLR) are all tagged. The busy bits in the FLRs marking their status (1 for busy and 0 for idle) can be used to determine the dependence of instructions in subsequent executions.

The CDB is used to transfer operands to the FLRs, the reservation stations, and the SDB. There are 11 units that can supply information to the CDB, including six floating-point buffers (FLB), three adder stations, and two multiply-divide stations. The tag fields of these units are binary-coded as FLBs 1 ~ 6, add stations 10 ~ 12, and multiply-divide stations 8 ~ 9. A tag is generated by the CDB priority controls to identify the unit whose result will next appear on the CDB.

This common data busing and register-tagging scheme permits simultaneous execution of independent instructions while preserving the essential precedences inherent in the instruction stream. The CDB can function with any number of execution units and any number of accumulators. It provides a hardware algorithm for the automatic efficient exploitation of multiple arithmetic units. The following example shows how internal forwarding can be achieved with the tagging scheme on the CDB.

Example 3.2 Consider the consecutive execution of two floating-point instructions in the Model 91 (Figure 3.33), where F refers to an FLR which is being used as an accumulator and B_i stands for the ith FLB. Their contents are represented by (F) and (B_i), respectively:

$$\begin{array}{lll} \text{ADD} & \text{F, B}_1 & \text{F} \leftarrow (\text{F}) + (\text{B}_1) \\ \text{MPY} & \text{F, B}_2 & \text{F} \leftarrow (\text{F}) * (\text{B}_2) \end{array}$$

In the processing of the *add* instruction, set the busy bit of F to 1, send the contents (F) and (B_1) to the adder station A_1, set the tag field of F to 1010 (the tag value of station A_1), and then carry out the addition.

In the meantime, the decode of the *mpy* (multiply) instruction reveals the fact that F is busy. This implies that the *mpy* depends on the result of the *add*. However, the execution should not be halted. Instead, the tag of F should be sent to the multiply station M_1 to set the tag of M_1 to be also 1010. Then the tag of F should be changed to 1000 (the tag value of station M_1) and the content (B_2) sent to M_1. When the *add* instruction is completed, the CDB finds that the addition result should be sent directly to M_1 (instead of F). The multiply-divide unit begins its execution when both operands become available. After the *mpy* operation is done, the CDB finds F via the tag 1000 of M_1, and thus sends the multiply result to F. In this process, the intermediate result (after addition) will not be sent to F before sending it to M_1. This is exactly a consequence of internal forwarding, using the tag as a vehicle to identify source and destination in successive computations.

3.3.4 Hazard Detection and Resolution

Pipeline hazards are caused by resource-usage conflicts among various instructions in the pipeline. Such hazards are triggered by interinstruction dependencies. In this section, we characterize various hazard conditions. Hazard-detection methods and approaches to resolve hazards are then introduced. Hazards discussed in this section are known as *data-dependent hazards*. Methods to cope with such hazards are needed in any type of lookahead processors for either synchronous-pipeline or asynchronous-multiprocessing systems. Another type of hazard is due to a job scheduling problem and will be described in Section 3.3.5.

When successive instructions overlap their fetch, decode and execution through a pipeline processor, interinstruction dependencies may arise to prevent the sequential data flow in the pipeline. For example, an instruction may depend on the results of a previous instruction. Until the completion of the previous instruction, the present instruction cannot be initiated into the pipeline. In other instances, two stages of a pipeline may need to update the same memory location. Hazards of this sort, if not properly detected and resolved, could result in an *interlock* situation in the pipeline or produce unreliable results by overwriting.

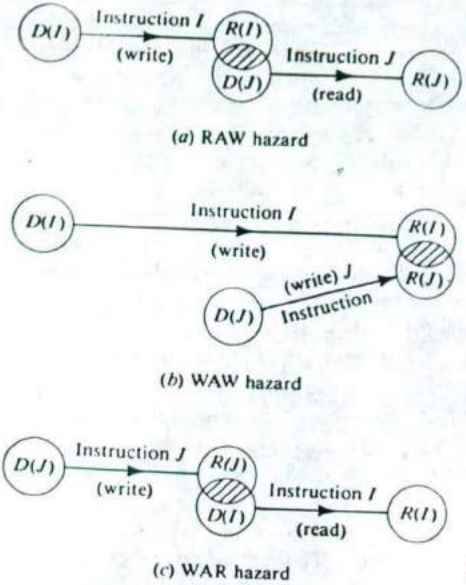
There are three classes of data-dependent hazards, according to various data update patterns: *write after read* (WAR) hazards, *read after write* (RAW) hazards, and *write after write* (WAW) hazards. Note that *read-after-read* does not pose a problem, because nothing is changed.

We use *resource objects* to refer to working registers, memory locations, and special flags. The contents of these resource objects are called *data objects*. Each instruction can be considered a mapping from a set of data objects to a set of data objects. The *domain* $D(I)$ of an instruction I is the set of resource objects whose data objects may affect the execution of instruction I. The *range* $R(I)$ of an instruction I is the set of resource objects whose data objects may be modified by the execution of instruction I. Obviously, the operands to be used in an instruction execution are retrieved (read) from its domain, and the results will be stored (written) in its range. In what follows, we consider the execution of the two instructions I and J in a program. Instruction J appears after instruction I in the program. There may be none or other instructions between instructions I and J. The latency between the two instructions is a very subtle matter. Instruction J may enter the execution pipe before or after the completion of the execution of instruction I. The improper timing and data dependencies may create some hazardous situations, as shown in Figure 3.36.

A RAW hazard between the two instructions I and J may occur when J attempts to read some data object that has been modified by I. A WAR hazard may occur when J attempts to modify some data object that is read by I. A WAW hazard may occur if both I and J attempt to modify the same data object. Formally, the necessary conditions for these hazards are stated as follows (Figure 3.32):

$$\begin{aligned} &R(I) \cap D(J) \neq \phi \qquad \text{for RAW} \\ &R(I) \cap R(J) \neq \phi \qquad \text{for WAW} \qquad\qquad (3.18) \\ &D(I) \cap R(J) \neq \phi \qquad \text{for WAR} \end{aligned}$$

Possible hazards for the four types of instructions (Table 3.1) are listed in Table 3.3. Recognizing the existence of possible hazards, computer designers wish to detect the hazard and then to resolve it effectively. Hazard detection can be done



(*a*) RAW hazard

(*b*) WAW hazard

(*c*) WAR hazard

Figure 3.36 Illustration of RAW, WAW, and WAR hazard conditions.

in the instruction-fetch stage of a pipeline processor by comparing the domain and range of the incoming instruction with those of the instructions being processed in the pipe. Should any of the conditions in Eq. 3.18 be detected, a warning signal can be generated to prevent the hazard from taking place. Another approach is to allow the incoming instruction through the pipe and distribute the detection to all the potential pipeline stages. This distributed approach offers better flexibility at the expense of increased hardware control. Note that the necessary conditions in Eq. 3.18 may not be sufficient conditions.

Table 3.3 Possible hazards for ṽarious instruction types

	Instruction *I* (first)			
Instruction *J* (second)	Arithmetic and load type	Store type	Branch type	Conditional branch type
Arithmetic and load type	RAW WAW WAR	RAW WAR	WAR	WAR
Store type	RAW WAR	WAW		
Branch type	RAW		WAW	WAW
Conditional branch type	RAW		WAW	WAW

Once a hazard is detected, the system should resolve the interlock situation. Consider the instruction sequence $\{\ldots I, I+1, \ldots, J, J+1, \ldots\}$ in which a hazard has been detected between the current instruction J and a previous instruction I. A straightforward approach is to stop the pipe and to suspend the execution of instructions J, $J+1$, $J+2, \ldots$, until the instruction I has passed the point of resource conflict. A more sophisticated approach is to suspend only instruction J and continue the flow of instructions $J+1$, $J+2, \ldots$, down the pipe. Of course, the potential hazards due to the suspension of J should be continuously checked as instructions $J+1$, $J+2, \ldots$ move ahead of J. Multilevel hazard detection may be encountered, requiring much more complex control mechanisms to resolve a stack of hazards.

In order to avoid RAW hazards, IBM engineers developed a *short-circuiting* approach which gives a copy of the data object to be written directly to the instruction waiting to read the data. This concept was generalized into a technique, known as *data forwarding*, which forwards multiple copies of the data to as many waiting instructions as may wish to read it. A data-forwarding chain can be established in some cases. The internal-forwarding and register-tagging techniques presented in the previous section should be helpful in resolving logic hazards in pipelines.

3.3.5 Job Sequencing and Collision Prevention

Once a task is initiated in a static pipeline, its flow pattern is fixed. An *initiation* refers to the start of a single function evaluation. When two or more initiations attempt to use the same stage at the same time, a *collision* results. Thus the job-sequencing problem is to properly schedule queued tasks awaiting initiation in order to avoid collisions and to achieve high throughput. The reservation table introduced in Section 3.1.3 identifies the space-time flow pattern of one complete data through the pipeline for one function evaluation. In a static pipeline, all initiations are characterized by the same reservation table. On the other hand, successive initations for a dynamic pipeline may be characterized by a set of reservation tables, one per each function being evaluated.

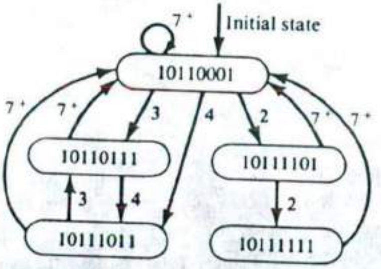
Figure 3.37 shows the reservation table for a unifunction pipeline. The multiple ×'s in a row pose the possibility of collisions. The number of time units between two initiations is called the *latency*, which may be any positive integer. For a static pipeline, the latency is usually one, two, or greater. However, zero latency is allowed in dynamic pipelines between different functions. The sequence of latencies between successive initiations is called *latency sequence*. A latency sequence that repeats itself is called a *latency cycle*. The procedure to choose a latency sequence is called a *control strategy*. A control strategy that always minimizes the latency between the current initiation and the very last initiation is called a *greedy strategy*. A greedy strategy is made independent of future initiations.

A collision occurs when two tasks are initiated with a latency (initiation interval) equal to the column distance between two ×'s on some row of the reservation table. The set of column distances $F = \{l_1, l_2, \ldots, l_r\}$ between all possible

s \ t	0	1	2	3	4	5	6	7	8
1	x								x
2		x	x					x	
3				x					
4					x	x			
5							x	x	

Forbidden list: $F = \{1,5,6,8\}$
Collision vector: $C = (10110001)$

(*a*) Reservation table and related terms



The notation 7^+ means any integer (latency) equal to 7 or greater than 7.

(*b*) State diagram with MAL $= (3 + 4)/2 = 3.5$

Figure 3.37 Reservation table and state diagram for a unifunction pipeline.

pairs of ×'s on each row of the reservation table is called the *forbidden set of latencies*. The forbidden set contains all possible latencies that cause collisions between two initiations. The *collision vector* is a binary vector, shown below:

$$C = (C_n \cdots C_2 C_1) \qquad \text{where } C_i = 1 \text{ if } i \in F \text{ and } C_i = 0 \text{ if otherwise} \quad (3.18)$$

For the example in Figure 3.37, the forbidden list $F = \{1, 5, 6, 8\}$, and the collision vector $C = (10110001)$, where $n = 8$ is the largest forbidden latency obtained from the reservation table. This means $C_n = 1$ is always true. The collision vector shows both permitted and forbidden latencies from the same reservation table. One can use an n-bit shift register to hold the collision vector for implementing a control strategy for successive task initiations in the pipeline. Upon initiation of the first task, the collision vector is parallel-loaded into the shift register as the initial state. The shift register is then shifted right one bit at a time, entering 0s from the left end. A collision-free initiation is allowed at time instant $t + k$ if, and only if, a bit "0" is being shifted out of the register after k shifts from time t. A *state diagram* is used to characterize the successive initiations of tasks in the pipeline in order to find the shortest latency sequence to optimize the control strategy. A *state* on the diagram is represented by the contents of the shift register after the proper number of shifts is made, which is equal to the latency between the current and next task initiations.

As shown in Figure 3.37*b*, the initial state corresponds to the collision vector (10110001). There are four outgoing branches from the initial state, labeled by latencies 2, 3, 4, and 7, corresponding to, respectively, zero-bit positions C_2, C_3, C_4, and C_7 in the vector (10110001). By shifting right the vector (10110001) two positions, we obtain the vector (00101100). This vector is then bitwise *or*ed with the collision vector (10110001) to produce a new collision vector (10111101) as the new state pointed to by the arc labeled 2. Similarly, one obtains the new state vectors (10110111) and (10111011) after shifting the latencies 3 and 4, respectively. The arc 7 branches back to the initial state. This shifting process should continue until no more new states can be generated. The shift register will be set to the initial state, if the latency (shift) is greater than or equal to n.

The successive collision vectors are used to prevent future task collisions with previously initiated tasks, while the collision vector C is used to prevent possible collisions with the current task. If a collision vector has a "1" in the ith bit (from the right) at time t, then the task sequence should avoid the initiation of a task at time $t + i$. The bitwise *or*ing operations will avoid collisions in any workable latency sequence that can be traced on the state diagram. Closed loops or cycles in the state diagram indicate the steady-state sustainable latency sequences of task initiations without collisions. The *average latency* of a cycle is the sum of its latencies (period) divided by the number of states in the cycle. Any cycle can be entered from the initial state.

The cycle consisting of states (10110111) and (10111011) in Figure 3.37*b* has two latencies, three and four. This cycle has a period equal to $7 = 3 + 4$. The average latency of this cycle is $\frac{7}{2} = 3.5$. Another cycle, which consists of the states (10110001), (10111101), and (10111111), has the three latencies 2, 2, and 7, with a period of 11. Its average latency cycle equals $\frac{11}{3} = 3.66$. The throughput of a pipeline is inversely proportional to the reciprocal of the average latency. A latency sequence is called *permissible* if no collisions exist in the successive initiations governed by the given latency sequence. The maximum throughput is achieved by an optimal scheduling strategy that achieves the *minimum average latency* (MAL) without collisions. Thus, the job-sequencing problem is equivalent to finding a permissible latency cycle with the MAL in the state diagram. The maximum number of ×'s in any single row of the reservation table is a lower bound of the MAL. In other words, the MAL is always greater than or equal to the maximum number of check marks in any row of the reservation table.

Table 3.4 Simple cycles in Figure 3.37*b*

Simple cycle	Average latency
(7)	7
(3, 7)	5
(3, 4)†	3.5
(4, 3, 7)	4.6
(4, 7)	5.5
(2, 7)	4.5
(2, 2, 7)†	3.6
(3, 4, 7)	4.6

† Greedy cycles.

Simple cycles are those latency cycles in which each state appears only once per each iteration of the cycle. Listed in Table 3.4 are simple cycles and their average latencies for the state diagram shown in Figure 3.37*b*. A simple cycle is a *greedy cycle* if each latency contained in the cycle is the minimal latency (outgoing arc) from a state in the cycle. For Figure 3.37*b*, the cycles (3, 4) and (2, 2, 7) are both greedy, with average latencies of 3.5 and 3.6, respectively. A good task-initiation sequence should include the greedy cycle.

The procedure to determine the greedy cycles on the state diagram is rather straightforward. From each node of the state diagram, one simply chooses the arc with the smallest latency label until a closed simple cycle can be formed. The average latency of any greedy cycle is no greater than the number of latencies in the forbidden set, which equals the number of 1s in the initial collision vector. The average latency of any greedy cycle is always lower-bounded by the MAL. In the above example, the greedy cycle (3, 4) has an average latency equal to the MAL = 3.5, which is smaller than 4, the number of 1s in the initial collision vector.

The job-sequencing method for static unifunction pipelines can be generalized for designing multifunction pipelines. A pipeline processor which can perform p distinct functions can be describeđ by p reservation tables overlaid together. In order to perform multiple functions, the pipeline must be reconfigurable. One example of a static multifunction pipeline is the arithmetic pipelines in TI-ASC, which has eight stages with about 20 possible functional configurations. Each task to be initiated can be associated with a function tag identifying the reservation table to be used. Collisions may occur between two or more tasks with the same function tag or from distinct function tags.

The stage-usage pattern for each function can be displayed with a different tag in the overlaid reservation table. For a p-function pipeline, an overlaid reservation table is formed by overlaying p unifunctional reservation tables. An overlaid reservation table for a two-function pipeline is shown in Figure 3.38*a*, where A and B stand for two distinct functions. Each task-requesting initiation must be associated with a function tag. A *forbidden set of latencies* for a multifunction pipeline is the collection of collision-causing latencies. A task with function tag

s \ t	0	1	2	3	4
1	A	B		A	B
2		A		B	
3	B		AB		A

(*a*) Reservation table of a two-function pipeline

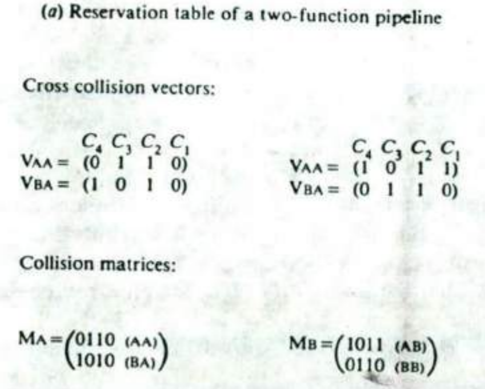
Cross collision vectors:

$$\begin{array}{lcccc} & C_4 & C_3 & C_2 & C_1 \\ V_{AA} = & (0 & 1 & 1 & 0) \\ V_{BA} = & (1 & 0 & 1 & 0) \end{array} \qquad \begin{array}{lcccc} & C_4 & C_3 & C_2 & C_1 \\ V_{AA} = & (1 & 0 & 1 & 1) \\ V_{BA} = & (0 & 1 & 1 & 0) \end{array}$$

Collision matrices:

$$M_A = \begin{pmatrix} 0110 & \text{(AA)} \\ 1010 & \text{(BA)} \end{pmatrix} \qquad M_B = \begin{pmatrix} 1011 & \text{(AB)} \\ 0110 & \text{(BB)} \end{pmatrix}$$

(*b*) Cross collision vectors and collision matrices



(*c*) State diagram

Figure 3.38 Reservation table, cross collision vectors, collision matrices, and state diagram for a multifunction pipeline.

A may collide with a previously initiated task with function tag B if the latency between these two initiations is a member of the forbidden list.

A *cross-collision vector* V_{AB} marks the forbidden latencies between the function pair A and B. The binary vector V_{AB} may be calculated by overlaying the reservation tables for A and B. A component $C_k = 1$ if some row of the overlaid reservation table contains an A in column t (for some t) and a B in column $t + k$; the component C_k equals 0, if otherwise. Thus, Figure 3.38 has four cross-collision vectors: $V_{AA} = (0\ 1\ 1\ 0)$, $V_{AB} = (1\ 0\ 1\ 1)$, $V_{BA} = (1\ 0\ 1\ 0)$, and $V_{BB} = (0\ 1\ 1\ 0)$. In general, there are p^2 cross-collision vectors for a p-function pipeline. The p^2 cross-collision vectors can be rewritten into p *collision matrices*, as shown in Figure 3.34*b*. The collision matrix M_R indicates forbidden latencies for all functions initiated after the initiation of a task with the function tag R. The ith row in matrix M_R is the cross-collision vector V_{IR}, where $i = 1, 2, \ldots, p$.

A p-function pipeline can be controlled by a bank of p shift registers. Shift register Q controls the initiation of function Q. The control bits for function initiations are the righmost bit of each shift register. Initiation of a task with function tag Q is allowed at the next time instant if the rightmost bit of the corresponding shift register Q is 0. The shift registers shift right one position per each cycle, with 0s entering from the left. Immediately after the initiation of a task with function tag Q, the collision matrix M_Q is *ored* with the matrix formed by the bank of shift registers. The state of the shift register Q is bitwise *ored* with the cross-collision vector V_{IQ} for all $1 \leq Q \leq p$.

A state diagram is constructed in Figure 3.38*c* for the two-function pipeline. Arcs are labeled with the latency and the function tag of the initiation. The initial state can be one of the p collision matrices. Cycles in the state diagram correspond to collision-free patterns of task initiations. Any cycle can be entered from at least one of the initial states. For example, the cycle (A3, B1) in Figure 3.38*c* can be reached by an arc labelled A3 from initial state I_2 or by an arc B1 from initial state I_1. The method of finding the greedy cycles and the MAL on the state diagram of a multifunction pipeline can be extended from that for a unifunction pipeline.

3.3.6 Dynamic Pipelines and Reconfigurability

A dynamic pipeline may initiate tasks from different reservation tables simultaneously to allow multiple numbers of initiations of different functions in the same pipeline. Two methods for improving the throughput of dynamic pipeline processors have been proposed by Davidson and Patel (1978). The reservation of a pipeline can be modified with the insertion of *noncompute delays* or with the use of internal buffers at each stage. The utilization of the stages and, hence, the throughput of the pipe can be greatly enhanced with a modified reservation table yielding a more desirable latency pattern.

It is assumed that any computation step can be delayed by inserting a noncompute stage. We consider first a unifunction pipeline. A *constant latency cycle* is a cycle with only one latency. A latency between two tasks is said to be *allowable* if these two tasks do not collide in the pipeline. Consequently, a cycle is allowable in a pipeline if all the latencies in the cycle are allowable. Our main concern so far has been to find an allowable cycle which results in the MAL. However, an allowable cycle with the MAL does not necessarily imply 100 percent utilization of the pipeline where utilization is measured as the percentage of time the busiest stage remains busy. When a latency cycle results in a 100 percent utilization of at least one of the pipeline stages, the periodic latency sequence is called a *perfect cycle*. Of course, pipelines with perfect cycles can be better utilized than those with nonperfect initiation cycles. It is trivial to note that constant cycles are all perfect.

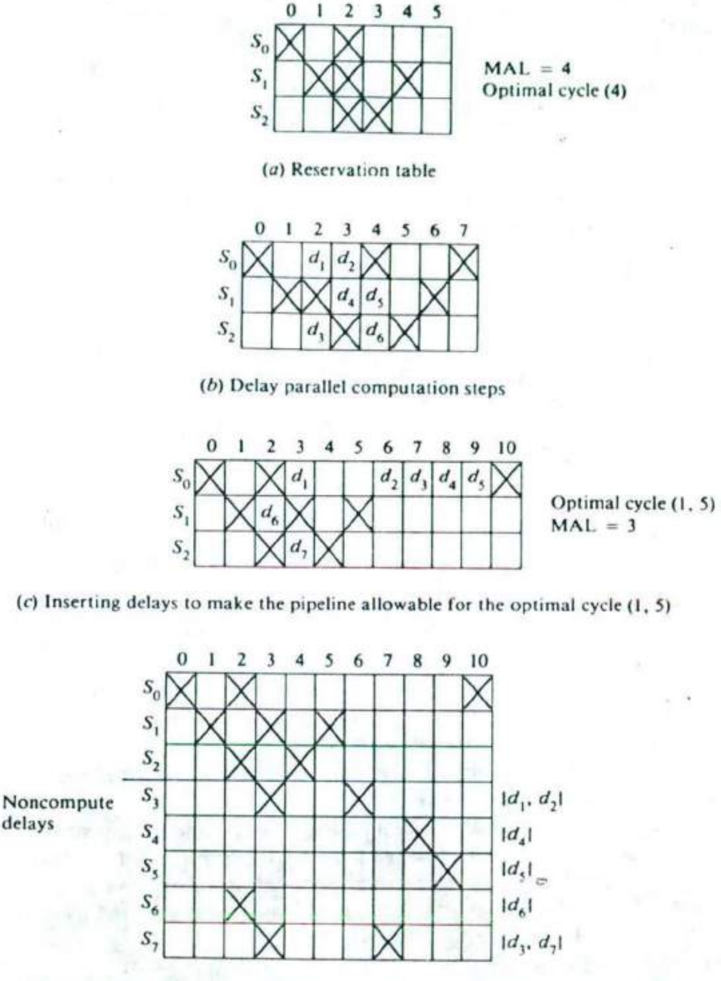
Consider a latency cycle C. The set G_C of all possible time intervals between initiations derived from cycle C is called an *initiation interval set*. For example, $G_C = \{4, 8, 12, \ldots\}$ for $C = (4)$, and $G_C = \{2, 3, 5, 7, 9, 10, 12, 14, 15, 17, 19, 21, 22, 24, 26, \ldots\}$ for $C = (2, 3, 2, 5)$. Note that the interval is not restricted to two adjacent initiations. Let $G_C(\text{mod } p)$ be the set formed by taking mod p equivalents of all elements of set G_C. For the cycle (2, 3, 2, 5) with period $p = 12$, the set $G_C(\text{mod } 12) = \{0, 2, 3, 5, 7, 9, 10\}$. The complement set $\bar{G}_C$ equals $Z - G_C$ where Z is the set of positive integers. Clearly, we have $\bar{G}_C(\text{mod } p) = Z(\text{mod } p) - G_C(\text{mod } p)$, where Z_p is the set of positive integers of modulo p. A latency cycle C with a period p and an initiation interval set G_C is allowable in a pipeline with a forbidden latency set F if, and only if,

$$F(\text{mod } p) \cap G_C(\text{mod } p) = \phi \tag{3.19}$$

This means that there will be no collision if none of the initiation intervals equals a forbidden latency. Thus, a constant cycle (l) with a period $p = l$ is allowed for a pipeline processor if, and only if, l does not divide any forbidden latency in the set F. Another way of looking at the problem is to choose a reservation table whose forbidden latency set F is a subset of the set $\bar{G}_C(\text{mod } p)$. Then the latency cycle C will be an allowable sequence for the pipeline. For example, the latency cycle $C = (2, 3, 2, 5)$, $G_C(\text{mod } 12) = \{0, 2, 3, 5, 7, 9, 10\}$ and $\bar{G}_C(\text{mod } 12) = \{1, 4, 6, 8, 11\}$, so C can be applied to a pipeline with a forbidden latency set F equal to any subset of $\{1, 4, 6, 8, 11\}$. This condition is very effective to check the applicability (allowability) of an initiation sequence (or a cycle) to a given pipeline, or one can modify the reservation table of a pipeline to yield a forbidden list which is confined within the set $\bar{G}_C(\text{mod } p)$, if the cycle C is fixed.

Adding noncompute stages to a pipeline can make it allowable for a given cycle. The effect of delaying some computation steps can be seen from the reservation table by writing a d before the step being delayed. Each d indicates one unit of delay, called an *elemental delay*. It is assumed that all steps in a column must complete before any steps in the next column are executed. In Figure 3.39*a*, the effect of delaying the step in row 0 and column 2 by two time units and the step in row 2 and column 2 by one time unit is shown in Figure 3.39*b*. The elemental delays d_1, d_2, and d_3 require the use of the additional delays d_4, d_5, and d_6 to make all the outputs simultaneously available in column 2 of the original reservation table.

For a given constant latency cycle (l), a pipeline can be made allowable by delaying some of the steps if, and only if, there are no more than l marks in each



(*a*) Reservation table

(*b*) Delay parallel computation steps

(*c*) Inserting delays to make the pipeline allowable for the optimal cycle (1, 5)

(*d*) Assignment of elemental delays to noncompute segments

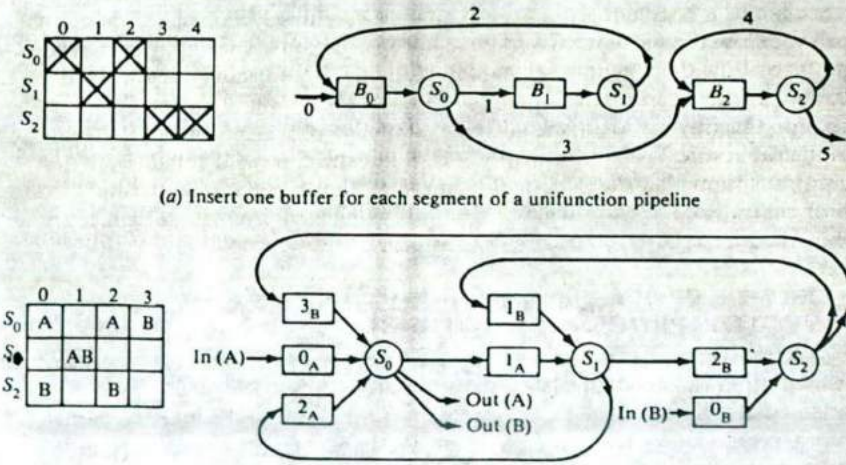
Figure 3.39 Pipeline with inserted noncomputer delays.

row of the table. Thus by adding elemental delays, a unifunction pipeline can always be fully utilized through the use of a cycle that has a constant latency equal to the maximum number of marks in any row of the reservation table. The maximum achievable throughput of that pipeline is thereby attained. On the other hand, for an arbitrary cycle, a pipeline can be made allowable by delaying some steps. The reservation table of Figure 3.39*a* can be made allowable with respect to the optimal cycle (1, 5) by adding some elemental delays. The resulting table is shown in Figure 3.39*c*. Once a modified table is obtained, it is necessary to assign the elemental delays to noncompute stages. Noncompute stages may be shared by various elemental delays. Figure 3.39*d* shows the modified reservation table after the introduction of the noncompute stages S_3, S_4, S_5, S_6, and S_7.

The task arrivals in a pipeline processor may be periodic for a program with inner loops. If we assume that each task can only occupy one stage at a time, no parallel computations can be done within a single task. Such an assumption stems from the practical difficulties encountered in implementing a priority scheme involving parallel computations of a task. Once some buffers are provided internally, the task-scheduling problem can be greatly simplified. Whenever two or more tasks are trying to use the same stage, only one of the tasks is allowed to use the stage, while the rest wait in the buffers according to some priority schemes.

There are two different implementations of internal buffers in a pipeline: The first uses one buffer for each stage (Figure 3.40*a*), and the second uses one buffer per computation step (Figure 3.40*b*). For one buffer per stage, two priority schemes, *FIFO-global* and *LIFO-global*, can be used. In the FIFO-global scheme, a task has priority over all tasks initiated later. In the LIFO-global scheme, a task has priority over all tasks initiated earlier. For one buffer per computation step, multiple buffers may be used in each segment with the following priorities: MPF: most processed first; LPF: least processed first; LWRF: least work remaining first; and MWRF: most work remaining first.

Reconfigurable pipelines with different function types are more desirable. Such an approach requires extensive resource-sharing among different functions. To achieve this, a more complicated structure of pipeline segments and their interconnection controls is needed. Bypass techniques can be used to avoid unwanted



	0	1	2	3	4
S_0	☒		☒		
S_1		☒			
S_2				☒	☒

(*a*) Insert one buffer for each segment of a unifunction pipeline

	0	1	2	3
S_0	A		A	B
S_1		AB		
S_2	B		B	

(*b*) Insert one buffer for each computation step of a two-function pipeline

Figure 3.40 Inserting buffers to improve the pipeline utilization rate.

stages. This may cause a collision when one instruction, as a result of bypassing, attempts to use the operands fetched for preceding instructions. To alleviate this problem, one solution has each instruction activate a number of consecutive stages down the pipeline which satisfy its need.

A dynamic pipeline would allow several configurations to be simultaneously present. For example, a dynamic-pipeline arithmetic unit could perform addition and multiplication at the same time. Tremendous control overhead and increased interconnection complexity would be expected. None of the existing pipeline processors has achieved this dynamic capability. Most commercial pipelines are static. In TI-ASC, the desired control allows different instructions to assume different data paths through the arithmetic pipeline at different times. All path-control information is stored in a *read-only memory* (ROM), which can be accessed at the initiation of an instruction.

The configuration for floating-point addition in TI-ASC (Figure 3.28*b*) requires four ROM words for its path-interconnection information. This forces the instruction execution logic to access the ROM for control signals. The ROM words for a *floating-point add* may be located at 100, 101, 102, and 103, while the words for a *floating-point subtract* could be located at 200, 101, 102, and 103. The common ROM words (101, 102, 103) used by both operations represent similar suboperations contained in these two instructions. The starting ROM address is supplied by the instruction-execution logic directly after the decode of the instruction.

The pipeline configuration for a floating-point vector *dot product* in TI-ASC was depicted in Figure 3.28*c*. If the dot product operated upon 1000 operands, the pipeline would be in this configuration for 1000 clock periods. Scalar instructions in ASC use different control sequences. When several scalar instructions in a sequence are of a common type, the instructions streaming through the arithmetic pipeline can be treated as vectors. This requires a careful selection of ROM output signals to allow the maximum overlapping of instructions. The ability to overlap instructions of the same type is achieved by studying the utilization of each pipeline segment. Overlaying identical patterns gives the minimum number of clock periods per result. The two static arithmetic pipeline processors in STAR-100 are reconfigurable with variable structures. Variable structure and resource sharing are of central importance to designing multifunction pipelines. Systematic procedures are yet to be developed for designing dynamically reconfigurable pipelines.

3.4 VECTOR PROCESSING REQUIREMENTS

In this section, we explain the basic concepts of vector processing and the necessary implementation requirements. We distinguish vector processing from scalar processing, present the characteristics of vector instructions, and define the performance measures of vector processors. We present a parallel vector scheduling model for multipipeline supercomputers. Three vector processing methods will be introduced for pipeline computers. After examining the architectures of various pipeline computers, we will study in Chapter 4 various vectorization methods and compiler-optimization problems.

3.4.1 Characteristics of Vector Processing

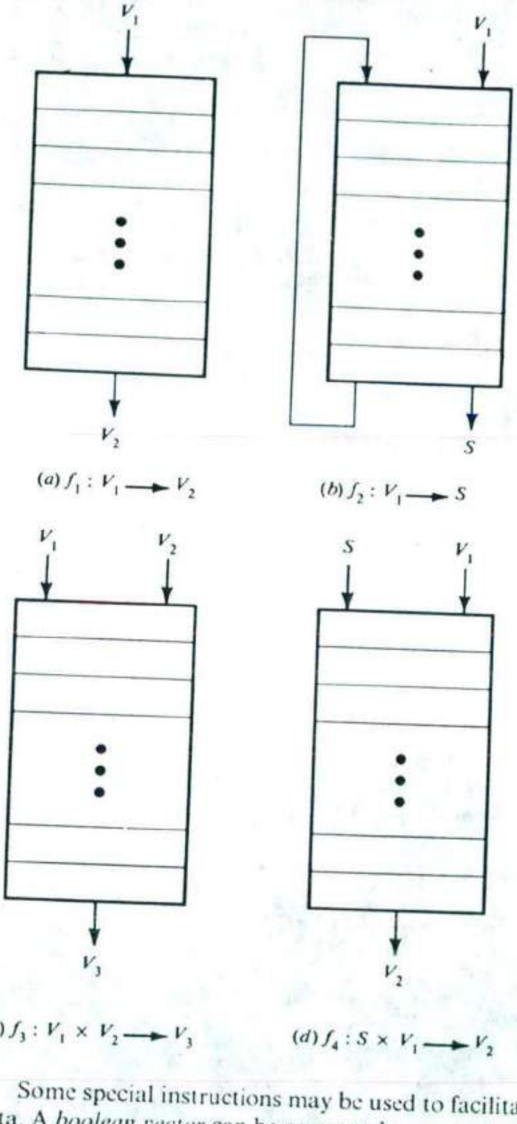
A vector operand contains an ordered set of n elements, where n is called the *length* of the vector. Each element in a vector is a scalar quantity, which may be a floating-point number, an integer, a logical value, or a character (byte). Vector instructions can be classified into four primitive types:

$$\begin{aligned} f_1 &: V \rightarrow V \\ f_2 &: V \rightarrow S \\ f_3 &: V \times V \rightarrow V \\ f_4 &: V \times S \rightarrow V \end{aligned} \tag{3.20}$$

where V and S denote a vector operand and a scalar operand, respectively. The mappings f_1 and f_2 are unary operations and f_3 and f_4 are binary operations. As shown in Table 3.5, the VSQR (*vector square root*) is an f_1 operation, VSUM (*vector summation*) is an f_2 operation, SVP (*scalar-vector product*) is an f_4 operation, and VADD (*vector add*) is an f_3 operation. The *dot product* of two vectors $V_1 \cdot V_2 = \sum_{i=1}^{n} V_{1i} \cdot V_{2i}$ is generated by applying f_3 (vector multiply) and then f_2 (vector sum) operations in sequence. Listed in Table 3.5 are some representative vector operations that can be found in a modern vector processor. Pipelined implementation of the four basic vector operations is illustrated in Figure 3.41. Note that a feedback connection is needed in the f_2 operation.

Table 3.5 Some representative vector instructions

Type	Mnemonic	Description ($I = 1$ through N)	
f_1	VSQR	Vector square root:	$B(I) \leftarrow \sqrt{A(I)}$
	VSIN	Vector sine:	$B(I) \leftarrow \sin(A(I))$
	VCOM	Vector complement:	$A(I) \leftarrow \overline{A(I)}$
f_2	VSUM	Vector summation:	$S = \sum_{I=1}^{N} A(I)$
	VMAX	Vector maximum:	$S = \max_{I=1,N} A(I)$
f_3	VADD	Vector add:	$C(I) = A(I) + B(I)$
	VMPY	Vector multiply:	$C(I) = A(I) \bullet B(I)$
	VAND	Vector and:	$C(I) = A(I)$ and $B(I)$
	VLAR	Vector larger:	$C(I) = \max(A(I), B(I))$
	VTGE	Vector test >:	$C(I) = 0$ if $A(I) < B(I)$ $C(I) = 1$ if $A(I) > B(I)$
f_4	SADD	Vector-scalar add:	$B(I) = S + A(I)$
	SDIV	Vector-scalar divide:	$B(I) = A(I)/S$



(a) $f_1 : V_1 \longrightarrow V_2$

(b) $f_2 : V_1 \longrightarrow S$

(c) $f_3 : V_1 \times V_2 \longrightarrow V_3$

(d) $f_4 : S \times V_1 \longrightarrow V_2$

Figure 3.41 Four vector instruction types for pipelined processor.

Some special instructions may be used to facilitate the manipulation of vector data. A *boolean vector* can be generated as a result of comparing two vectors, and can be used as a *masking vector* for enabling or disabling component operations in a vector instruction. A *compress* instruction will shorten a vector under the control of a masking vector. A *merge* instruction combines two vectors under the control of a masking vector. Compress and merge are special f_1 and f_3 operations because the resulting operand may have a length different from that of the input operands. Several examples are shown below to characterize these special vector operations.

Example 3.3 Let $X = (2, 5, 8, 7)$ and $Y = (9, 3, 6, 4)$. After the *compare* instruction $B = X > Y$ is executed, the boolean vector $B = (0, 1, 1, 1)$ is generated.

Let $X = (1, 2, 3, 4, 5, 6, 7, 8)$ and $B = (1, 0, 1, 0, 1, 0, 1, 0)$. After the execution of the *compress* instruction $Y = X(B)$, the compressed vector $Y = (1, 3, 5, 7)$ is generated.

