GOVERNMENT ARTS AND SCIENCE COLLEGE,KOMARAPALAYAM

Department of Computer science

17PCS02-Advanced Computer Architecture

UNIT – II

Handled BY

K.SHANMUGA VADIVU

6

Pipelining and Superscalar Techniques

This chapter deals with pipelining and superscalar design in processor development [illegible] discussion of conventional linear pipelines and analyze their performance. [illegible] pipeline model is introduced to include nonlinear interstage connections. Collision-free scheduling techniques are described for performing dynamic functions.

Specific techniques for building instruction pipelines, arithmetic pipelines, and memory-access pipelines are presented. The discussion includes instruction prefetching, [illegible] data forwarding, software interlocking, hardware scoreboarding, hazard avoidance, [illegible] handling, and instruction-issuing techniques. Both static and multifunctional arithmetic pipelines are designed. Superscalar design techniques are studied along with performance analysis.

6.1 LINEAR PIPELINE PROCESSORS

A *linear pipeline processor* is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other. In modern computers, linear pipelines are applied for instruction execution, arithmetic computation, and memory-access operations.

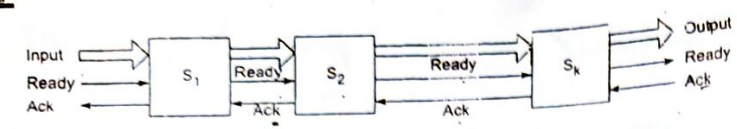
6.1.1 Asynchronous and Synchronous Models

A linear pipeline processor is constructed with k processing stages. External inputs (operands) are fed into the pipeline at the first stage S_1. The processed results are passed from stage S_i to stage S_{i+1}, for all $i = 1, 2, \ldots, k-1$. The final result emerges from the pipeline at the last stage S_k.

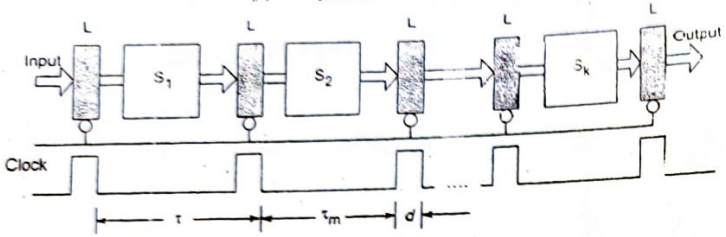
Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: *asynchronous* and *syncnronous*.

Asynchronous Model As shown in Fig. 6.1a, data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol. When stage S_i is ready to transmit, it sends a *ready* signal to stage S_{i+1}. After stage S_{i+1} receives the incoming data, it returns an *acknowledge* signal to S_i.

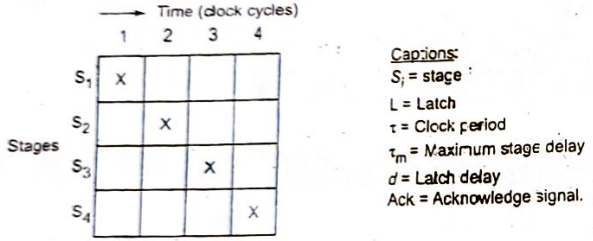
Asynchronous pipelines are useful in designing communication channels in message-passing multicomputers where pipelined wormhole routing is practiced (see Chapter 9). Asynchronous pipelines may have a variable throughput rate. Different amounts of delay may be experienced in different stages.



(a) An asynchronous pipeline model



(b) A synchronous pipeline model



(c) Reservation table of a four-stage linear pipeline

Fig. 6.1 Two models of linear pipeline units and the corresponding reservation table

Synchronous Model Synchronous pipelines are illustrated in Fig. 6.1b. Clocked latches are used to interface between stages. The latches are made with master-slave flip-flops, which can isolate inputs from outputs. Upon the arrival of a clock pulse, all latches transfer data to the next stage simultaneously.

The pipeline stages are combinational logic circuits. It is desired to have approximately equal delays in all stages. These delays determine the clock period and thus the speed of the pipeline. Unless otherwise specified, only synchronous pipelines are studied in this book.

The utilization pattern of successive stages in a synchronous pipeline is specified by a *reservation table*. For a linear pipeline, the utilization follows the diagonal streamline pattern shown in Fig. 6.1c. This table is essentially a space-time diagram depicting the precedence relationship in using the pipeline stages. For a k-stage linear pipeline, k clock cycles are needed for data to flow through the pipeline.

Successive tasks or operations are initiated one per cycle to enter the pipeline. Once the pipeline is filled up, one result emerges from the pipeline for each additional cycle. This throughput is sustained only if the successive tasks are independent of each other.

6.1.2 Clocking and Timing Control

The *clock cycle* τ of a pipeline is determined below. Let τ_i be the time delay of the circuitry in stage S_i and d the time delay of a latch, as shown in Fig. 6.1b.

