CHAPTER NINE

Pipeline and Vector Processing

IN THIS CHAPTER

9-1 Parallel Processing

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time. Furthermore, the system may have two or more processors operating concurrently. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing, and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

throughput

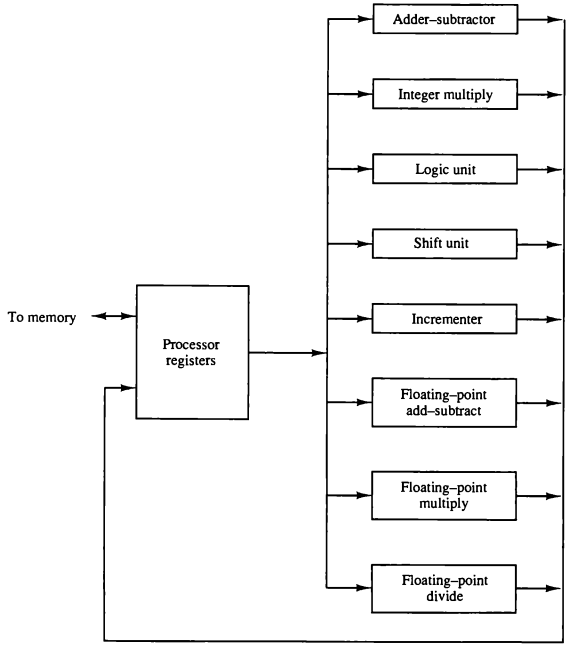
Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time,

while registers with parallel load operate with all the bits of the word simultaneously. Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

multiple functional units

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruc-

Figure 9-1 Processor with multiple functional units.



tion associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented. A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an *instruction stream*. The operations performed on the data in the processor constitutes a *data stream*. Parallel processing may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)
Single instruction stream, multiple data stream (SIMD)
Multiple instruction stream, single data stream (MISD)
Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only of theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.

MIMD

Flynn's classification depends on the distinction between the performance of the control unit and the data-processing unit. It emphasizes the be-

havioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only two categories used from this classification are SIMD array processors discussed in Sec. 9-7, and MIMD multiprocessors presented in Chap. 13.

In this chapter we consider parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

Pipeline processing is an implementation technique where arithmetic suboperations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

9-2 Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

an example

The pipeline organization will be demonstrated by means of a simple

example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A_1 and B_1 into $R1$ and

Figure 9-2 Example of pipeline processing.

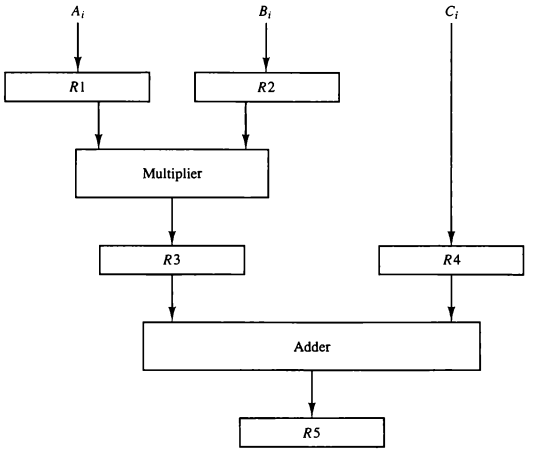


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

$R2$. The second clock pulse transfers the product of $R1$ and $R2$ into $R3$ and C_1 into $R4$. The same clock pulse transfers A_2 and B_2 into $R1$ and $R2$. The third clock pulse operates on all three segments simultaneously. It places A_3 and B_3 into $R1$ and $R2$, transfers the product of $R1$ and $R2$ into $R3$, transfers C_2 into $R4$, and places the sum of $R3$ and $R4$ into $R5$. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

General Considerations

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R_i that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a *task* as the total operation performed going through all the segments in the pipeline.

task

space-time diagram

The behavior of a pipeline can be illustrated with a *space-time* diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis displays the time in clock cycles and the vertical axis gives the

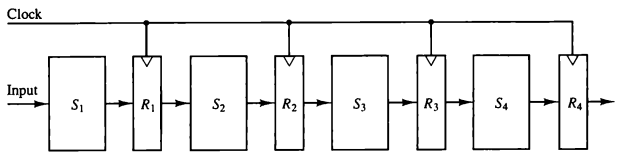


Figure 9-3 Four-segment pipeline.

segment number. The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1, while segment 1 is busy with task T_2. Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Now consider the case where a k-segment pipeline with a clock cycle time t_p is used to execute n tasks. The first task T_1 requires a time equal to kt_p to complete its operation since there are k segments in the pipe. The remaining $n - 1$ tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$. Therefore, to complete n tasks using a k-segment pipeline requires $k + (n - 1)$ clock cycles. For example, the diagram of Fig. 9-4 shows four segments and six tasks. The time required to complete all the operations is $4 + (6 - 1) = 9$ clock cycles, as indicated in the diagram.