Let $X = (1, 2, 4, 8)$, $Y = (3, 5, 6, 7)$, and $B = (1, 1, 0, 1, 0, 0, 0, 1)$. After the *merge* instruction $Z = X, Y, (B)$, the result is $Z = (1, 2, 3, 4, 5, 6, 7, 8)$. The first 1 in B indicates that $Z(1)$ is selected from the first element of X. Similarly, the first 0 in B indicates that $Z(3)$ is selected from the first element of Y.

In general, machine operations suitable for pipelining should have the following properties:

a. Identical processes (or functions) are repeatedly invoked many times, each of which can be subdivided into subprocesses (or subfunctions).
b. Successive operands are fed through the pipeline segments and require as few buffers and local controls as possible.
c. Operations executed by distinct pipelines should be able to share expensive resources, such as memories and buses, in the system.

These characteristics explain why most vector processors have pipeline structures. Vector instructions need to perform the same operation on different data sets repeatedly. This is not true for scalar processing over a single pair of operands. One obvious advantage of vector processing over scalar processing is the elimination of the overhead caused by the loop-control mechanism. Because of the startup delay in a pipeline, a vector processor should perform better with longer vectors. Vector instructions are usually specified by the following fields:

1. The *operation code* must be specified in order to select the functional unit or to reconfigure a multifunctional unit to perform the specified operation. Usually, microcode control is used to set up the required resources.
2. For a memory-reference instruction, the *base addresses* are needed for both source operands and result vectors. If the operands and results are located in the vector register file, the designated *vector registers* must be specified.
3. The *address increment* between the elements must be specified. Some computers, like the Star-100, restrict the elements to be consecutively stored in the main memory, i.e., the increment is always 1. Some other computers, like TI-ASC, can have a variable increment, which offers higher flexibility in application.
4. The *address offset* relative to the base address should be specified. Using the base address and the offset, the effective memory address can be calculated. The offset, either positive or negative, offers the use of skewed vectors to achieve parallel accesses.

5. The *vector length* is needed to determine the termination of a vector instruction. A masking vector may be used to mask off some of the elements without changing the contents of the original vectors.

We can classify pipeline vector computers into two architectural configurations according to where the operands are retrieved in a vector processor. One class is the *memory-to-memory* architecture, in which source operands, intermediate and final results are retrieved directly from the main memory. For memory-to-memory vector instructions, the information of the base address, the offset, the increment, and the vector length must be specified in order to enable streams of data transfers between the main memory and the pipelines. Vector instructions in the TI-ASC, the CDC STAR-100, and the Cyber-205 have a memory-to-memory format. The other class has a *register-to-register* architecture, in which operands and results are retrieved indirectly from the main memory through the use of a large number of vector or scalar registers. Vector instructions in the Cray-1 and the Fujitsu VP-200 use a register-to-register format. The example below demonstrates the difference between these two vector-instruction formats.

To examine the efficiency of vector processing over scalar processing, we compare the following two programs, one written for vector processing and the other written for scalar processing.

Example 3.4 In a conventional scalar processor, the Fortran DO loop

```
          DO 100 I=1, N
          A(I)=B(I)+C(I)
100       B(I)=2*A(I+1)
```

is implemented by the following sequence of scalar operations:

```
          INITIALIZE I=1
10        READ B(I)
          READ C(I)
          ADD B(I)+C(I)
          STORE A(I)←B(I)+C(I)
          READ A(I+1)
          MULTIPLY 2*A(I+1)
          STORE B(I)←2*A(I+1)
          INCREMENT I←I+1
          IF I≤N GO TO 10
          STOP
```

In a vector processor, the above DO loop operations can be vectorized into three vector instructions in a sequence:

```
A(1:N)=B(1:N)+C(1:N)
TEMP(1:N)=A(2:N+1)
B(1.N)=2*TEMP(1:N)
```

where A(1:N) refers to the N-element vector $A(1), A(2), \ldots, A(N)$. The introduction of the TEMP(1:N) vector is necessary to enable the vectorization.

The execution of the scalar loop repeats the loop-control overhead in each iteration. In vector processing using pipelines, the overhead is reduced by using hardware or firmware controls. A vector-length register can be used to control the vector operations. The overhead of pipeline processing is mainly the *setup time*, which is needed to route the operands among functional units. For example, in the ASC and Star-100 systems, each vector instruction needs to get some vector-parameter registers or control vectors before the instruction can be initiated. Thus, many additional memory fetches are needed to load the control registers. Another overhead is the *flushing time* between the decoding of a vector instruction and the exit of the first result from the pipeline. The flushing time exists for both vector and scalar processing; however, a vector pipe has to check the termination condition and the control vectors. Therefore, a vector pipe may have a longer flush time than its sequential counterpart.

The vector length affects the processing efficiency because of the additional overhead caused by subdividing a long vector. In order to enhance the vector-processing capability, an optimized object code must be produced to maximize the utilization of pipeline resources. The following approaches have been suggested:

Enrich the vector instruction set With a richer instruction set, the processing capability will be enhanced. One can avoid excessive memory accesses and poor resource utilization with an improved instruction set. The compress instruction in Example 3.3 was a good example of saving memory.

Combine scalar instructions Using a pipeline for processing scalar quantities, one should group scalar instructions of the same type together as a batch instead of interleaving them. The overhead due to the pipeline reconfiguration can be greatly reduced by grouping scalar instructions.

Choose suitable algorithms Often a fast algorithm that is implemented in a serial processor may not be at all effective in a pipelined processor. For example, the merge-sort algorithm is more suitable for pipelining because the machine can merge two ordered vectors in one pass.

Use a vectorizing compiler An intelligent compiler must be developed to detect the concurrency among vector instructions which can be realized with pipelining or with the chaining of pipelines. A vectorizing compiler would regenerate parallelism lost in the use of sequential languages. It is desirable to use high-level programming languages with rich parallel constructs on vector processors. The following four stages have been recognized in the development of parallelism in advanced programming. The parameter in parentheses indicates the degree of parallelism explorable at each stage:

- Parallel *a*lgorithm (A)
- High-level *l*anguage (L)
- Efficient *o*bject code (O)
- Target *m*achine code (M)

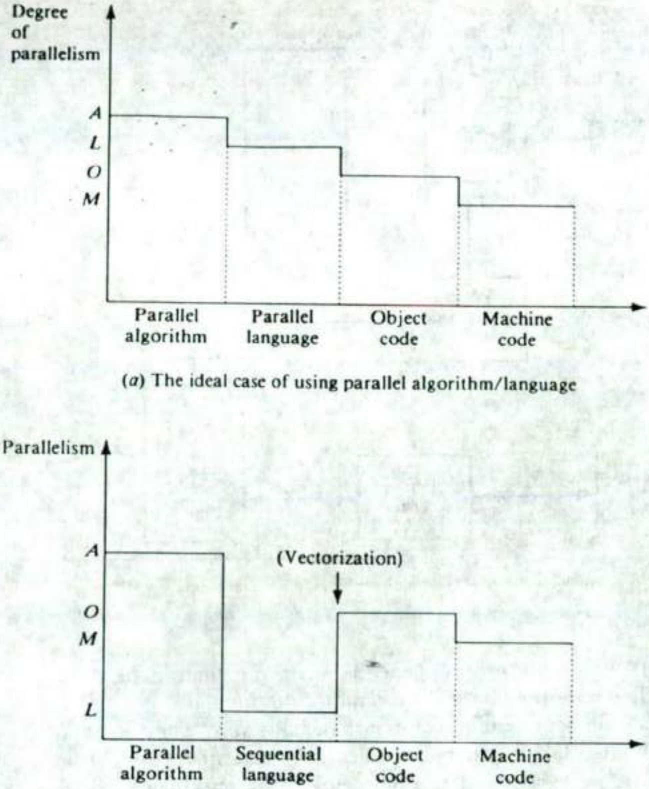
The degree of parallelism refers to the number of independent operations that can be performed simultaneously. We wish to find a suitable algorithm with a high parallelism (A) to solve large-scale matrix problems. We also need to develop *parallel languages* to express parallelism (L). Unfortunately, no parallel language standards have yet been universally accepted. At present, most users still write their source code in sequential languages.

In sequential languages like Fortran, Pascal, and Algol, we still have L = 1. The natural parallelism in a machine is determined by the hardware. For example, the Cray-1 has O = M = 64 or 32. In the ideal situation with well-developed parallel user languages, we should expect A ≥ L ≥ O ≥ M, as illustrated in Figure 3.42*a*. At present, any parallelism in an algorithm is lost when it is expressed in a sequential high-level language. In order to promote parallel processing in machine hardware, an intelligent compiler is needed to regenerate the parallelism through vectorization, as illustrated by Figure 3.42*b*. The process to replace a block of sequential code by vector instructions is called *vectorization*. The system software which does this regeneration of parallelism is called a *vectorizing compiler*. In Chapter 4, we will study attempts at developing parallel constructs in high-level languages and then discuss desired features in vectorizing compilers.

3.4.2 Multiple Vector Task Dispatching

A parallel task-scheduling model is presented for multi-pipeline vector processors. This model can be applied to explore maximum concurrency in vector supercomputers. The functional block diagram of a modern multiple-pipeline vector computer is shown in Figure 3.43. This structure is generalized from the existing modern vector processors. The main memory is often interleaved to minimize the access time of vector operands. Instructions and data may appear in either vector or scalar formats. The *instruction processing unit* (IPU) fetches and decodes scalar and vector instructions. All scalar instructions are dispatched to the *scalar processor* for execution. The scalar processor itself contains multiple scalar pipelines.

A *task system* contains a set of vector instructions (tasks) with a precedence relation determined only by data dependencies. A long vector task can be partitioned into many subvectors, to be processed by several pipelines concurrently. An increase in system overhead may be incurred with vector segmentations. It has been proved by Hwang and Su (1983) that the multi-pipeline scheduling problem is *NP*-complete, even for restricted task classes. Heuristic-scheduling algorithms are thus desirable for parallel vector processing.



(*a*) The ideal case of using parallel algorithm/language

(*b*) The case of using vectorizing compiler and sequential language

Figure 3.42 Parallelism regeneration in using a vectorizing compiler for programs written in sequential language.

After a vector instruction is recognized by the IPU, the *vector instruction controller* takes over in supervising its execution. The functions of this controller include decoding vector instructions, calculating effective vector-operand addresses, setting up the vector access controller and the *vector processor*, and monitoring the execution of vector instructions. We consider here a very capable vector instruction controller which can partition a vector task and schedule different instructions to different functional pipelines. In most commercial vector processors, identical pipelines must execute the same vector instruction at the same time. The vector machine model being presented has a structure generalized from the commercial machines. The *vector access controller* is responsible for fetching vector operands by a series of main memory accesses. The *vector registers* are used to close up the speed gap between the main memory and the vector processor. In

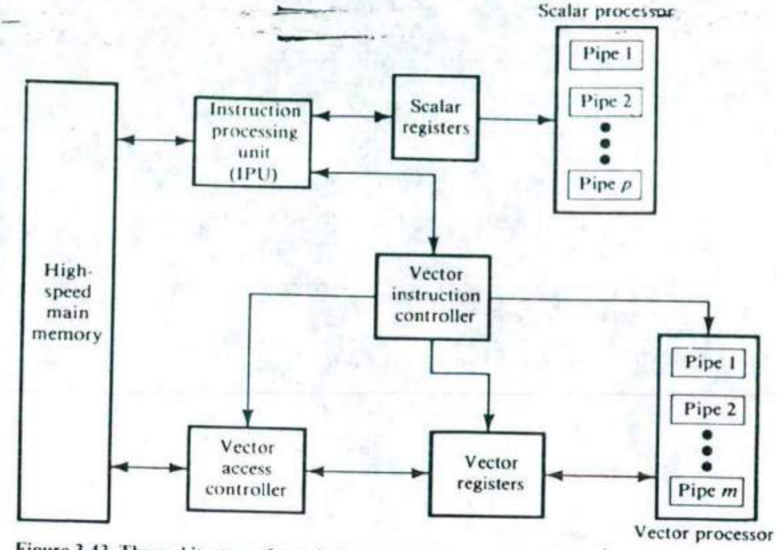


Figure 3.43 The architecture of a typical vector processor with multiple functional pipes.

the following discussions, we assume *m* homogeneous vector pipelines in the vector processor, each of which is unifunctional or static-multifunctional.

We shall concentrate on scheduling vector tasks exclusively. The vector instruction controller in Figure 3.43 is capable of scheduling several vector instructions simultaneously. The time required to complete the execution of a single vector task is measured by $t_o + \tau$, where t_o is the *pipeline overhead time* due to startup and flushing delays, $\tau = t_l \cdot L$ is the *production delay*, t_l is the *average latency* between two successive operand pairs, and L is the *vector length* (the number of component operands in a vector). The startup time is measured from the initiation of the vector instruction to the entrance of the first operand pair into the pipeline. Parameters t_o and t_l vary with different vector instructions. The overhead time t_o may vary from tens to several hundreds of pipeline cycles. The average latency t_l is usually one or two pipeline cycles. It is reasonable to assume that $t_o \gg t_l$.

Given a task system, we wish to schedule the vector tasks among *m* identical pipelines such that the total execution time is minimized. To simplify the modeling, we assume equal overhead time t_o for all vector tasks. A *vector task system* can be characterized by a triple:

$$V = (T, <, \tau) \tag{3.21}$$

where

1. $T = \{T_1, T_2, \ldots, T_n\}$ is a set of *n* vector tasks.

2. $<$ is a partial ordering relation, specifying the precedence relationship among the tasks in set T.
3. $\tau: T \rightarrow R^+$ is a time function defining the production delay $\tau(T_i)$ for each T_i in T. We shall denote the value $\tau(T_i)$ simply as τ_i for all $i = 1, 2, \ldots, n$.

Let $P = \{P_1, P_2, \ldots, P_m\}$ be the set of vector pipelines and R^2 be the set of possible time intervals. The utilization of a pipeline P_i within interval $[x, y]$ is denoted by $P_i(x, y)$. The set of all possible pipeline-utilization patterns is called the *resource space*, which is equal to the cartesian product $P \times R^2 = \{P_i(x, y) | P_i \in P$ and $(x, y) \in R^2\}$. A parallel schedule f for a vector task system $V = (T, <, \tau)$ is a total function defined by

$$f: T \rightarrow 2^{P \times R^2} \tag{3.22}$$

where $2^{P \times R^2}$ is the power set of the resource space $P \times R^2$. Typically, we have the following mapping for each $T_i \in T$. The index $i_j \in \{1, 2, \ldots, n\}$ could be repeated

$$f(T_i) = \{P_{i1}(x_1, y_2), P_{i2}(x_2, y_2), \ldots, P_{i_p}(x_p, y_p)\} \tag{3.23}$$

This mapping actually subdivides the task T_i into p subtasks $T_{i1}, T_{i2}, \ldots, T_{ip}$. Subtask T_{ij} will be executed by pipeline P_{ij} for each $j = 1, 2, \ldots, p$. We call $\{T_{ij} | j = 1, 2, \ldots, p\}$ a *partition* of the task T_i. The following conditions must be met in order to facilitate multiple pipeline operations:

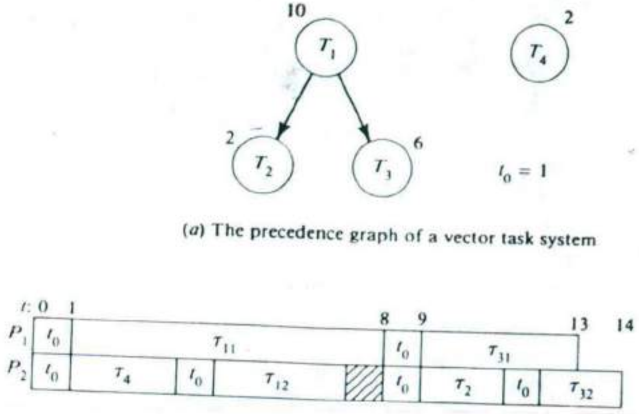
1. For all intervals $[x_j, y_j]$, $j = 1, 2, \ldots, p$, $y_j - x_j > t_o$ and the total production delay $\tau_i = \sum_{j=1}^{p} (y_j - x_j - t_o)$.
2. If $P_{ij} = P_{il}$, then $[x_j, y_j] \cap [x_l, y_l] = \phi$. This implies that each pipeline is static, performing only one subtask at a time.

The *finish time* for vector task T_i is $F(T_i) = \max\{y_1, y_2, \ldots, y_p\}$. The finish time ω of a parallel schedule for an n-task system is defined by

$$\omega \equiv \max\{F(T_1), F(T_2), \ldots, F(T_n)\} \tag{3.24}$$

The purpose is to find a "good" parallel schedule such that ω can be minimized. This deterministic scheduling concept is clarified by the following example:

Example 3.5 Given a vector task system V, as specified in Figure 3.44*a*, $T = \{T_1, T_2, T_3, T_4\}$, $t_0 = 1$, $\tau_1 = 10$, $\tau_2 = 2$, $\tau_3 = 6$, and $\tau_4 = 2$. These delays are marked beside each node of the task graph. We want to schedule four tasks on two ($m = 2$) pipelines. A parallel schedule f is shown in Figure 3.44*b*, where the shaded area denotes the idle periods of the pipelines. The vector task T_1 is partitioned into the two subtasks T_{11} and T_{12}, with $\tau_{11} = 7$ and $\tau_{12} = 3$. Similarly, the vector task T_3 is partitioned into the two subtasks



(*a*) The precedence graph of a vector task system

(*b*) A parallel schedule for the task system in (*a*)

Figure 3.44 Parallel scheduling of the task system of vector instructions in Example 3.5.

T_{31} and T_{32}, with $\tau_{31} = 4$ and $\tau_{32} = 2$. The parallel schedule f is specified by the following mappings with a finish time $\omega = F(T_3) = 14$:

$$f(T_1) = \{P_1(0, 8), P_2(3, 7)\} \qquad \text{with } S(T_1) = 0 \text{ and } F(T_1) = 8$$

$$f(T_2) = \{P_2(8, 11)\} \qquad \text{with } S(T_2) = 8 \text{ and } F(T_2) = 11$$

$$f(T_3) = \{P_1(8, 13), P_2(11, 14)\} \qquad \text{with } S(T_3) = 8 \text{ and } F(T_3) = 14$$

$$f(T_4) = \{P_2(0, 3)\} \qquad \text{with } S(T_4) = 0 \text{ and } F(T_4) = 3$$

The multiple-pipeline scheduling problem can be formally stated as a feasibility problem: *Given a vector task system V, a vector computer with m identical pipelines, and a deadline D, does there exist a parallel schedule f for V with finish time ω such that $\omega \leq D$?* This scheduling problem has been proven to be computationally intractable. In practice, the production delays of different vector tasks are different. These unequal production delays lead to the intractability of the multi-pipeline scheduling problem. Therefore, we have to seek heuristic algorithms in real-life system designs. The heuristics must be simple to implement, with low system overhead, and witlr nearly optimal performance.

Consider a vector processor with m pipelines with a fixed overhead time t_o for all instructions. The input to the scheduler is an independent task system V with n vector tasks which are totally unrelated. The task scheduler is a built-in part of the vector instruction controller. The output is the parallel schedule f for V. Let t_j be the time span of using pipeline P_j for the execution of various tasks in a given task system V. This time span includes the overhead time t_o every time the pipeline is reconfigured to assume a new task (or a new subtask), the production times τ_i(or τ_{ij}), and some idle times between successive tasks.

We denote the number of subtasks in a partition of task T_i as p_i. This task partitioning process requires $p_i - 1$ subdivisions of the original task. The total number of subdivisions of all tasks in a parallel schedule is expressed by:

$$k = \sum_{i=1}^{n} (p_i - 1) = \sum_{i=1}^{n} p_i - n \tag{3.25}$$

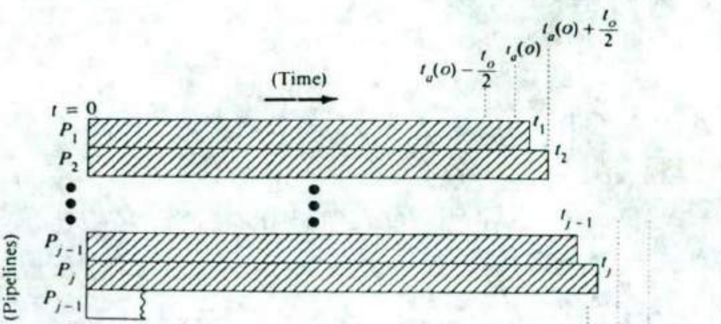
The average time span $t_a(k)$ for partitioning n tasks into $n + k$ subtasks over m pipelines is defined by

$$t_a(k) = \frac{\sum_{i=1}^{n} (\tau_i + t_o) + kt_o}{m} \tag{3.26}$$

If there is no subdivision of the original tasks in a schedule, the average time span $t_a(k)$ is reduced as follows, when $k = 0$.

$$t_a(0) = \sum_{i=1}^{n} \frac{\tau_i + t_o}{m} \tag{3.27}$$

This quantity $t_a(0)$ is an absolute lower bound of the finish time ω, defined in Eq. 3.24. This means that an optimal schedule is generated when $\omega = t_a(0)$. Scheduling n independent tasks among the m pipelines is done by making the time span t_j (for $j = 1, 2, \ldots, m$) as close to $t_a(k)$ as possible. As demonstrated in Figure 3.45, a *bin-packing* approach is used to generate a parallel schedule for independent tasks. First, we assign some tasks to pipeline P_1 until time $t_1 \geq t_a(0) - t_o/2$. Then we switch to pipeline P_2 for assigning the remaining tasks until $t_2 \geq t_a(k) - t_o/2$, where $k = 0$ or 1 depending on how many subdivisions of tasks have been



Note: shaded areas correspond to pipelines that have been assigned with vector tasks.

Figure 3.45 Multipipeline scheduling for independent vector tasks with a bin-packing approach.

performed. This generating process will repeat in a sequential manner for the remaining pipelines.

In general, we will switch to the next pipeline, P_{j+1}, when the following boundary condition is met:

$$t_j \leq t_a(k) - \frac{t_o}{2} \tag{3.28}$$

Furthermore, we will subdivide the current task and update $t_a(k)$ if the following condition is met, before switching to pipeline P_{j+1}:

$$t_j \geq t_a(k) + \frac{t_o}{2} \tag{3.29}$$

We consider below, as an example, the schedule of a tree structured task system based on the partitioned bin-packing procedure. This procedure generates a partition, $\{E_1, E_2, \ldots, E_l\}$, of all n tasks in the tree system. The first block E_1 consists of all tasks on leave nodes. The second block E_2 consists of those tasks on the "new" leave nodes after removing tasks in E_1 from the tree. This process continues until reaching the root, which forms the last block E_l where l equals the tree height. We shall process tasks in E_i before E_j, if $i < j$. In this sense, each E_i can be considered as a set of independent tasks, which can be dispatched concurrently as described above.

Example 3.6 We are given a tree task system $V = (T, \theta, \tau)$, where $T = \{T_1, \ldots, T_9\}$ follows the tree relationship shown in Figure 3.46*a*. Suppose $t_o = 1, \tau_1 = 2, \tau_2 = 4, \tau_3 = 6, \tau_4 = 8, \tau_5 = 8, \tau_6 = 2, \tau_7 = 6, \tau_8 = 4$, and $\tau_9 = 4$, as marked in the tree graph. To schedule this tree task system on $m = 4$ identical pipelines, we first obtain the partition $E_1 = \{T_1, T_2, T_3, T_4\}$, $E_2 = \{T_5, T_6, T_8\}$, $E_3 = \{T_7\}$, $E_4 = \{T_9\}$ as circled by dashed lines in the figure. A parallel schedule f_B is generated, as depicted in Figure 3.46*b*. Shaded areas indicate the idle periods of pipelines. Tasks T_2, T_3, T_4, T_5, T_7, and T_9 have been subdivided into subtasks:

$$f_B(T_1) = \{P_1(0, 3)\}$$

$$f_B(T_2) = \{P_1(3, 6.5), P_2(0, 2.5)\}$$

$$f_B(T_3) = \{P_2(2.5, 6.75), P_3(0, 3.75)\}$$

$$f_B(T_4) = \{P_3(3.75, 7), P_4(0, 6.75)\}$$

$$f_B(T_5) = \{P_1(7, 11.75), P_2(7, 12), P_3(7, 8.25)\}$$

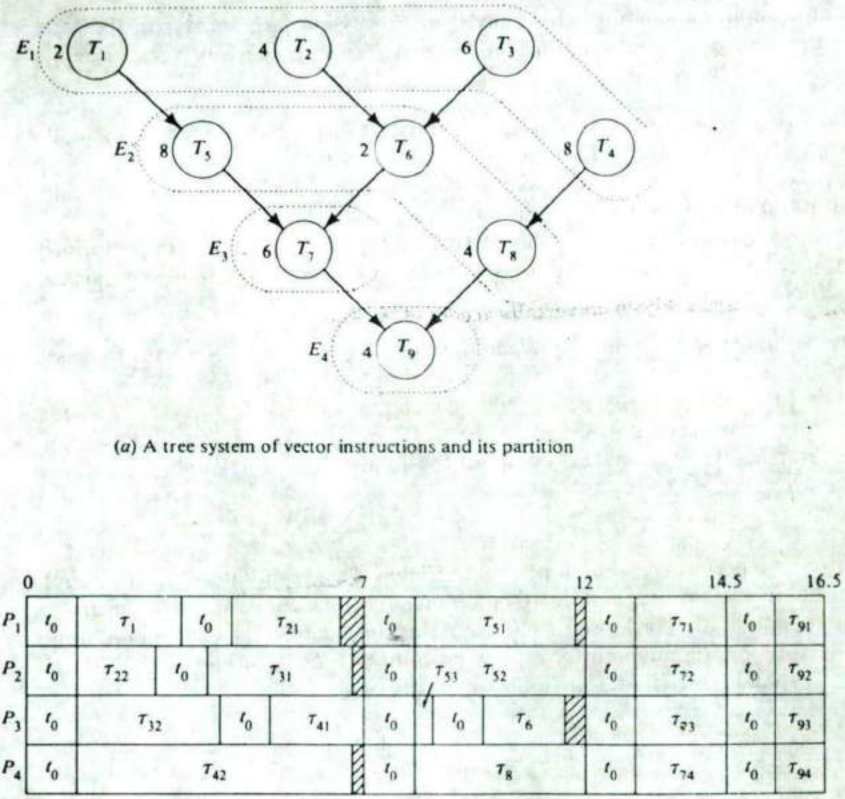
$$f_B(T_6) = \{P_3(8.25, 11.25)\}$$

$$f_B(T_8) = \{P_4(7, 12)\}$$

$$f_B(T_7) = \{P_i(12, 14.5): 1 \leq i \leq 4\}$$

$$f_B(T_9) = \{P_i(14.5, 16.5): 1 \leq i \leq 4\}$$

The finish time $\omega = 16.5$ has the same order of magnitude as $\omega_o = 13.25$, the finish time of an optimal schedule.


E_1 2 T_1
4 T_2
6 T_3
E_2 8 T_5
2 T_6
8 T_4
E_3 6 T_7
4 T_8
E_4 4 T_9
0
7
12
14.5
16.5
P_1 t_0 T_1 t_0 T_{21} t_0 T_{51} t_0 T_{71} t_0 T_{91}
P_2 t_0 T_{22} t_0 T_{31} t_0 T_{53} T_{52} t_0 T_{72} t_0 T_{92}
P_3 t_0 T_{32} t_0 T_{41} t_0 t_0 T_6 t_0 T_{73} t_0 T_{93}
P_4 t_0 T_{42} t_0 T_8 t_0 T_{74} t_0 T_{94}

(*a*) A tree system of vector instructions and its partition

(*b*) A parallel schedule

Figure 3.46 A tree task system of vector instructions and a parallel schedule for it.

Concurrent processing allows a vector to be partitioned into several subvectors for simultaneous execution by parallel pipelines. The fact that the parallel pipeline-scheduling problem is *NP*-complete precludes us from insisting on finding an optimal pipeline-scheduling algorithm. Since the average time span of all tasks is usually much longer than the overhead time, nearly optimal performance is guaranteed in the above heuristic-scheduling algorithms. With proper refinement of the precedence relations on the task graphs, one can extend this method to schedule vector task systems with arbitrary precedence relations. The partitioning of a vector by time units is equivalent to partitioning by vector lengths. The above pipeline-scheduling methodology can be applied to the design and evaluation of pipeline supercomputers for parallel vector processing.

3.4.3 Pipelined Vector Processing Methods

Vector computations are often involved in processing large arrays of data. By ordering successive computations in the array, we can classify vector (array) processing methods into three types:

1. *Horizontal processing*, in which vector computations are performed horizontally from left to right in row fashion
2. *Vertical processing*, in which vector computations are carried out vertically from top to bottom in column fashion
3. *Vector looping*, in which segmented vector loop computations are performed from left to right and top to bottom in a combined horizontal and vertical method

We use a simple vector-summation computation to illustrate these vector processing methods.

Let $\{a_i \text{ for } 1 \leq i \leq n\}$ be n scalar constants, $\mathbf{x}_j = (x_{1j}, x_{2j}, \ldots, x_{mj})^T$ for $j = 1, 2, \ldots, n$ be n column vectors, and $\mathbf{y} = (y_1, y_2, \ldots, y_m)^T$ be a column vector of m components. We need to compute the following linear combination of n vectors:

$$\mathbf{y} = a_i \cdot \mathbf{x}_1 + a_2 \cdot \mathbf{x}_2 + \cdots + a_n \cdot \mathbf{x}_n \tag{3.30}$$

One needs to perform $m \cdot n$ multiplications and $m \cdot (n - 1)$ additions in the above vector computations. Expanding all component computations will help visualize different computation orderings to be used. For simplicity, we specify all multiplications by the shorthand notation: $z_i = a_j \cdot x_{ji}$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. We can expand Eq. 3.30 into the following array of additions:

$$\begin{aligned} y_1 &= z_{11} + z_{12} + \cdots + z_{1n} \\ y_2 &= z_{21} + z_{22} + \cdots + z_{2n} \\ &\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ y_m &= z_{m1} + z_{m2} + \cdots + z_{mn} \end{aligned} \tag{3.31}$$

Let us now consider the implementation of the above array computations in a pipeline processor. This processor has a static two-function arithmetic pipeline with five stages. Suppose that the pipeline can perform either addition or multiplication of two numbers, but not both simultaneously. One should first implement the mn multiplications through the pipeline, then follow with the implementation of the $m(n - 1)$ additions specified in Eq. 3.31. This separation of multiply and add functions will result in a minimum reconfiguration cost by eliminating unnecessary pipeline setup delays and, thus, increasing the overall pipeline throughput.

In a nonpipeline scalar processor, each addition or multiplication requires $5t$, where t is the clock period or one stage delay. The total execution time (without pipelining) is thus equal to

$$T_1 = [mn + m(n - 1)] \cdot 5t = (10mn - 5m)t \tag{3.32}$$

We will compare the total execution times of various pipeline processing methods with T_1 in order to reveal their relative speedups.

The way that addition pairs (operands) are scheduled distinguishes the three processing methods. In what follows, we assume that all multiplications have already been carried out by the pipeline in T_m pipeline cycles:

$$\begin{aligned} T_m &= 5t \text{ (setup time)} + 5t \text{ (time for the first product to come out of the pipe)} \\ &\quad + (mn - 1)t\text{(time for producing all the remaining products)} \\ &= mnt + 9t \end{aligned} \tag{3.33}$$

It is assumed that the main memory is large enough to hold all intermediate results. There is a feedback path from the output of the pipeline to one of the two inputs if needed for cumulative additions. Let T_a be the total number of clock periods needed for the pipelined addition in each of the following methods. It is assumed $m \gg 5$ and $n \gg 5$.

Horizontal vector processing In this method, all components of the vector **y** are calculated in a sequential order, y_i for $i = 1, 2, \ldots, m$. Each summation $y_i = \sum_{j=1}^{n} z_{ij}$ involving $(n - 1)$ additions must be completed before switching to the evaluation of the next summation $y_{i+1} = \sum_{j=1}^{n} z_{i+1,j}$. To evaluate each y_i requires $(n + 14)t$ clock periods. The total *add time* for m outputs equals

$$T_a(\text{horizontal}) = (mn + 14m)t \tag{3.34}$$

This method is frequently used in a scalar pipeline processor. The above sequence of computations corresponds to the following Fortran program, provided that all initial values of y_i for $i = 1, 2, \ldots, n$, are set to zero.

```
DO 100 i=1,m,1
DO 100 j=1,n,1
yi=yi+ai*xij
100 CONTINUE
```

The speedup of this horizontal pipelining on a vector processor over serial processing in a uniprocessor is derived below:

$$S_{\text{horizontal}} = \frac{T_1}{T_m + T_a(\text{horizontal})} = \frac{(10mn - 5m)t}{(2mn + 14m + 9)t} = \frac{10mn - 5m}{2mn + 14m + 9} \tag{3.35}$$

Vertical vector processing The sequence of additions in this method is specified below with respect to the m-by-n array shown in Eq. 3.31:

Step 1. Compute the partial sums $(z_{i1} + z_{i2}) = y^i_{12}$ for $i = 1, 2, \ldots, m$ sequentially through the pipeline.

Step 2. Compute the partial sums $(y^i_{12} + z_{i3})$ for $i = 1, 2, \ldots, m$ by loading y^i_{12} into one input port in stage 1 and loading z_{i3} into the second input port.

Step 3 to Step $n - 1$. Repeat Step 2 for $n - 3$ times by feeding successive columns $(z_{1j}, z_{2j}, \ldots, z_{mj})^T$ for $j = 4, 5, \ldots, n$, into the second input port. The values of y_i for $i = 1, 2, \ldots, m$ emerge from the pipeline at the end of Step $n - 1$.

The total add time of this vertical approach equals

$$T_a(\text{vertical}) = (mn - m + 10)t \tag{3.36}$$

Therefore, the speedup of vertical vector processing over uniprocessing equals

$$S_{\text{vertical}} = \frac{T_1}{T_m + T_a(\text{vertical})} = \frac{10mm - 5m}{2mn - m + 19} \tag{3.37}$$

This method has been applied to vector processing in the STAR-100.

Vector looping method This method combines the horizontal and vertical approaches into a "block" approach. The steps are specified below.

Step 1. Apply the vertical processing method to generate the first block of five outputs, $y_1, y_2, \ldots, y_5$, in column fashion.

Step 2 to Step k. Repeat Step 1 for generating the remaining five-output blocks as listed below:

Step 2: $y_6, \quad y_7, \quad \ldots, y_{10}$

Step 3: $y_{11}, \quad y_{12}, \quad \ldots, y_{15}$

. . . .

Step k: $y_{5k-4}, y_{5k-3}, \ldots, y_{5k}$

Step $k + 1$. Repeat Step 1 for generating the last block of r outputs, $y_{5k+1}, y_{5k+2}, \ldots,$ and y_{5k+r}, where $m = 5k + r$ and $0 < r < 5$.

The total add time of this vector-looping method is given below, where $k = (m - r)/5$.

$$\begin{aligned} T_a(\text{vector looping}) &= 5t + (5n - 1)t + (k - 1)[5(n - 1)t] + 5t \\ &= mnt - mt - nrt + rt + 14t \end{aligned} \tag{3.38}$$

The speedup of the vector-loop method over a uniprocessing method equals:

$$S_{\text{vector looping}} = \frac{T_1}{T_m + T_a(\text{loop})} = \frac{10mn - 5m}{2mn - m - rn + r + 23} \tag{3.39}$$

This method has been applied in the Cray-1 for segmented vector processing.

The horizontal method is suitable for use in scalar processors but unfit for parallel processing in a vector processor. Both vertical and vector-looping methods are attractive for vector processors. In vertical processing, the number of vector components m is unrestricted. However, many intermediate results (partial sums in the example) have to be stored in the memory. This poses the problem of increased demand for memory bandwidth. Vertical processing is more suitable for memory-to-memory pipeline operations, like those in the Star-100 and the Cyber-205. The vector-looping method is also not restricted by vector length. Since the intermediate results appear as small blocks of data, one can use a cache memory or fast-register arrays to hold the intermediate results. Thus vector looping is more suitable for register-to-register pipeline operations, such as in the Cray-1 and the Fujitsu VP-200. It is interesting to note that all the speedups approach 5, the number of stages in the sample pipeline, when *n* and *m* are very large in the performance analysis.

The speed of a scalar processor is usually measured by the number of instructions executed per unit time, such as the use of a *million instructions per second* (MIPS) as a measure. For a vector processor, it is universally accepted to measure the number of arithmetic operations performed per unit time, such as the use of *mega floating-point operations per second* (megaflops). Note that the conversion between mips and megaflops depends on the machine type. There is no fixed relationship between the two measures. In general, to perform a floating-point operation in a scalar processor may require two to five instructions. If we consider the average as three, then one megaflops may imply three mips. This conversion constant is machine dependent. Other authors compare the speeds of different computers by choosing a reference machine. Readers should be aware of the difference between the *peak speed* and the *average speed* when benchmark programs or test computations are executed on each machine. The peak speed corresponds to the maximum theoretical CPU rate, whereas the average speed is determined by the processing times of a large number of mixed jobs including both CPU and I/O operations.

3.5 BIBLIOGRAPHIC NOTES AND PROBLEMS

An earlier survey of pipeline computer architecture was given by Ramamoorthy and Li (1977). A recent assessment of pipeline processors and vectorization methods can be found in Hwang et al. (1981). The concept of overlapped parallelism was studied in Chen (1971a, b; 1975). Pipeline processors were described in Hayes (1978), Kuck (1978), and Stone (1980). The classification of pipeline computers is based on the work of Händler (1977). Good examples of instruction and arithmetic pipelines can be found in Anderson et al. (1967), Hintz and Tate (1972), Majithia (1976), Hwang (1979), and Waser and Flynn (1982). Interleaved memory systems for pipelining and parallel computers have been studied in Hellerman (1967), Burnett and Coffman (1970), Knuth and Rao (1975), Chang et al., (1977), Briggs and Davidson (1977), and Briggs and Dubois, (1983). A comprehensive treatment of pipeline computer systems can be found in Kogge (1981).

Pipeline models and task scheduling problems have been studied in Davidson (1971), Reddi (1972), Thomas and Davidson (1974), Ramamoorthy and Kim (1974), Li (1975), and Lang et al. (1979). Instruction prefetch techniques were treated by Rau (1977) and Grohoski and Patel (1982). Different aspects of busing, branching, and interrupt-handling of pipeline operations were also treated in Ramamoorthy and Li (1977). The dynamic pipelines with improved throughput using noncompute delays and internal buffers were proposed by Patel (1976, 1978a, b). Lookahead techniques such as hazard resolution and data forwarding have been treated by Keller (1975) and Tomasulo (1967). The modeling of a vector processor with multiple pipelines is based on the work of Hwang and Su (1983). Static pipes are commercially designed because of less control and hardware costs. However, systems requiring reliable and flexible designs may have to use dynamic pipes in order to enhance fault-tolerance capability and to increase the resource utilization.