Next consider a nonpipeline unit that performs the same operation and takes a time equal to t_n to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

speedup

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

This shows that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to $t_p = 20$ ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take $(k + n - 1)t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a nonpipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that $t_n = 60$ ns, the speedup becomes 60/20 = 3.

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel. Each P circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same time. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Fig. 9-5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock. Moreover, it is not always

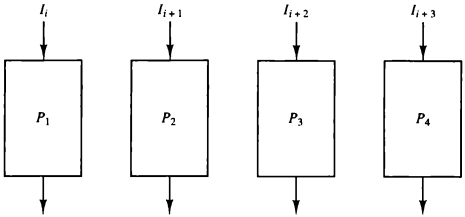


Figure 9-5 Multiple functional units in parallel.

correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An *arithmetic pipeline* divides an arithmetic operation into suboperations for execution in the pipeline segments. An *instruction pipeline* operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

9-3 Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

A and B are two fractions that represent the mantissas and a and b are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6. The registers labeled R are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the suboperations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$
$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain $3 - 2 = 1$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 \times 10^3$$
$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

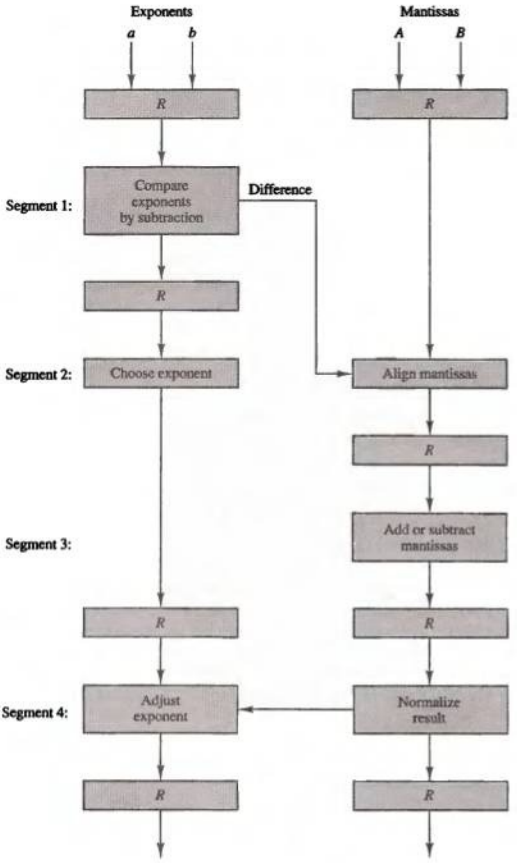


Figure 9-6 Pipeline for floating-point addition and subtraction.

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent nonpipeline floating-point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

9-4 Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue from which control then extracts the instructions for the execution unit.

instruction cycle

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

Example: Four-Segment Instruction Pipeline

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Figure 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

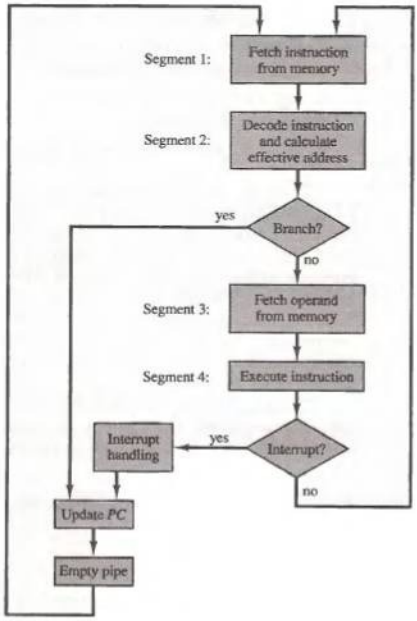


Figure 9-7 Four-segment CPU pipeline.

Figure 9-8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence

	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 9-8 Timing of instruction pipeline.

of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

pipeline conflicts

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. *Resource conflicts* caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. *Data dependency* conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. *Branch difficulties* arise from branch and other instructions that change the value of *PC*.

Data Dependency

A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when

an instruction cannot proceed because previous instructions did not complete certain operations. A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

hardware interlocks

The most straightforward method is to insert *hardware interlocks*. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

operand forwarding

Another technique called *operand forwarding* uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program. The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as *delayed load*. An example of delayed load is presented in the next section.

delayed load

Handling of Branch Instructions

One of the major problems in operating an instruction pipeline is the occurrence of branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address. In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.

Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching.

prefetch target instruction

One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction. An extension of this procedure is to continue fetching instructions from both places until the branch decision is made. At that time control chooses the instruction stream of the correct program flow.

branch target buffer

Another possibility is the use of a *branch target buffer* or BTB. The BTB is an associative memory (see Sec. 12-4) included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. If it is in the BTB, the instruction is available directly and prefetch continues from the new path. If the instruction is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

loop buffer

A variation of the BTB is the *loop buffer*. This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches. The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out.

branch prediction

Another procedure that some computers use is *branch prediction*. A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties.

delayed branch

A procedure employed in most RISC processors is the *delayed branch*. In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions. An example of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of the no-operation instruction, allowing a continuous flow of the pipeline. An example of delayed branch is presented in the next section.

9-5 RISC Pipeline

The reduced instruction set computer (RISC) was introduced in Sec. 8-8. Among the characteristics attributed to RISC is its ability to use an efficient instruction pipeline. The simplicity of the instruction set can be utilized to

implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle. Because of the fixed-length instruction format, the decoding of the operation can occur at the same time as the register selection. All data manipulation instructions have register-to-register operations. Since all operands are in registers, there is no need for calculating an effective address or fetching of operands from memory. Therefore, the instruction pipeline can be implemented with two or three segments. One segment fetches the instruction from program memory, and the other segment executes the instruction in the ALU. A third segment may be used to store the result of the ALU operation in a destination register.

The data transfer instructions in RISC are limited to load and store instructions. These instructions use register indirect addressing. They usually need three or four stages in the pipeline. To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories: one for storing the instructions and the other for storing the data. The two memories can sometime operate at the same speed as the CPU clock and are referred to as cache memories (see Sec. 12-6).

As mentioned in Sec. 8-8, one of the major advantages of RISC is its ability to execute instructions at the rate of one per clock cycle. It is not possible to expect that every instruction be fetched from memory and executed in one clock cycle. What is done, in effect, is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution. The advantage of RISC over CISC (complex instruction set computer) is that RISC can achieve pipeline segments, requiring just one clock cycle, while CISC uses many segments in its pipeline, with the longest segment requiring two or more clock cycles.

single-cycle instruction execution

Another characteristic of RISC is the support given by the compiler that translates the high-level language program into machine language program. Instead of designing hardware to handle the difficulties associated with data conflicts and branch penalties, RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems. The following examples show how a compiler can optimize the machine language program to compensate for pipeline conflicts.

compiler support

Example: Three-Segment Instruction Pipeline

A typical set of instructions for a RISC processor are listed in Table 8-12. We see from this table that there are three types of instructions. The data manipulation instructions operate on data in processor registers. The data transfer instructions are load and store instructions that use an effective address obtained from the addition of the contents of two registers or a register and a displacement constant provided in the instruction. The program control instructions use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter PC.

Now consider the hardware operation for such a computer. The control section fetches the instruction from program memory into an instruction register. The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected. The processor unit consists of a number of registers and an arithmetic logic unit (ALU) that performs the necessary arithmetic, logic, and shift operations. A data memory is used to load or store the data from a selected register in the register file. The instruction cycle can be divided into three suboperations and implemented in three segments:

I: Instruction fetch
A: ALU operation
E: Execute instruction

The I segment fetches the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. The ALU is used for three different functions, depending on the decoded instruction. It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction. The E segment directs the output of the ALU to one of three destinations, depending on the decoded instruction. It transfers the result of the ALU operation into a destination register in the register file, it transfers the effective address to a data memory for loading or storing, or it transfers the branch address to the program counter.