Problems

3.1 Describe the following terminologies associated with pipeline computers and vector processing:

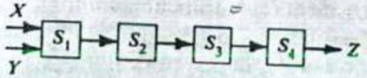
(*a*) Static pipeline
(*b*) Dynamic pipeline
(*c*) Unifunctional pipeline
(*d*) Multifunctional pipeline
(*e*) Instruction pipeline
(*f*) Arithmetic pipeline
(*g*) Pipeline efficiency
(*h*) Pipeline throughput
(*i*) Forbidden latencies
(*j*) Collision vector
(*k*) Minimum average latency
(*l*) Precise vs. imprecise interrupts
(*m*) Perfect cycle
(*n*) Greedy cycle
(*o*) Data-dependent hazards
(*p*) Short circuiting
(*q*) Internal forwarding
(*r*) Vectorizer
(*s*) Branch target buffering
(*t*) Register tagging

3.2 Compare the advantages and disadvantages of the three interleaved memory organizations: the S-access, the C-access, and the C/S-access described in Section 3.1.4 for pipelined vector accessing. In the comparison, you should be concerned with the issues on effective memory bandwidth, storage schemes used, access conflict resolution, and cost-effectiveness tradeoffs.

3.3 Consider a four-segment normalized floating-point adder with a 10-ns delay per each segment, which equals the pipeline clock period.

(*a*) Name the appropriate functions to be performed by the four segments.

(*b*) Find the minimum number of periods required to add 100 floating-point numbers $A_1 + A_2 + \cdots + A_{100}$ using this pipeline adder, assuming that the output Z of segment S_4 can be routed back to any of the two inputs X or Y of the pipeline with delays equal to any multiples of the period.



3.4 A certain dynamic pipeline with the four segments S_1, S_2, S_3, and S_4 is characterized by the following reservation table:

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
S_1	☒					☒	
S_2			☒				☒
S_3		☒		☒			
S_4			☒		☒		

(*a*) Determine latencies in the forbidden list F and the collision vector C.

(*b*) Determine the minimum constant latency L by checking the forbidden list.

(*c*) Draw the state diagram for this pipeline. Determine the *minimal average latency* (MAL) and the *maximum throughput* of this pipeline.

3.5 For the following reservation table of a pipeline processor, give the forbidden list of avoided latencies F, the lower bound on latency, the collision vector, the state diagram, the MAL and all greedy cycles.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
S_1	☒								☒
S_2		☒	☒					☒	
S_3				☒					
S_4				☒	☒				
S_5						☒	☒		

3.6 The following overlayed reservation table corresponds to a two-function pipeline:

	t_0	t_1	t_2	t_3	t_4
S_1	A	B		A	B
S_2		A		B	A
S_3	B		AB		

(*a*) List all four cross forbidden lists of latencies and corresponding combined cross-collision matrices.

(*b*) Draw the state diagram for the two-functional pipeline.

3.7 Assume that instructions are executed in a k-segment pipeline. The delay of each segment is one time unit. If an instruction depends on one or more of its predecessors, then all these predecessors must complete execution before the current instruction can begin execution. If such a predecessor is N instructions preceding the current instruction, a delay is added as $k - N$ time units for $N \leq k$ and no delay for $N > k$. Let p_n be the probability of encountering a data dependency from the nth predecessor. Assume an integer $L > k$. Suppose that p_n has the distribution $P_n = 1/L$ for $n = 1, 2, \ldots, L$ and p_n equal zero otherwise.

(*a*) Find the expected value of the total time T to execute a block of M instructions.

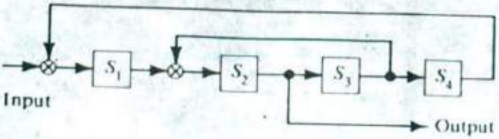
(*b*) Determine the performance P of the instruction pipeline, where

$$P = \lim_{M \to \infty} \frac{M}{T}$$

3.8 (*a*) Suppose that only two 4-segment pipelined adders and a number of noncompute delay elements are available. The delay of each segment is one time unit and the noncompute delay element can have either a one- or two-time unit delay. Using available resources, construct a pipeline with only one input, a's, to compute $b(i) = a(i) + a(i-1) + a(i-2) + a(i-3)$. Show the schematic block diagram of your design.

(*b*) Given one additional four-segment pipelined adder, use this adder together with the pipeline obtained from (*a*) to design a pipeline for computing the recurrence function $x(i) = a(i) + x(i-1)$. The pipeline constructed should have a feedback. Show your schematic block diagram. *Hint*: $x(i) = a(i) + x(i-1) = a(i) + [a(i-1) + x(i-2)] = a(i) + a(i-1) + [a(i-2) + x(i-3)] = a(i) + a(i-1) + a(i-2) + [a(i-3) + x(i-4)] = b(i) + x(i-4)$.

3.9 Consider the following pipelined processor with four stages. All successor stages after each stage must be used in successive clock periods.



Answer the following questions associated with using this pipeline with an *evaluation time* of six pipeline clock periods.

(*a*) Write out the reservation table for this pipeline with six columns and four rows.

(*b*) List the set of forbidden latencies between task initiations.

(*c*) Show the initial collision vector.

(*d*) Draw the state diagram which shows all the possible latency cycles.

(*e*) List all the simple cycles from the state diagram.

(*f*) List all the greedy cycles from the state diagram.

(*g*) What is the value of the minimal average latency (MAL)?

(*h*) Indicate the minimum constant latency cycle for this pipeline.

(*i*) What is the maximal throughput of this pipeline?

3.10 (*a*) How does the IBM 360/91 avoid problems due to data dependencies involving the contents of floating-point registers within the floating-point execution unit? In your answer, especially address each type of hazard, indicating how each is controlled.

(*b*) The floating-point execution unit in the 360/91 handles data dependencies involving floating-point register contents. What data dependencies can arise in the execution of floating-point instructions (including loads to and stores from the floating-point registers) that involve the contents of some memory word? How can these dependencies be managed? Efficiency is a prime consideration. Use a block diagram to illustrate the organization of the major hardware units that your solution requires. Explain the operation of each of these units.

3.11 Answer the following equations related to the task initiation cycle (2, 3, 7) for a given pipelined processor.

(*a*) What are the period p and the average latency l_a of this initiation cycle?

(*b*) Specify the initiation interval set G (mod p).

(*c*) What is the necessary and sufficient condition that a given task initiation cycle is allowed by a pipeline with a forbidden latency set F? Repeat the same question for a constant initiation cycle with period p.

3.12 Suppose that scalar operations take 10 times longer to execute per result than vector operations. Given a program which is originally written in scalar code:

(*a*) What are the percentages of the code needed to be vectorized in order to achieve the speedup factors of 2, 4, and 6 respectively?

(*b*) Suppose the program contains 15% of code that cannot be vectorized such as sequential I/O operations. Now repeat question (*a*) for the remaining code to achieve the three speedup factors.

CHAPTER

FOUR

PIPELINE COMPUTERS AND VECTORIZATION METHODS

This chapter describes the system architectures and vector processing techniques developed with existing pipeline computers. The first section gives a historical retrospective of pipeline computers in two architectural categories: *vector supercomputers* and *attached array processors*. We will examine three attached processors: the AP-120B (FPS-164), the IBM 3838, and the MATP. Vector supercomputers to be studied include the early systems Star-100 and TI-ASC, and the recent systems Cray-1, Cyber-205, and VP-200, and their possible extensions. Finally, we will study vectorizing compiling techniques, optimization methods, and performance evaluation issues in designing or using pipeline computers.

4.1 THE SPACE OF PIPELINE COMPUTERS

Pipeline computers refer to those digital machines that provide overlapped data processing in the central processor, in the I/O processor, and in the memory hierarchy. Pipelining is practiced not only in program execution but also in program loading and data fetching operations. Univac-1 was the first machine that overlapped program execution with some I/O activities. With the development of interleaved memory, memory words in successive memory modules could be fetched in a pipelined fashion. These pipeline memory fetches prompted the overlapped instruction fetches and instruction executions as pioneered in the IBM 7094 series in the Stretch project and in the Univac-Larc system.

The performance of a pipeline processor may be significantly degraded by the data dependency holdup problem. The evolution of the CDC 6000/7000 series has contributed to the development of hardware/software mechanisms to overcome this difficulty. In addition to further partitioning the instruction execution process, the CDC 6000 series uses a "status checkboard" to indicate the availabilities of various resources in the computer required to execute various stages of subsequent instructions. Resource conflicts are recorded in the checkboard. Instructions being interrupted are temporarily queued for deferred executions. A single instruction may be deferred several times because of sequence of resource conflicts. Multiple arithmetic units are employed in the CDC 6600/7600 to alleviate the resource-conflict problem in the overlapped executions of multiple instructions in the system.

The development of the IBM System/360 Model 91 scientific processor has greatly enhanced the design methodology of pipeline computers. A hierarchy of pipelines is employed in the Model 91 for instruction fetch, preprocessing, and execution, as described in Chapter 3. Mechanisms are provided to prefetch instructions at both alternative program paths after a conditional branch instruction. Continued instruction execution can be sustained with prefetching to increase the system throughput. High-speed instruction and data buffers are used to make the above approach possible. Fast internal data forwarding techniques were also implemented in the IBM System/360 Model 91 and its successor IBM System/370 models to overcome the difficulty caused by hazards or out-of-sequence executions. Many of these pipeline design techniques have appeared in later machines like the Amdahl 470 V/6 and the IBM 3081.

4.1.1 Vector Supercomputers

A *supercomputer* is characterized by its high computational speed, fast and large main and secondary memory, and the extensive use of parallel structured software. Most of today's supercomputers are designed to perform large-scale vector or matrix computations in the areas of structural engineering, petroleum exploration, VLSI circuit design, aerodynamics, hydrodynamics, meteorology, nuclear research, tomography, and artificial intelligence. The demand for high speed and large internal memory is obvious in these scientific applications. Large amounts of data are often processed by a supercomputer. Usually the data elements are arranged in array, vector, or matrix forms. The large data arrays are collected from, for example, seismic echo signals after the set off of a sonic shock wave into the ground. In 1979 alone, 10^{15} bits of seismic data were processed in the United States. Similar examples can be found in radar and sonar signal processing for detection of space and underwater targets, in remote sensing for earth resource exploration, in computational wind tunnel experiments, in three-dimensional stop-action computer-assisted tomography, in numerical weather forecasting, and in many real-time applications. In terms of speed, current supercomputers should be able to operate at a speed of 100 megaflops or higher.

The first generation of vector supercomputers is marked by the development of the Star-100, TI-ASC, and the Illiac-IV in the 1960s. By 1978, there were seven installations of ASC, four installations of Star-100, and only one Illiac-IV system installed at user sites. We will first study the Star-100 and ASC systems. Both the Star-100 and the ASC systems are equipped with multiple functional pipeline processors to achieve parallel vector processing. The Star-100 has a memory-to-memory architecture with two pipeline processors. The ASC can handle up to three-dimensional vector computations in pipeline mode. The peak speed of both systems is around 40 megaflops. We will study Illiac-IV in Chapter 6.

Vector processors entered the second generation with the development of the Cray-1, the Cyber-200 series, and the Fujitsu VP-200. The Cray-1, evolved from the CDC 6600/7600 series, is considered one of the fastest supercomputers that has ever been built. The maximum CPU rate of the Cray-1 is 160 megaflops if all the resources are fully utilized. The Cyber-200 series is extended from the Star-100. The Cyber 205 has both vector and scalar pipelines, with the potential to perform 800 megaflops. As of September 1982, there were over 60 Cray-1 and Cyber-205 machines installed all over the world. Recently, Fujitsu in Japan announced a vector processor, VP-200, which can perform up to 500 megaflops.

For the future, it is highly necessary to have a vector processor which can perform 1000 megaflops or more. Cray Research is currently extending the Cray-1 to a multiprocessor configuration, called Cray X-MP. This Cray X-MP, consisting of dual processors, is expected to be five times more powerful than the Cray-1, with an expected peak speed of 400 megaflops. Eventually, Cray Research plans to further upgrade Cray X-MP to a four-processor model, called Cray-2, which will be 12 times more powerful than Cray-1 in vector processing mode. CDC has proposed to upgrade the Cyber-205 eventually to a vector processor that can provide 3000 megaflops for numerical aerodynamic simulations. Each uniprocessor in the multiprocessor system S-1, under construction at the Lawrence Livermore National Laboratory, is also highly pipelined, with an expected CPU rate twice as fast as the Cray-1. Cray X-MP, Cray-2, and S-1 will be introduced in Chapter 9, along with other multiprocessor systems.

4.1.2 Scientific Attached Processors

Most scientific processors are designed as back-end machines attached to a host computer. Most of these *attached processors* are pipeline structured. The most used system is the AP-120B and FPS-164 array processors manufactured by Floating-Point Systems. We will study three attached processors, including the AP-120B (FPS-164), the IBM 3838, and the Datawest array transform processor.

These attached processors are mostly designed to enhance the floating-point and vector-processing capabilities of the host machine. They can be attached to minicomputers and mainframes. For example, an AP-120B can be attached to a VAX 11/780, increasing the computing power of the VAX to 12 megaflops. Attached processors cost much less than the mainframes or supercomputers. However, most attached array processors must be microcoded for specific vector applications. The application software costs are much higher than the bare hardware costs, especially when microcode packages must be developed by users for special-purpose computations.

In a host back-end computer organization, the host is a program manager which handles all I/O, code compiling, and operating system functions, while the back-end attached processor concentrates on arithmetic computations with data supplied by the host machine. High-speed interface is often needed between the host and the back-end machine. In this sense, the supercomputers Cray-1 and Cyber-205 are also back-end machines driven by a host machine.

The projected speed performances of the aforementioned pipeline supercomputers and attached processors are compared in Figure 4.1. We use the measure *million operations per second* (mops) to refer to either megaflops or a million integer operations per second. All speeds indicated within the parentheses refer to the theoretical peak performance, if the machine is used sensibly. In the late sixties, only pipeline scalar processors were available with a maximum speed of 5 mops, as represented by the IBM 360/91 and 370/195 series, and by the CDC 6600/7600 series. The first-generation vector processors Star-100 and TI-ASC have a speed ranging from 30 to 50 mops. The second-generation vector processors Cray-1, Cyber-205, and VP-200 have a speed between 100 and 800 mops.

The attached processors AP-120B and FPS-164 have a peak speed of 12 megaflops. The FACOM 230/75 can perform 22 megaflops. MATP is a four-pipeline multiprocessor which can operate up to 120 megaflops. The IBM 3838 has a peak speed of 30 megaflops. The Fujitsu VP-200 is extended from its predecessor FACOM 230/75. The first Cray X-MP became available in 1983.

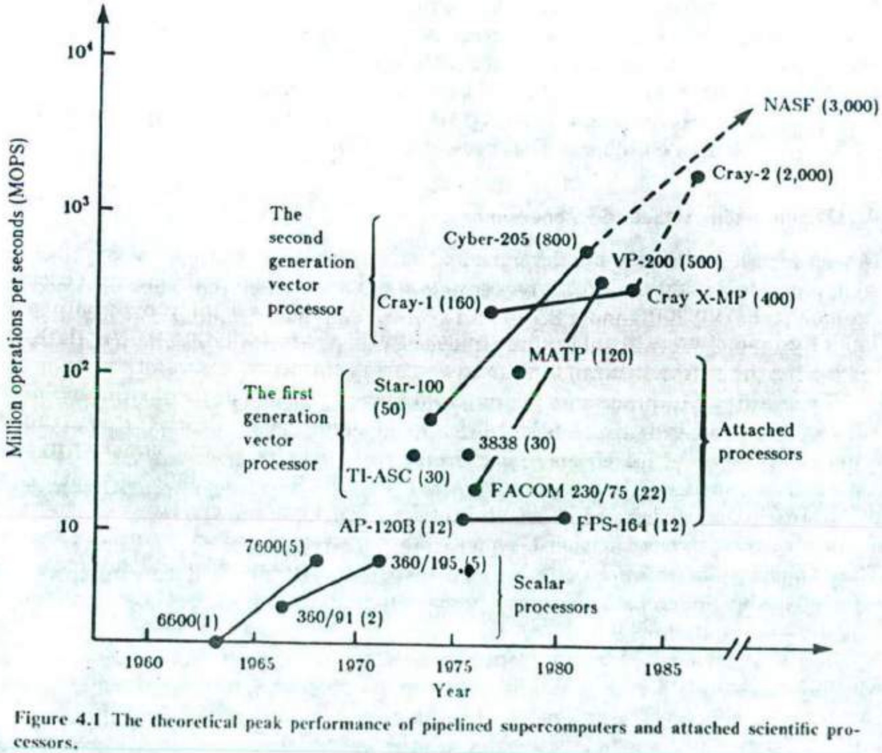


Figure 4.1 The theoretical peak performance of pipelined supercomputers and attached scientific processors.

The future supercomputers Cyber 2xx, Cray 2, and S-1 are expected to perform over 3000 megaflops for applications in the 1990s.

Of course, the peak speeds in Figure 4.1 may not always be attainable. For average programs written by nonspecialists, the speed is much lower, than those peak values indicated. For mixed programs, it has been estimated that the average rate of the Cray-1 is 24 megaflops, of the Star-100 is 16 megaflops, and of the AP-120B is only 6 megaflops. These measured operating speeds are low because the software is not properly tuned to explore the hardware. These issues will be studied in Section 4.5 along with language, compiling, vectorization, and optimization facilities for vector processors.

4.2 EARLY VECTOR PROCESSORS

Dedicated computers for vector processing started with the introduction of two supercomputer systems known as the CDC-Star and the TI-ASC, both of which have multiple pipeline processors for stand-alone operations. Based on the employed technology and architectural features, the two generations of vector processors differ in many aspects. In this section, architectural structures, pipelined arithmetic designs and vector processing in the Star and ASC are described. In the next section, we will study more recent vector processors.

4.2.1 Architectures of Star-100 and TI-ASC

The Star-100 is a vector-oriented processor with two nonhomogeneous arithmetic pipelines. Control Data Corporation started the design of Star in 1965 and delivered it to user sites in 1973. It is structured around a four million byte (eight million optional) high-bandwidth core memory for stand-alone operations. Special features designed in the Star-100 include stream processing, virtual addressing, hardware microinstructions, semiconductor memory-register file, and pipelined floating-point arithmetic. The core memory in the Star has a cycle time of 1.28 μs. It has 32 interleaved memory banks, each containing 2048 words of 512 bits each. The memory cycle is divided into 32 minor cycles, with a rate of 40 ns each. This implies that the memory supplies 512 bits of data per minor cycle.

The pipelined arithmetic units are especially designed for sequential and parallel operations on single bits, 8-bit bytes, and 32- or 64-bit floating-point operands and vectors. Virtual addressing employs a high-speed mapping technique to convert a logical address to an absolute memory address. In the ideal case, the system has the capability of producing 100 million 32-bit floating-point results per second. The system architecture of the Star-100 is shown in Figure 4.2. The memory banks are organized into eight groups of four banks each. During the streaming operations, all four buses will be active, with each bus transferring data at a rate of 128 bits per minor cycle. Two of the buses are used for transferring

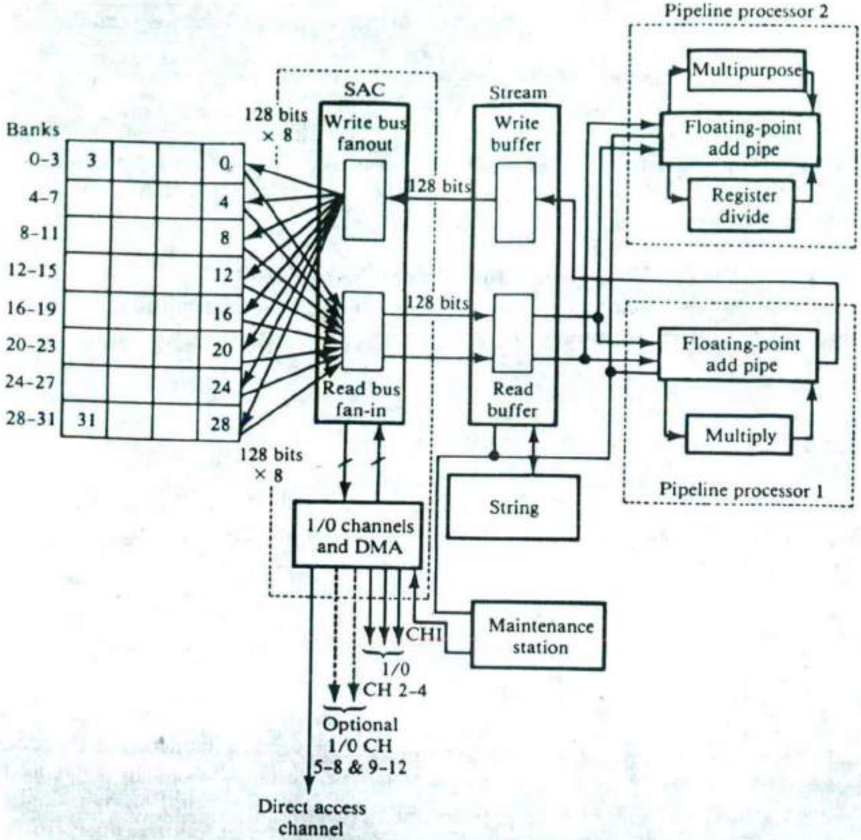


Figure 4.2 The system architecture of Star-100. (Courtesy of Control Data Corp.)

operand streams to the pipeline processors. The third bus is used for storing the result stream, and the fourth bus is shared between input-output storage requests and the references of control vectors.

The *Storage Access Control* unit controls the transmission of all data to and from the memory. It is responsible for memory sharing among the various buses shared by the stream and I/O units. Its principal function is to perform virtual memory address comparison and translation. The *Stream unit* provides basic control for the system. All memory references and many control signals originate from this unit. It has the facilities for instruction buffering and decoding. The *Read Buffer* and *Write Buffer* are used to synchronize the four active buses to maintain a smooth data transfer. The memory requests are buffered eight banks apart to avoid access conflicts. As a result, the maximum pipeline rate can be sustained regardless of distribution of addresses on the four active buses.

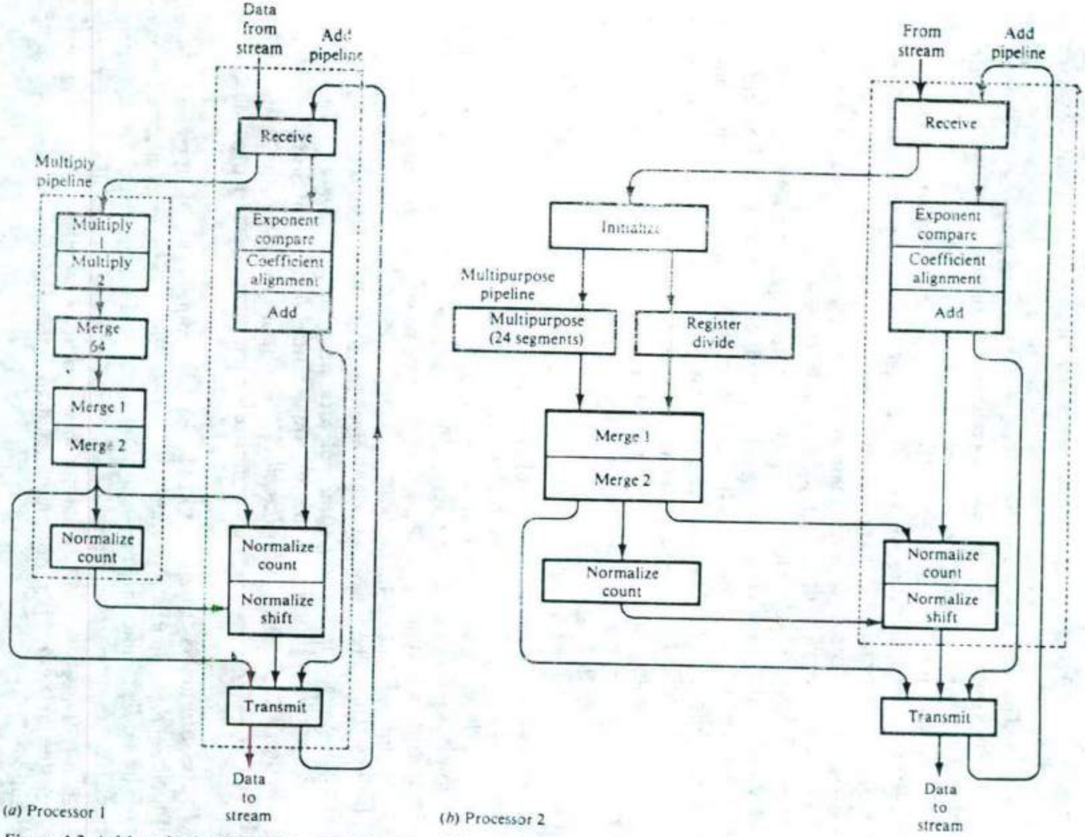
Other functional units in the Stream unit include the register file and the microcode memory. The register file supplies necessary addressing for all source operands and results. It also has the capability of performing simple logical and arithmetic operations. The semiconductor microcode memory is used as part of the stream control. The control signals and enable conditions produced by the microcode are used together with the hardwired control to process instructions and interrupts. The *String unit* processes strings of decimal or binary digits and performs bit-logical and character-string operations. It contains several adders to execute binary coded decimal (BCD) and binary arithmetic.

In the Star-100 are two independent arithmetic pipelines (Figure 4.3). The pipeline processor 1 consists of a 64-bit *floating-point* (henceforth FLP) *add unit* and a 32-bit FLP *multiply unit*. The add pipeline on the right contains four segment groups in cascade. The *exponent compare* segment compares exponents and saves the larger. The difference between the exponents is then saved as a shift count by which the fraction with the smaller exponent is right-shifted in the *coefficient alignment* segment. In the add segment, the shifted and unshifted fractions are added. The sum and the larger exponent are then gated to the *normalized* segments. The *transmit* segment selects the desired upper or lower half of the sum, checks for any fraction overflow, and transmits the results to the designated data bus. There is a path from the output of the transmit segment to the input of the *receive* segment. This feedback feature is especially useful for continuous addition of multiple floating-point numbers. However, when nonstreaming-type operations are performed, the execution time can be decreased by 50 percent if the output of an operation is needed as an input operand for subsequent operations.

With little additional hardware, it is possible to split the 64-bit add pipeline into two independent 32-bit ones. Consequently, half-width (32-bit) arithmetic can be available. The 32-bit *multiply pipeline* is implemented with multiplier-recoding logic, multiplicand-gating network, and several levels of carry-save adders. A resultant product of the multiplication is formed by adding the final partial sum and the saved carry vector. The required post-normalization after FLP multiply is done using the normalize segments of the add pipeline on the right.

Processor number 2, depicted in Figure 4.3*b*, contains a pipelined *add unit*, a nonpipelined *divide unit*, a pipelined *multipurpose unit*, and some pipelined *merge units*. The add pipeline in processor number 2 is similar to that in processor number 1. The multipurpose pipeline has 24 segments and is capable of performing multiply, divide, square root, and a number of other arithmetic logic operations. The *register divide unit* is a nonpipelined divider which can also perform BCD arithmetic.

Two 32-bit multiply pipelines can be combined to form a 64-bit multiply pipeline. This combined unit can simultaneously execute two 32-bit multiplications or execute one 64-bit multiplication. In order to perform a 64-bit multiplication, the multiplicand A and multiplier B are each split into two parts, $A = A_0 + A_1 \cdot 2^w$,



(*a*) Processor 1

(*b*) Processor 2

Figure 4.3 Arithmetic pipelines in Star-100. (Courtesy of Control Data Corp.)

and $B = B_0 + B_1 \cdot 2^w$, where $w = 32$ bits, the width of the basic multiply pipeline. Then the following four multiplications are performed:

$$A \times B = A_0 \times B_0 + (A_0 \times B_1 + A_1 \times B_0) \cdot 2^w + (A_1 \times B_1) \cdot 2^{2w} \quad (4.1)$$

$A_0 \times B_0$ and $A_0 \times B_1$ are executed during the first cycle of multiplication, and $A_1 \times B_0$ and $A_1 \times B_1$ during the second cycle. Afterward, all partial sums and partial carries are merged in a 64-bit merge section, which is essentially a set of carry-save adder trees (pipelines). The partial sum and partial product from the 64-bit merge section are then added together by two adders to yield the final 64-bit product.

The Star-100 has 130 scalar instructions and 65 vector instructions, as categorically listed in Table 4.1. Vectors in the Star-100 are formed as *strings* of binary numbers or characters, or as *arrays* of 32- or 64-bit FLP numbers. The *sparse vector* instructions can process compressed sparse vectors. When the pipeline enters streaming operations, it is possible to maintain a 40-ns output rate. The input-to-output time for the FLP add pipeline is 160 ns, because there are essentially four pipeline segments. The time delay of the FLP multiply pipeline equals 320 ns. The maximum throughput for different arithmetic operations is summarized in Table 4.2. These are peak CPU speeds. In practice, the measured average speed of the Star-100 is only 0.5 to 1.5 megaflops for scalar operations and 5 to 10 megaflops for vector operations, lower than its designed capabilities. It is quite obvious that double-precision FLP operations require more time to complete, twice the add/subtract time and four times the multiply or divide time compared to their single-precision counterparts.

Texas Instruments *Advanced Scientific Computer* (ASC) was delivered in 1972. The central processor of ASC is incorporated with a high degree of pipelining in both instruction and arithmetic levels. The basic components of the ASC system

Table 4.1 Instruction types in Star-100

Scalar instructions	Vector instructions
Load and Store	Arithmetic
Arithmetic	Compare
Index	Search
Increment and Test	Move
Bit operations	Normalize
Normalize	Data type conversion
Data type conversion	Sparse vector
Branch	Vector macros
String	Dot product
Logical	Polynomial evaluation
Monitor call	Average difference
	Average
	Adjacent mean

Table 4.2 Maximum numbers of arithmetic operations executable in the CDC Star-100 system for short and long word lengths (in mops)

Floating-point operations	32-bit (short)	64-bit (long)
Add-subtract	100	50
Multiply	100	25
Divide	50	12.5
Square root	50	12.5

are shown in Figure 4.4. The central processor is used for its high speed to process a large array of data. The peripheral processing unit is used by the operating system. Disk channels and tape channels support a large number of storage units. Data concentrators are included for support of remote batch and interactive terminals. The memory banks and an optional memory extension are managed by the memory control unit. The main memory has eight interleaved modules, each with a cycle time of 160 ns and a word length of 32 bits. Eight memory words can be transferred in one memory access. The memory control unit is an interface between eight independent processor ports and nine memory buses. Each processor port has full accessibility to all memories.

The central processor can execute both scalar and vector instructions. Figure 4.5 illustrates the functional pipelines in the central processor. The processor includes the instruction processing unit (IPU), the memory buffer unit

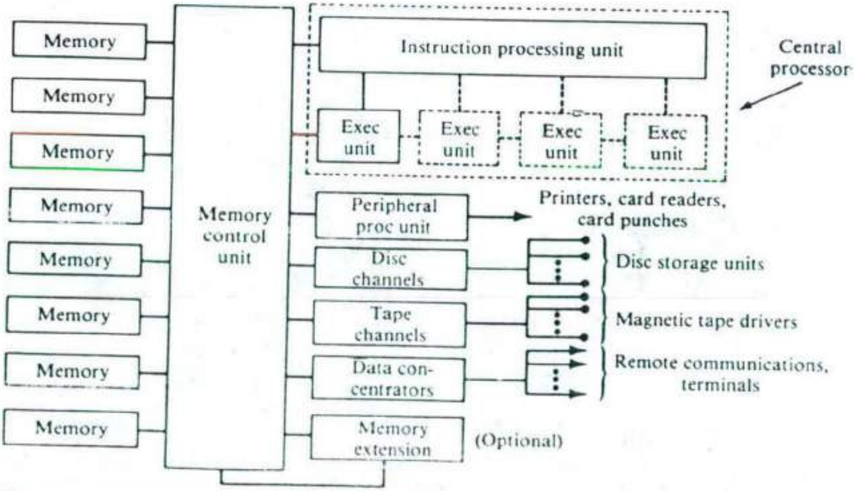


Figure 4.4 Basic Texas Instruments ASC systems configuration.

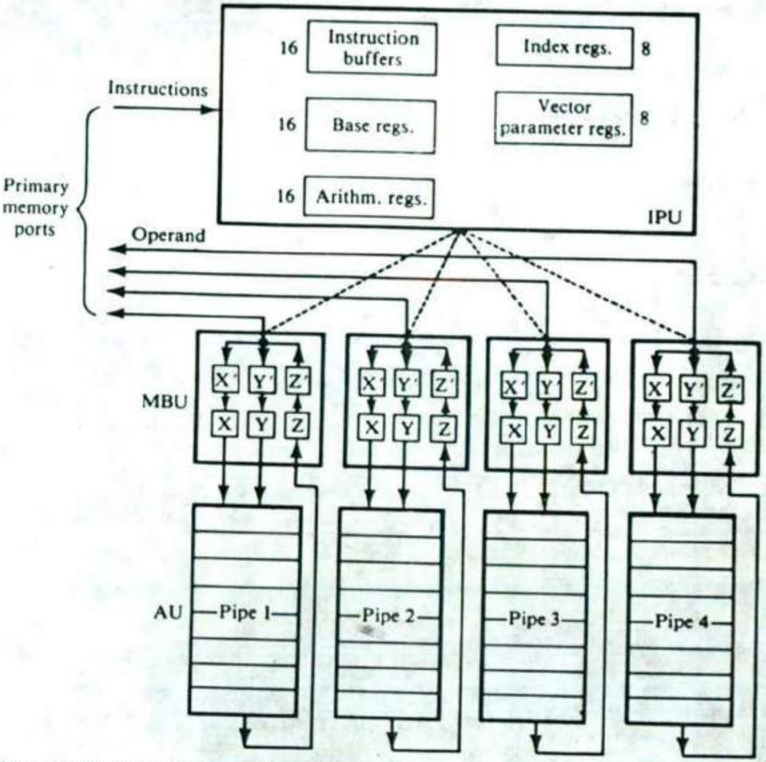


Figure 4.5 Central processor of the TI-ASC with four arithmetic logic pipelines. (Courtesy of Texas Instruments, Inc.)

(MBU), and the arithmetic unit (AU). Up to four arithmetic pipelines (MBU-AU pairs) can be built into the central processor. The ASC instruction types are listed in Table 4.3. The maximum ASC speed per arithmetic pipeline is given in Table 4.4. On the average, only 0.5 to 1.5 megaflops and 3 to 10 megaflops per pipeline can be expected for scalar and vector operations, respectively.

The primary function of the IPU is to supply the rest of the central processor with a continuous stream of instructions. Internally, the IPU is a multisegment pipeline which has 48 program-addressable registers for fetching and decoding instructions and generating the operand address. Instructions are first fetched in *octets* (8 words) from memory into the instruction buffers of 16 registers. Then the IPU performs assignment of instructions to the MBU-AU pairs to achieve optimal use of the arithmetic pipelines. The MBU is an interface between main memory and the arithmetic pipelines. Its primary function is to support the arithmetic units with continuous streams of operands. The MBU has three double buffers, with each buffer having eight registers. *X* and *Y* buffers are used for inputs, and a *Z* buffer is used for output. The fetch and store of data are made in 8-word

Table 4.3 TI-ASC instruction types

Scalar instructions	Vector instructions
Load and Store	Arithmetic
Arithmetic	Logical
Logical	Shift
Shift	Compare
Compare	Merge and Order
Branch	Move
Increment and Test	Search
Stack	Normalize
Normalize	Data conversion
Data type conversion	Peak picking
Monitor call	Select
	Replace

Table 4.4 Maximum floating-point speed of TI-ASC (in mops)

Arithmetic operations	32-bit operands	64-bit operands
Add	16.6	9.5
Multiply	16.6	9.5
Divide	1.1	0.67

Table 4.5 Comparison of major architectural features between Star-100 and TI-ASC

Characteristic	STAR-100	ASC
Data word size	1/8/32/64/128 bits	16/32/64 bits
Instruction size	32/64 bits	32 bits
Memory size	1 M 64-bit words	8 M 64-bit words
Clock rate	40 ns	60 nsec
Functional pipeline unit	2 nonhomogeneous pipelines (1) Add pipe + Multiply pipe in parallel (2) Add pipe + Divide pipe + Multipurpose pipe	1 to 4 homogeneous pipelines: eight exclusive segments in each pipe with bypasses to execute a number of arithmetic functions in FLP or FXP formats
Average speed per pipeline	0.5–1.5 Mflops/s (scalar) 5–10 Mflops/s (vector)	0.5–1.5 Mflops/s (scalar) 3–10 Mflops/s (vector)

increments. The AU has a pipeline structure to enable efficient arithmetic computations. This unit is reconfigurable with variable interconnecting paths among eight segments, as described in Section 3.2.3. Many similarities exist between the Star-100 and the ASC systems. Table 4.5 summarizes the major architectural features of these two early vector processors.

4.2.2 Vector Processing in Streaming Mode

Continuous streaming of data from the high-bandwidth interleaved memories to multiple pipelines makes Star very efficient for the processing of long vectors. The key issue here is to structure the computations into vector mode, such as those frequently done in matrix multiplication, polynomial evaluation, and the solution of large-scale linear systems of equations. The system software of Star provides aids to enable the user to take full advantage of hardware capabilities. The Fortran compiler in Star has been extended to detect loops and vectorize them into simplified vector codes. The lines of Fortran code which can be vectorized must be well isolated and easy to recognize. Of course, the programmer can escape from Fortran code by directly using the mnemonic assembly language to achieve maximal hardware performance.

The vector instruction format of the Star-100 is shown in Figure 4.6. Each instruction has 64 bits divided into eight fields. Fields *F* and *G* specify function

							C+1 (offset for *C*, *Z*)
F (8X, 9X)	*G* (subfunction)	*X* (offset for *A*)	*A* (field length 8: base address)	*Y* (offset for *B*)	*B* (field length 8: base address)	*Z* (CV base address)	*C* (field length 8: base address)

Field	Contents of the field in the cited register
X	The offset of source operand 1
Y	The offset of source operand 2
A	The base address and field length of source operand 1
B	The base address and field length of source operand 2
Z	The base address of the control vector
C	The base address and field length of the result vector
C+1	The offset of the control or result vector

Figure 4.6 The instruction format of Star-100.

and subfunction codes. The rest of the fields designate the working registers to be used. The field $C + 1$ automatically specifies the register holding the offset for control of the result vectors. The effective starting address is calculated as the sum of the base address and the offset. The effective field length is calculated as the offset subtracted from the field length. Thus; the ending address is the sum of the effective starting address and the effective field length. With offset capability, the ith element in the source operand can operate with the $(i + d)$th element of another source operand, where d is the difference between the two offsets. The following example shows the streaming operations for vector addition in the Star-100:

Example 4.1 Consider the execution of a *Vector Add* instruction in the Star-100:

$$\text{VADD A,B,C} \qquad (\text{A}+\text{B}\rightarrow\text{C}) \tag{4.2}$$

where A = field length of A vector = 12 halfwords (32 bits each)
B = field length of B vector = 4 halfwords, base address = 20000_{16}
X = offset for A vector = 4 halfwords
Y = offset for B vector = -4 halfwords
Z = base address of control vector = 40004_{16}
C = base address of result vector = 30000_{16}, field length = 12 halfwords
C + 1 = control vector and result vector offset = 4 halfwords

The starting address and effective field length of the A vector are calculated in Figure 4.7. Note that bit addressing is used and a "1" in the control vector permits storing the corresponding element in the resulting vector. For example, the memory location 40005 is stored with a "1," so C_5 is transformed into $A_5 + B_{-3}$. The skewing effect is apparent in this example. After the instruction has been decoded at the stream unit, the appropriate microcode sequence is initiated by the *microcode unit* (MIC) in the stream unit.