Delayed Load

Consider now the operation of the following four instructions:

1. LOAD: $R1 \leftarrow M$[address 1]
2. LOAD: $R2 \leftarrow M$[address 2]
3. ADD: $R3 \leftarrow R1 + R2$
4. STORE: M[address 3] $\leftarrow R3$

If the three-segment pipeline proceeds without interruptions, there will be a data conflict in instruction 3 because the operand in $R2$ is not yet available in the A segment. This can be seen from the timing of the pipeline shown in Fig. 9-9(a). The E segment in clock cycle 4 is in a process of placing the memory data into $R2$. The A segment in clock cycle 4 is using the data from $R2$, but the value in $R2$ will not be the correct value since it has not yet been transferred from memory. It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction. This is a type of instruction that is fetched from

Clock cycles:	1	2	3	4	5	6
1. Load *R*1	I	A	E			
2. Load *R*2		I	A	E		
3. Add *R*1 + *R*2			I	A	E	
4. Store *R*3				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

Figure 9-9 Three-segment pipeline timing.

memory but has no operation, thus wasting a clock cycle. This concept of delaying the use of the data loaded from memory is referred to as *delayed load*.

Figure 9-9(b) shows the same program with a no-op instruction inserted after the load to *R*2 instruction. The data is loaded into *R*2 in clock cycle 4. The add instruction uses the value of *R*2 in step 5. Thus the no-op instruction is used to advance one clock cycle in order to compensate for the data conflict in the pipeline. (Note that no operation is performed in segment A during clock cycle 4 or segment E during clock cycle 5.) The advantage of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware. This results in a simpler hardware segment since the segment does not have to check if the content of the register being accessed is currently valid or not.

Delayed Branch

It was shown in Fig. 9-8 that a branch instruction delays the pipeline operation until the instruction at the branch address is fetched. Several techniques for reducing branch penalties were discussed in the preceding section. The method used in most RISC processors is to rely on the compiler to redefine the branches so that they take effect at the proper time in the pipeline. This method is referred to as *delayed branch*.

The compiler for a processor that uses delayed branches is designed to analyze the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps. For example, the compiler can determine that the program dependencies allow one or more instructions preceding the branch to be moved into the delay steps after the branch. These instructions are then fetched from memory and executed through the pipeline while the branch instruction is being executed in other segments. The effect is the same as if the instructions were executed in their original order, except that the branch delay is removed. It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert no-op instructions.

An example of delayed branch is shown in Fig. 9-10. The program for this example consists of five instructions:

Load from memory to *R*1
Increment *R*2
Add *R*3 to *R*4
Subtract *R*5 from *R*6
Branch to address *X*

In Fig. 9-10(a) the compiler inserts two no-op instructions after the branch. The branch address *X* is transferred to *PC* in clock cycle 7. The fetching of the instruction at *X* is delayed by two clock cycles by the no-op instructions. The instruction at *X* starts the fetch phase at clock cycle 8 after the program counter *PC* has been updated.

The program in Fig. 9-10(b) is rearranged by placing the add and subtract instructions after the branch instruction instead of before as in the original program. Inspection of the pipeline timing shows that *PC* is updated to the value of *X* in clock cycle 5, but the add and subtract instructions are fetched from memory and executed in the proper sequence. In other words, if the load instruction is at address 101 and *X* is equal to 350, the branch instruction is fetched from address 103. The add instruction is fetched from address 104 and executed in clock cycle 6. The subtract instruction is fetched from address 105 and executed in clock cycle 7. Since the value of *X* is transferred to *PC* with clock cycle 5 in the E segment, the instruction fetched from memory at clock cycle 6 is from address 350, which is the instruction at the branch address.

9-6 Vector Processing

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems are characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to *X*					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in *X*								I	A	E

(a) Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to *X*			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in *X*						I	A	E

(b) Rearranging the instructions

Figure 9-10 Example of delayed branch.

applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

applications

Computers with vector processing capabilities are in demand in specialized applications. The following are representative application areas where vector processing is of the utmost importance.

Long-range weather forecasting
Petroleum explorations
Seismic data analysis
Medical diagnosis
Aerodynamics and space flight simulations

Artificial intelligence and expert systems
Mapping the human genome
Image processing

Without sophisticated computers, many of the required computations cannot be completed within a reasonable amount of time. To achieve the required level of high performance it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

Vector Operations

Many scientific problems require arithmetic operations on large arrays of numbers. These numbers are usually formulated as vectors and matrices of floating-point numbers. A vector is an ordered set of a one-dimensional array of data items. A vector V of length n is represented as a row vector by $V = [V_1 \quad V_2 \quad V_3 \quad \cdots \quad V_n]$. It may be represented as a column vector if the data items are listed in a column. A conventional sequential computer is capable of processing operands one at a time. Consequently, operations on vectors must be broken down into single computations with subscripted variables. The element V_i of vector V is written as $V(I)$ and the index I refers to a memory address or register where the number is stored. To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

```
      DO 20 I = 1, 100
20    C(I) = B(I) + A(I)
```

This is a program for adding two vectors A and B of length 100 to produce a vector C. This is implemented in machine language by the following sequence of operations.

```
      Initialize I = 0
20    Read A(I)
      Read B(I)
      Store C(I) = A(I) + B(I)
      Increment I = I + 1
      If I ≤ 100 go to 20
      Continue
```

This constitutes a program loop that reads a pair of operands from arrays A and B and performs a floating-point addition. The loop control variable is then updated and the steps repeat 100 times.

A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program

loop. It allows operations to be specified with a single vector instruction of the form

$$C(1:100) = A(1:100) + B(1:100)$$

The vector instruction includes the initial address of the operands, the length of the vectors, and the operation to be performed, all in one composite instruction. The addition is done with a pipelined floating-point adder similar to the one shown in Fig. 9-6.

A possible instruction format for a vector instruction is shown in Fig. 9-11. This is essentially a three-address instruction with three fields specifying the base address of the operands and an additional field that gives the length of the data items in the vectors. This assumes that the vector operands reside in memory. It is also possible to design the processor with a large number of registers and store all operands in registers prior to the addition operation. In that case the base address and length in the vector instruction specify a group of CPU registers.

Matrix Multiplication

Matrix multiplication is one of the most computational intensive operations performed in computers with vector processors. The multiplication of two $n \times n$ matrices consists of n^2 inner products or n^3 multiply–add operations. An $n \times m$ matrix of numbers has n rows and m columns and may be considered as constituting a set of n row vectors or a set of m column vectors. Consider, for example, the multiplication of two 3×3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3×3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting $i = 1, j = 1$, to obtain

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Figure 9-11 Instruction format for vector processor.

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length

This requires three multiplications and (after initializing c_{11} to 0) three additions. The total number of multiplications or additions required to compute the matrix product is $9 \times 3 = 27$. If we consider the linked multiply–add operation $c + a \times b$ as a cumulative operation, the product of two $n \times n$ matrices requires n^3 multiply–add operations. The computation consists of n^2 inner products, with each inner product requiring n multiply–add operations, assuming that c is initialized to zero before computing each element in the product matrix.

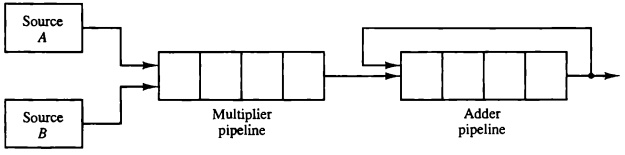
In general, the inner product consists of the sum of k product terms of the form

$$C = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4 + \cdots + A_k B_k$$

In a typical application k may be equal to 100 or even 1000. The inner product calculation on a pipeline vector processor is shown in Fig. 9-12. The values of A and B are either in memory or in processor registers. The floating-point multiplier pipeline and the floating-point adder pipeline are assumed to have four segments each. All segment registers in the multiplier and adder are initialized to 0. Therefore, the output of the adder is 0 for the first eight cycles until both pipes are full. A_i and B_i pairs are brought in and multiplied at a rate of one pair per cycle. After the first four cycles, the products begin to be added to the output of the adder. During the next four cycles 0 is added to the products entering the adder pipeline. At the end of the eighth cycle, the first four products $A_1 B_1$ through $A_4 B_4$ are in the four adder segments, and the next four products, $A_5 B_5$ through $A_8 B_8$, are in the multiplier segments. At the beginning of the ninth cycle, the output of the adder is $A_1 B_1$ and the output of the multiplier is $A_5 B_5$. Thus the ninth cycle starts the addition $A_1 B_1 + A_5 B_5$ in the adder pipeline. The tenth cycle starts the addition $A_2 B_2 + A_6 B_6$, and so on. This pattern breaks down the summation into four sections as follows:

$$\begin{aligned} C = {} & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \cdots \\ & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \cdots \\ & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \cdots \\ & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \cdots \end{aligned}$$

Figure 9-12 Pipeline for calculating an inner product.

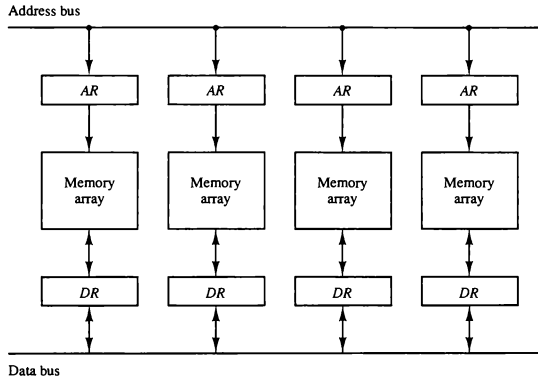


When there are no more product terms to be added, the system inserts four zeros into the multiplier pipeline. The adder pipeline will then have one partial product in each of its four segments, corresponding to the four sums listed in the four rows in the above equation. The four partial sums are then added to form the final sum.