When the CPU initiates an instruction requiring microcode control, it sends the F (function) code and a microcode pulse to the MIC. The MIC then takes over control of the startup and termination of the instruction. In case of interrupts, it has to branch to save all the operands and parameters necessary to resume execution afterward. The MIC is the heart of the vector processing control, consisting of the following sequence of control steps:

1. The reading of addresses from the register file (in the stream unit) for the vector parameters according to designations specified in the instruction
2. The calculation of the effective addresses and field lengths for monitoring the starting of vector operations
3. The setting up of the usage of read-write buses as specified by the G (subfunction) field for the operands and results

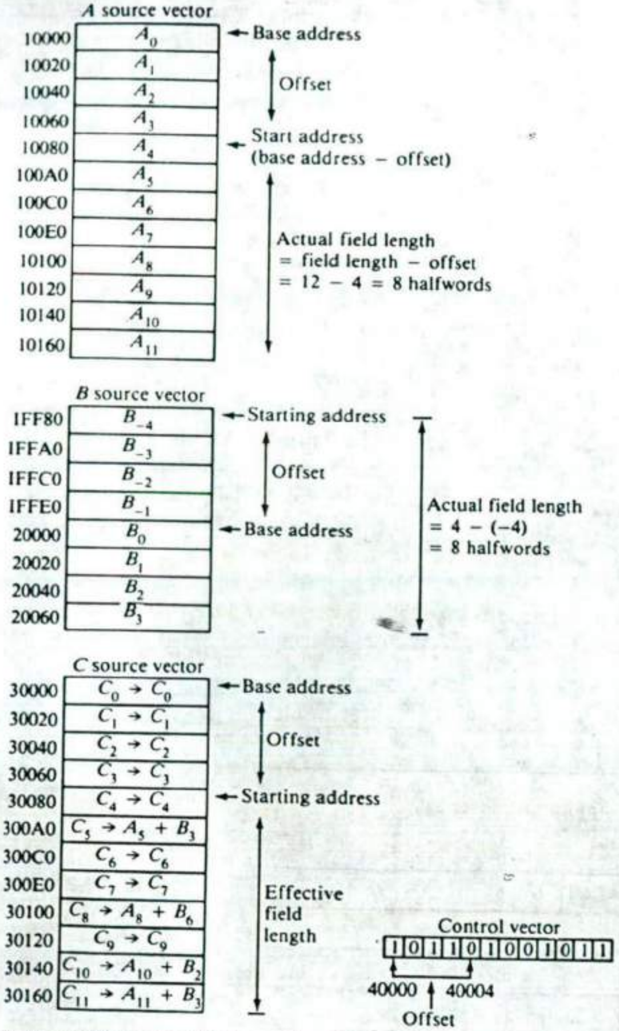


Figure 4.7 The vector ADD instruction of Star-100 in Example 4.1.

4. The transfer of addresses and other information whenever needed to appropriate interrupt-count registers

Once effective addresses are computed, the operand elements are fetched and paired for the operations involved. The static configuration of the execution pipe will remain active until vector instruction is terminated. A termination is marked by either of the following events: (*a*) a vector is exhausted when the effective field length becomes 0; (*b*) some other data fields or strings have been exhausted.

To support vector and scalar processing in Star, its operating system provides time sharing, utilizing the concepts of virtual memory. Prepaging is allowed by a feature known as *advise* to alleviate the I/O-bound problem. The Star operating system handles the functions of input, compilation, assembly, loading, execution, and output of all programs submitted, as well as the allocation of main memory. In addition to Star Fortran, an interactive interpreter called Star APL is also implemented in the system, which upgrades system capability to handle a large area of scientific computations.

Instructions in TI-ASC have 32 bits, as shown in Figure 4.8, where F is the opcode, R, T, and M specify the arithmetic, index, and base registers, and N is the symbolic address. ASC differs from Star in the way vector instructions are implemented. Instead of using certain registers to retrieve the operand addresses and control information, the ASC uses a *vector parameters file* (VPF), which consists of eight 32-bit registers in the IPU, as shown in Figure 4.5. The function of each VPR register is fixed, as shown in Figure 4.8. The register V_0 holds the opcode, the vector-operand type, and the length; V_1, V_2, and V_3 indicate the base address and the displacement of each operand vector; V_4 and V_5 hold the increment of the vector index and the interaction number of inner loops; and V_6 and V_7 hold similar information for outer loops.

The above control information is loaded into these V registers from the main memory before the execution of each vector instruction. Microcode will be

8	4	4	4	12
F	R	T	M	N

(*a*) Instruction format

V_0	OPR		ALCT	SV	L
V_1	—	XA	SAA		
V_2	HS	XB	SAB		
V_3	VI	XC	SAC		
V_4	DA_1				DB_1
V_5	DC_1				N_1
V_6	DA_0				DB_0
V_7	DA_0				N_0

(*b*) Vector parameter file (VPF)

Figure 4.8 The TI-ASC instruction format and vector parameter file.

generated by the timing-sequence circuitry in the MBU to control the entire pipeline processing in the AU. This sequence includes the fetch of operands, the sending of pairs of operands to the pipeline, the execution phase, and the return of successive results. The increment in ASC is variable, while only an increment of 1 is possible in Star. This offers more flexibility in addressing operands. Three-dimensional indexing is also possible in ASC to process the inner loop and outer loop more efficiently. However, ASC does not use control vector, sparse, and macro-vector instructions, as found in Star. Both the Star and ASC are structured to execute memory-to-memory instructions in streaming mode.

4.3 SCIENTIFIC ATTACHED PROCESSORS

Attached processors are becoming popular because their costs are low and yet they provide significant improvement on the host machines. The AP-120B and FPS-164 are back-end attached arithmetic processors specially designed to process large vectors or arrays (matrices) of data. Operationally, these processors must work with a host computer, which can be either a minicomputer (such as the VAX-11 series) or a mainframe computer (IBM 308X series). While the host computer handles the overall system control and supervises I/O and peripheral devices, the attached processor is responsible for heavy floating-point arithmetic computations. Such a functional distribution can result in a 200 times speedup over a minicomputer, and a 20 times speedup over a mainframe computer. Other scientific attached processors include the IBM 3838 and the low-cost Datawest processor. We describe in this section the architectural features of these attached processors and assess their potential applications in the scientific and engineering areas.

4.3.1 The Architecture of AP-120B

The combination of an AP-120B and a host computer is shown in Figure 4.9. All the peripheral devices like printers, display terminals, disk and tape units are attached to the host computer. In fact, the AP-120B is itself a peripheral attachment to the host. Since the host and back-end may have different data formats and even unequal word lengths, an *interface unit* is needed to convert the data "on the fly" and to help implement the *direct-memory access* (DMA) and *programmed input-output* (PIO) data transfers. These are two sets of registers in the interface unit. One set is devoted to control functions via programmed I/O; the other to block data transfers via the DMA. The programmed I/O section of the interface unit provides the array processor with a simulated front panel of the host. It contains a *switches register* used by the host to enter control or parameter data and addresses into the array processor, a *light register* to display contents of registers in the array processor, and a *functional register* for typical front panel commands such as *start*, *stop*, or *reset*.

The DMA register set includes a *host memory address* register, an *AP memory address* register, a *word count* register, a *control* register, and a *format* register. The

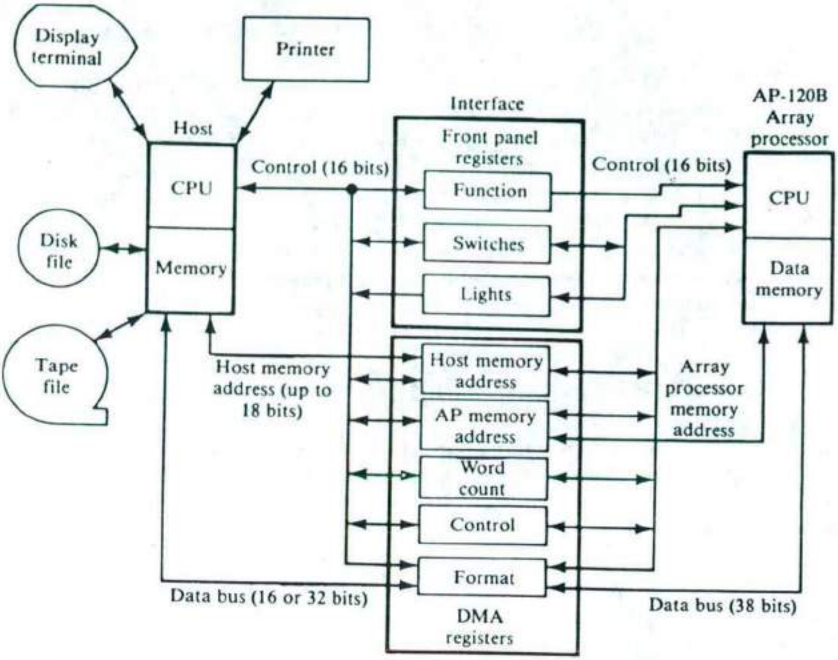


Figure 4.9 The AP-120B host and interface organization.

control register governs the direction of data transfer and the mode of transfer. The format register performs conversion between the FLP format of the host and that of the AP-120B. Interface logic permits data transfer to occur under control of either the host or the AP-120B. The floating-point format in the AP-120B is 38 bits long, with a 28-bit 2's complement mantissa and a 10-bit exponent biased by 512. Using such a format, the precision and dynamic range are improved over the conventional 32-bit floating-point format. If the host has different floating-point data formats, the format conversion is done "on the fly" through the interface. Consequently, the AP-120B can concentrate on useful computational tasks.

A detailed functional diagram of the AP-120B processor is shown in Figure 4.10. The processor is divided into six sections, the *I/O section*, *memory section*, *control memory*, *control unit*, *data bus*, and two *arithmetic units*. The memory section consists of the *data memory* (MD), *table memory* (TM), and two *data pads* (DPX and DPY). The control memory or *program memory* (PM) has 64-bit words with a 50-ns cycle time. The program memory consists of up to 4K words in 256-word increments. Instructions residing in the PM are fetched, decoded, and executed in the control unit. The data memory is interleaved with a cycle time of either 167 or 333 ns. The choice of a particular speed depends on the trade-off between cost

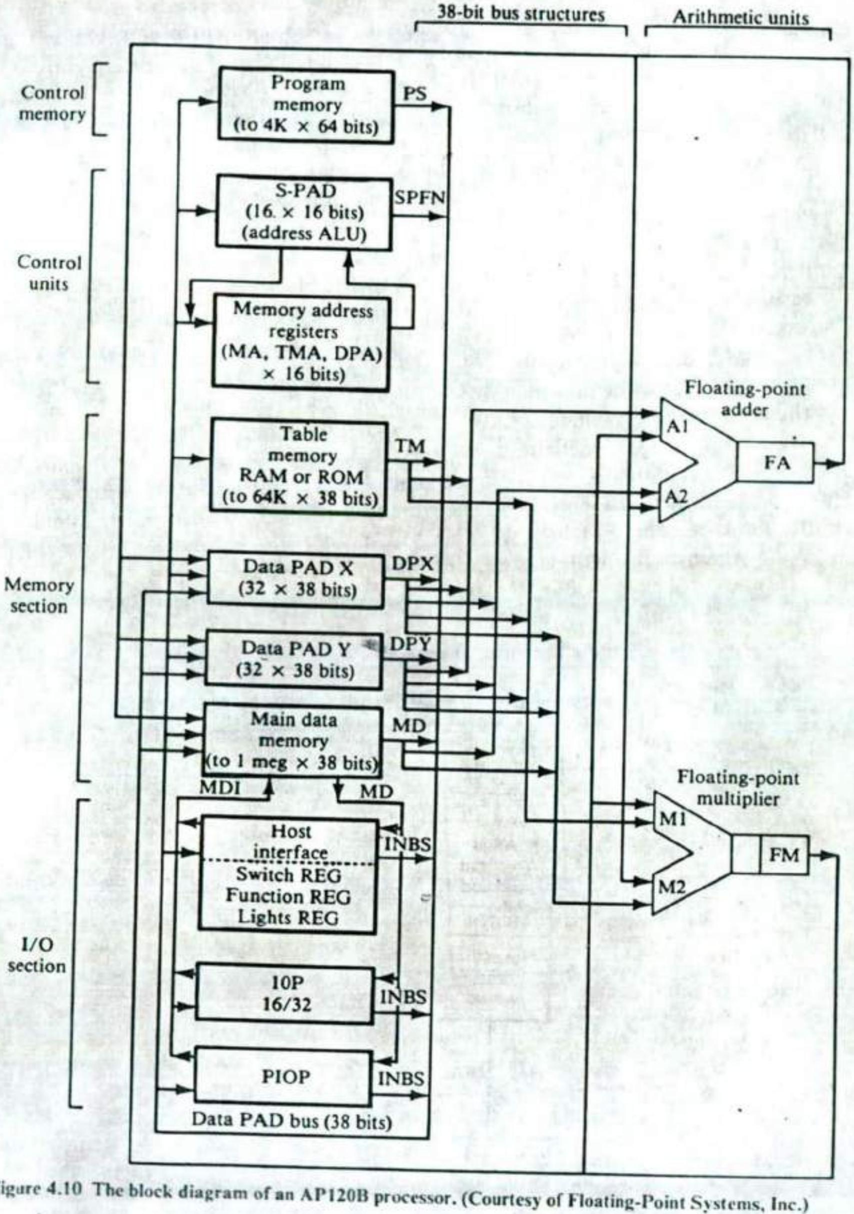

38-bit bus structures
Arithmetic units
Control memory
Program memory (to 4K × 64 bits)
PS
Control units
S-PAD (16 × 16 bits) (address ALU)
SPFN
Memory address registers (MA, TMA, DPA) × 16 bits)
Memory section
Table memory RAM or ROM (to 64K × 38 bits)
TM
Data PAD X (32 × 38 bits)
DPX
Data PAD Y (32 × 38 bits)
DPY
Main data memory (to 1 meg × 38 bits)
MD
MDI
MD
I/O section
Host interface
Switch REG
Function REG
Lights REG
INBS
10P 16/32
INBS
PIOP
INBS
Data PAD bus (38 bits)
Floating-point adder
A1
A2
FA
Floating-point multiplier
M1
M2
FM

Figure 4.10 The block diagram of an AP120B processor. (Courtesy of Floating-Point Systems, Inc.)

and performance. The data memory is the main data storage unit with 38-bit words. It is directly addressable by the 1 million words in 2K-word (167 ns) or 8K-word (333 ns) increments. The TM has up to 64K 38-bit ROM or RAM words of 167-ns cycle time. The table memory is used for the storage of frequently used constants (e.g., FFT constants). It is associated with a special data path which does not interfere with the data path associated with the data memory. The data pads X and Y are two blocks of 38-bit accumulators. There are 16 accumulators in each block. These accumulators are directly addressable by the AP processor. Any accumulator can be accessed in a single machine cycle of 167 ns. Simultaneous *read* and *write* are possible in each data pad within the same cycle.

The S pad in the control unit contains two parts: an *S-pad memory* and an *integer ALU*. The S-pad memory contains 16 directly addressable integer registers. These registers feed the address ALU to produce effective operand addresses. The address ALU performs 16-bit integer arithmetic. The outputs of the address ALU can be routed to any one of the following address registers: MA for the data memory, TMA for the table memory, and DPA for the data pads. Other functions of the address ALU include *clear*, *increment*, *decrement*, *logical and*, and *logical or*.

Two pipeline arithmetic units are the *FLP adder* (FA) and the *FLP multiplier* (FM). The FA consists of two input registers, A1 and A2, and a two-segment pipeline, as shown in Figure 4.11. The sum output is a 38-bit normalized floating-point number. The FM has M1 and M2 input registers and a three-segment pipeline which performs floating-point multiply operations. Once the pipeline is

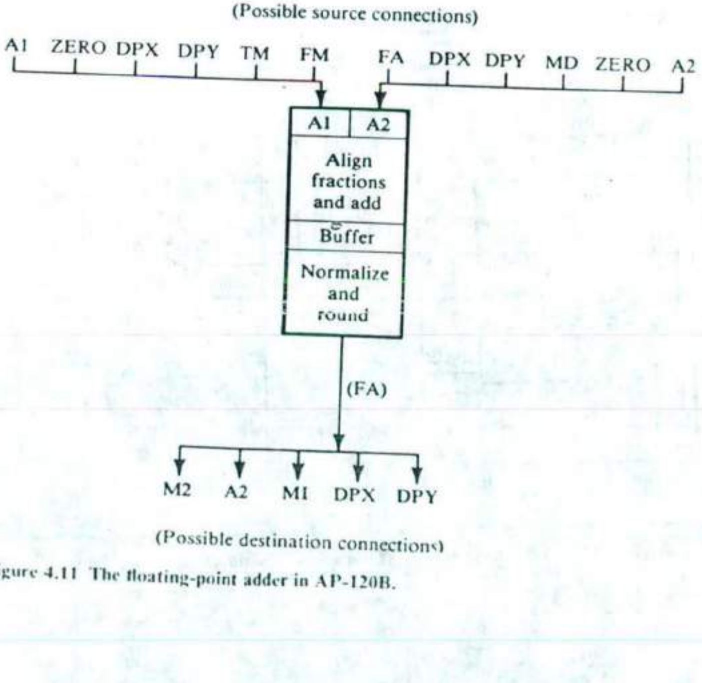


Figure 4.11 The floating-point adder in AP-120B.

full, a new result (sum or product) is produced for every machine cycle of 167 ns. Consequently, the maximum throughput rate for the AP-120B is 12 mega floating-point computations per second.

The AP-120B derives its high computing power from multiplicities in all sections of its processor organization. It uses two pipeline arithmetic units (FA and FM), one integer ALU, multiple memories (PM, MD, TM) which can be independently addressed, a large number of registers and accumulators (A1, A2, M1, M2, MA, TMA, DPA, DPXs, and DPYs), and seven data paths, as shown in the bus structures section of Figure 4.10.

The two floating-point arithmetic units FA and FM can operate simultaneously and the 16-bit integer ALU can operate independently of the FA and FM. The use of two independent blocks of accumulators (DPX and DPY) provides the desired flexibility in handling operands and intermediate and final results. For instance, each block can hold a vector operand with 16 components so that a 16-element dot product can be performed within the FA and the FM in pipeline mode. In other cases, one block provides data for the FA or FM, while the other block transfers data to and from data memory or table memory.

The pipeline structures of the FA and FM are described below. The first stage of the FA compares exponents, shifts the fraction of the smaller number, and adds the fractions. In the second stage, the resulting fraction is normalized and rounded. Because of different processing speeds in the two stages, a buffer is inserted to hold the intermediate result. The output of the FLP adder, denoted by FA, can be routed to five different destinations. Possible source connections to the input registers A1 and A2 are shown at the top of Figure 4.11. The FM has three stages. In the first stage, the 56-bit product of the two 28-bit fractions is partially completed. The second stage completes the product of the fractions. The third stage adds the exponents, rounds, and normalizes the fraction of the product. All possible source and destination connections to the FLP multiplier are identified in Figure 4.12.

Seven buses are used in the AP-120B simultaneously to enable parallel processing. Both the FA and the FM have multibus input ports. In other words, multiple operands and results can be moved between different functional units at the same machine cycle. Thereby, the total data path bandwidth will match the execution speed of the pipeline adder and multiplier.

Several levels of parallelism in the AP-120B have been described. Another aspect worthy of mentioning is the control of parallel functional units. This is provided by the long instruction word of the AP-120B. An AP-120B instruction has 64 bits, which are subdivided into 10 command fields (Figure 4.13). Each command field controls a specific unit; therefore, a single AP-120B instruction can initiate as many as ten operations per machine cycle, as listed in Figure. 4.13. Multiple memory accesses, register transfers, integer arithmetic, and floating-point computations can occur at the same time.

In summary, multiple memories, multiple functional units, parallel data paths, and the multiple command fields in the instruction have made the AP-120B a fast attached processor for scientific computations.

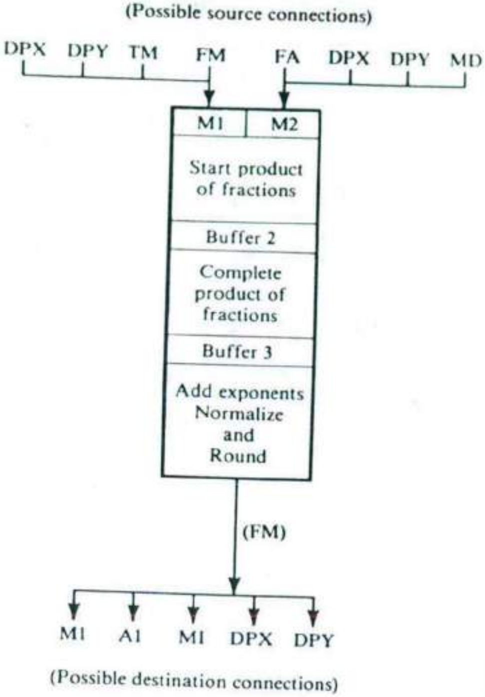

(Possible source connections)
DPX
DPY
TM
FM
FA
DPX
DPY
MD
M1
M2
Start product of fractions
Buffer 2
Complete product of fractions
Buffer 3
Add exponents Normalize and Round
(FM)
M1
A1
M1
DPX
DPY
(Possible destination connections)

Figure 4.12 The floating-point multiplier in AP-120B.

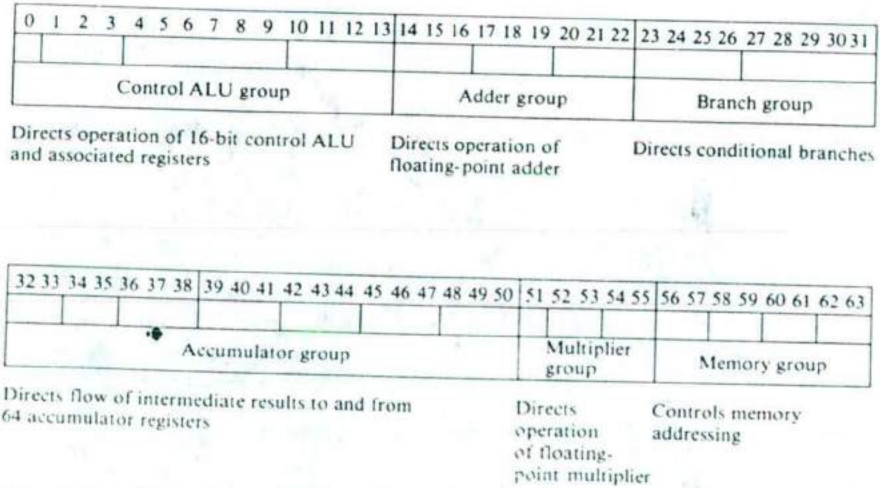

0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31
Control ALU group
Adder group
Branch group
Directs operation of 16-bit control ALU and associated registers
Directs operation of floating-point adder
Directs conditional branches
32 33 34 35 36 37 38
39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55
56 57 58 59 60 61 62 63
Accumulator group
Multiplier group
Memory group
Directs flow of intermediate results to and from 64 accumulator registers
Directs operation of floating-point multiplier
Controls memory addressing

Figure 4.13 The instruction format in AP-120B.

4.3.2 Back-End Vector Computations

The AP-120B, unlike vector supercomputers, does not have vector instructions. Instead, long instructions containing many concurrent microoperations are used to specify the parallel activities. More than 200 application software packages have been developed for complicated vector computations. These routines are called a *mathematical library*, which is devoted to mainly vector and matrix manipulations. These vector-processing routines are Fortran callable from the host computer. All calls are handled by the *array processor executive* (APEX) software, which decodes the subroutine calls from Fortran programs residing in the host and automatically passes control parameters and routines to the AP-120B for execution. After completing the computation of a routine, the AP-120B returns the results to the host computer for further use. The user can add new or special routines to the mathematics library in Fortran or in AP assembly language code. Some program development software have been provided for such purposes. In addition, there is the *signal processing library*, dedicated to digital signal-processing applications. Important Fortran callable routines in these libraries are summarized in Table 4.6, with the time measured in microseconds and the program sizes in numbers of AP-120B microinstruction words.

In the AP-120B, different functions can be performed by the floating-point adder at different times. Listed below are some typical functions:

1. A1 + A2
2. A1 − A2
3. A2 − A1
4. A1 EQV A2
5. A1 AND A2
6. A1 OR A2
7. Convert A2 from signed magnitude to 2's complement format
8. Convert A2 from 2's complement to signed magnitude format
9. Scale A2
10. Absolute value of A2
11. Fix A2

Table 4.6 Floating-point arithmetic timing for some functions in AP-120B

Operation	Travel time	Pipeline interval
Add-subtract	333 ns	167 ns
Multiply	500 ns	167 ns
Multiply-add	833 ns	167 ns
Complex add-subtract	500 ns	333 ns
Complex multiply	1,333 ns	667 ns
Complex multiply-add	1,667 ns	667 ns

Table 4.7 Important FORTRAN callable routines for AP-120B

Operation	Name	Timing (μs per point)	Size (AP-120B prog. words)
Real vector operations			
Vector add	VADD	1.2	8
Vector subtract	VSUB	1.2	8
Vector multiply	VMUL	1.2	11
Vector divide	VDIV	1.8	44
Vector exponential	VEXP	5.1	42
Vector sine	VSIN	5.1	46
Vector cosine	VCOS	5.6	46
Sum of vector squares	SVR	0.4	11
Dot product of two vectors	DOTPR	0.8	9
Sum of vector elements	SVE	0.4	7
Complex vector operations			
Complex vector multiply	CVMUL	2.0	26
Complex vector reciprocal	CVRCIP	5.0	51
Matrix operations			
Matrix transpose	MTRANS	0.8	17
Matrix multiply	MMUL	•	58
Matrix multiply (dimension ≤ 32)	MMUL32	•	27
Matrix inverse	MATINV	•	130
Matrix vector multiply (3×3)	MVML3	2.5/vector	30
Matrix vector multiply (4×4)	MVML4	4.6/vector	39
Fast fourier transform operations			
Complex FFT	CFFT	•	187
Real FFT	RFFT	•	235
Signal processing operations			
Convolution (or correlation)	CONV	•	102
Wiener-Levinson algorithm	WIENER	•	68
Bandpass filter	BNDPS	•	287
Power spectrum	PWRSPC	•	268

• Timing unknown.

Similarly, the FM can perform many different functions. The timing for some floating-point arithmetic operations in the AP-120B is summarized in Table 4.7, where the *travel time* is the total time required to transfer data from source to destination, and the *pipeline interval* is the time between successively available results. The pipeline interval indicates the maximum throughput rate for vector-oriented computations.

A detailed example of vector processing in the AP-120B is given below. First some notations are established. A semicolon " ; " separates parallel operations within an instruction word. A comma " , " is used to separate operands. A double slash bar "//" denotes a comment. An arrow "←" refers to the replacement operator for data transfers. Some operations required in presenting the example are specified below:

```
FADD  A1,A2         //A1 + A2 (floating-point add)
DPX(m)←FA           //Save FA in location m of data pad DPX.
FMUF   M1,M2        //M1 × M2 (floating-point multiply)
DPY(m)←FM           //Save FM in location m of data pad DPY.
```

where the inputs A1, A2, M1, and M3 to the adder and multiplier come from the input sources specified in Figures 4.11 and 4.12, respectively.

Example 4.2 The following sequence is used to compute the dot product of two vectors, $\sum_{i=0}^{4} X_i Y_i$, where X_i and Y_i are obtained from DPX and DPY, respectively. The resulting sum of the products is to be stored in DPX:

(1) FMUL DPX(0), DPY(0)	//Multiply X_0Y_0.
(2) FMUL DPX(1), DPY(1)	//Multiply X_1Y_1.
(3) FMUL DPX(2), DPY(2)	//Multiply X_2Y_2.
(4) FMUL DPX(3), DPY(3); FADD FM, ZERO	//Multiply X_3Y_3, X_0Y_0 is now done. Save it in adder.
(5) FMUL DPX(4), DPY(4); FADD FM, ZERO	//Multiply X_4Y_4, X_1Y_1 is now done. Save it in adder.
(6) FMUL; FADD FM, FA	//X_2Y_2 is coming out of the multiplier and X_0Y_0 from the adder. Add them together.
(7) FMUL; FADD FM, FA	//X_3Y_3 is coming out of the multiplier and X_1Y_1 from the adder. Add them together.
(8) FADD FM, FA	//X_4Y_4 is coming out of the multiplier and $(X_0Y_0+X_2Y_2)$ from the adder. Add them together.
(9) FADD; DPX(4)←FA	//$(X_1Y_1+X_3Y_3)$ is coming out of the adder. Save it in DPX(4).
(10) FADD DPX(4), FA	//$(X_0Y_0+X_2Y_2+X_4Y_4)$ is coming out of the adder. Add it to $(X_1Y_1+X_3Y_3)$.
(11) FADD	//Push result out of adder pipeline.
(12) DPX(4)←FA	//The result $\sum_{i=0}^{4} X_iY_i$ is stored in DPX(4).

In the above sequence of computations, cycles 1 to 3 are used to fill the FM pipeline; cycles 4 to 5 to fill the FA pipeline; cycles 6 to 8 to drain the FM pipeline; cycles 9 to 11 to drain the FA pipeline, and the final result is stored in data pad X.

The dummy add "FADD" without arguments in cycle 11 is used only to push the last computation out of the pipeline. Remember that there are three stages and two buffer registers in the FM pipeline, hence two dummy multiplies are needed to push the last two computations out of the pipeline. For long vectors, the speed to execute dot product in the AP-120B is much faster than in a serial processor.

The AP-120B has been applied extensively in the field of digital-signal processing. The execution sequence of *fast Fourier transform* (FFT) in the AP-120B is shown below as an example. The FFT program resides in the program memory of the AP-120B. The array of data to be transformed is stored in the main memory of the host computer. The FFT computation sequence consists of the following steps:

1. The host computer issues an I/O instruction to initiate the FFT program in the AP-120B.
2. The AP-120B requests host DMA cycles to transfer the array of data from host memory to data memory in the AP-120B. The floating-point format is converted during the flow of data through the interface unit.
3. The FFT computations are performed over a 38-bit floating-point data array.
4. The AP-120B requests the host DMA cycles to return the results of the FFT frequency-domain coefficients array.

Example 4.3 The above operations are called by a host machine with the following four Fortran statements:

```
CALL APCLR              //Clear AP-120B.
CALL APPUT (······)     //Transfer data to AP-120B.
CALL CFFT (······)      //Perform FFT.
CALL APGET (······)     //Transfer results to host.
```

where "……" denotes the parameters used in the routines.

For the convolution of two arrays, say A and B, all required operations can also be done by the AP-120B. Once the transfer of data arrays is initiated, there is no need to wait until completion of the entire array transfer. Such convolution requires a sequence of forward FFT and inverse FFT operations, as listed below:

1. Transfer arrays A and B to AP-120B.
2. Perform FFT on A array.
3. Perform FFT on B array.
4. Multiply the results of steps 2 and 3.
5. Perform inverse FFT of the result obtained from step 4.
6. Return the final result to the host computer.

4.3.3 FPS-164, IBM 3838 and Datawest MATP

The FPS-164 is evolved from the proven architecture of its predecessor products, the AP-120B, the AP-190L, and the FPS-100 by Floating-Point Systems, Inc. It is attached to either the input-output channel or the DMA channel of a host computer by means of a hardware and software interface similar to that for the AP-120B. The host machine can be a DEC VAX 11/780, an IBM 4341, or an IBM 3081, ranging from superminis to large mainframes. The FPS-164 improves its performance over the AP-120B by extended precision (64-bit floating-point numbers instead of 38 bits, as in the AP-120B) and a much enlarged memory of 16 million 64-bit words. The FPS-164 can be programmed with either a Fortran-77 subset, FPS-164 symbolic assembly language, or the extensive library of preprogrammed mathematics, matrix, and applications routines.

A functional block diagram of the FPS-164 is given in Figure 4.14. There are eight independent pipeline functional units (the FLP multiplier, the FLP adder, the data pads X and Y, table memory, main memory, integer ALU, and the data pad bus) interconnected by seven dedicated data paths. The peak speed is still 12 megaflops. The 64-bit data word provides 15 decimal-digit accuracy. The 64-bit address space covers 16 million words. Multi-user protection is provided by using memory base and limit registers and privileged instructions. The vectored priority interrupts allow real-time applications. The dynamic range and accuracy of the FPS-164 improves significantly over the AP-120B. Furthermore, the processor has instructions which assist software implementation of double-word floating-point arithmetic. Diagnostic and reliability features are also built into the FPS-164 to enhance dependability of the system in case of hardware or software failures.

The IBM 3838 is a multiple-pipeline scientific processor. It is evolved from the earlier IBM 2938 array processor. Both processors are specially designed to attach to IBM mainframes, like the System/370, for enhancing the vector-processing capability of the host machines. These attached pipeline processors reflect recent progress in scientific processing at IBM beyond the level of the 360/91 and the 370/195. Vector instructions that can be executed in the 3838 include the componentwise *vector add*, *vector multiply*, the *inner product*, the *sum of vector components*, *convolving multiply*, *vector move*, *vector format conversion*, *fast Fourier transforms*, *table interpolations*, *vector trigonometric* and *transcendental* functions, *polynomial* evaluation, and *matrix* operations. Like the AP-120B and the FPS-164, both the IBM 2938 and the 3838 are microprogrammed pipeline processors which can be supplied with custom-ordered instruction sets for specific vector applications.

The hardware architecture of the IBM 3838 array processor is shown in Figure 4.15. The processor can attach to a System/370 via a block-multiplexer I/O channel with a data transfer rate of 1.5 M bytes per second. With an optional two-type interface, the maximum data-transfer rate can be doubled to 3 M bytes/s. The 3838 appears to the host processor I/O channel as a shared control unit. Up to seven users can be simultaneously active in the 3838. The tasks defined by each user are pipelined at various subsystems in the 3838. The control processor can assist the user with a set of scalar instructions and the necessary registers in preparing vector instructions. The *bulk memory* is used to hold a large volume of vector operands. The *I/O unit* supervises the transfer of data or programs between

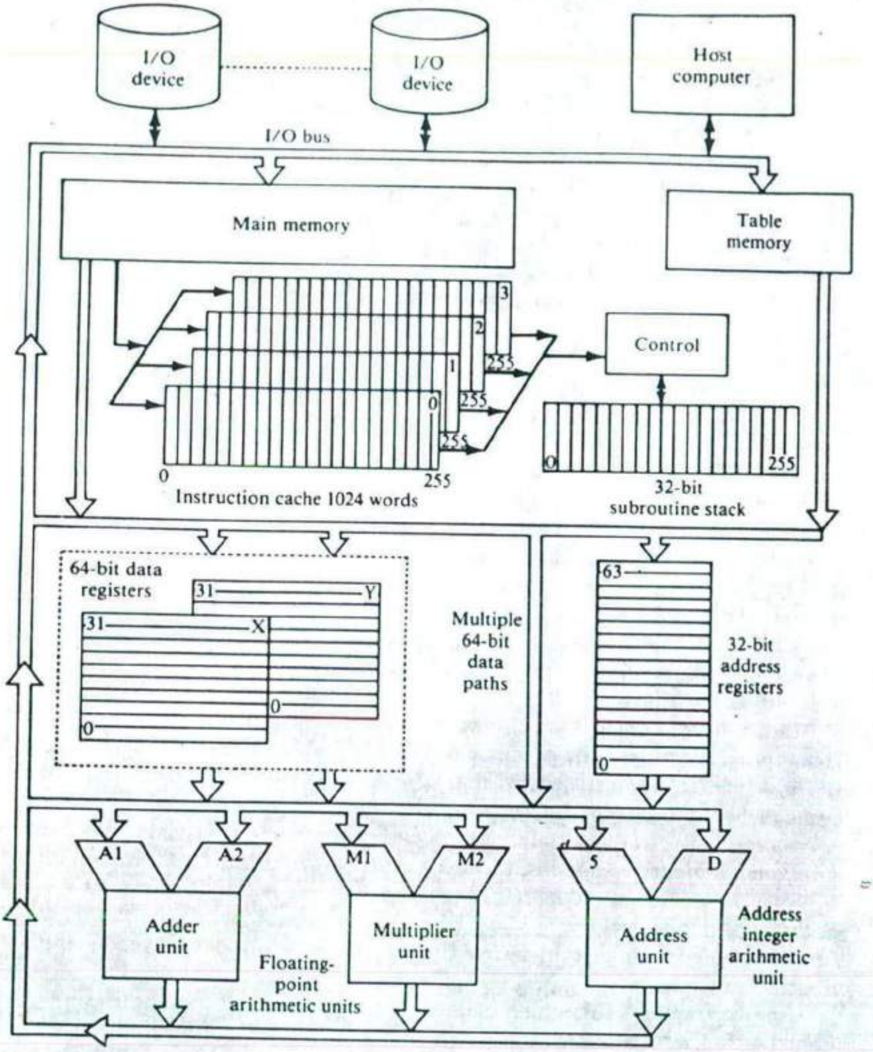


Figure 4.14 The FPS-164 system diagram. (Courtesy of Floating-Point Systems, Inc.)