Memory Interleaving

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments. Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows a memory unit with four modules. Each memory array has its own address register *AR* and data register *DR*. The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

Figure 9-13 Multiple module memory organization.



The advantage of a modular memory is that it allows the use of a technique called *interleaving*. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other. When the number of modules is a power of 2, the least significant bits of the address select a memory module and the remaining bits designate the specific location to be accessed within the selected module.

A modular memory is useful in systems with pipeline and vector processing. A vector processor that uses an n-way interleaved memory can fetch n operands from n different modules. By staggering the memory access, the effective memory cycle time can be reduced by a factor close to the number of modules. A CPU with instruction pipeline can take advantage of multiple memory modules so that each segment in the pipeline can access memory independent of memory access from other segments.

Supercomputers

A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a *supercomputer*. Supercomputers are very powerful, high-performance machines used mostly for scientific computations. To speed up the operation, the components are packed tightly together to minimize the distance that the electronic signals have to travel. Supercomputers also use special techniques for removing the heat from circuits to prevent them from burning up because of their close proximity.

The instruction set of supercomputers contains the standard data transfer, data manipulation, and program control instructions of conventional computers. This is augmented by instructions that process vectors and combinations of scalars and vectors. A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing. It is equipped with multiple functional units and each unit has its own pipeline configuration. Although the supercomputer is capable of general-purpose applications found in all other computers, it is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.

Supercomputers are not suitable for normal everyday processing of a typical computer installation. They are limited in their use to a number of scientific applications, such as numerical weather forecasting, seismic wave analysis, and space research. They have limited use and limited market because of their high price.

A measure used to evaluate computers in their ability to perform a given number of floating-point operations per second is referred to as *flops*. The term *megaflops* is used to denote million flops and *gigaflops* to denote billion flops. A typical supercomputer has a basic cycle time of 4 to 20 ns. If the processor can calculate a floating-point operation through a pipeline each cycle time, it will have the ability to perform 50 to 250 megaflops. This rate would be

sustained from the time the first answer is produced and does not include the initial setup time of the pipeline.

The first supercomputer developed in 1976 is the Cray-1 supercomputer. It uses vector processing with 12 distinct functional units in parallel. Each functional unit is segmented to process the incoming data through a pipeline. All the functional units can operate concurrently with operands stored in the large number of registers (over 150) in the CPU. A floating-point operation can be performed on two sets of 64-bit operands during one clock cycle of 12.5 ns. This gives a rate of 80 megaflops during the time that the data are processed through the pipeline. It has a memory capacity of 4 million 64-bit words. The memory is divided into 16 banks, with each bank having a 50-ns access time. This means that when all 16 banks are accessed simultaneously, the memory transfer rate is 320 million words per second. Cray research extended its supercomputer to a multiprocessor configuration called Cray X-MP and Cray Y-MP. The new Cray-2 supercomputer is 12 times more powerful than the Cray-1 in vector processing mode.

Another early model supercomputer is the Fujitsu VP-200. It has a scalar processor and a vector processor that can operate concurrently. Like the Cray supercomputers, a large number of registers and multiple functional units are used to enable register-to-register vector operations. There are four execution pipelines in the vector processor, and when operating simultaneously, they can achieve up to 300 megaflops. The main memory has 32 million words connected to the vector registers through load and store pipelines. The VP-200 has 83 vector instructions and 195 scalar instructions. The newer VP-2600 uses a clock cycle of 3.2 ns and claims a peak performance of 5 gigaflops.

9-7 Array Processors

An array processor is a processor that performs computations on large arrays of data. The term is used to refer to two different types of processors. An *attached array processor* is an auxiliary processor attached to a general-purpose computer. It is intended to improve the performance of the host computer in specific numerical computation tasks. An *SIMD array processor* is a processor that has a single-instruction multiple-data organization. It manipulates vector instructions by means of multiple functional units responding to a common instruction. Although both types of array processors manipulate vectors, their internal organization is different.

Attached Array Processor

An attached array processor is designed as a peripheral for a conventional host computer, and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications. It achieves

high performance by means of parallel processing with multiple functional units. It includes an arithmetic unit containing one or more pipelined floating-point adders and multipliers. The array processor can be programmed by the user to accommodate a variety of complex arithmetic problems.

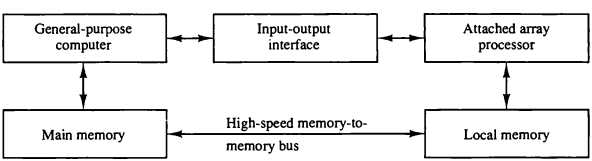
Figure 9-14 shows the interconnection of an attached array processor to a host computer. The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer. The array processor is connected through an input–output controller to the computer and the computer treats it like an external interface. The data for the attached processor are transferred from main memory to a local memory through a high-speed bus. The general-purpose computer without the attached processor serves the users that need conventional data processing. The system with the attached processor satisfies the needs for complex arithmetic applications.

Some manufacturers of attached array processors offer a model that can be connected to a variety of different host computers. For example, when attached to a VAX 11 computer, the FSP-164/MAX from Floating-Point Systems increases the computing power of the VAX to 100 megaflops. The objective of the attached array processor is to provide vector manipulation capabilities to a conventional computer at a fraction of the cost of supercomputers.

SIMD Array Processor

An SIMD array processor is a computer with multiple processing units operating in parallel. The processing units are synchronized to perform the same operation under the control of a common control unit, thus providing a single instruction stream, multiple data stream (SIMD) organization. A general block diagram of an array processor is shown in Fig. 9-15. It contains a set of identical processing elements (PEs), each having a local memory M. Each processor element includes an ALU, a floating-point arithmetic unit, and working registers. The master control unit controls the operations in the processor elements. The main memory is used for storage of the program. The function of the master control unit is to decode the instructions and determine how the instruction is to be executed. Scalar and program control instructions are

Figure 9-14 Attached array processor with host computer.



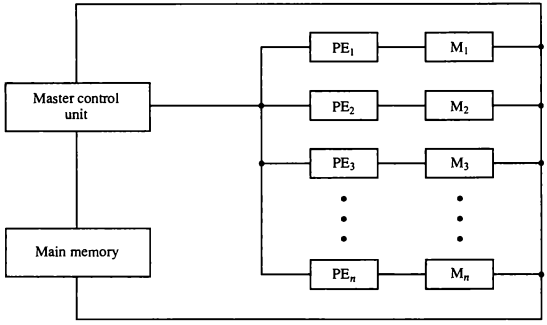


Figure 9-15 SIMD array processor organization.

directly executed within the master control unit. Vector instructions are broadcast to all PEs simultaneously. Each PE uses operands stored in its local memory. Vector operands are distributed to the local memories prior to the parallel execution of the instruction.

Consider, for example, the vector addition $C = A + B$. The master control unit first stores the ith components a_i and b_i of A and B in local memory M_i for $i = 1, 2, 3, \ldots, n$. It then broadcasts the floating-point add instruction $c_i = a_i + b_i$ to all PEs, causing the addition to take place simultaneously. The components of c_i are stored in fixed locations in each local memory. This produces the desired vector sum in one add cycle.

Masking schemes are used to control the status of each PE during the execution of vector instructions. Each PE has a flag that is set when the PE is active and reset when the PE is inactive. This ensures that only those PEs that need to participate are active during the execution of the instruction. For example, suppose that the array processor contains a set of 64 PEs. If a vector length of less than 64 data items is to be processed, the control unit selects the proper number of PEs to be active. Vectors of greater length than 64 must be divided into 64-word portions by the control unit.

The best known SIMD array processor is the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp. This computer is no longer in operation. SIMD processors are highly specialized computers. They are suited primarily for numerical problems that can be expressed in vector or matrix form. However, they are not very efficient in other types of computations or in dealing with conventional data-processing programs.

PROBLEMS

9-1. In certain scientific computations it is necessary to perform the arithmetic operation $(A_i + B_i)(C_i + D_i)$ with a stream of numbers. Specify a pipeline configuration to carry out this task. List the contents of all registers in the pipeline for $i = 1$ through 6.

9-2. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

9-3. Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.

9-4. A nonpipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

9-5. The pipeline of Fig. 9-2 has the following propagation times: 40 ns for the operands to be read from memory into registers *R*1 and *R*2, 45 ns for the signal to propagate through the multiplier, 5 ns for the transfer into *R*3, and 15 ns to add the two numbers into *R*5.

a. What is the minimum clock cycle time that can be used?

b. A nonpipeline system can perform the same operation by removing *R*3 and *R*4. How long will it take to multiply and add the operands without using the pipeline?

c. Calculate the speedup of the pipeline for 10 tasks and again for 100 tasks.

d. What is the maximum speedup that can be achieved?