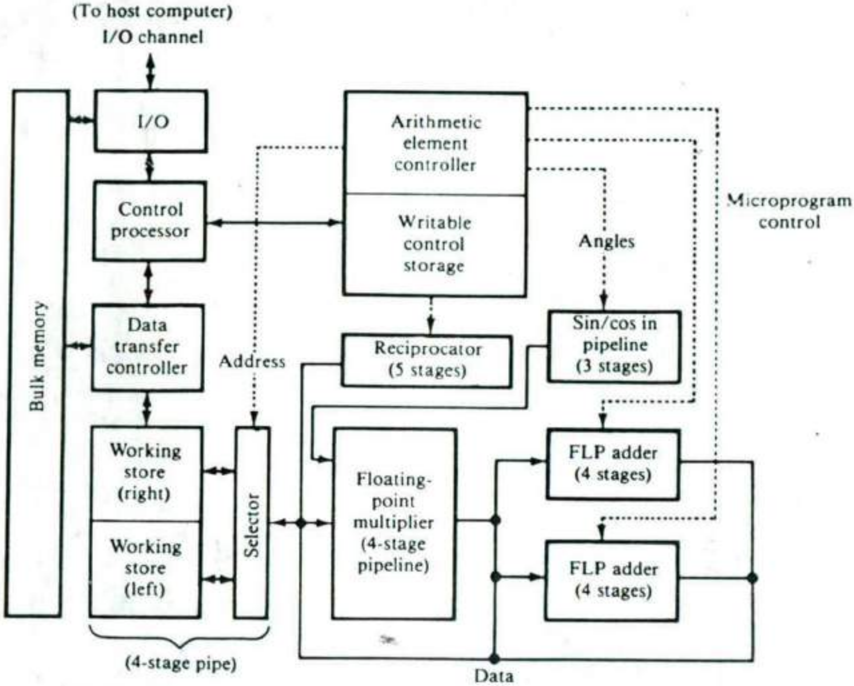


Figure 4.15 The arithmetic processor in IBM/3838. (Courtesy of International Business Machines Corp.)

the host and the bulk memory. Data-word size of the 3838 is 32 bits, matching that of the System/370 machines.

The transfer of the working sets of the vector segments between the bulk memory and the *working stores* is supervised by the *data transfer controller* (DTC). Each working store can hold 8192 bytes. Vector-addressing parameters are supplied to the DTC by the control processor. This DTC is microprogrammed to generate the effective memory addresses for both the bulk and working memories before data can be properly transferred. Furthermore, the DTC can perform data-format conversion during the data flow. The *arithmetic controller* is also a microprogrammed unit. The microprogram sequences performed by the arithmetic pipelines are initialized by this controller. The use of the working stores by the arithmetic pipelines and by the DTC is synchronized. The basic pipeline cycle time is 100 ns in the 3838.

There are five pipeline arithmetic units in the 3838. The pipeline units as diagrammed in Figure 4.15 include two floating-point adders of four stages each; a four-stage floating-point multiplier; a three-stage sine/cosine pipeline; and a five-stage reciprocal estimator. Even the working stores appear as a four-stage pipeline. The delay of each stage is 100 ns. The interconnection paths between these functional pipes are under the microprogrammed control of the *arithmetic* *element controller*. The access of the writable control storage is also pipelined with two stage delays.

The programs and data to be processed by the 3838 are prepared by the host computer. Both vector and scalar instructions can be contained in these 3838 programs. The host sends the programs and data to the 3838 through the I/O channel. Data will be stored in the bulk store. The instructions will be executed by the control processor. After the decoding of each instruction, the control processor provides linked lists of microprogram sequences for supervising the pipelined execution of the instructions. While the arithmetic pipelines are updating vector data from one working store, the DTC can load the other working store. Therefore, data loading and instruction execution can be done simultaneously at the two banks of the working stores. This facilitates the multiprogrammed use of the 3838. Concurrent pipelinings allow multiple users to share the hardware resources in achieving high system throughput. The maximum speed of the 3838 has been estimated to be 30 megaflops.

Datawest, Inc. at Scottsdale, Arizona, has built a very sophisticated attached processor called MATP for large scientific computations. The MATP consists of up to four pipeline processors. These processors, forming a hybrid MIMD-SIMD system, are microprogrammable and share a common data memory. Each processor can be controlled by separate writable control stores. The primary means of host communication is through a set of program channels that connect to host I/O channels.

A schematic functional block diagram of the Datawest MATP is shown in Figure 4.16. This processor is designed to work with a Univac 1184 computer. Using a Univac and an MATP at a cost of $4 million, Datawest claims that it can attain a peak rate of 120 megaflops. This compares favorably with the 160-megaflop Cray-1 with a $10 million cost. The Fujitsu FACOM 230/75 is another attached array processor with a peak performance of 22 megaflops when attached to a FACOM 200M mainframe.

A comparison of three competing attached processors manufactured in the United States is given in Table 4.8. All three processors, FPS' AP-120B, IBM's 3838, and Datawest's MATP, are pipelined and microprogrammable. The speeds shown are theoretical peak speeds in megaflops. The speed of the MATP corresponds to a maximum configuration of four processors. It is interesting to note the multiprocessor structure in the MATP. This concept of pipelining in a multiprocessing mode is also seen in other supercomputers like the Cray X-MP and HEP to be introduced in Chapter 9.

Attached array processors are effective in seismic-signal processing. If one enlarges the instruction repertoire of array processors, they can be turned into general-purpose scientific processors. The attempt by Datawest is a good example. Most scientific computers remain outside the mainstream of developing large computers for business use. The peak speed shows only a theoretical limit. It is the degree to which parallelism is exploited in the application programs that determines the effectiveness of a scientific processor. In general, attached processors have specialized architectures that appeal better to programs containing many

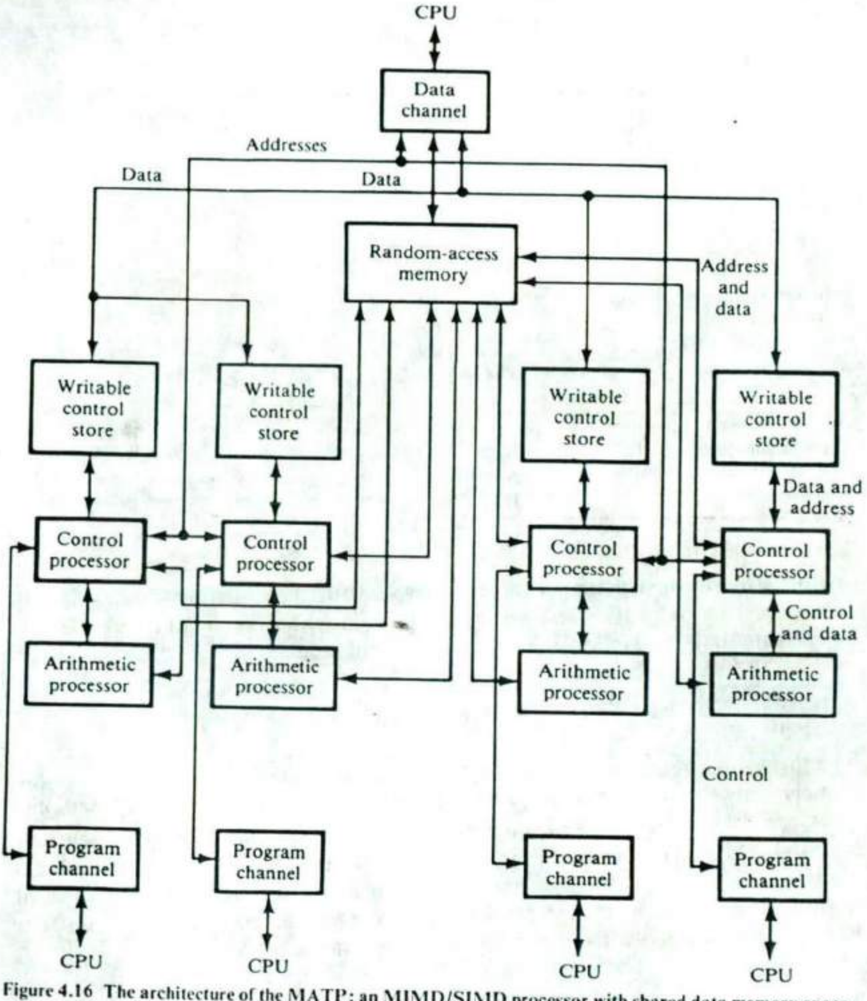


Figure 4.16 The architecture of the MATP: an MIMD/SIMD processor with shared data memory space. (Courtesy Datawest, Inc.)

Table 4.8 Comparison of attached processor capabilities

Features	AP-120B	IBM 3838	MATP
Data word size	38 bits	32bits	32 bits
Processor	Pipelined	Pipelined	Pipelined
Number of controllers	1	1	1 to 4
Number of processors	1	1	1 to 4
Memory size	32K to 1M bytes	2 16K-byte sections	65K bytes
Clock rate	167 ns	100 ns	100 ns
Microprogrammed	Yes	Yes	Yes
Writable control store	Yes	Manufacturer only	Yes
Architecture	Pipeline	Pipeline	Pipeline MIMD/SIMD
Maximum speed (in megaflops)	12	30	120
Add	6	20	2 × 40
Multiply	6	10	40

vector or matrix instructions with regularly structured parallelism. Programs with arbitrary scalar operations may not be suitable for execution in attached scientific processors that are available in the computer market.

4.4 RECENT VECTOR PROCESSORS

The three most recently developed vector processors are described in this section, namely Cray Research's Cray-1, Control Data's Cyber-205, and Fujitsu's VP-200. All three are commercial supercomputers with multiple pipelines for concurrent scalar and vector processing. Possible extensions to these vector supercomputers will be elaborated at the end. We focus on the architectural structures, special hardware functions, software supports, and parallel processing techniques that have been developed with these second generation vector processors.

4.4.1 The Architecture of Cray-1

The Cray-1 has been available as the first modern vector processor since 1976. The architecture of Cray-1 consists of a number of working registers, large instruction buffers and data buffers, and 12 functional pipeline units. With the "chaining" of pipeline units, interim results are used immediately once they become available. The clock rate in the Cray-1 is 12.5 ns. The Cray-1 is not a "stand-alone" computer. A front-end host computer is required to serve as the system manager. A Data General Eclipse computer or a Cray Research "A" processor has been used as the front end, which is connected to the Cray-1 CPU via I/O channels. Figure 4.17 shows the front-end system interface and the Cray-1 memory and functional sections. The CPU contains a *computation section*, *a memory section*, and an *I/O section*. Twenty-four I/O channels are connected to the front-end computer, the I/O stations, peripheral equipment, the mass-storage subsystem, and a *maintenance control unit* (MCN). The front-end system will collect data, present it to the Cray-1 fo; processing, and receive output from the Cray-1 for distribution to slower devices. Table 4.9 summarizes the key characteristics of the three sections in the CPU of the Cray-1.

The memory section in the Cray-1 computer is organized in 8 or 16 banks with 72 modules per bank. Bipolar RAMs are used in the main memory with, at most, one million words of 72 bits each. Each memory module contributes 1 bit of a 72-bit word, out of which 8 bits are parity checks for *single error correction and double error detection* (SECDED). The actual data word has only 64 bits. Sixteen-way interleaving is constructed for fast memory access with small bank conflicts. The bipolar memory has a cycle time of 50 ns (four clock periods). The transfer of information from this large bipolar memory to the computation section can be done in one, two, or four words per clock period. With a memory cycle of 50 ns, the memory bandwidth is 320 million words/s, or 80 million words per clock period.

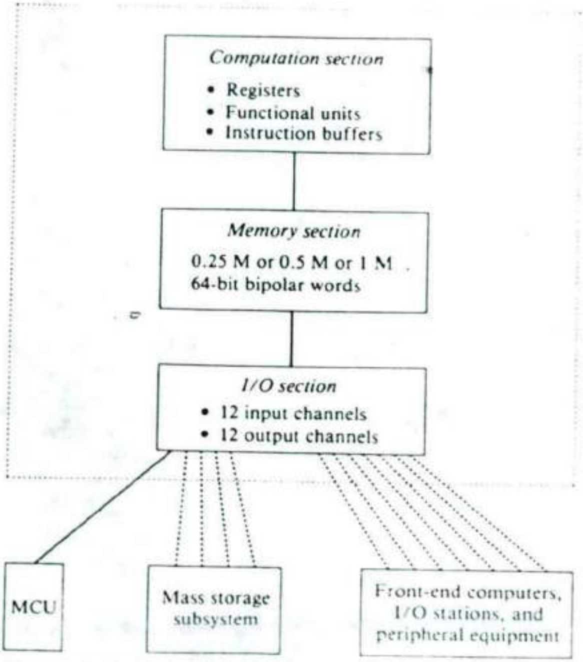


Figure 4.17 Front-end system interface and Cray-1 connections.

Table 4.9 Characteristic of the Cray-1 computer system

Computation section

64-bit word length
12.5-ns clock period
2's complement arithmetic
Scalar and vector processing modes
Twelve fully segmented functional units
Eight 24-bit address (A) registers
Sixty-four 24-bit intermediate address (B) registers
Eight 64-bit scalar (S) registers
Sixty-four 64-bit intermediate scalar (T) registers
Eight 64-element vector (V) registers, 64-bits per element
Four instruction buffers of 64 16-bit parcels each
Integer and floating-point arithmetic
128 instruction codes

Memory section

Up to 1,048,576 words of bipolar memory
(64 data bits and eight error correction bits)
8 or 16 banks of 65,536 words each
Four-clock-period bank cycle time
One word per two clock periods transfer rate to A and S registers
Four words per clock period transfer rate to instruction buffers
Single error correction and double error detection (SECDED)

Input-output section

Twelve input channels and twelve output channels
Channel groups contain either six input or six output channels
Channel groups served equally by memory (scanned every four clock periods)
Channel priority resolved within channel groups
Sixteen data bits, three control bits per channel, and four parity bits
Lost data detection

Such high-speed data-transfer rates are necessary to match the high processing bandwidth of the functional pipelines.

The I/O section contains 12 input and 12 output channels. Each channel has a maximum transfer rate of 80 M bytes/s. The channels are grouped into six input or six output channel groups and are served equally by all memory banks. At most, one 64-bit word can be transferred per channel during each clock period. Four input channels or four output channels operate simultaneously to achieve the maximum transfer of instructions to the computation section. The MCU in Figure 4.17 handles system initiation and monitors system performance. The mass storage subsystem provides large secondary storage in addition to the one million bipolar main memory words.

A functional block diagram of the computation section is shown in Figure 4.18. It contains 64 × 4 instruction buffers and over 800 registers for various purposes. The 12 functional units are all pipelines with one to seven clock delays except for the reciprocal unit, which has a delay of 14 clock periods. Arithmetic operations include 24-bit integer and 64-bit floating-point computations. Large

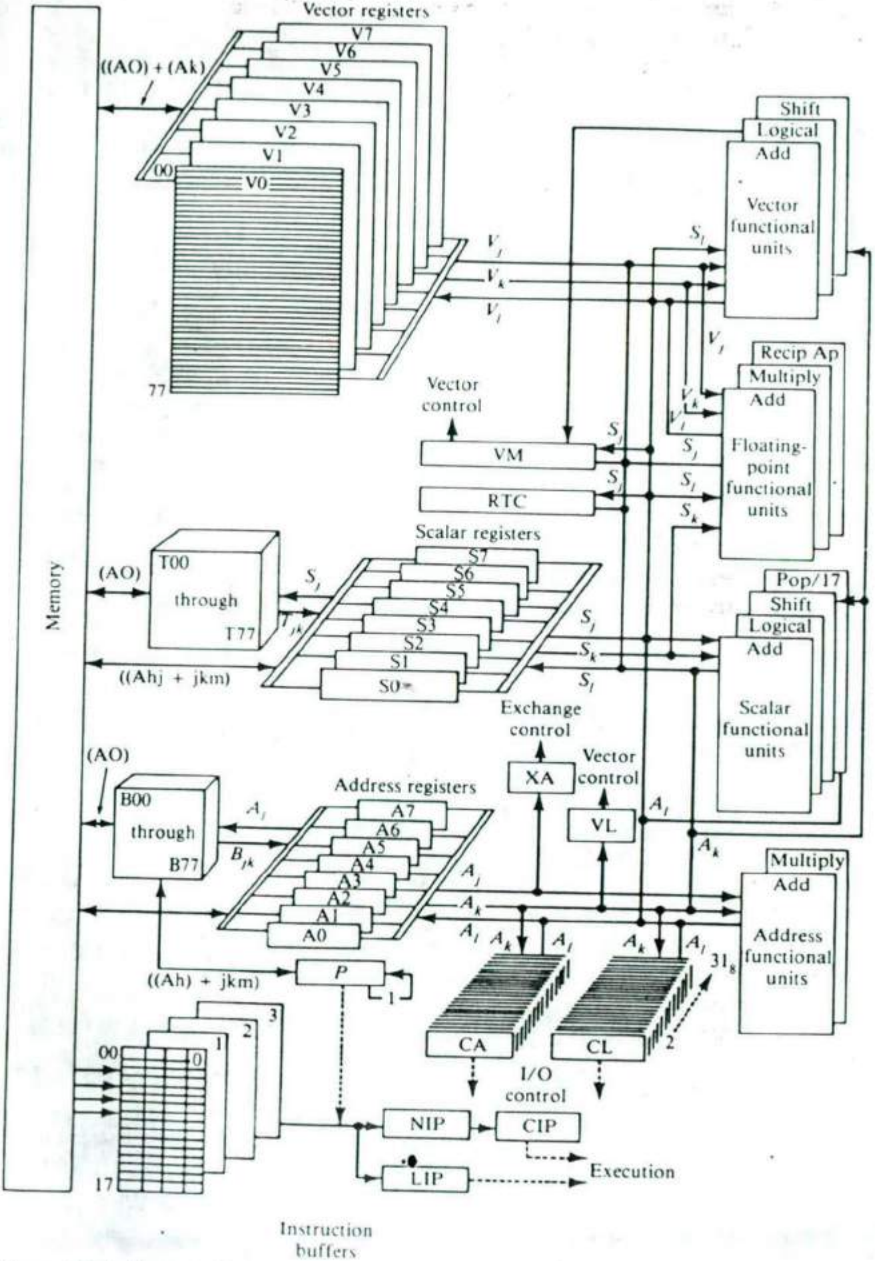


Figure 4.18 Arithmetic logic pipelines, registers, buffers, memory, and data paths in the Cray-1. (Courtesy of Cray Research, Inc.)

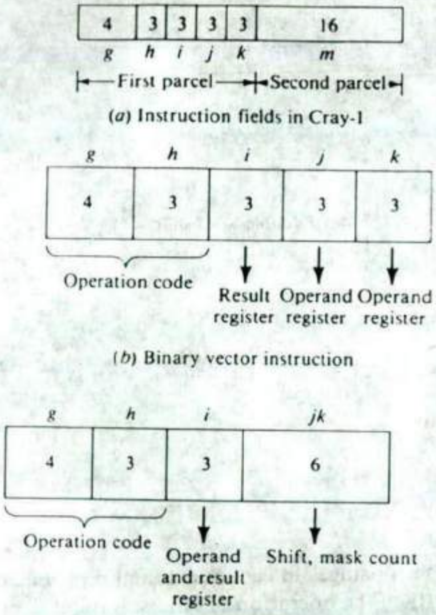
numbers of high-speed registers contribute to the vector and scalar processing capability of the Cray-1. Without these many registers, the functional units cannot operate with a clock rate of 12 ns. According to usage, there are five types of registers: three primary types and two intermediate types. The three primary types of registers are *address* (A), *scalar* (S), and *vector* (V) *registers*. The functional units can directly access primary registers. To support the scalar and address registers, an intermediate level of registers exists which is not accessible directly by the functional units. These registers act as buffers between the bipolar memory and the primary registers. The intermediate registers that support address registers are called *address-save registers* (B), and those supporting scalar registers are called *scalar-save registers* (T). Block transfers are made possible between B and T registers and the bipolar memory banks.

There are eight address registers with 24 bits each used for memory addressing, indexing, shift counting, loop control, and I/O channel addressing. Data can be moved directly between bipolar memory and A registers or can be placed in B registers first and then moved into A registers. There are sixty-four 24-bit B registers. The B registers hold data to be referenced repeatedly over a sufficiently long period. It is not desirable to retain such data in the A registers or in the bipolar memory. Examples of such uses are

There are eight 64-bit S registers serving as the storage of source and destination operands for the execution of scalar arithmetic and logical instructions. S registers may furnish one operand in vector instructions. Data can be moved directly between memory and S registers or can be placed in T registers as an intermediate step before transfer to S registers. There are sixty-four 64-bit T registers. T registers access the bipolar memory by block read or block write instructions. Block transfers occur at a maximum rate of one word per clock period.

There are eight V registers; each has 64 component registers. A group of data is stored in component registers of a V register to form a vector operand. Vector instructions demand the iterative processing of components in the subregisters. A vector operation begins with fetching operands from the first component of a V register and ends with delivering the vector result to a V register. Successive component operands are supplied for each clock period and the result is delivered to successive elements of the result V register. The vector operation continues until the number of operations performed equals a count specified by the *vector length* (VL) register. Vectors having a length greater than 64 are handled under program control in groups of 64 plus a remainder. The contents of a V register are transferred to or from memory in a block mode by specifying the address of the first word in memory, the increment for the memory address, and the vector length.

All instructions, either 16 or 32 bits long (Figure 4.19*a*), are first loaded from memory into one of four instruction buffers, each having sixty-four 16-bit registers. Substantial program segments can be prefetched with the large instruction buffers. Forward and backward branching within the buffers is possible. When the current instruction does not reside in a buffer, one instruction buffer is replaced with a new block of instructions from memory. Four memory words are



(*a*) Instruction fields in Cray-1

(*b*) Binary vector instruction

(c) Unary vector instruction

Figure 4.19 Instruction format of the Cray-1.

fetched per clock period to the least recently used instruction buffer. To allow fast issuing of instructions, the memory word containing the current instruction is the first to be fetched. The Cray-1 has 120 instructions with 10 vector types and 13 scalar types, the majority of which are three-address instructions (Figure 4.19*b*). Figure 4.19*c* shows the format of a unary vector instruction.

The P register is a 22-bit program counter indicating the next parcel of program code to enter the *next instruction parcel* (NIP) register in a linear program sequence. The P register is entered with a new value on a branch instruction or on an exchange sequence. The *current instruction parcel* (CIP) register is a 16-bit register holding the instruction waiting to be issued. The NIP register is a 16-bit register which holds a parcel of program code prior to entering the CIP register. If an instruction has 32 bits, the CIP register holds the upper half of the instruction. The *lower instruction parcel* (LIP) register, which is also 16 bits long, holds the lower half. Other registers, such as the *vector mask* (VM) registers, the *base address* (BA), the *limit address* (LA) registers, the *exchange address* (XA) register, the *flag* (F) register, and the *mode* (M) register, are used for masking, addressing, and program control purposes.

The twelve functional units in the Cray-1 are organized into four groups: *address, scalar, vector*, and *floating-point pipelines*, as summarized in Table 4.10. Each functional pipe has several stages. The register usage and the number of pipeline stages for each functional unit are specified in the table. The number of

Table 4.10 Cray-1 functional pipeline units

Functional pipelines	Register usage	Pipeline delays (clock periods)
Address functional units		
Address add unit	A	2
Address multiply unit	A	6
Scalar functional units		
Scalar add unit	S	3
Scalar shift unit	S	2 or 3 if double-word shift
Scalar logical unit	S	1
Population/leading zero count unit	S	3
Vector functional units		
Vector add unit	V or S	3
Vector shift unit	V or S	4
Vector logical unit	V or S	2
Floating-point functional units		
Floating-point add unit	S and V	6
Floating-point multiply unit	S and V	7
Reciprocal approximation unit	S and V	14

required clock periods equals the number of stages in each functional pipe. Each functional pipe can operate independently of the operation of others. A number of functional pipes can operate concurrently as long as there are no register conflicts. A functional pipe receives operands from the source registers and delivers the result to a destination register. These pipelines operate essentially in three-address mode with limited source and destination addressing.

The *address pipes* perform 24-bit 2's complement integer arithmetic on operands obtained from A registers and deliver the results back to A registers. There are two address pipes: the *address add* pipe and the *address multiply* pipe. The *scalar pipes* are for *scalar add*, *scalar shift*, *scalar logic*, and *population-leading zero count*, performing operations over 64-bit operands from S registers, and in most cases delivering the 64-bit result to an S register. The exception is the population-leading zero count, which delivers a 7-bit integer result to an A register. The scalar shift pipe can shift either the 64-bit contents of an S register or the 128-bit contents of two S registers concatenated together to form a double precision word. The population count pipe counts the number of "1" bits in the operand, while the leading zero count counts the number of "0" preceding a 1 bit in the operand. The scalar logical pipe performs the mask and boolean operations.

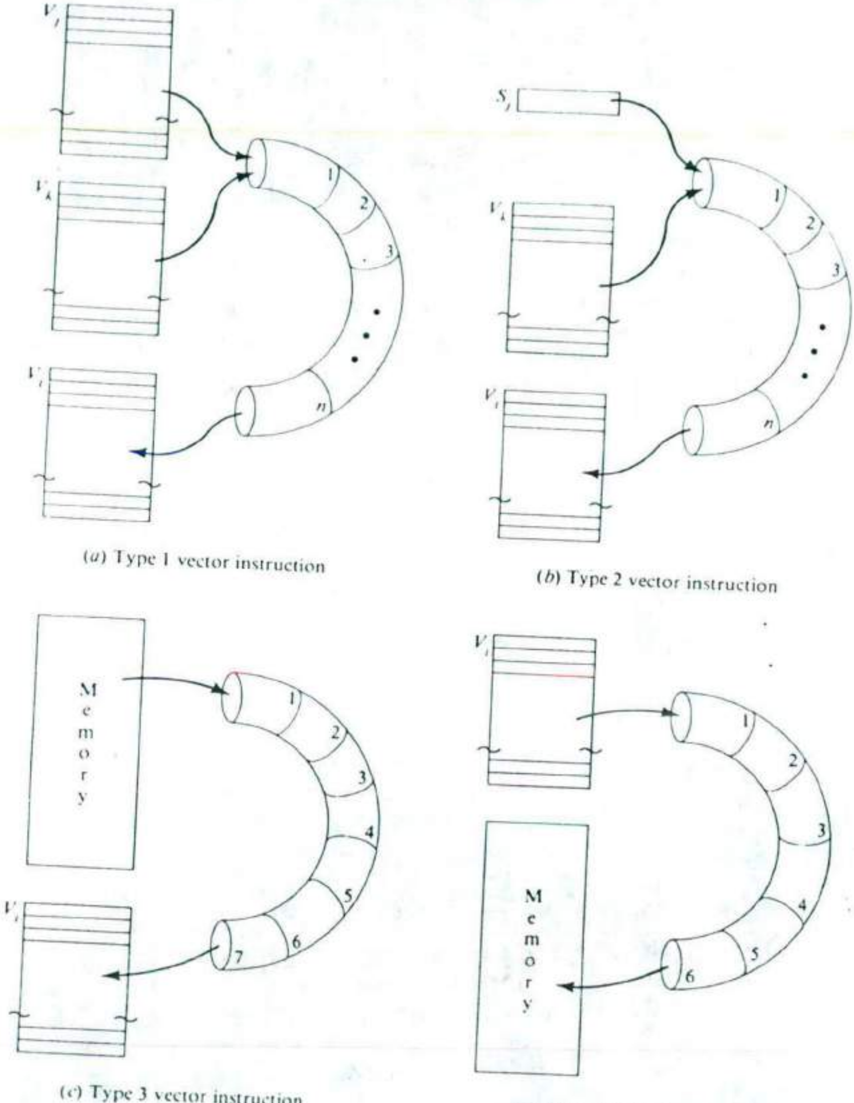
The *vector pipes* include the *vector add*, *vector logic*, and *vector shift*. These units obtain operands from one or two V registers and an S register. Results from a vector pipe are delivered to a V register. When a floating-point pipe is used for a vector operation, it can function similar to a vector pipe. The three *floating-point pipes* are for FLP *add*, FLP *multiply*, and *reciprocal approximation* over floating-point operands. The reciprocal approximation pipe finds the approximated reciprocal of a 64-bit operand in floating-point format. Note that no divide p exists in the Cray-1. The Cray-1 performs floating-point division by multiplyi the reciprocal of the divisor with the dividend. Add pipes of various types have ea two, three or six stages. All logical pipes have only one or two stages. Multi pipes require six or seven clocks for completion. The reciprocal approximati pipe has the longest delay of 14 clock periods. The two shifters have two, three, four clock delays.

The scalar add pipe and vector add pipe perform 64-bit integer arithmetic 2's complement mode. Multiplication of two fractional operands is accomplish by the floating-point multiply pipe. The floating-point multiply pipe recognizes a special case the condition of both operands having zero exponents and retur the upper 48-bits of the product of the fractions as the fraction of the result, leavi the exponent field zero. Division of integers would require that they first be co verted to FLP format and then divided by the floating-point reciprocal approxim tion pipe. The floating-point data format contains a 48-bit binary coefficient ar a 15-bit exponent field. Sign magnitude mantissa is assumed. Double-precisio computations having 95-bit fractions are performed with the aid of softwa routines, since the Cray-1 has no special hardware supporting multiple precisio operations. Logical functions are bitwise *and*, exclusive *or*, and inclusive operations.

In the Cray-1, the startup time for vector operations is nominal; thereby, eve for short vectors the performance is quite good. Because of the short startup tim of the vector pipes, there is little loss of speed in processing short vectors. Fo typical operations, vector lengths of three elements or less run faster in scalar mod while those of four elements or more run faster in vector mode. The vector mode definitely faster than scalar mode for long vectors. Of course, a vector operatio can also be processed as an iterative scalar operation, as it is done on any scala processor.

4.4.2 Pipeline Chaining and Vector Loops

The Cray-1 is designed to allow many arithmetic operations performed on operand that are resident in registers before returning them to memory. Resources lik registers and functional pipes must be properly reserved to enable multiple vecto processing. In register-to-register architecture, all vector operands are preloade into fast vector registers before feeding them into the pipelines. Intermediate an final results (outputs from pipeline) are also loaded into vector registers befor storing them in the main memory. We consider below the resource reservatio problem associated with a register-to-register vector processor like the Cray-1. A illustrated in Figure 4.20, vector instructions can be classified into four types. Th *type 1* instruction obtains operands from one or two vector registers and return results to another vector register. The *type 2* vector instruction consumes a scala operand from an S_j register and a vector operand from a V_k register and returns th vector results to another vector register, V_i. The *type 3* and *type 4* instruction transfer data from memory to a vector register and vice versa, respectively. A dat



(*a*) Type 1 vector instruction

(*b*) Type 2 vector instruction

(*c*) Type 3 vector instruction

(*d*) Type 4 vector instruction

Figure 4.20 **Four types of vector instruction in the Cray-1.**

path between memory and working registers can be considered a *data transmit pipeline* with a fixed-time delay.

When a vector instruction is issued, the required functional pipes and operand registers are reserved for a number of clock periods determined by the vector length. Subsequent vector instructions using the same set of functional units or operand registers cannot be issued until the reservations are released. Two or more vector instructions may use different functional pipelines and different vector registers at the same time, if they are independent. Such concurrent instructions can be issued in consecutive clock periods. Figure 4.21*a* shows two independent instructions, one using the add pipe and the other using the multiply pipe. Figure 4.21*b* depicts the demand on the add pipe by two independent vector additions. When the first *add* instruction is issued, the add pipe is reserved. Therefore, the issue of the second *add* instruction is delayed until the add pipe is freed. Figure 4.21*c* shows two different vector instructions sharing the same operand register V_1. The first *add* instruction reserves the operand register V_1, causing the issue of the *multiply* instruction to be delayed until the operand register V_1 is freed. Figure 4.21*d* illustrates the reservations of both the add pipe and the operand register V_1. Like the reservation required for operand registers, the result register needs also to be reserved for the number of clock periods determined by the vector length and the pipeline delays. This reservation ensures the proper transmittal of the final result to the result register.

A result register may become the operand register of a succeeding instruction. In the Cray-1, the technique is called *chaining* of two pipelines. Pipeline chaining

$$V_0 \leftarrow V_1 + V_2$$

$$V_3 \leftarrow V_4 \cdot V_5$$

(*a*) Independent instructions

$$V_3 \leftarrow V_1 + V_2$$

$$V_6 \leftarrow V_4 + V_5$$

(*b*) Functional unit reservation

$$V_3 \leftarrow V_1 + V_2$$

$$V_6 \leftarrow V_1 + V_5$$

(*c*) Operand register reservation

$$V_0 \leftarrow V_1 + V_2$$

$$V_3 \leftarrow V_1 + V_5$$

(*d*) Functional unit and operand register reservations

Figure 4.21 The reservation of functional units and operand registers.

is expanded from the concept of internal forwarding, discussed in Section 3.3.3. Basically, chaining is a linking process that occurs when results obtained from one pipeline unit are directly fed into the operand registers of another functional pipe. In other words, intermediate results do not have to be restored into memory and can be used even before the vector operation is completed. Chaining permits successive operations to be issued as soon as the first result becomes available as an operand. Of course, the desired functional pipes and operand registers must be properly reserved; otherwise, chaining operations have to be suspended until the demanded resources become available. The following example is used to illustrate pipeline chaining in the Cray-1. Because only eight vector registers are available, the number of pipeline functions that can be linked together is bounded by eight. Usually only two to five functions can be linked in a cascade.

Example 4.4 The following sequence of four vector instructions are chained together to be executed as a compound function:

$V_0 \leftarrow$ Memory (Memory fetch)

$V_2 \leftarrow V_0 + V_1$ (Vector add)

$V_3 \leftarrow V_2 < A_3$ (Left shift)

$V_5 \leftarrow V_3 \wedge V_4$ (Logical product)

A pictorial illustration is given in Figure 4.22 to show the chaining of the *memory fetch* pipe, the *vector add* pipe, the *vector shift* pipe, and the *vector logical* pipe into a longer pipeline processor. The contents of the register A_3 determine the shift count. A timing diagram of the chaining operations is shown in Figure 4.23. The memory fetch instruction is issued at time t_0. Each horizontal line shows the production of one component of the result in register V_5. The time spans in four pipelines are indicated by solid heavy line sections (marked by b, e, h, and k). The dashed lines represent the transit times (marked by a, c, d, f, g, i, j, and l) between memory fetch and functional pipelines or between transfers among vector registers. One operand is fetched from memory to the pipeline cascade per clock period. The first result emerges at clock period t_{23} and a new component result enters the V_5 register for each clock period thereafter.

In a vector operation, the results are normally not restored to the same vector register used by the source operands. Under certain circumstances, it may be desirable to route results directly back to one of the operand registers. Such recursive operations on functional pipelines require special precautions to avoid the data-jamming problem. To see how recursive computation can be realized in a pipeline, component operations must be properly monitored. Associated with each vector register is a *component counter*. When a vector instruction is issued, all component counters are set to zero. Normally, sending an operand from a source register to a functional pipeline causes the associated component counter

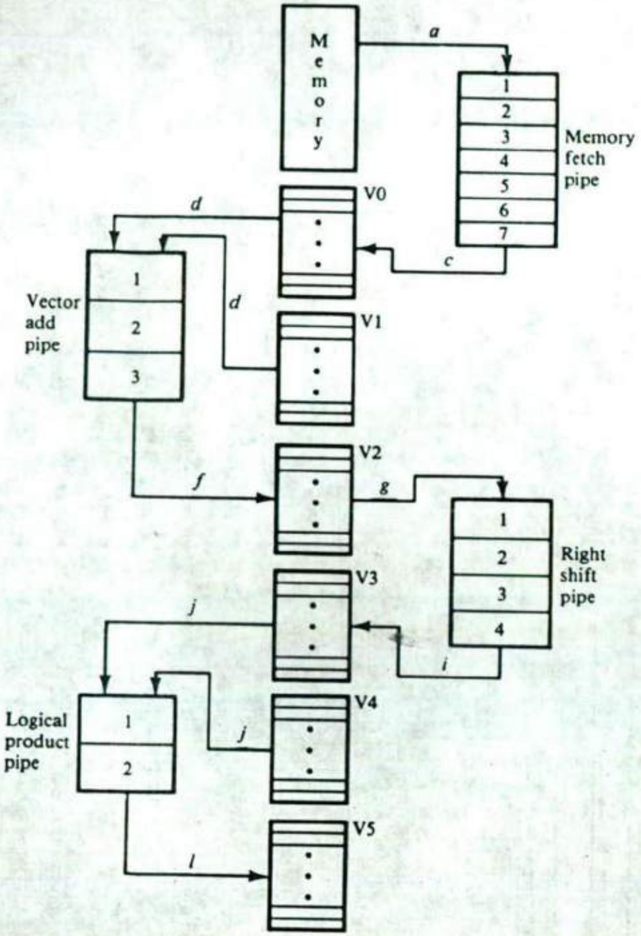
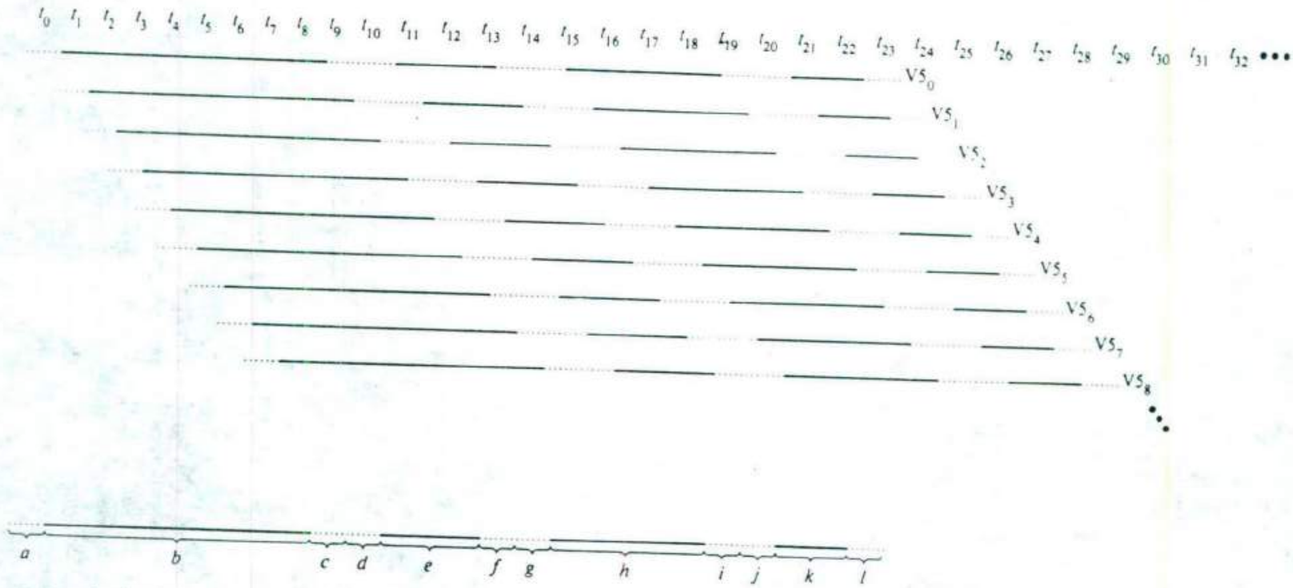


Figure 4.22 A pipeline chaining example in Cray-1. (Courtesy of Cray Research, Inc., Johnson 1977.)

to be incremented by one. Similarly, a component result arriving at a vector register from a functional pipe causes the associated component counter to be incremented by one. When a vector register serves as both operand and result register, its component counter will not be updated until the first result returns from the functional pipe. While the counter is held at zero, the same operand component is repeatedly sent to the functional pipe.

The recursive use of functional pipelines can be applicable to many vector operations, either arithmetic or logical. The initial value in a component register depends on the operation to be performed. Since each vector register in the Cray-1



a: transit of memory word to "read functional unit"

b: transit of memory word through "read functional unit"

c: transit of memory word from "read functional unit" to element of V0

d: transit of operand elements in V0 and V1 to integer add functional unit

e: computation of sum by integer add functional unit

f: transit of sum from integer add functional unit to element of V2

g: transit of operand element in V2 to shift functional unit

h: shift operation performed by shift functional unit

i: transit of shifted sum from shift functional unit to element of V3

j: transit of operand elements in V3 and V4 to logical functional unit

k: logical operation performed by logical functional unit

l: transit of final result to element of V5

Figure 4.23 Timing diagram for the chaining example in Figure 4.22.

can accommodate at most 64 elements, long vectors are processed in segments. The program construct for processing long vectors is called a *vector loop*. The segmentation of long vectors into loops is done by the system hardware and software control. The programmer never sees this segmentation into vector loops. Each pass through the loop processes a 64-element (or smaller) section of the long vectors. Generally, the loop count is computed from the vector length before entering the loop. Inside the loop, each of the twelve functional pipes can be fully utilized to process the current section. The following is an implemented example demonstrating vector looping in the Cray-1.

Example 4.5 Let A and B be vectors of length N. Consider the following loop operations:

```
DO 10 I=1,N
10 A(I)=5.0*B(I)+C
```

When N is 64 or less, a sequence of seven instructions generates the A array:

$S_1 \leftarrow 5.0$	Set constant in scalar register
$S_2 \leftarrow C$	Load constant C in scalar register
$\mathrm{VL} \leftarrow N$	Set vector length into VL register
$V_0 \leftarrow B$	Read B vector into vector register
$V_1 \leftarrow S_1 * V_0$	Multiply each component of the B array by a constant
$V_2 \leftarrow S_2 + V_1$	Add C to $5 * B(I)$
$A \leftarrow V_2$	Store the result vector in A array

The fourth and sixth instructions use different functional pipelines with shared intermediate registers. They can be chained together. The outputs of the chain are finally stored in the A array. When N exceeds 64, vector loops are required. Before entering the loop, N is divided by 64 to determine the loop count. If there is a remainder, the remainder elements of the A array are generated in the first loop. The loop consists of the fourth to the seventh instructions for each 64-element segment of the A and B arrays.

Recursion can be implemented by vector operations in the Cray-1 with the help of the component counters. Let C_1 be the component counter associated with a vector register V1. To introduce recursions, one vector register, say V0, must serve as both an operand register and a result register. Under such circumstances, C_0 will be held to zero until the first component result is returned from the functional pipe to V0. In other words, C_0 updates according to its function as a result register. During this period, the same component operand from $\mathrm{C0}_0$, the first component register in V0, is repeatedly sent to the functional pipe until C_0 increments after the first result. The use of the same component from V0 can repeat for

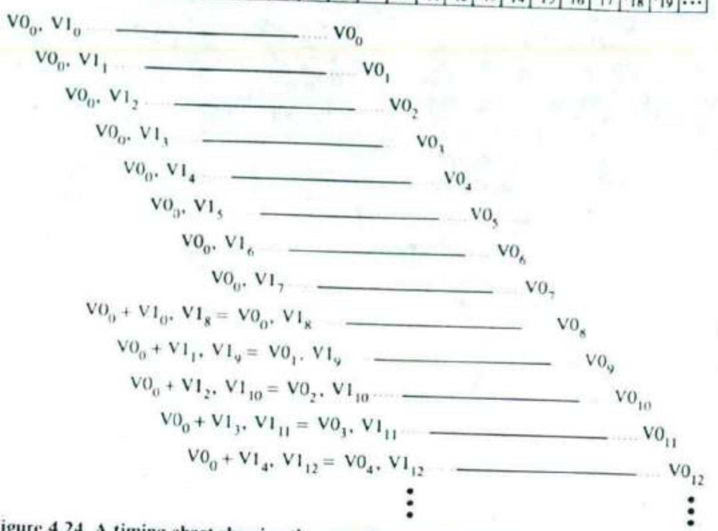

t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15} t_{16} t_{17} t_{18} t_{19} ···
$V0_0, V1_0$ — $V0_0$
$V0_0, V1_1$ — $V0_1$
$V0_0, V1_2$ — $V0_2$
$V0_0, V1_3$ — $V0_3$
$V0_0, V1_4$ — $V0_4$
$V0_0, V1_5$ — $V0_5$
$V0_0, V1_6$ — $V0_6$
$V0_0, V1_7$ — $V0_7$
$V0_0 + V1_0, V1_8 = V0_0, V1_8$ — $V0_8$
$V0_0 + V1_1, V1_9 = V0_1, V1_9$ — $V0_9$
$V0_0 + V1_2, V1_{10} = V0_2, V1_{10}$ — $V0_{10}$
$V0_0 + V1_3, V1_{11} = V0_3, V1_{11}$ — $V0_{11}$
$V0_0 + V1_4, V1_{12} = V0_4, V1_{12}$ — $V0_{12}$

Figure 4.24 A timing chart showing the recursive summation of the vector components in Example 4.6.

subsequent components in V0 until the entire vector operation is done. The following example will clarify this recursion concept.

Example 4.6 Consider the use of the floating-point add pipe for the recursive vector summation V0 ← V0 + V1, where the vector register V1 holds an array of floating-point numbers to be added recursively. The timing chart of the recursion is shown in Figure 4.24. Initially, both counters C_0 and C_1 are set to be zero. The initial value in the first component register $V0_0$ of V0 is also set to zero. The FLP add pipe requires six clock periods to pass through. Register transfer to or from the FLP add pipe takes another clock period. Therefore, the total cycle is $1 + 6 + 1 = 8$ clock periods, as shown in Figure 4.24. The vector-length register is assumed to have a value of 64 for a single vector loop.

The counter C_0 is kept at zero until time t_8. During this cycle, $V0_0$ (which is set to 0) is sent to the pipeline. However, the counter C_1 keeps incrementing after each clock period. Therefore, $V1_0, V1_1, \ldots, V1_{63}$ are sent to the pipeline in subsequent 64 clock periods after t_0. After t_8, C_0 gets incremented by one after each clock period. This means the successive output sums are added recursively with one additional component from V1 in every eight clock periods. When the computations are completed, the component registers of V0 should be loaded as shown in Table 4.11. The 64 components are divided into eight groups of eight component sums each. The last summation group from $V0_{56}$ to $V0_{63}$ holds the eight summations of the eight components of V1, each.

Table 4.11 Successive contents of vector register V0 in Example 4.6

$(V0_0) = (V0_0) + (V1_0) = 0 + (V1_0)$	The first group
$(V0_1) = (V0_0) + (V1_1) = 0 + (V1_1)$	
$(V0_2) = (V0_0) + (V1_2) = 0 + (V1_2)$	
$(V0_3) = (V0_0) + (V1_3) = 0 + (V1_3)$	
$(V0_4) = (V0_0) + (V1_4) = 0 + (V1_4)$	
$(V0_5) = (V0_0) + (V1_5) = 0 + (V1_5)$	
$(V0_6) = (V0_0) + (V1_6) = 0 + (V1_6)$	
$(V0_7) = (V0_0) + (V1_7) = 0 + (V1_7)$	
$(V0_8) = (V0_0) + (V1_8) = (V1_0) + (V1_8)$	The second group
$(V0_9) = (V0_1) + (V1_9) = (V1_1) + (V1_9)$	
$(V0_{10}) = (V0_2) + (V1_{10}) = (V1_2) + (V1_{10})$	
$(V0_{11}) = (V0_3) + (V1_{11}) = (V1_3) + (V1_{11})$	
$(V0_{12}) = (V0_4) + (V1_{12}) = (V1_4) + (V1_{12})$	
$\vdots$	
$(V0_{15}) = (V0_7) + (V1_{15}) = (V1_7) + (V1_{15})$	
$(V0_{16}) = (V0_8) + (V1_{16}) = (V1_0) + (V1_8) + (V1_{16})$	↑ Group 3
$\vdots$	⋮ through
$(V0_{55}) = (V0_{47}) + (V1_{55}) = (V1_7) + (V1_{15}) + (V1_{23}) + (V1_{31}) + (V1_{30}) + (V1_{47}) + (V1_{55})$	↓ group 7
$(V0_{56}) = (V0_{48}) + (V1_{56}) = (V1_0) + (V1_8) + (V1_{16}) + (V1_{24}) + (V1_{32}) + (V1_{40}) + (V1_{48}) + (V1_{56})$	The eighth group (result)
$(V0_{57}) = (V0_{49}) + (V1_{57}) = (V1_1) + (V1_9) + (V1_{17}) + (V1_{25}) + (V1_{33}) + (V1_{41}) + (V1_{49}) + (V1_{57})$	
$(V0_{58}) = (V0_{50}) + (V1_{53}) = (V1_2) + (V1_{10}) + (V1_{18}) + (V1_{26}) + (V1_{34}) + (V1_{42}) + (V1_{50}) + (V1_{58})$	
$(V0_{59}) = (V0_{51}) + (V1_{59}) = (V1_3) + (V1_{11}) + (V1_{19}) + (V1_{27}) + (V1_{35}) + (V1_{43}) + (V1_{51}) + (V1_{59})$	
$(V0_{60}) = (V0_{52}) + (V1_{60}) = (V1_4) + (V1_{12}) + (V1_{20}) + (V1_{28}) + (V1_{36}) + (V1_{44}) + (V1_{52}) + (V1_{60})$	
$(V0_{61}) = (V0_{53}) + (V1_{61}) = (V1_5) + (V1_{13}) + (V1_{21}) + (V1_{29}) + (V1_{37}) + (V1_{45}) + (V1_{53}) + (V1_{61})$	
$(V0_{62}) = (V0_{54}) + (V1_6) + (V1_{14}) + (V1_{22}) + (V1_{30}) + (V1_{36}) + (V1_{46}) + (V1_{54}) + (V1_{62})$	
$(V0_{63}) = (V0_{55}) + (V1_{63}) = (V1_7) + (V1_{15}) + (V1_{23}) + (V1_{31}) + (V1_{39}) + (V1_{47}) + (V1_{55}) + (V1_{63})$	

The above recursive vector summation is very useful in scientific computations. For an example, the dot product of vectors $A \cdot B = \sum a_i \cdot b_i$ can be implemented in the Cray-1 by chaining the following two operations: a *vector multiply* V1 ← V3 $*$ V4 followed by a *floating-point add* V0 ← V0 + V1. If the vector length is 64, the dot product can be reduced to 8 sums (from 64) after the *chained multiply and add* operations. The next iteration is to find the sum of the eight subsums to product the final result. For *recursive vector multiplication*, similar operations can be implemented with a floating-point multiple pipe, except the initial value of $V0_0$ should be 1 instead of 0. This operation should be useful in polynomial evaluations.

The performance of the Cray-1 may vary from 3 to 160 megaflops, depending on the applications and programming skills. Scalar performance of 12 megaflops was observed for matrix multiplication. Vector performance of 22 megaflops was observed in vector dot product operations. Supervector performance of 153 megaflops was observed in assembly-code matrix multiplication. These speeds are special peak values. The Cray-1 will more likely have an average vector-supervector performance in the range of 20 to 80 megaflops, depending of course on the work load distribution.

In order to achieve even better supercomputer performance, Cray Research has extended the Cray-1 to the Cray X-MP, a dual-processor system with loosely coupled multiprogramming and single-program multiprocessing. The Cray X-MP has eight times the Cray-1 memory bandwidth and a reduced clock period of 9.5 ns. It has guaranteed chaining. Furthermore, the software for the Cray X-MP is compatible with that of the Cray-1. The first customer shipments of the Cray X-MP took place in 1983, with full production in 1984.

The Cray X-MP offers impressive speedup over the Cray-1. For mixed jobs, it has been estimated that the Cray X-MP has a 2.5 to 5 times throughput gain over the Cray-1. For scalar processing, it is 1.25 to 2.5 times faster than the Cray-1. Again, it is excellent for both short and long vector processing, as is the Cray-1. When the Cray X-MP gets upgraded to the Cray-2 after the mid-80s, the performance is expected to increase six times in scalar and 12 times in vector operations over the Cray-1. The Cray-2 will have four processors with a basic pipeline clock rate of 4 ns, 32 M words of main memory, and 20 times improved I/O. We will study various Cray Research's multiprocessors in detail in Chapter 9.

4.4.3 The Architecture of Cyber-205

The Cyber-205 represents more than 20 years of evolution in scientific computing by the Control Data Corporation from the early CDC-1604, through the CDC 6600/7600 series, to the Star-100 and Cyber-203. The Cyber-203 improves over the Star-100 by the use of semiconductor memory, concurrent scalar processing, and a memory-interface design that permits instructions to be issued every 20 ns. The improvements of the Cyber-205 over the 203 stem from the use of entire LSI circuitry with large bipolar memory, additional vector instructions, and support by the NOS-based operating system. The Cyber 205 became available in 1981.

The system architecture of Cyber-205 (Figure 4.25) differs from that of the Star-100 (Figure 4.2) in the addition of a powerful scalar processor and two more vector pipelines (for a total of four vector pipes). The basic pipeline clock period of the Cyber-205 pipelines is 20 ns and the memory cycle is 80 ns, half of that of the Star. Only 26 distinct LSI-chip types are used, which significantly increases system reliability and maintainability. Instead of using slow core memory, the Cyber-205 uses bipolar main memory of up to four million 64-bit words with an 80-ns cycle time. Memory-access patterns include 512-bit superwords (eight 64-bit words) for vector operands and fullwords (64 bits) and halfwords (32 bits) for scalar operands. The main memory bandwidth is 400 MW/s, significantly

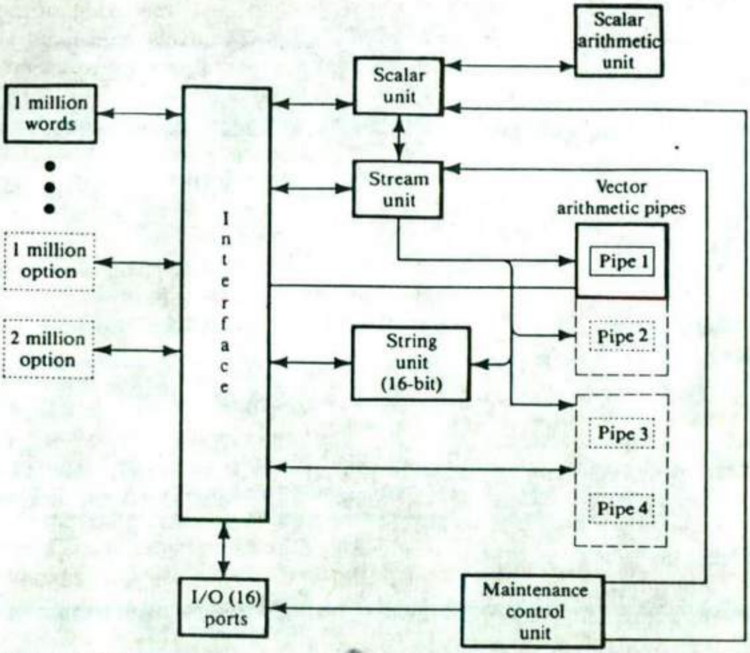


Figure 4.25 The Cyber-205 computer system configuration. (Courtesy of Control Data Corp.)

higher than that in the Cray-1. The high memory bandwidth is needed to support the memory-to-memory pipeline operations.

Instruction-execution control resides in the *scalar unit*, which receives and decodes all instructions from memory, directly executes scalar instructions, and dispatches vector and string instructions to the four *vector pipes* and the *string unit* for execution. It also provides orderly buffering and execution of the data and instructions. With independent vector and scalar instruction controls to a single-instruction stream, the scalar unit can execute scalar instructions in parallel with the execution of most vector instructions.

The *scalar arithmetic unit* contains five independent functional pipes for *add/subtract*, *multiply*, *log*, *shift*, and *divide/sqrt* operations over 32- or 64-bit scalars. The peak speed of the scalar processor is 50 megaflops. The *vector processor* has the option of having one, two, or four floating-point arithmetic pipes. The *stream unit* manages the data streams between central memory and the vector pipelines. A vector arithmetic pipe can perform *add/subtract*, *multiply*, *divide*, *sqrt*, *logical*, and *shift* over 32- or 64-bit vector operands. Each vector pipeline is directly connected to the main memory without using vector registers (Figure 4.25*a*). The *string unit* processes the control vectors during streaming operations. It provides the capability for BCD and binary arithmetic-address arithmetic and boolean operations.

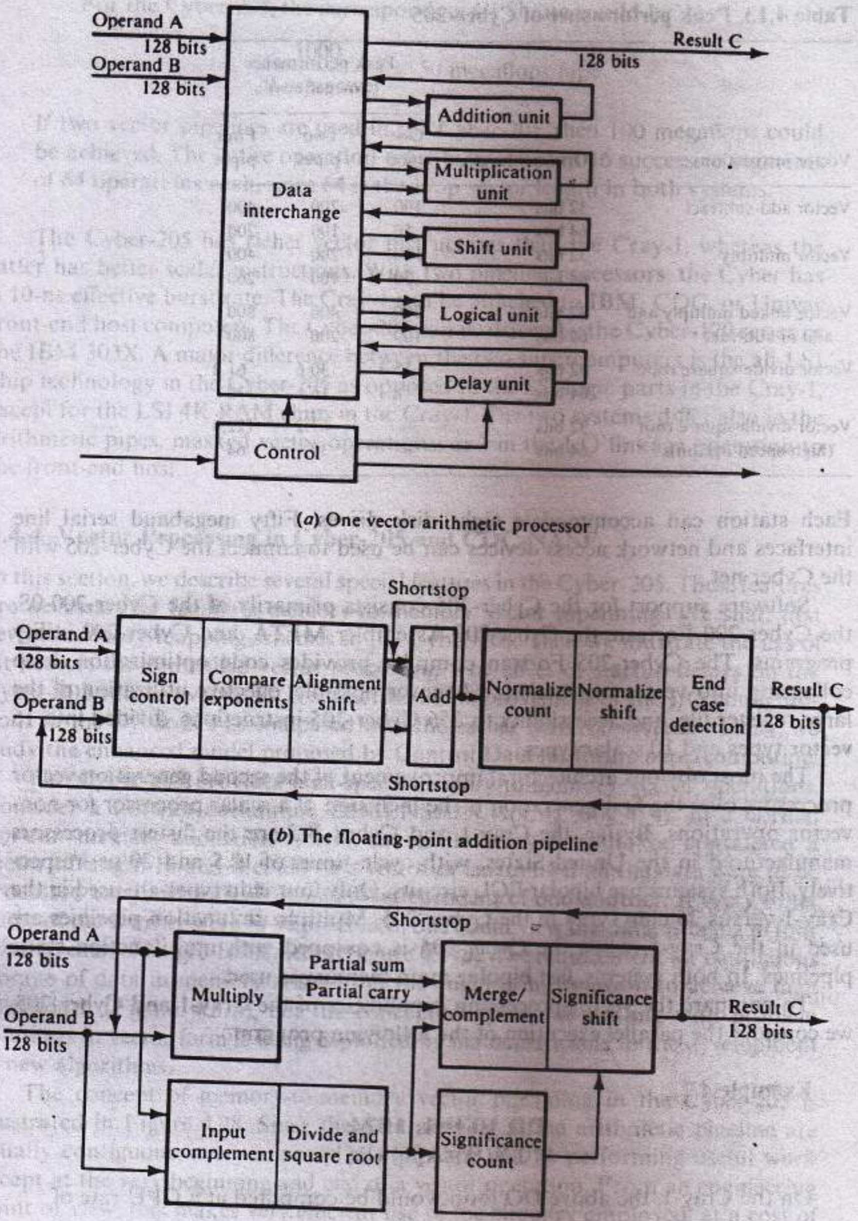
The vector startup time in the Cyber-205 is much longer than that of the Cray-1. A vector may comprise up to 65,635 consecutive memory words. Control vectors are used to address data that is stored in nonconsecutive locations. Each pipeline receives two input streams and generates one output stream of floating-point numbers. Each stream is 128 bits wide, supporting a 100-megaflops computation rate of 32-bit results or 50 megaflops for 64-bit results per each vector processor. With four vector pipes, the Cyber-205 can produce 200 megaflops for 64-bit results and 400 megaflops for 32-bit results in *vector add/subtract* or in *vector multiply* operations. The Cyber-205 can also be used to perform *linked vector multiply* and *vector add/subtract* operations with a maximum rate of 800 or 400 megaflops for 64- or 32-bit results, respectively, on a four-processor configuration. The vector divide and *square root* operations are much slower than the *add/subtract* or *multiply* operations.

Each vector arithmetic unit consists of five functional pipes, as shown in Figure 4.26*a*. The detailed pipeline stages in the floating-point *add unit* and the *multiply unit* are shown in Figure 4.26*b* and *c*. Both pipeline units have feedback connections for accumulative add or multiply operations. These two units are improved over the designs in the Star-100 (Figure 4.3). The pipeline delays in both vector-scalar units are summarized in Table 4.12. With a 20-ns clock rate, the number of required clock periods is also shown in each case. The *load/store* is also pipelined. Pipelining produces one result per each clock period of 20 ns. The result from any of the above units can be routed directly to the input of other units without stopping in some intermediate registers. This process is called *short-stopping*, as facilitated by the feedback connections in Figure 4.25. The theoretical peak performance of the Cyber-205 is summarized in Table 4.13 for 32- and 64-bit results.

The bipolar memory is four-way interleaved, giving an effective cycle time of 20 ns per word. The central memory is a virtual memory system with advanced memory management features such as *key* and *lock* for memory protection and separation, hardware mapping from virtual to physical address, and user program-data sharing capability. The page sizes vary from small (1K, 2K, 8K words) to large (65K words). The Cyber-205 has 16 I/O channels, each 16 bits wide. The I/O system consists of multiple minicomputers for handling up to 10 disk stations.

Table 4.12 Pipeline delays in the Cyber-205

Functional pipe	Time delays (ns)	Clock periods
Load-store	300	15
Add-subtract	100	5
Multiply	100	5
Logical	60	3
Divide-square root	1080 (64 bits)	54
Conversion	600 (32 bits)	30



(a) One vector arithmetic processor

(b) The floating-point addition pipeline

(c) The floating-point multiply pipeline

Figure 4.26 Pipelined structure of one vector processor in the Cyber-205. (Courtesy of Control Data Corp.)

Table 4.13 Peak performance of Cyber-205

Vector instructions	Operand length	Peak performance (megaflops) One pipe	 Two pipes	 Four pipes
Vector add-subtract	32 bits	100	200	400
	64 bits	50	100	200
Vector multiply	32 bits	100	200	400
	64 bits	50	100	200
Vector linked multiply and add or subtract	32 bits	200	400	800
	64 bits	100	200	800
Vector divide-square root	32 bits	15.3	30.6	61.2
	64 bits	8	16	32
Vector divide-square root (high-speed option)	32 bits	—	61.2	122.4
	64 bits	—	32	64

Each station can accommodate eight disk drives. Fifty megabaud serial line interfaces and network access devices can be used to connect the Cyber-205 with the Cyber net.

Software support for the Cyber-205 consists primarily of the Cyber-200-0S, the Cyber-200 Fortran, the Cyber-200 Assembler META, and Cyber-200 utility programs. The Cyber-205 Fortran compiler provides code optimization, loop collapsing into vector instructions whenever possible, effective utilization of the large register file, and accessibility to 256 Cyber-205 instructions, divided into 16 vector types and 10 scalar types.

The most obvious architectural improvement of the second generation vector processors over the first generation is the inclusion of a scalar processor for non-vector operations. By far, the Cray-1 and Cyber-205 are the fastest processors manufactured in the United States, with cycle times of 12.5 and 20 ns, respectively. Both systems use bipolar ECL circuits. Only four chip types are used in the Cray-1 versus 26 chip types in the Cyber-205. Multiple unifunction pipelines are used in the Cray-1, while the Cyber-205 is equipped with multifunction static pipelines. In both systems, fast bipolar main memory is used.

To compare the vector-processing capabilities of the Cray-1 and Cyber-205, we consider the parallel execution of the following program.

Example 4.7

```
DO 10 I=1, 1024
10 Y(I)=A(1)*B(1)
```

On the Cray-1, the above DO loop would be computed at a CPU rate of

$$\frac{1000}{12.5 \text{ ns/answer}} = 80 \text{ megaflops}$$

For the Cyber-205, the corresponding CPU rate would be

$$\frac{1000}{20 \text{ ns/answer}} = 50 \text{ megaflops/pipe}$$

If two vector pipelines are used in the Cyber-205, then 100 megaflops could be achieved. The entire operation must be grouped in 16 successive segments of 64 operations each, since 64 is the loop vector length in both systems.

The Cyber-205 has richer vector instructions than the Cray-1, whereas the latter has better scalar instructions. With two pipeline processors, the Cyber has a 10-ns effective burst rate. The Cray-1 can be attached to IBM, CDC, or Univac front-end host computers. The Cyber-205 can be driven by the Cyber-170 series or the IBM 303X. A major difference between the two supercomputers is the all-LSI chip technology in the Cyber-205 as opposed to the SSI logic parts in the Cray-1, except for the LSI 4K RAM chips in the Cray-1. The two systems differ also in the arithmetic pipes, masked vector operations, and in the I/O linkage operation to the front-end host.

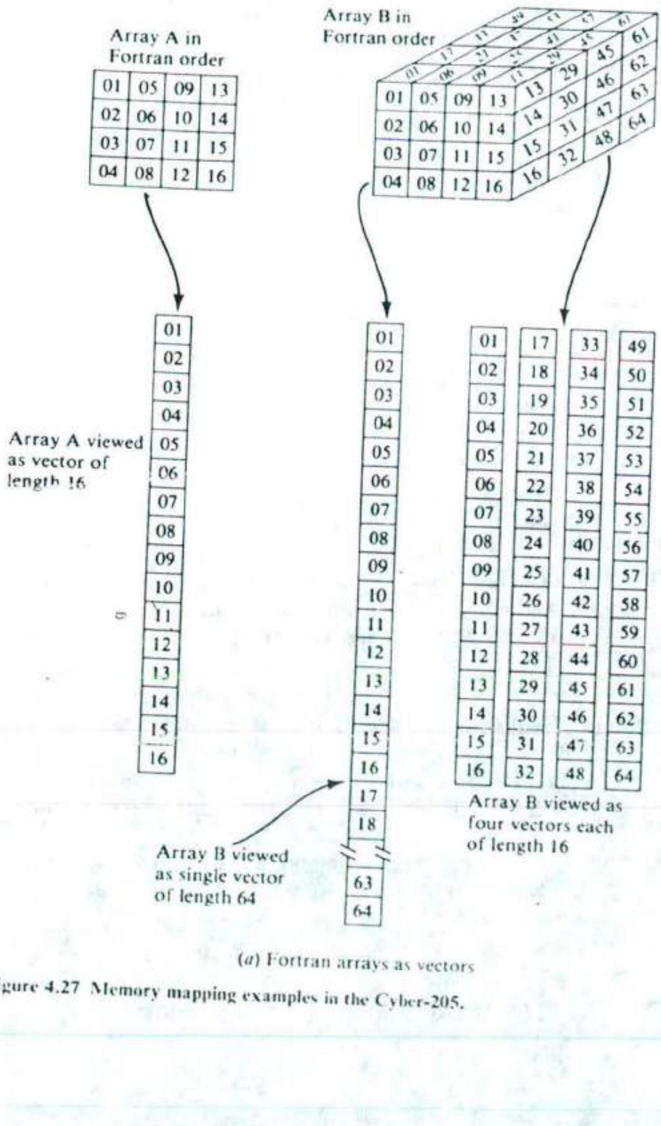
4.4.4 Vector Processing in Cyber-205 and CDC-NASF

In this section, we describe several special features in the Cyber-205. These features are necessary to facilitate memory-to-memory vector pipelining. We shall first review memory-mapping schemes in the Cyber-205. Then we illustrate the use of bit vectors for controlled vector processing. The effect of startup delays on the Cyber-205 will be evaluated. We describe the improvement of the I/O configurations in the Cyber-205 as compared with the earlier Star-100 system. Finally, we study the enhanced model proposed by Control Data for future supercomputing.

The Cyber-205 provides high-speed memory-to-memory vector operations. Consider a Fortran declaration DIMENSION A(4, 4), B(4, 4, 4). In a normal Fortran memory-allocation system, the entire array **A** could be considered a vector of length 16 and **B** could be a vector of length 64 if all the data were to be processed as a unit with all elements independent of one another. If every other plane of the **B** array were to be processed, one could view the data as being in four vectors, each of length 16 elements (Figure 4.27*a*). The requirement for contiguous storage of data in memory arises from the engineering solution to achieve high bandwidth (Figure 4.27*b*), but the conceptual notion of dealing with problem solutions in vector form is being exploited by mathematicians in the development of new algorithms.

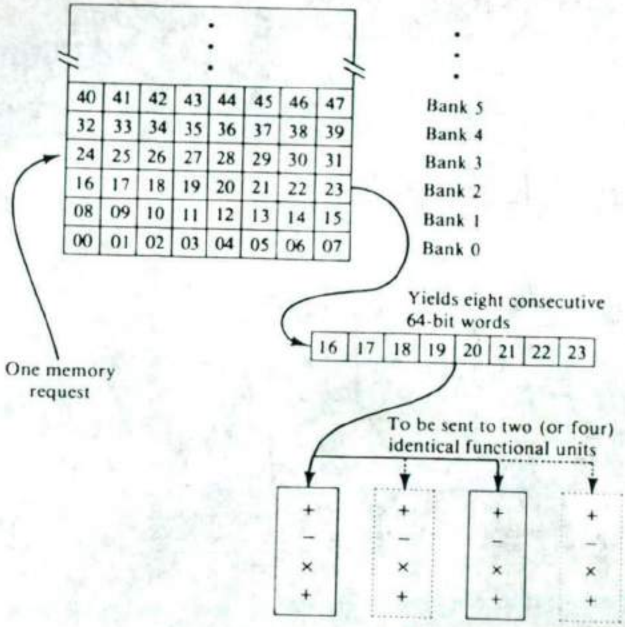
The concept of memory-to-memory vector pipelining in the Cyber-205 is illustrated in Figure 4.28. Since the data arriving at the arithmetic pipeline are usually contiguous, every segment in a pipeline will be performing useful work except at the very beginning and end of a vector operation. From an engineering point of view, this makes very efficient use of the circuitry employed, at a cost of some time to "get the operation rolling" (first result returned to memory or registers) due to the length (in number of clock cycles) of the pipeline.

Experience has shown that few supercomputers totally operate on a single massive problem in a dedicated manner. The normal installation finds the waking hours being consumed by algorithm development, program debugging, and interactive execution of research programs and even large production programs. The evening and midnight hours are usually more structured, with one or more major programs monopolizing the machine's resources and with, perhaps, a modest amount of interactive debugging being pursued in a background mode.


Array A in Fortran order
Array B in Fortran order
Array A viewed as vector of length 16
Array B viewed as single vector of length 64
Array B viewed as four vectors each of length 16

(*a*) Fortran arrays as vectors

Figure 4.27 Memory mapping examples in the Cyber-205.



(*b*) Cyber-20 memory allocation

Figure 4.27 (continued) (Courtesy of *IEEE Trans. Computers*, Lincoln 1982.)

Virtual memory plays a major supporting role in a supercomputer. Listed below are the important memory functions in the Cyber-205:

1. *Assist in memory management*: The operating system can commit and decommit arbitrary blocks of real memory without having to ensure the physical contiguity of a user's workspace. This reduces overhead for actions such as the accumulation of unused space, which could be quite costly in large memory systems (on the order of eight million words).
2. *Provide identically appearing execution of all jobs*: This means that a program's *dimension* statements and input parameters can remain unchanged whether or not a 4-hour run is being contemplated or a simple debugging run of a particular phase is intended.
3. *Eliminate working space constraints from algorithm development*: Mathematicians and programmers can begin developing an algorithm as if they had available an infinite workspace in which to put data and temporary results. Once the algorithm is developed, the programmer must introduce the means to handle paging of the information in order to optimize the performance of the system and to eliminate the thrashing that can occur in virtual memory machines moving data to and from real central memory.

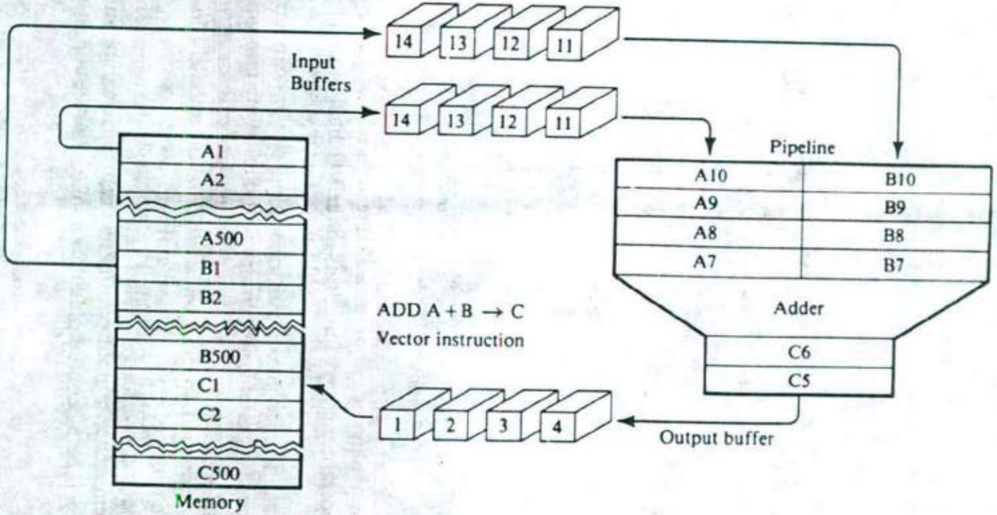
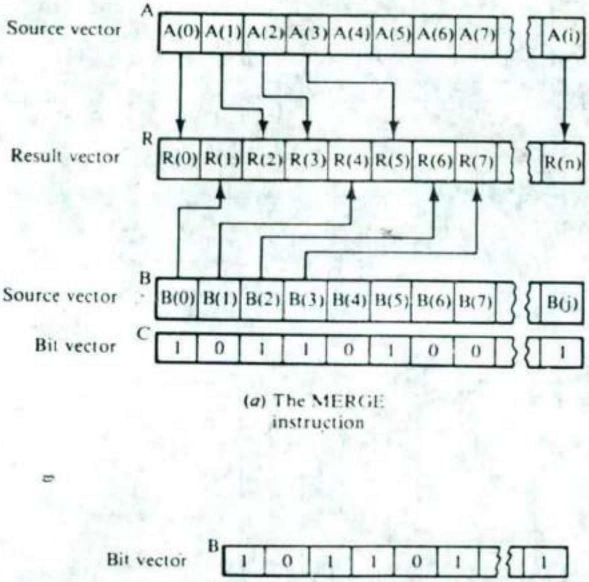


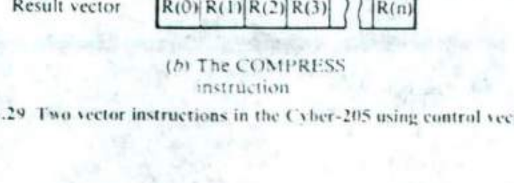
Figure 4.28 Memory-to-memory vector pipelining in the Cyber-205. (Courtesy of *IEEE Trans. Computers*, Lincoln 1982.)

A most significant contribution of the Cyber-205 was the notion that strings of binary bits (called *bit vectors*) could be used to carry information about vectors and could be applied to those vectors to perform some key functions. Since the bit strings became the key to the vector-restructuring concept, a means had to be provided to manipulate bit vectors as well as numeric vectors; thus was added the *string functional unit* to the hardware ensemble in both the Star-100 and the Cyber-205. The functions of *compress, mask, merge, scatter*, and *gather* were incorporated. In addition, many of the reduction operations like *sum, product*, and *inner product* were implemented directly in the hardware.

Two special vector instructions using the control bit vectors are illustrated in Figure 4.29. In part *a*, the two source vectors A and B are merged under the control of the bit vector C to give the result vector R. The merging is conducted so as to select from A on "1" in C, and from B on "0" in C. In part *b* the source vector A



(*a*) The MERGE instruction



(*b*) The COMPRESS instruction

Figure 4.29 Two vector instructions in the Cyber-205 using control vectors.

is being compressed to give the result vector R under the bit vector B. The compression is done on "0" in B. These instructions are extremely useful in manipulating sparse matrices.

Startup time of vector operations includes the instruction translation time and the delay imposed by fetching and aligning the input streams and by aligning and moving output streams to memory. Figure 4.30 illustrates the impact of startup time on the effective performance of the Cyber-205 architecture. A major improvement in the overall performance of this type of memory-to-memory vector architecture requires careful attention to startup time. In addition to the raw improvement in clock speed, the designers were directed to other methods for reducing the delay in vector initiation.

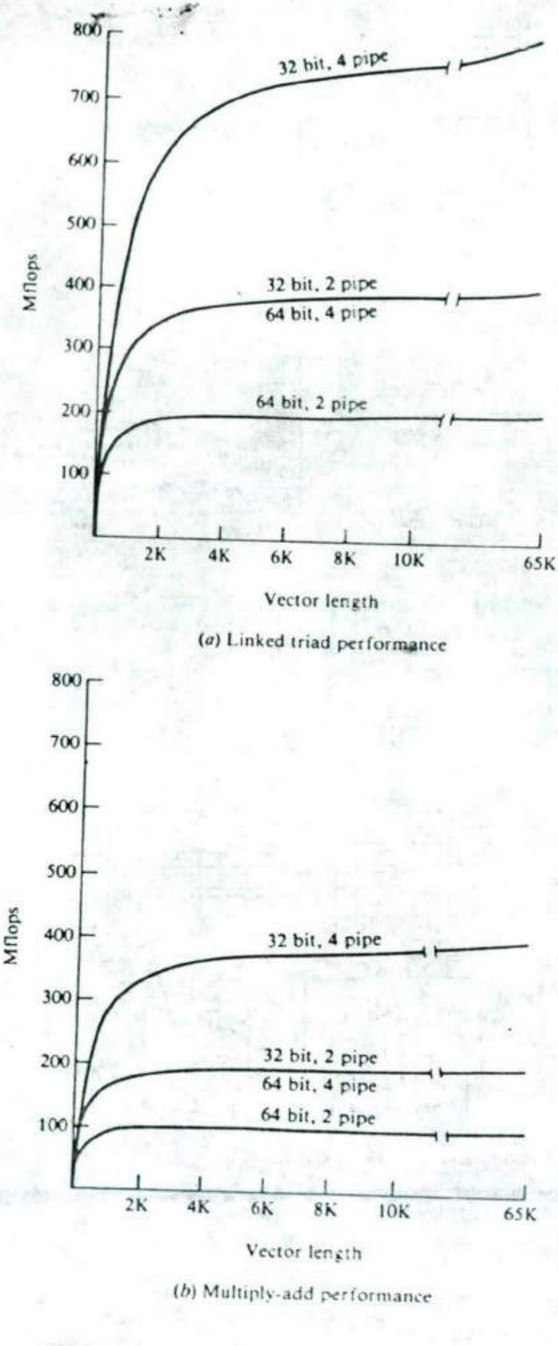
The actions of address setup and management of vector arithmetic control lines require substantial speedup. In addition, providing separate and independent functional units in each of the scalar and vector processors permits execution of those functions in parallel. Hence, the apparent startup time becomes smaller than what the hardware provides. This feature, in the best cases, results in parallel execution of both scalar and vector floating-point operations with a consequent increase in overall performance.