9-6. It is necessary to design a pipeline for a fixed-point multiplier that multiplies two 8-bit binary integers. Each segment consists of a number of AND gates and a binary adder similar to an array multiplier as shown in Fig. 10-10.

a. How many AND gates are there in each segment, and what size of adder is needed?

b. How many segments are there in the pipeline?

c. If the propagation delay in each segment is 30 ns, what is the average time that it takes to multiply two fixed-point numbers in the pipeline?

9-7. The time delay of the four segments in the pipeline of Fig. 9-6 are as follows: $t_1 = 50$ ns, $t_2 = 30$ ns, $t_3 = 95$ ns, and $t_4 = 45$ ns. The interface registers delay time $t_r = 5$ ns.

a. How long would it take to add 100 pairs of numbers in the pipeline?

b. How can we reduce the total time to about one-half of the time calculated in part (a)?

9-8. How would you use the floating-point pipeline adder of Fig. 9-6 to add 100 floating-point numbers $X_1 + X_2 + X_3 + \cdots + X_{100}$?

9-9. Formulate a six-segment instruction pipeline for a computer. Specify the operations to be performed in each segment.

9-10. Explain four possible hardware schemes that can be used in an instruction pipeline in order to minimize the performance degradation caused by instruction branching.

9-11. Consider the four instructions in the following program. Suppose that the first instruction starts from step 1 in the pipeline used in Fig. 9-8. Specify what operations are performed in the four segments during step 4.

```
Load    R1←M[312]
ADD     R2←R2 + M[313]
INC     R3←R3 + 1
STORE   M[314]←R3
```

9-12. Give an example of a program that will cause data conflict in the three-segment pipeline of Sec. 9-5.

9-13. Give an example that uses delayed load with the three-segment pipeline of Sec. 9-5.

9-14. Give an example of a program that will cause a branch penalty in the three-segment pipeline of Sec. 9-5.

9-15. Give an example that uses delayed branch with the three-segment pipeline of Sec. 9-5.

9-16. Consider the multiplication of two 40 × 40 matrices using a vector processor.

a. How many product terms are there in each inner product, and how many inner products must be evaluated?

b. How many multiply–add operations are needed to calculate the product matrix?

9-17. How many clock cycles does it take to process an inner product in the pipeline of Fig. 9-12 when used to evaluate the product of two 60 × 60 matrices? How many inner products are there, and how many clock cycles does it take to evaluate the product matrix?

9-18. Assign addresses to an array of data of 1024 words to be stored in the memory described in Fig. 9-13.

9-19. A weather forecasting computation requires 250 billion floating-point operations. The problem is processed in a supercomputer that can perform 100 megaflops. How long will it take to do these calculations?

9-20. Consider a computer with four floating-point pipeline processors. Suppose that each processor uses a cycle time of 40 ns. How long will it take to perform 400 floating-point operations? Is there a difference if the same 400 operations are carried out using a single pipeline processor with a cycle time of 10 ns?

REFERENCES

1. Dasgupta, S., *Computer Architecture: A Modern Synthesis*, Vol. 2. New York: John Wiley, 1989.

2. DeCegama, A. L., *Parallel Processing Architecture and VLSI Hardware*. Englewood Cliffs, NJ: Prentice Hall, 1989.

3. Gibson, G. A., *Computer Systems Concepts and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
4. Hays, J. F., *Computer Architecture and Organization*, 2nd ed. New York: McGraw-Hill, 1988.
5. Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.
6. Kain, R., *Computer Architecture: Software and Hardware*. Vol. 2. Englewood Cliffs, NJ: Prentice Hall, 1989.
7. Lee, J. K. F., and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design." *Computer*, Vol. 17, No. 1 (January 1984), pp. 6–22.
8. Lilja, D. J., "Reducing the Branch Penalties in Pipeline Processors." *Computer*, Vol. 21, No. 7 (July 1988), pp. 47–55.
9. Patterson, D. A., and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1990.
10. Pollard, L. H., *Computer Design and Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1990.
11. Stone, H. S., *High-Performance Computer Architecture*, 2nd ed. Reading, MA: Addison-Wesley, 1990.
12. Tabak, D., *Multiprocessors*. Englewood Cliffs, NJ: Prentice Hall, 1990.