Identified below are three major changes in the architecture of the Star-100 to yield the Cyber-205:

1. In the Star-100, the operation of the scalar unit was coupled with the vector unit such that only one type of operation could be performed at a time. The Cyber-205 has the ability to run both vector unit and scalar unit in parallel.
2. To fit a variety of operating environments, the Cyber-200 family was provided with a range of small page sizes beginning with 4096 bytes and ending with 65,536 bytes. The large page size of the Star-100 was retained since it appeared to be optimum for large production programs.
3. The input-output system employed in the Star-100 was of the "star network" type, with node-to-node communication between the CPU and attached peripherals. The change to a network form of I/O which is called the *loosely coupled network* (LCN), was a major switch for hardware and software alike on the Cyber-205.

Figure 4.31 illustrates the star network connection of the Star-100 and attached peripherals and contrasts this with the Cyber-205. Note that the connectivity of the Cyber-205 is potentially much greater than that of its predecessor. In addition, data transfers between elements of the Cyber-205 system can bypass the front-end elements; in the Star-100, data rates between permanent file storage and the CPU are limited by the capacity of the front-end processor, which typically is on the order of 1 to 4 Mbits/s.

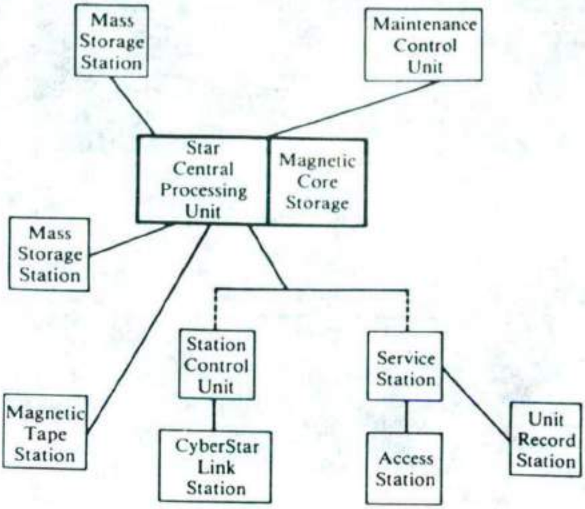
Transmission of data is accomplished on a high-speed, bit-serial trunk to which 2 to 16 system elements can be coupled. The method of establishing a link is based on addresses in the serial message which can be recognized by the hardware trunk coupler, called the *network access device* (NAD). The most significant



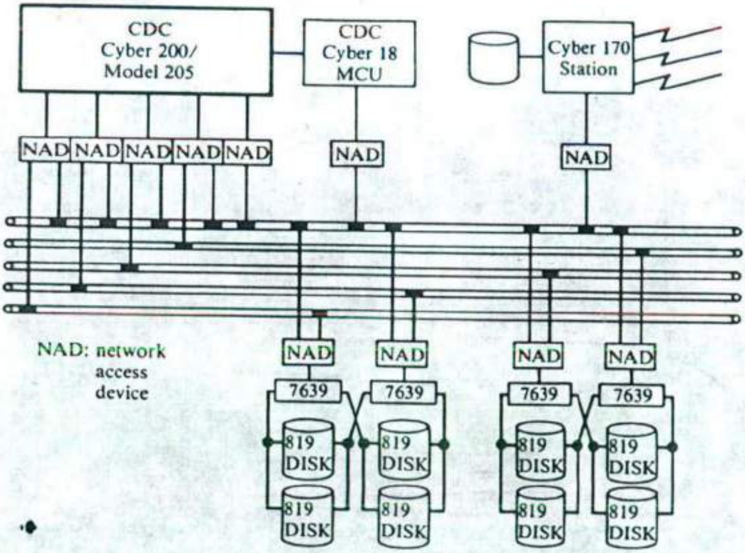
(*a*) Linked triad performance

(*b*) Multiply-add performance

Figure 4.30 Effect of startup del on the Cyber-205 performanc (Courtesy of *IEEE Trans. Co puters*, Lincoln 1982.)



(*a*) The "star-network" for Star-100



(*b*) The "loosely coupled network" for Cyber-205.

Figure 4.31 Input-output subsystem configurations in the Star-100 and the Cyber-205. (Courtesy of *IEEE Trans. Computers*, Lincoln 1982.)

aspect of this decision has been the philosophical departure from dedicated peripherals to shared peripherals, which are accessed on a "party line" basis. Once this change has been incorporated in the system software, the actual transmission media and hardware form of NAD is invisible to the user.

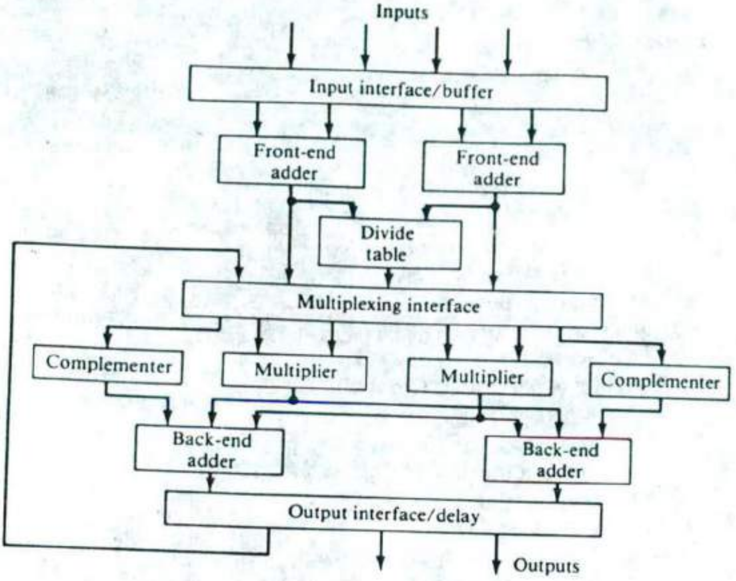
In 1979, Control Data Corporation proposed to the NASA Ames Research Center a supercomputer design, called the *Numerical Aerodynamic Simulation Facility* (NASF), to be used in the 1990s for aerospace vehicle or superjet designs. The purpose is to provide predictive three-dimensional modeling of the wind tunnel experiments characterized by viscous Navier-Stokes fluid equations. This computational approach to solve fluid dynamic problems is only constrained by processor speed and memory space. It costs much less than building a huge wind tunnel, which is limited by so many physical factors. The speed requirement of the NASF was set to be at least 1000 megaflops. Feasibility has been established and U.S. government funding is being awaited before proceeding with the design and construction of such a supercomputer.

The CDC/NASF design extends the structure of the Cyber-205, as shown in Figure 4.32*a*. There are five vector pipelines in NASF, with one serving as a spare unit. Functional components in one vector pipeline are shown in Figure 4.32*b*. A separate scalar processor is used. The clock rate of this proposed design is 8 ns. The memory hierarchy has three levels: an 8 M word of ECL cache, a 32 M word MOS intermediate memory, and a CCD sequential memory of 128 M words. Within each vector pipeline, adders, multipliers and complementers are all duplicated in pairs to facilitate parallel real or complex number calculations and error checkings. The spare vector pipe can be switched in automatically whenever a failure is detected. This allows on-line repairing of the failing unit.

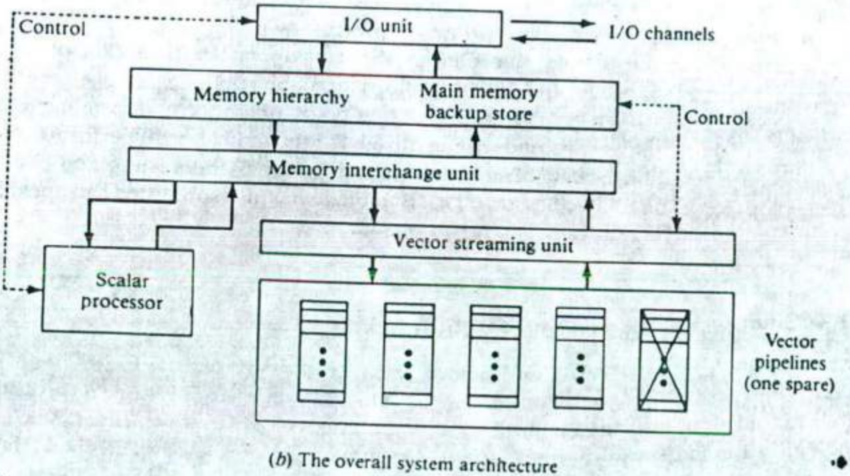
With an 8-ns clock rate, the CDC/NASF can operate with a rate of 500 megaflops for 64-bit results and 1000 megaflops for 32-bit results. Since each result may be produced with one to three floating-point operations, depending on whether real or complex operands are involved, the theoretical peak performance of the NASF should be tripled, as 1500 megaflops for 64-bit results and 3000 megaflops for 32-bit results. Besides CDC, Burroughs has also submitted a proposal to build the NASF as an SIMD array processor.

4.4.5 Fujitsu VP-200 and Special Features

Fujitsu announced the FACOM vector processors VP-100 and VP-200 in July 1982. High performance in this machine is achieved by LSI technology, improved architecture, a sophisticated compiler, and a number of advanced features in both the hardware and software areas. It can be used as a loosely coupled back-end system. The block diagram of the VP-200 is shown in Figure 4.33. The system has a scalar processor and a vector processor which can operate concurrently. Like in the Cray-1, large and fast registers, buffers, and multiple pipes are used to enable register-to-register operations. The main memory has up to 256 M bytes connected to the vector registers via two load-store pipelines. Each of the two load-store pipes



(*a*) Functional design of one vector pipeline



(*b*) The overall system archhecture

Figure 4.32 The proposed CDC NASF supercomputer. (Courtesy of Control Data Corp.)

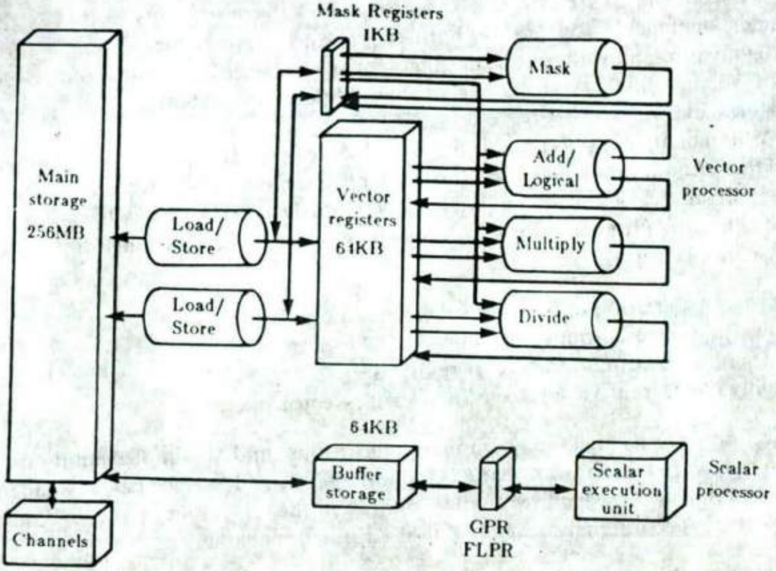


Figure 4.33 The FACOM vector process VP-200. (Courtesy of Fujitsu Limited, Japan.)

has a data bandwidth of 267 M words in either direction. This rate matches the maximum throughput of the arithmetic pipes. There are four execution pipelines in the vector processor. Data format for vector instructions can be bit strings, 32-bit fixed-point, and 32- or 64-bit floating-point operands. There are 83 vector instructions and 195 scalar instructions in the VP-200. Most of the scalar instructions are IBM 370 compatible and the scalar unit interfaces with the main memory via larger buffer storage.

In the VP-200, the vector registers can be dynamically reconfigured by concatenation to assume variable lengths up to 1024 words. Vector instructions include *vector compare*, *masking*, *compress*, *expand*, *macros*, and *controls*, in addition to arithmetic logic operations. Concurrent operations include two load-store, mask, two out of three arithmetic pipes, and scalar operations. The throughput of the *add-logical* pipe and the *multiply* pipe is 267 megaflops each, whereas that of the *divide* pipe is 38 megaflops. Hence 533 megaflops is the maximum throughput when the add-logical and the multiply pipes run concurrently. The VP-200 has advanced optimization facilities to generate efficient object code through vectorization of sequential constructs, pipeline parallelization, vector-register allocation, and generation optimization.

The strength of the VP-200 lies in its impressive throughput and easy programming environment. The Fortran 77 compiler is developed with advanced automatic optimization feature and convenient tuning tools and application library. The system can be a simple add-on to a front-end processor with MVS/OS as a loosely coupled multiprocessor. It can utilize many of the existing software assets. With more reliable circuit and packaging technology, the system can self-recover from hardware errors. The Fortran 77/VP components in the VP-200 include language processors for both scalar and vector objects, tuning tools, debugging tools, and subroutine packages for special scientific computations, such as for solving linear systems, eigenvalues, differential equations, fast Fourier transform, etc.

Although high-speed components, high degree of parallelism and/or pipelining, and large-sized main memory are the basic requirements to stretch the computational capabilities of a supercomputer, the VP-200 designers consider the following features as also important for such a machine to be versatile enough for a wide range of applications:

1. Efficient processing of DO loops which contain IF statements
2. Powerful vector editing capabilities
3. Efficient utilization of the vector registers
4. Highly concurrent vector-vector and scalar-vector operations

Technology, architecture, compiling algorithms, and the implementation of special hardware and software features in the VP-200 are described below. Readers may find some of these features similar to those in the Cray-1 and Cyber-205, and some features are uniquely developed in the Fujitsu machine.

Technologies utilized Fujitsu's latest technologies are utilized in the FACOM vector processor. Logic LSIs contain 400 gates per chip, with some special functional chips such as the register files containing 1300 gates. Signal propagation delay per gate of these LSIs are 350 picoseconds (ps). Memory LSIs containing 4K bits per module with an access time of 5.5 ns are used where extremely high speed is necessary. Up to 121 LSIs can be mounted on a 14-layered printed circuit board called MCC (multichip carrier). Forced air cooling is employed throughout the system. These are well-proven technologies with Fujitsu's FACOM M-380 mainframes. It is with these technologies that a 7.5-ns clock is realized for the vector unit and a 15-ns clock is realized for the scalar unit. As for the main memory, 64K-bit MOS static RAM LSIs are used with chip access time of 55 ns. The main memory is 256-way interleaved. Vectors can be accessed in contiguous, constant-strided, and indirect addressed fashion.

Vectorizing compiler—Fortran 77/VP A vectorizing compiler, Fortran 77/VP, has been developed for the FACOM vector processor. Fortran 77 has been chosen as the language for this machine so that the large software assets can become readily available. In order to obtain high vectorization ratio for a wide range of application programs, the Fortran 77/VP compiler vectorizes not only the simple DO loops but nested DO loops and the macro operations such as the inner product efficiently. It also detects and separates the recurrences.

Ease of use is another objective of the compiler. Debugging aids, a performance analyzer, an interactive vectorizer, and a vectorized version of the scientific subroutine library are some of the software included in Fortran 77/VP. With the interactive feature, for example, a programmer can provide the compiler with useful information for higher vectorization.

Conditional vector operations The analysis of the application programs indicates that the conditional statements are frequently encountered within DO loops. The FACOM vector processor provides three different methods to efficiently execute conditional branch operations for vectors: masked arithmetic operations, compress-expand functions, and vector indirect addressing.

In order to control conditional vector operations and vector editing functions, bit strings (called mask vectors) are also provided. A total of 256 mask registers, 32 bits each, the store mask vectors, and the *mask pipe* perform logical operations associated with the mask vectors.

Example 4.8 This example describes the *masked* operations in VP-200:

1	1	0	1	0	0	1	1	Mask vector
A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	Vector A
B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	Vector B
+	+		+			+	+	Add operation
A_1	A_2	C_3	A_4	C_5	C_6	A_7	A_8	Result on vector C
+	+		+			+	+	
B_1	B_2		B_4			B_7	B_8	

In the case of the masked arithmetic operations, the add-logical pipe generates mask vectors to indicate the true-false values of conditional statements, and the arithmetic pipeline units take such mask vectors via mask registers as the control inputs. The arithmetic pipeline units store the results back to vector registers only for the elements having 1s in the corresponding locations of the mask vector: Old values are retained otherwise. For a given vector length, the execution time of a masked arithmetic operation is constant regardless of the ratio of true values to the vector length (called "true ratio").

Since the true ratio of conditional statements widely varies from one case to another, the compiler must select the best method for individual cases. More precisely, there are two key parameters in selecting the best one to use: the *true ratio* and the *relative frequency of load-store instruction executions* over the total instruction executions within the DO loop. If the true ratio is medium to high, the masked arithmetic operation is selected; otherwise, the compress-expand method is selected when the frequency of load-store operations is low, and the indirect addressing is selected when such frequency is high. The Fortran 77/VP compiler analyzes the DO loop, compares the estimated execution times for all three methods, and selects the best one. A programmer can also interactively provide the compiler with the information on the true ratio.

Vector editing functions The FACOM vector processor provides two types of editing functions: compress-expand operations and vector indirect addressing.

These functions can be used not only for the conditional vector operations but for sparse matrix computations and other data editing applications. Vectors on vector registers can be edited by compress and expand functions by using load-store pipes as data alignment circuits; no access to the main memory is involved in these cases. Compressing a vector A means that the elements of A marked with 1s in the corresponding locations of the mask vector are copied into another vector B, where these elements are stored in contiguous locations with their order preserved. Expanding a vector means the opposite operation, as the example below shows:

Example 4.9 This example describes the *compress* operation in VP-200.

1	1	0	1	0	0	1	1	Masked vector
A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	Vector A
A_1	A_2	A_4	A_7	A_8				Compress A

In the vector indirect addressing, a vector J on vector registers holds the indices for the elements of another vector A stored in the main memory, which is to be loaded into vector C defined on vector registers. Namely, $C(I) = A(J(I))$. This is a very versatile and powerful operation, since the order of the elements can be scrambled in any manner. The data transfer rate for vector indirect addressing, however, is lower than that for the contiguous vectors, due to possible bank and/or bus conflicts.

Example 4.10 This example describes the *vector indirect addressing* in VP-200.

1	1	0	1	0	0	1	1	Mask vector
1	2	4	7	8				List vector generation
A_1	A_2	A_4	A_7	A_8				Indirect vector load A

Vector compress-expand operations are performed in two steps: mask vector generation and actual operations. Frequently used mask patterns may be stored in mask registers. The compiler also checks whether the domains occupied by the source vector and the destination vector overlap. The indirect addressing, on the other hand, takes three steps: mask vector generation, index list vector generation from the mask vector, and loading the vector from the main memory. Frequently used list vectors may also be stored in the main memory.

Vector registers optimization One of the most unusual features of the FACOM vector processor is the dynamically configurable vector registers. The concept of vector register is very important for efficient vector processing, since it drastically reduces the frequency of accesses to the main memory. The results of our study indicate that the requirements for the length and the number of vectors vary from one program to another. To make the best utilization of the total size of 64K bytes, the vector registers can be concatenated to take the following configurations: 32 (length) × 256 (vector counts), 64 × 128, 128 × 64, . . . , 1024 × 8. The length of vector registers is specified by a special hardware register, and it can be altered by an instruction in the program.

The compiler must know the frequently used hardware vector length for each program, or even within one program the vector length may have to be adjusted. When the vector length is too short, load-store instructions will be issued more frequently, whereas if it is unnecessarily long, the number of available vectors will be small and vector registers will be wasted. As a general strategy, the compiler puts a higher priority on the number of vectors in determining the register configuration. A programmer can also interactively provide the compiler with the information on the vector length.

High-level concurrency The FACOM vector processor allows concurrent operations at different levels. In the vector unit, five functional pipelines can operate concurrently: two out of three arithmetic pipeline units, two load-store pipes, and mask pipe. Within each arithmetic pipeline unit, vector operands associated with consecutive instructions can flow continuously without flushing the pipe.

The vector unit and the scalar unit can also operate concurrently, as illustrated in Figure 4.34. Without such a feature, the scalar operations between the vector operations could cause considerable performance degradation. Serialization instructions are provided to preserve the data dependency relations among instructions.

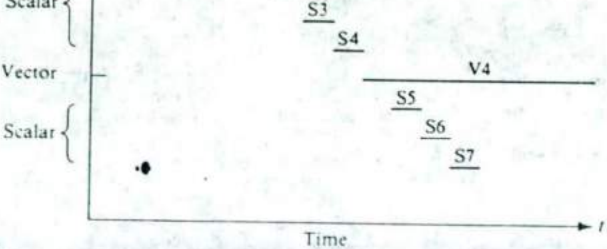


Figure 4.34 Concurrent processing of vector and scalar instructions in VP-200. (Courtesy of Fujitsu Limited, Japan.)

The compiler performs the extensive data-flow analysis of the Fortran source programs and schedules the instruction stream, so that the vector arithmetic pipeline units are kept as busy as possible. This process includes the reordering of instruction sequence, balanced assignments of two load-store pipes, and insertion of serialization instructions, wherever necessary.

A comparison of the modern pipeline vector supercomputers that we have studied is given in Table 4.14. This table summarizes the instruction repertoire, basic system specifications, functional pipes, vector registers, main memory, peak CPU speed, vectorizing facilities, front-end host computers, and possible future extensions for the Cray-1, the Cyber-205, and the VP-200. The option of one vector processor in the Cyber-205 is assumed in the comparison. With the introduction of the Cray X-MP and the options of having two or four vector processors in a Cyber-205, one can conclude that the three vector supercomputers have essentially the same computing power. It is interesting to watch for their future upgraded models.

Table 4.14 Comparison of three pipeline vector supercomputers

Architecture and capability	Cray Research's Cray-1	Control Data's Cyber-205 (1 vector processor)	Fujitsu's VP-200
Organization	Register-to-register	Memory-to-memory	Register-to-register
Instruction repertoire	128 instructions, 10 vector types 13 scalar types	256 instructions, 16 types 10 scalar types	83 vector and 195 scalar instructions
Functional pipelines, pipeline cycle	12 pipelines: 3 vector, 4 scalar, 2 floating point, and 2 integer, clock period: 12.5 ns	11 pipelines: 6 vector and 5 scalar functions, clock period: 20 ns	6 pipelines: add-logical, multiply, divide, mask, and two load-store, clock period 7.5 ns
Vector registers, main memory	8 × 64 × 8 = 4K bytes, 32 Mbytes	Vector register unused, 32 Mbyte main memory	32 × 256 × 8 = 64K bytes, 256 Mbytes
Peak CPU speed	160 mflops	200 mflops	500 mflops
Vectorizing facilities	Cray Fortran compiler (CFT) with automatic vectorization by user intervention	Vector arithmetic with automatic vectorization	Fortran 77/VP compiler with interactive vectorizer
Front-end host computers	IBM/MVS, CDC/NOS, Univac	Cyber-170 series IBM 303x	FACOM M series IBM/MVS
Possible future extended models	Cray X-MP with 420 megaflops, Cray-2 with 2000 megaflops	CDC Cyber 2XX with over 10 gegaflops	(unknown)

In the next 10 years (1984 to 1994), a new generation of supercomputer will emerge, driven by customer demands and other competitive pressures. Memory capacities will be 2 to 4 times greater than now possible, and processing speeds will be improved from 2.5 to 20 times current rates. These goals will be achieved by employing another generation of device technology, with emphasis on larger scales of integration.

The major element of this next generation will be continued evolution of architectures involving parallelism and the incorporation of many artificial intelligence functions and more intelligent I/O interfaces. Vector processors, multiprocessors, and similar parallel structures will be required to keep pace with the demand for computational power. The exploitation of these architectures through new algorithms, operating systems, and compiler technology is the key to achieving the goals of the next generation consumers.

4.5 VECTORIZATION AND OPTIMIZATION METHODS

In this section, we study four issues towards performance enhancement of vector processors. We begin with an introduction of parallel language features for vector processing. Parallel constructs in extending high-level languages are described by examples, instead of abstract declarations. The design phases of a vectorizing compiler for generating vector codes from sequentially written source codes are characterized with various vectorizing facilities. Then we study various optimization methods to generate efficient object code. Finally, analytical tools for evaluating the performance of a pipeline computer are presented.

4.5.1 Parallel Languages for Vector Processing

In recent years, substantial efforts are being exerted on developing high-level languages with parallel constructs to facilitate vector processing. As discussed in Section 3.5.1, the use of sequential languages will lose the parallelism specified in a good algorithm. Thus vectorization (Figure 3.28) is highly needed to restore the concurrency in parallel algorithms so that they can be efficiently implemented on a vector processor. Most commercial vector processors have built-in hardware to support extended high-level languages, like the extended Fortran in the Cray-1, and the Fortran 77 extension in the FACOM VP-200.

Two vector processing languages have been proposed recently: one is the *Actus* by Perrott (1979) and the other is the *Vectran* by Paul and Wilson (1975). Unfortunately, neither of these parallel languages has been successfully tested on a real machine. Parallel languages are far from being standardized. The desired features include *flexibility* in declaring and selecting array objects in columns, rows, blocks, diagonals, and in various subarray expressions; *effectiveness* in manipulating sparse and dense matrices; *array conformity* to allow transportability; and mechanisms to break *vectorization barriers*.

Of course, the usefulness of a new language depends on its application areas We briefly characterize below, by Fortran extension examples, some of th attractive features in a typical parallel language. A vector may be identifiec implicitly by the appearance of an array name followed by specific subscripts. Th extended notation may be specified through an implied DO notation as follows:

$$\begin{array}{c} e_1{:}e_2{:}e_3 \\ e_1{:}e_2 \\ * \\ e_1{:}*{:}e_3 \end{array} \qquad (4.3$$

where e_1, e_2, and e_3 are expressions of indexing parameters as they appear in a DO statement: e_1 indicates the first element or the *initial index* value, e_2 indicates the *terminal index* value, and e_3 is the *index increment* or skip distance. If e_3 is omitted, the increment is one; this includes all of the elements from e_1 to e_2. The single symbol "*" indicates that all of the elements are in a particular dimension. If the elements are to be used in reverse order, the notation "-*" may be used.

Example 4.11 *Given*: DIMENSION X(8), Y(10, 4)
Then: X(2:8:2) represents the elements X(2), X(4), X(6), X(8);
Y(3:5, 3) represents the elements Y(3,3), Y(4,3), Y(5,3);
Y(*,3) represents the third column of the matrix Y;
Y(5,2:*) represents the elements Y(5,2), Y(5,3), Y(5,4).

A vector statement allows different portions of an array to be identified explicitly by separate names. No extra storage is allocated for an identified vector. Each identified vector is simply a virtual name for a collection of elements in the original vector.

Example 4.12 *Given*: REAL X(10,10)
Then: VECTOR X ROW 2(1:10) is a vector consisting of the second row of X;
VECTOR X DIAG(1:10) represents the diagonal elements of X;
VECTOR X COL 5(1:10:3) is a vector consisting of X(1,5), X(4,5), X(7,5)

In a binary vector operation, the two operands must have equal length with only a few exceptions. Each vector operation may be associated with a logical array that serves as a control vector. A WHERE statement may allow the programmer to indicate the assignment statements to be executed under the control of a logical array. The following PACK and UNPACK operations demonstrate the use of control vectors.

Example 4.13 *Given*: DIMENSION A(6), B(6), C(6); DATA A/−3, −2,1,3, −2,5/
Then: PACK WHERE (A .GT. 0) B=C causes elements of C in positions corresponding to "trues" in A.GT.0 to be assigned to B elements such that B(1) = C(3), B(2) = C(4), B(3) = C(6);

UNPACK WHERE (A.GT.0) A = B inserts the elements of B into A in positions indicated by A.GT.0. Thus, A(3) = B(1), A(4) = B(2), A(6) = B(3).

An intrinsic function needs to compute with each element of a vector operand. For example A(1:10) = SIN(B(1:10)) is a vector intrinsic function. Several special vector instructions are shown in the following example:

Example 4.14 *Given*: DIMENSION A(50), B(50), C(50)
Then: C(2:9) = VADD(A(2:9),B(1,8)) performs the *vector addition*;
S = SIZE(A(1:50:4)) equals the *length* of the sparse vector A(1:50·4)
S = DOTPD(A,B) forms the *dot product* of vectors A and B;
S = MAXVAL(A) finds the *largest value* of vector A.

A *Fortran vectorizer* has the capability of detecting parallelism in serially coded Fortran programs. It recognizes Fortran constructs that can be executed in parallel. Basic operations performed by the vectorizer program are precedence analysis and code generation. In performing the analysis, the vectorizer analyzes Fortran instruction sequences for possible translation into a vector syntax. This phase is extremely machine dependent since it must consider special characteristics of the hardware. An ideal vectorizer performs sophisticated analysis of data dependencies and determines the possibility of vectorization. General guidelines in designing a vectorizer include:

1. Determining the flow pattern between subprograms
2. Checking the precedence relationship among the subprograms
3. Checking the locality of variables
4. Determining the loop variables
5. Checking the independence of variables
6. Replacing the inner loop with vector instructions

Described below are six examples for converting conventional Fortran statements into vectorized codes, presumably by a vectorizing compiler.

Example 4.15 A simple DO loop containing independent instructions and no branch statements can be converted to a single vector instruction. The following DO-loop statements:

```
   DO 20 1=8,120,2
20 A(I)=B(I+3)+C(I+1)
```

are being converted into a single vector statement:

```
A(8:120:2)=B(11:123:2)+C(9:121:2)          (4.4)
```

Example 4.16 A recurrence computation can be converted into vector form, subject only to precedence constraint. The recursion

```
      A(0)=X
      DO 20 I=1,N
   20 A(I)=A(I-1)*B(I)+C(I+1)
```

is being converted to be:

```
A(0)=X
A(1:N)=A(0:N:1)*B(1:N)+C(2:N+1)            (4.5)
```

Example 4.17 An IF statement in a loop can be eliminated by setting a corresponding control vector together with a WHERE statement, such as converting

```
      DO 20 I=1,N
   20 IF(L(I).NE.0) A(I)=A(I)-1
```

to

```
WHERE (L(I).NE.0) A(1:N)=A(1:N)-1           (4.6)
```

Example 4.18 Exchanging the execution sequence sometimes will enable parallel computations, such as converting

```
      DO 20 I=1,N
      A(I)=B(I-1)
   20 B(I)=2*B(I)
```

to the following code:

```
B(1:N)=2*B(1:N)
A(1:N)=B(0:N-1)                             (4.7)
```

Example 4.19 Temporary storage can be used to enable parallel computations, such as converting the statements

```
      DO 20 I=1,N
      A(I)=B(I)+C(I)
   20 B(I)=2*A(I+1)
```

to vector code

```
TEMP (1:N)=A(2:N+1)
A(1:N)=B(1:N)+C(1:N)                        (4.8)
B(1:N)=2*TEMP(1:N)
```

Example 4.20 Prolonging the vector length is always desirable for pipeline processing. The two levels of array computations

```
   DO 20 I=1,80
   DO 20 J=1,10
20 A(I,J)=B(I,J)+C(I,J)
```

can be rearranged to promote better pipelining:

```
   DO 20 J=1,10
   DO 20 I=1,80
20 A(I,J)=B(I,J)+C(I,J)
```

Other techniques such as register allocation, vector hazard, and instructions rearrangement are also machine dependent. For example, we want to allocate the vector registers in the Cray-1 to result in minimal execution time. Rearranging the execution sequence to execute the same vector operations repeatedly can reduce the pipeline reconfiguration overhead in a multifunctional pipe. A vectorizer informs the programmer of the possibility of parallel operations. It provides also a learning tool, in that the programmer can examine the output of the vectorizer and tune the computations for better pipelining. Automatic vectorization and code optimization will increase the programming productivity on vector processors.

4.5.2 Design of a Vectorizing Compiler

A vectorizing compiler analyzes whether statements in DO loops can be executed in parallel and generates object codes with vector instructions. The higher the vectorization ratio, the higher will be the performance. To achieve this, the compiler vectorizes complicated data accesses and restructures program sequences, subject to machine hardware constraints. Barriers to vectorization exist in conditional and branch statements, sequential dependencies, nonlinear and indirect indexing, and subroutine calls within loops. In this subsection, we outline the design considerations of a vectorizing compiler and illustrate some vectorization techniques.

Let us first review the phases of a conventional Fortran compiler. The first is a lexical-scan and *syntax-parsing* phase which converts the source program to some intermediate code, quadruples, for example. Quadruples usually have fields for specifying the operations, up to two source operands, a result operand, and auxiliary information; the auxiliary field will be used by the next two phases of *code optimization* and *code generation*. The purpose is to specify which operands are found in registers, which registers are occupied, whether the instruction should be deleted, and so on. For an example, the lexical parser converts a Fortran statement $A = A + B * C$ to the following quadruples:

$$T1 \leftarrow B * C$$
$$A \leftarrow A + T1$$

where T1 is a compiler generated temporary identifier.

In the second phase for optimization, the compiler accepts quadruples as inputs and produces modified quadruples as outputs. The optimizer might find that the subexpression B * C is redundant, one whose value had been previously computed and called as identifier T2. The multiplication can thus be eliminated:

```
T1←T2
A←A+T1
```

The optimizer may also discover that T1 will never be used again, so all of its use can be directly replaced by the original T2 and its definition be eliminated, resulting in the single statement A ← A + T2. A register allocator attempts to assign heavily used quantities to CPU registers. If it is discovered that A is frequently used, A will probably be assigned to a register, say R3, and will be recorded as auxiliary information:

```
A(R3)←A(R3)+T2
```

In the third phase, the code generator translates the final intermediate code into a machine-language program based on the auxiliary information. A code fragment is selected to represent the quadruple in machine code. The code fragment, along with all other fragments and some initializing code, would be written in linkage-editor format. In the above example, machine addresses have been assigned to identifiers A and T2. Thus we can write the quadruple in a machine code:

```
ADD R3, M(T2)
```

where M(T2) refers to the memory location of the identifier T2.

In a vectorizing Fortran compiler, the scanner and parser need not be modified. With the scalar code already in place, we want to convert a series of scalar operations into vector code. Let us consider some optimization techniques which can extend the intermediate code. Since the optimization techniques are machine dependent, we will consider a Cray-like structure in which all the vector operations are register-to-register.

A vectorizing compiler will analyze the structure of Fortran programs being compiled. We have seen many examples in previous sections on converting DO loops and other scalar operations in the source program to vector instructions in the object code. In general, the higher the *vectorizing ratio*, the better will be the performance. This is due to the fact that vector speed is much higher than scalar speed in a vector processor. As illustrated in Figure 4.35, we assume the speed ratio of vector to scalar operations to be 50. The shaded areas correspond to reduced execution time for vector instructions; one vector process is reduced from 50 scalar processes as shown. Two levels of vectorization are shown in the figure.

Simple scalar operations like inner product, random-access integer operations, and simple DO arithmetic can be vectorized easily at the first level for a vectorizing ratio of 50 percent. The remaining complicated operations like scatter, conditional statements, gather and others can only be vectorized by a very intelligent

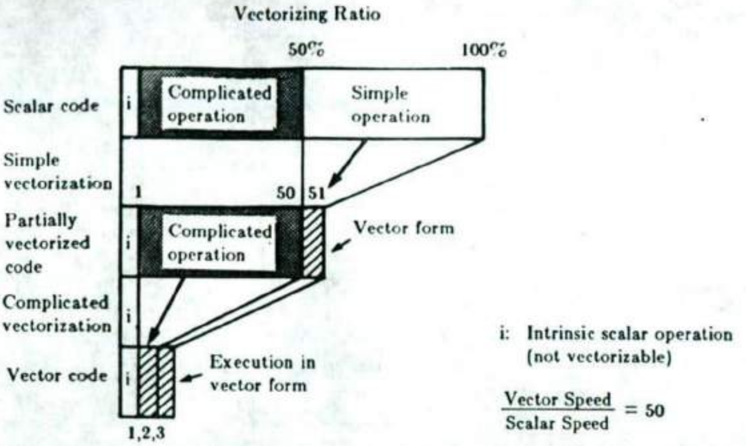


Figure 4.35 Vectorization ratio and saving in execution time. (Courtesy of Fujitsu Limited, Japan.)

compiler which can efficiently access complicated data structures and tune branch-disturbed program structures. The compiler requires sophisticated optimization techniques and improved hardware architecture. Major optimizing functions are listed below according to the levels of sophistication in generating efficient scalar and vector code modules.

1. General optimization
   (*a*) Common expression elimination
   (*b*) Invariant expression movement
   (*c*) Strength reduction
   (*d*) Register optimization
   (*e*) Constant folding
2. Extended optimization
   (*a*) Intrinsic function integration such as SQRT, SINE, etc.
   (*b*) User Fortran subprogram integration
   (*c*) Reductions of iteration numbers in nested DO loops
   (*d*) Reorder of execution sequence to reduce pipeline overhead
   (*e*) Temporal storage management
   (*f*) Code avoidance
3. Vector extended optimization
   (*a*) Full vectorization
   (*b*) Pipeline chaining
   (*c*) Pipeline antichaining
   (*d*) Vector register optimization
   (*e*) Parallelization

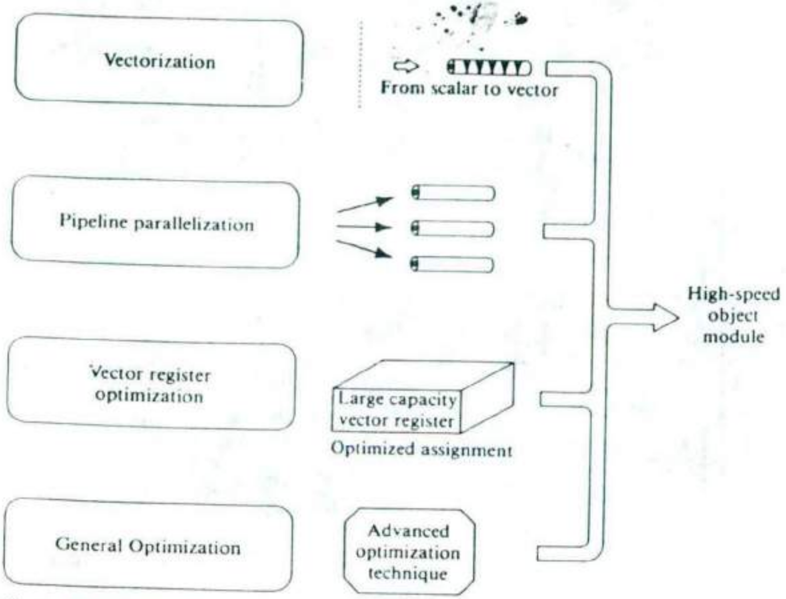


Figure 4.36 Vectorizing compiler design techniques in the Fujitsu FACOM VP-200. (Courtesy of Fujitsu Limited, Japan.)

The *general optimization* and *extended optimization* contribute to the generation of efficient scalar code. The *vector extended optimization* is added to produce fast vector instructions. The three levels of optimization features are demonstrated in Figure 4.36. The *vectorization* converts scalar to vector. The *parallelization* utilizes multiple pipes simultaneously. The vector *register optimization* allocates large-capacity registers properly. The *general optimization* can be achieved with many of the existing compiler assets. The above optimizing features have been implemented in vectorizer packages in most commercial supercomputers.

4.5.3 Optimization of Vector Functions

Described below are nine common practices in optimizing the vector functions. Some of these techniques apply not only to pipeline computers but also to array processors. Tuning tools for an interactive vectorizer are then introduced. Again, we use some examples to illustrate various optimization functions and tuning facilities.

(A) Redundant expression elimination After the scan and parse phase, some redundant operations in the intermediate code could be eliminated. The number of memory accesses and the execution time can be reduced by often eliminating redundant expressions.

Example 4.21 Consider the following extended Fortran code:

```
DIMENSION A(100), B(50), C(50)
C(1:50)=A(1:99:2)*B(1:50)+A(1:99:2)*C(1:50)
```

A possible set of intermediate code is generated on the left below. With elimination of redundant statements, the simplified code is generated on the right below, if the compiler detected the fact that the B array and C array are identical.

```
V1←A(1:99:2)
V2←B(1:50)
V3←V1*V2                 V1←A(1:99:2)
V4←A(1:99:2)             V2←B(1:50)
V5←C(1:50)       →       V1←V1*V2                (4.9)
V6←V4*V5                 V1←2*V1
V7←V3+V6                 C(1:50)←V1
C(1:50)←V7
```

(B) Constant folding at compile time In its full generality, constant folding means shifting computations from run time to compile time. Although the opportunities to perform operations on constant arrays is not often, such opportunities will crop up occasionally, particularly as initialized tables. For example, a DO loop for generating the array A(I) = I for I = 1, 2, . . . , 100 can be avoided in the execution phase because the array A(I) can be generated by a constant vector (1, 2, 3, . . . , 100) initialized in the compile time.

(C) Invariant expression movement The innermost loops are often just scalar encodings of vector operations. In such a case, the innermost loop can be replaced by some vector operations if there is no data dependence relation or no loop variance. If some of the statements are loop-variant, the loop-invariant expressions may be moved out of the loop. This is called *code motion*. Consider the following two-level DO loop:

```
DO 20 I=1,N
   ...
DO 20 J=1,M
   ...
B(I,J)=B(I,J)+A(J)*C(J)
20 CONTINUE
```

Assuming no other computations affecting the variables A, B, C, I and J, the above program can be vectorized as follows:

```
DO 20 I=1,N
   ...
B(I,*)=B(I,*)+A(*)*C(*)
DO 20 J=1,M
   ...
20 CONTINUE
```

The vector multiplication can be moved out of the outer loop using a temporary vector T(*):

```
T(*)=A(*)*C(*)
DO 20 I=1,N
   ...
B(I,*)=B(I,*)+T(*)
DO 20 J=1,M
   ...
20 CONTINUE
```

(D) Pipeline chaining and parallelization In a vector processor with multiple functional pipes, the performance can be upgraded by chaining several pipelines. The result from one pipeline may be directed as input to another pipeline. The [illegible] delays due to storing intermediate results are thus eliminated. The intermediate results need not be stored back into the memory with the chaining. An intelligent compiler should have the capability of detecting sequences of operations that can be chained together. We have seen some chaining operations of multiple pipelines in Section 3.4.1. Figure 4.37 shows the parallel use of two load-store pipes which are chained with the *multiply pipe* and then the *add pipe* in the VP-200.

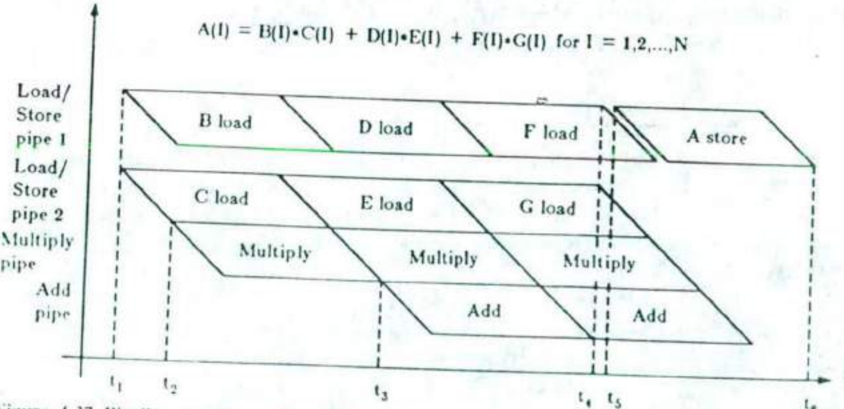


Figure 4.37 Pipeline chaining and parallelization for linked vector operations. (Courtesy of Fujitsu Limited, Japan.)

Example 4.22 Based on the hardware architecture of the VP-200 in Figure 4.33, we can implement the following vector operations, as demonstrated in Figure 4.37.

$$A(I)=B(I)*C(I)+D(I)*C(I)+F(I)*G(I) \quad \text{for } I=1,2,\ldots, N \tag{4.10}$$

Three pairs of *vector load* operations, (B, C), (D, E), and (F, G), are done in two load-store pipelines. Three *vector multiply* operations, B * C, D * E, and F * G, are carried out in the multiply pipeline from time t_2. The two *vector add* operations are executed in the add pipeline from time t_3. The first result becomes available at t_4. With a minor pipeline reconfiguration delay, the *store* operation begins at t_5. The entire operation requires $t_6 - t_1 = 4N + \Delta_1$ where $\Delta_1 = t_2 - t_1$ is the delay of one pipeline. It was assumed that all pipes have equal delays.

(E) Vector register allocation On a machine like the Cray-1 allowing the chaining of vector operations, the importance of register allocation cannot be overstated. To achieve the maximum computing power, the execution units must be fed a continuous stream of operands. Retaining vectors in registers between operations is one way to achieve this. However, the strategy of vector register allocation emphasizes local allocation rather than global allocation, because of the limited number of available vector registers in a system.

Example 4.23 The vector expression A − B * C can be executed in the Cray-1 by using the three vector registers V1, V2, and V3 as follows:

$$\begin{aligned}&V1 \leftarrow A\\&V2 \leftarrow B\\&V3 \leftarrow C\\&V2 \leftarrow V2 * V3\\&V1 \leftarrow V1 - V2\end{aligned}$$

If we do the multiply before loading the vector A into a register, only two vector registers are required:

$$\begin{aligned}&V1 \leftarrow B\\&V2 \leftarrow C\\&V1 \leftarrow V1 * V2\\&V2 \leftarrow A\\&V2 \leftarrow V2 - V1\end{aligned}$$

This sequence of five instructions requires 31 clock periods on the Cray-1, as opposed to 26 clock periods for the sequence using three vector registers.

In general, an optimizing compiler will simulate the instruction timing and keep assigning intermediate results to new registers until a previously assigned register becomes free. In the absence of a careful timing simulation, the best strategy that a compiler can adopt is the "round-robin" allocation of registers. Another solution to the register allocation problem is to generate code for a group of arithmetic expressions using large numbers of virtual registers, then to schedule the resulting code to minimize issue delays and map the virtual registers into a finite number of real registers. This process has the danger of requiring more than the number of available registers, in which case some spilling must be done. Besides, moving instructions around to minimize issue delays may destroy the use of common subexpressions. As done in FACOM VP-200, register concatenation offers another approach to processing vectors of variable lengths.

(F) Reorder the execution sequence In a multifunction pipeline, reconfiguring the pipe for different functions requires the overhead of flushing the pipe, establishing new data paths, etc. Instructions of the same type may be grouped together for pipeline execution. The instructions on the left of the following code block can be regrouped to yield the sequence on the right:

```
A1←A2+A3          A1←A2+A3
A4←A5*A6          B1←A7+A8
B1←A7+A8    →     B5←C1+C2          (4.11)
B2←B3*B4          A4←A5*A6
B5←C1+C2          B2←B3*B4
```

The sequence on the left requires three pipeline reconfigurations, while the right one requires only one. An intelligent compiler should be able to reorder the execution sequence to minimize the number of required pipeline reconfigurations.

(G) Temporary storage management In the optimization phase of a vectorizing compiler, the generation of too many intermediate vector quantities can quickly lead to a serious problem. For example, the execution of the vector instruction A(1:4000) = A(1:4000)*B(1:4000) + C(1:4000) in the Cray-1 may need 63 vector registers (with 64 components each) to temporarily hold all intermediate product terms. The Cray-1 does not have this many vector registers to store all the intermediate results. Therefore, the intermediate results must be temporarily stored in the memory. For this reason, the compiler must allocate and deallocate temporary storage dynamically since vector registers may be thought of as temporaries and spillage out of them needs memory. Temporary storage management is closely related to the policy of register allocation. A vector loop can be used to solve the storage allocation problems. Since each vector register can handle 64 elements, we partition the 4000-component vectors into 64-element groups as follows:

```
    DO 20 K=1,4000,64
    L=MIN(K+63,4000)
    A(K:L)=A(K:L)*B(K:L)+C(K:L)
 20 CONTINUE
```

In any case, run-time storage management is an expensive feature to be included in a Fortran-based language. There are two techniques to reduce the design cost. First, if the compiler allocates a specific area to fixed-length and variable-length temporaries, the allocation of the fixed-length area can be done at compile time, eliminating some run-time overhead and permitting access of the temporaries by some generated codes. Second, the number of fixed-length temporaries can be increased by "strip mining" with a width equal to the length of vector registers in the target machine.

(H) Code avoidance A somewhat radical approach to the optimization of vector operations is based on the technique of copy optimization. The idea is to avoid excessive copying arrays unless forced to do so by the semantics of the language. Consider the following code sequence:

```
A(1:50)=B(1:99:2)
     ...
     ...
C(1:50)=2.0*A(1:50)
```

A copy is avoided (or at least delayed), if the compiler adjusts the storage-mapping function for array A to reference the storage for array B. Thus, instead of producing the following code:

```
    V1←B(1:99:2)
A(1:50)←V1
     ...
     ...
    V1←A(1:50)
    V1←2.0*V1
C(1:50)←V1;
```

it would generate the following simplified code:

```
    V1←B(1:99:2)
    V1←2.0*V1
C(1:50)←V1
```

(I) Tuning for interactive vectorization Tuning tools are necessary to provide some user interactions in advanced vectorization. Both the Cray-1 and the VP-200 have some tuning facilities. The tuning facility in the VP-200 is illustrated in Figure 4.38. From the displayed tuning information and vectorizing effects, the user can modify the source program with the help of an *interactive vectorizer* package. The modified source program will be optimized towards full vectorization by the vectorizing compiler. A number of compiler directive lines are useful to check whether recurrence appeared in the source code, the true ratio in IF statements, the vector length distribution, and others. The vectorizing compiler generates the

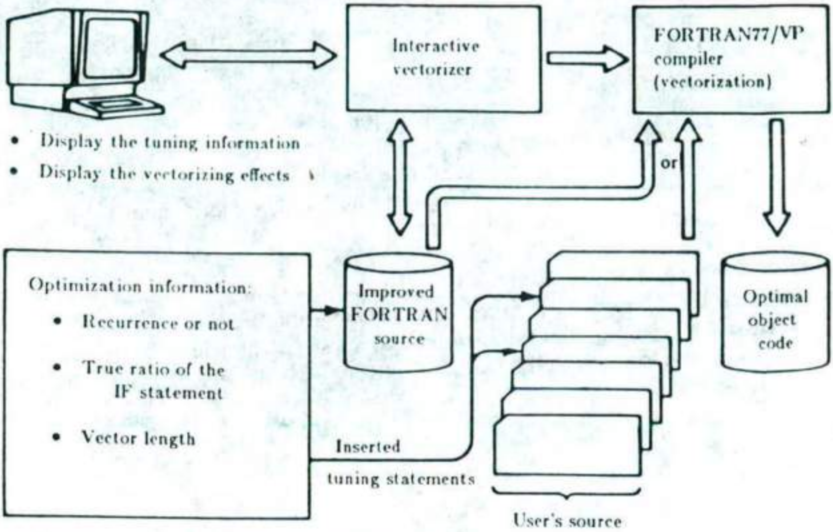

Interactive vectorizer
FORTRAN77/VP compiler (vectorization)
• Display the tuning information
• Display the vectorizing effects
or
Optimization information:
• Recurrence or not
• True ratio of the IF statement
• Vector length
Improved FORTRAN source
Optimal object code
Inserted tuning statements
User's source

Figure 4.38 The tuning facilities in FACOM VP-200. (Courtesy of Fujitsu Limited, Japan.)

optimal object code with vector instructions after the tuning process is completed. An example is given below to illustrate the tuning concept realized in the VP-200.

Example 4.24 Consider the following DO loop containing an IF statement:

```
   DO 20 I=1,N
      IF(A(I).GE.10) GO TO 20
      T=C1*A(I)+C2*B(I)
      A(I)=SIN(T)
20 CONTINUE
```

If one realizes the true ratio is 90 percent, the compiler chooses vector indirect addressing which results in several times better performance than the case without information for tuning. The compiler chooses masked arithmetic, if the optimization information is not available.

4.5.4 Performance Evaluation of Pipeline Computers

In this section, we evaluate the performance of pipelined vector processors. The system performance is measured in terms of *processor utilization* and the *speedup* over a serial computer. The efficiency of a computer depends heavily on the inherent parallelism in the programs it executes. Fully exploring the embedded parallelism is the responsibility of both system designers and programmers. The performance of a pipeline computer depends on the pipeline rate, the work load distributions, the vectorization ratio, and the utilization rate of system resources.

The system performance of a vector processor is measured by the *maximum throughput* (W), that is, the maximum number of results that can be generated per unit time, such as the *megaflops* we have used. The processing of a vector job in a pipeline occupies the equipment (segments) over a certain length of time. The enclosed area in a space-time diagram depicts the pipeline hardware utilization as a function of time. The segments of a pipeline may operate on distinct data operands simultaneously. Pipelining increases the bandwidth by a factor of k, since it may carry k independent operand sets in the k segments concurrently. A pipeline computer requires more hardware and complex control circuitry than a corresponding serial computer. The following notations are used in the performance analysis:

- k: the number of stages in a functional pipe
- T: the total pipeline delay in one instruction execution
- n: the number of instructions contained in a task (program)
- N_i: the length of vector operands used in the ith instruction ($1 \leq i \leq n$)
- W: the throughput of a pipeline computer
- T_i: the time required to finish the ith instruction in a pipeline computer ($1 \leq i \leq n$)
- T_p: the total time required to finish a task consisting of n instructions
- S_k: the speedup of a pipeline computer (with k stages) over a corresponding serial computer
- η: the efficiency of a pipeline computer

The parameter T is assumed to be independent of the type of instructions. $N(i) = 1$ means a scalar instruction; $N(i) > 1$ refers to a vector instruction. The average delay in a stage is $\tau = T/k$. We can write

$$T_i = (k - 1)\cdot\tau + N_i\cdot\tau = (N_i + k - 1)\cdot\frac{T}{k} \tag{4.12}$$

where $(k - 1)\tau$ is the time required to fill up the pipe. As N_i becomes long, T_i approaches $N_i \cdot \tau$ because k is usually a small integer. The system throughput W then approaches k/T, accordingly. The above derivation is for a single vector instruction with vector length N_i.

Consider a sequence of n vector instructions. The degree of parallelism in each vector instruction is represented by its vector length N_i, for $i = 1, 2, \ldots, n$. Suppose that the execution of different types of vector instructions takes the same amount of time if they have the same vector length. The total execution times required in a pipeline processor is equal to

$$T_p = \sum_{i=1}^{n} T_i = \frac{T}{k}\cdot\left[(k - 1)\cdot n + \sum_{i=1}^{n} N_i\right] \tag{4.13}$$

The same code if executed on an equivalent-function serial computer needs a time delay $T_s = T \cdot \sum_{i=1}^{n} N_i$. The following speedup is obtained over a serial computer that does the same job:

$$S_k = \frac{T_s}{T_p} = \frac{T \cdot \sum_{i=1}^{n} N_i}{T \cdot \left[(k-1)n + \sum_{i=1}^{n} N_i \right] \Big/ k} = \frac{k \cdot \sum_{i=1}^{n} N_i}{(k-1)n + \sum_{i=1}^{n} N_i} \qquad (4.14)$$

The *efficiency* of the pipeline computer is defined as the total space-time product required by the job divided by a total available space-time product:

$$S_k = \frac{T_s}{T_p} = \frac{S_k}{k} \qquad (4.15)$$

The pipeline efficient can be interpreted as the ratio of the actual speedup to the maximum possible speedup k. A numerical example is used to demonstrate the analytical measures. Consider a vector job with a vector length distribution $N_i = 7, 3, 10, 1, 4, 6, 2, 5, 2, 4$ for $n = 10$ vector instructions. Figure 4.39 plots S_k and η against different values of k with respect to the above distribution. When k increases beyond the average vector length of N_i (i.e., 4.4 in this example), the increase in speedup becomes rather flat while the processor utilization continues to decline.

In general, pipeline processors are in favor of long vectors. The longer the vector fed into a pipeline, the less the effect of the overhead will be exhibited. Figure 4.40 shows the relation between the vector length and the system performance on a pipeline computer with $k = 8$ stages. The speedup increases monotonically until reaching the maximum value of $k = 8$, where the length approaches infinity. The dashed line in Figure 4.41 displays the effect due to partitioning the vector operand into 16-element segments. The maximum speedup drops to $8 * 16/(16 + 7) = 5.5$ with vector looping. This occurs when the vector length is a multiple of 16, the number of component registers in the vector register.

The pipeline efficiency depends also on the vector length distribution. Too many scalar operations of different types will definitely downgrade the system performance. To overcome this drawback, an intelligent vectorizer can help improve the situation. The Cray-1 has a scalar processor which is more than two times faster than the CDC 7600. When the vector length is short, execution by a scalar processor should be faster than execution in a vector pipeline.

The throughput rate reflects the processing capability of a pipeline processor. A higher throughput rate may be obtained at the expense of higher hardware cost. Therefore, the cost effectiveness of a pipeline design should not be ignored. Cost effectiveness can be measured by *megaflops per million dollars*. Table 4.15 presents the performance and cost ratio of several pipeline computers. The efficiency of a pipeline computer may depend on both hardware cost and the

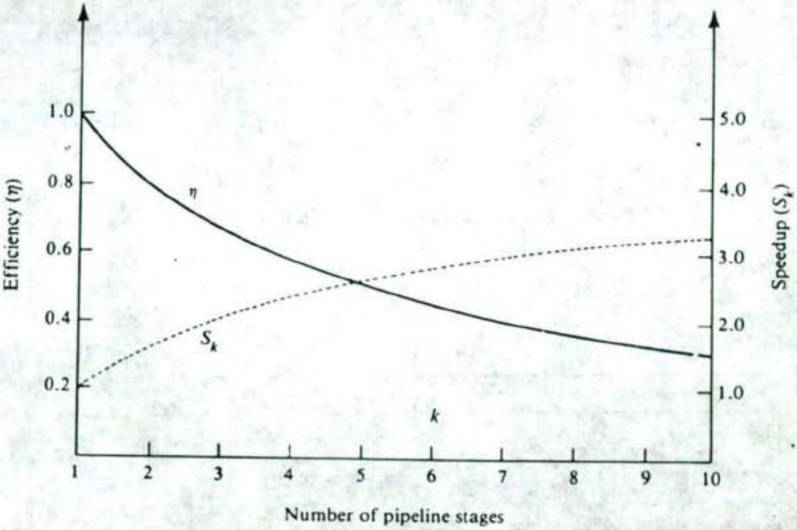


Figure 4.39 The speedup (S_k) and efficiency (η) of a pipelined processor with k stages.

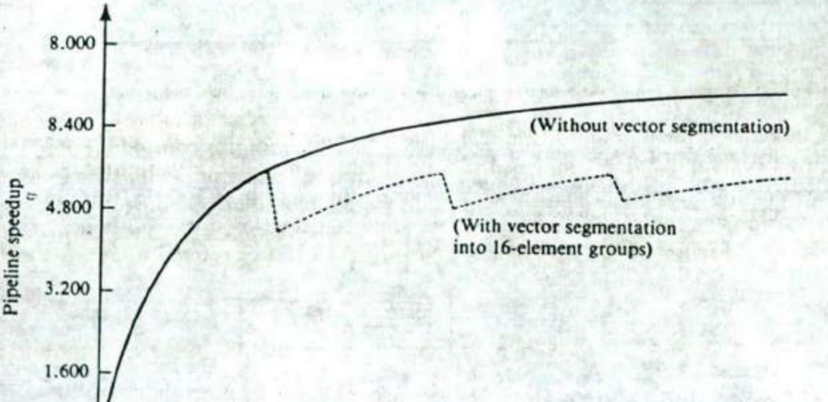


Figure 4.40 Pipeline speedup with and without vector looping. (Courtesy of *Advances in Computers*, Vol. 20, Hwang et al. 1981.)

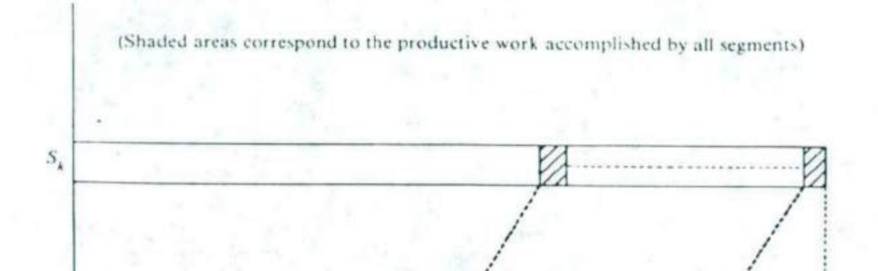

(Shaded areas correspond to the productive work accomplished by all segments)
S_k
$S_j + 1$
S_j
(Bottleneck)
S_3
S_2
S_1
Time
$\sum_{i=1}^{j} \tau_i$
$(n-1)\,\tau_j$
$\sum_{l=j+1}^{k} \tau_l$

Figure 4.41 The space-time diagram of a linear pipeline with nonuniform stage delays.

delay of each pipeline stage. Let C_i be the cost and τ_i be the delay of the ith stage S_i in a pipeline having k stages. Let n be the total number of jobs streamed into the pipeline during the period of measurement (assuming continuous input of jobs). Let τ be the pipeline clock period equal to the delay of the bottleneck stage ($\tau = \tau_j$ in Figure 4.41). The efficiency η in Eq. 4.15 can be refined to be:

$$\eta = \frac{n \cdot \sum_{i=1}^{k} C_i \cdot \tau_i}{\sum_{i=1}^{k} C_i \cdot \left[\sum_{i=1}^{k} \tau_i + (n-1) \cdot \tau \right]} \tag{4.16}$$

As illustrated in Figure 4.41, the numerator in Eq. 4.16 corresponds to the total weighted space-time span of n productive jobs. The denominator designates the total weighted space-time span of all k stages, including both productive and idle periods of all hardware facilities. In the ideal case with uniform delay ($\tau_i = \tau$ for all i), this efficiency can be reduced to the form in Eq. 3.5:

$$\eta = \frac{n}{k + (n-1)} \tag{4.17}$$

Table 4.15 Performance and cost of several pipeline computers

Pipeline computer	CPU cost (millions of of dollars)	Average performance (megaflops)	Performance and cost ratio (megaflops per millions of dollars)	Relative performance cost ratio
CDC6600	\$1.5	0.63	0.42	1
Star-100	\$8.0	16.6	2.10	5
AP-120B	\$0.1	7.9	79.0	188
Cray-1	\$4.5	20.0	4.44	11

Note that $\eta = S_k/k$, where S_k is the speedup defined in Eq. 4.14. When the pipeline approaches the steady state with sufficiently long vector input, the *limiting efficiency* becomes

$$\lim_{n \to \infty} \eta = \frac{\sum_{i=1}^{k} C_i \cdot \tau_i}{\tau \cdot \sum_{i=1}^{k} C_i} \tag{4.18}$$

In the ideal case of uniform delay, ($\tau_i = \tau$ for all i), the above limit will be 1 and the *maximal speedup* will be achieved ($S_k \to k$).

The cost effectiveness is indicated by the potential throughput performance relative to the total processor cost. The optimal pipeline design will maximize such a performance-cost ratio. Let T_s be the total time required to process a job in a nonpipeline serial processor. Consider the execution of the same job in an equivalent pipeline processor with k stages. The pipeline clock period is set to be $\tau = T_s/k + \theta$, where θ is the latch delay. Thus, in $n \cdot \tau = T_s + n \cdot \theta$ time units, n results can be produced. This implies the following system throughput:

$$W = \frac{n}{n \cdot (T_s/k + \theta)} = \frac{1}{T_s/k + \theta} \tag{4.19}$$

Let $C = \sum_{i=1}^{k} C_i$ be the total cost of all pipeline stages and d be the average latch cost. The cost of the entire pipeline is equal to $C + (k \cdot d)$. A *performance-cost ratio* (PCR) for the pipeline processor is defined as

$$\text{PCR} = \frac{W}{C + k \cdot d} = \frac{1}{(T_s/k + \theta) \cdot (C + k \cdot d)} \tag{4.20}$$

The optimal design of a static linear pipeline processor requires k_0 stages such that the PCR is maximized. Differentiating the PCR with respect to k, we obtain the first-order derivative

$$\frac{\partial(\text{PCR})}{\partial k} = \frac{T_s \cdot C/k^2 - \theta \cdot d}{(T_s/k + \theta)^2 \cdot (C + kd)^2} \tag{4.21}$$

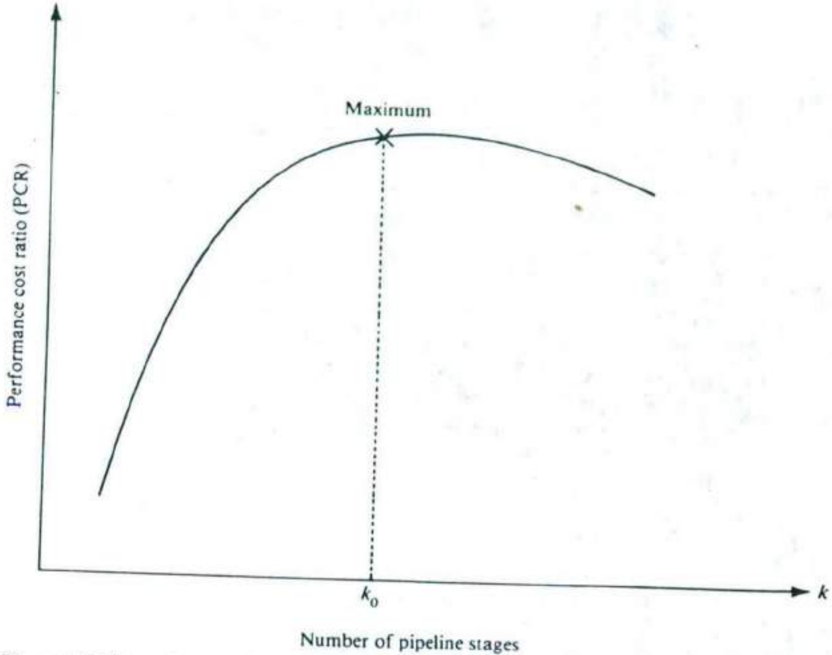


Figure 4.42 The performance/cost ratio (PCR) as a function of the number of stages in a pipeline.

When $\partial(\mathrm{PCR})/\partial k = 0$, we have $k = k_0$ from $T_s \cdot C/k_0^2 - \theta \cdot d = 0$. Therefore, the optimal design should have k_0 pipeline stages, where

$$k_0 = \sqrt{\frac{T_s \cdot C}{\theta \cdot d}} \tag{4.22}$$

This is indeed the maximum, because one can prove that $\partial^2(\mathrm{PCR})/\partial k^2 < 0$ when evaluated at $k = k_0$.

This optimal stage number is always greater than one for a reasonably complex pipeline. In the above discussion, we have emphasized local optimization of the pipeline processor. In practical design, one has to consider the global performance of the entire computer system, which may include parameters on memory access and program behavior. The PCR given in Eq. 4.20 is plotted in Figure 4.42 as a function of the number k of pipeline stages. The peak of the curve corresponds to the optimal number, k_0, of pipeline stages.

4.6 BIBLIOGRAPHIC NOTES AND PROBLEMS

Comparative studies of the early vector processors Star-100 and TI-ASC were given by Higbie (1973) and Theis (1974). The Cray-1 and Cyber-205 have been characterized in Kozdrawicki and Theis (1980). Hockney and Jesshope (1981) have treated pipeline vector processors and array processors. Others can be found in Thurber (1976, 1979a, b), Kuck (1977), Chen (1980), and Kogge (1981). Literature devoted to the CDC Star-100 and TI-ASC can be found in CDC manuals, Hintz and Tate (1972), Purcell (1974), Stone (1978), Ginsberg (1977), Watson (1972a, b), Watson and Carr (1974), and Texas Instruments manuals.

The Cray-1 computers were studied in Johnson (1978), Peterson (1979), Russell (1978). Cray Research manuals, Dorr (1978), and Baskett and Keller (1976). The Cyber-205 is described in Kascic (1979). The material presented in Section 4.4.4 is based on the work by Lincoln (1982). Manual information on the Cyber-205 can be found in CDC manuals. Material on the VP-200 is based on the report by Miura and Uchida (1983). The AP-120B has been described and assessed in Wittmayer (1978), and Floating Point Systems manuals. Other attached processors were reported in Datawest (1979), IBM (1977), and Thurber (1979b).

Vectorization methods for pipeline computers can be found in CDC and Cray Research manuals, Paul and Wilson (1978), Kennedy (1979), Loveman (1977), and Hwang et al. (1981). Vectorizing compilers are also studied in Arnold (1982), Brode (1981), and Kuck et al. (1983). Parallel programming languages and the optimization of vector operations are still wide open areas for further research and development. Performance of pipeline processors has been evaluated in Chen (1975), Bovet and Varneschi (1976), Baskett and Keller (1977), Larson and Davidson (1973), Ramamoorthy and Li (1975), Hwang and Su (1983), and Stokes (1977).

Problems

4.1 Design a pipeline multiplier using carry-save adders and a carry-lookahead adder to multiply a stream of input numbers $X_1, X_2, X_3, \ldots,$ by a fixed number Y. You may assume that all X_i and Y are n-bit positive integers. The output should be a stream of n-bit products $X_1 \cdot Y, X_2 \cdot Y, X_3 \cdot Y, \ldots$. Determine the pipeline clock rate in terms of the delays α, β, and γ, where

α = delay of one stage of a 3-input and 2-output carry-save adder

β = delay of the carry-lookahead adder

γ = delay of the interface latch between stages

4.2 Consider a static multifunctional pipeline processor with k stages, each stage having a delay of $1/k$ time units. The pipeline must be drained between different functions, such as addition and multiplication. Memory-access time, control-unit time, etc., can be ignored. There are sufficient numbers of temporary registers to use.

(*a*) Determine the number of unit-time steps T_1 required to compute the product of two $n \times n$ matrices on a nonpipeline scalar processor. Assume one unit time per each addition or each multiplication operation in the scalar processor.

(*b*) Determine the number of time steps T_k required to compute the matrix product, using the multifunction pipeline processor with a total pipeline delay of one time unit.

(*c*) Determine the speedup ratios T_1/T_k, when $n = 1$, $n = k$, and $n = m \cdot k$ for some large m.

4.3 Compare the second-generation vector processors Cray-1, Cyber-205, and VP-200 in the following aspects:

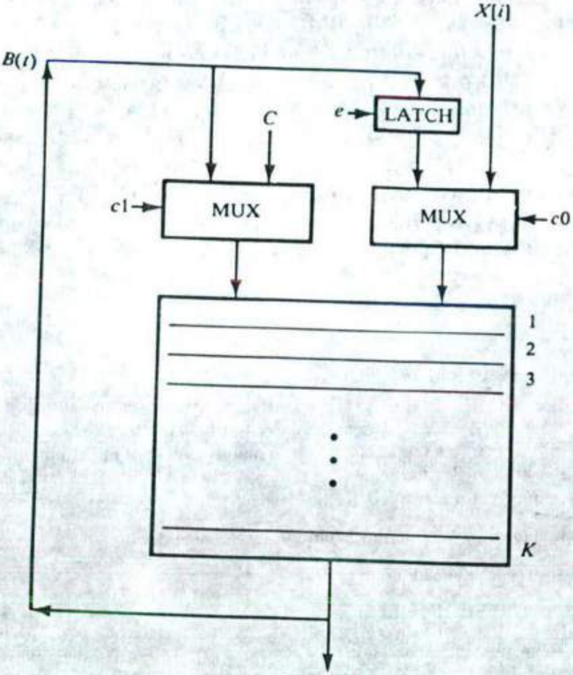
(*a*) Instruction sets: vector versus scalar instructions

(*b*) Functional pipeline structures and usage

(c) Register files and main memory organization
(d) Vectorization and optimization methods
(e) Strengths and weaknesses in using the machine

4.4 Pipeline chaining has demonstrated its advantages in many vector processors. From the viewpoints of resource reservation (registers and functional pipes) and the precedence graphs of arithmetic expressions, find necessary and sufficient conditions that allow the chaining of functional pipes. For different task systems, different chaining conditions may be found for different types of data dependence graphs. Assume some fixed delays of the pipes, registers and memory and some fixed vector-length distribution.

4.5 Conduct a thorough survey of various vectorizing compilers in existing vector supercomputers and compare their special features and relative strengths for a number of representative kernel computations such as those for solving partial differential equations, oil reservoir modeling, electric power flow analysis, linear programming, etc.



c1	c0	Input pair
0	0	$C, X[i]$
0	1	$C, B(t-j)$
1	0	$B(t), X[i]$
1	1	$B(t), B(t-j)$

Figure 4.43 The hardware organization of a vector reduction pipeline.

4.6 Conduct a thorough comparison of back-end computer systems, including both *vector supercomputers* and the *attached scientific processors*, in the following aspects:

(*a*) Peak scalar performance
(*b*) Peak vector performance
(*c*) Pipeline clock rate
(*d*) Memory bandwidth
(*e*) Cost of central processing unit
(*f*) Performance/cost ratio
(*g*) Main memory capacity
(*h*) Register files/buffers
(*i*) Functional pipelines

4.7 Let A(1 : 2*N*), B(1 : 2*N*), C(1 : 2*N*) and D(1 : 2*N*) be each a 2*N*-element vector stored in the main memory of a vector processor. Each vector register in the processor has *N* components. There are two *load-store* pipes, one *multiply* pipe, and one *add* pipe that are available to be used. Draw a space-time diagram (similar to that in Figure 4.37 for Example 4.22) to show the *pipeline chaining* and *parallelization* operations to be performed in the execution of the following linked vector instructions with minimum time delay:

$$\mathrm{A(I) = B(I) * C(I) + D(I)}$$
$$\text{for } \mathrm{I = 1, 2, \ldots, N, N+1, \ldots, 2N}$$

Assume that all pipeline units, regardless of their functions, have equal delays as assumed in Figure 4.37. Sufficient numbers of vector registers are assumed available, and they can be cascaded together to hold longer vectors, such as 2*N*, 3*N*, . . . , etc.

4.8 A *vector arithmetic reduction unit* is shown in Figure 4.43. This multifunction pipeline can accept vector inputs and produce a single scalar output at the end of computation. The feedback connection is needed for accumulated arithmetic operations. Develop four fast algorithms for scheduling the successive computations (*multiply*'s and *add-subtract*'s), needed in the following vector reduction arithmetic operations:

(*a*) The summation of the *n* components in a vector
(*b*) The dot product of two *n*-element vectors
(*c*) The multiplication of two $n \times n$ matrices
(*d*) The searching of the maximum among *n* components of a vector

Hint: Algorithm (*a*) may be used in algorithm (*b*). Both (*a*) and (*b*) may be used in algorithm (*c*). In part (*d*), comparison is done by subtraction through the pipeline unit.

4.9 Let *D* be a stream of data (tasks or operand sets). Suppose that we wish to perform two functions f_1 and f_2 on every task in *D*. That is, for each operand set *x* in *D*, we want to compute $f_2(f_1(x))$. The computations are to be performed on a machine with multiple pipeline functional units, one of which computes the function f_1 and another which computes f_2. The reservation tables for f_1 and f_2 are given below.

f_1:	1	2	3	4	5	6	7	8
S1	x						x	
S2		x	x					
S3				x				
S4		x			x			x
S5						x		

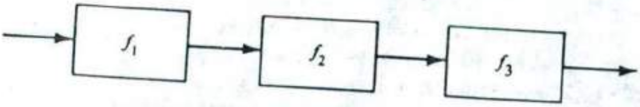
f_2:	1	2	3	4	5	6	7	8	9	10
S1	x									x
S2		x								x
S3			x				x			
S4				x			x			
S5					x			x		
S6						x			x	

f_1:	1	2	3	4	5	6
s_1	x					x
s_2		x		x		
s_3			x		x	

f_2:	1	2	3	4
s_1	x			x
s_2		x		x
s_3			x	

f_3:	1	2	3	4	5
s_1	x			x	
s_2		x			x
s_3			x		

(*a*) Reservation tables



(*b*) Pipeline chaining

Figure 4.44 The chaining of three pipelines in Problem 4.10.

(*a*) What are the maximum throughputs for the f_1 and g_2 pipelines, assuming that they work completely independently on one another? That is, assume that the two pipelines work on completely independent data streams D_1 and D_2.

(*b*) In chaining, the output of one pipeline is applied directly to the input of another pipeline. One can think of this as configuring the pipelines such that the output latch or buffer of the first pipeline becomes the input latch or buffer of the second. What is the maximum throughput for tasks in D if the f_1 and f_2 pipeline functional units are chained together?

(*c*) What can you conclude about the general effectiveness of chaining pipelines that have feedback? Consider the effect on memory contention and the demand on memory bandwidth as part of your answer.

4.10 Consider three functional pipelines f_1, f_2, and f_3 characterized by the reservation tables in Figure 4.44*a*.

(*a*) What are the *minimal average latencies* in using the f_1, f_2, and f_3 pipelines independently?

(*b*) What is the *maximum throughput* if three pipelines are chained into a linear cascade as shown in Figure 4.44*b*?

4.11 Show the timing diagrams for implementing the two sequences of vector instructions (described in Example 4.23) on the Cray-1 machine. Verify the total clock periods required in each of the two computing sequences.