Clock Cycle and Throughput Denote the *maximum stage delay* as τ_m, and we can write τ as

$$\tau = \max_i \{\tau_i\}_1^k + d = \tau_m + d \tag{6.1}$$

At the rising edge of the clock pulse, the data is latched to the master flip-flops of each latch register. The clock pulse has a width equal to d. In general, $\tau_m >> d$ by one to two orders of magnitude. This implies that the maximum stage delay τ_m dominates the clock period.

The *pipeline frequency* is defined as the inverse of the clock period:

$$f = \frac{1}{\tau} \tag{6.2}$$

If one result is expected to come out of the pipeline per cycle, f represents the *maximum throughput* of the pipeline. Depending on the initiation rate of successive tasks entering the pipeline, the *actual throughput* of the pipeline may be lower than f. This is because more than one clock cycle has elapsed between successive task initiations.

Clock Skewing Ideally, we expect the clock pulses to arrive at all stages (latches) at the same time. However, due to a problem known as *clock skewing*, the same clock pulse may arrive at different stages with a time offset of s. Let t_{max} be the time delay of the longest logic path within a stage and t_{min} that of the shortest logic path within a stage.

To avoid a race in two successive stages, we must choose $\tau_m \geq t_{max} + s$ and $d \leq t_{min} - s$. These constraints translate into the following bounds on the clock period when clock skew takes effect:

$$d + t_{max} + s \leq \tau \leq \tau_m + t_{min} - s \tag{6.3}$$

In the ideal case $s = 0$, $t_{max} = \tau_m$, and $t_{min} = d$. Thus, we have $\tau = \tau_m + d$, consistent with the definition in Eq. 6.1 without the effect of clock skewing.

6.1.3 Speedup, Efficiency, and Throughput

Ideally, a linear pipeline of k stages can process n tasks in $k + (n - 1)$ clock cycles, where k cycles are needed to complete the execution of the very first task and the remaining $n - 1$ tasks require $n - 1$ cycles. Thus the total time required is

$$T_k = [k + (n - 1)]\tau$$

where τ is the clock period. Consider an equivalent-function nonpipelined processor which has a *flow-through delay* of $k\tau$. The amount of time it takes to execute n tasks on this nonpipelined processor is $T_1 = nk\tau$.

Speedup Factor The speedup factor of a k-stage pipeline over an equivalent non pipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)}$$

The *maximum throughput* f occurs when $E_k \to 1$ as $n \to \infty$. This coincides with the speedup definition given in Chapter 3. Note that $H_k = E_k \cdot f = E_k/\tau = S_k/k\tau$. Other relevant factors of instruction pipelines will be discussed in Chapters 12 and 13.

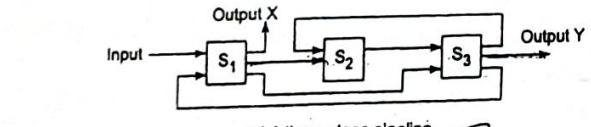
A *dynamic pipeline* can be reconfigured to perform variable functions at different times. The traditional linear pipelines are static pipelines because they are used to perform fixed functions. A dynamic pipeline allows feedforward and feedback connections in addition to the streamline connections. For this reason, some authors call such a structure a *nonlinear pipeline*.

6.2.1 Reservation and Latency Analysis

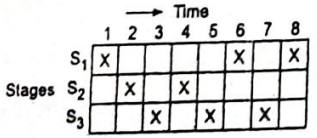
In a static pipeline, it is relatively easy to partition a given function into a sequence of linearly ordered subfunctions. However, function partitioning in a dynamic pipeline becomes quite involved because the pipeline stages are interconnected with loops in addition to streamline connections.

A multifunction dynamic pipeline is shown in Fig. 6.3a. This pipeline has three stages. Besides the *streamline connections* from S_1 to S_2 and from S_2 to S_3, there is a *feed forward connection* from S_1 to S_3 and two *feedback connections* from S_3 to S_2 and from S_3 to S_1.

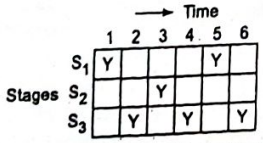
These feedforward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task. With these connections, the output of the pipeline is not necessarily from the last stage. In fact, following different dataflow patterns, one can use the same pipeline to evaluate different functions.



(a) A three-stage pipeline



(b) Reservation table for function X



(c) Reservation table for function Y

Reservation Tables The reservation table for a static linear pipeline is trivial in the sense that dataflow follows a linear streamline. The *reservation table* for a dynamic pipeline becomes more interesting because a nonlinear pattern is followed. Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of different functions.

Two reservation tables are given in Figs. 6.3b and 6.3c, corresponding to a function X and a function Y, respectively. Each function evaluation is specified by one reservation table. A static pipeline is specified by a single reservation table. A dynamic pipeline may be specified by more than one reservation table.

Each reservation table displays the time-space flow of data through the pipeline for one function evaluation. Different functions follow different paths through the pipeline.

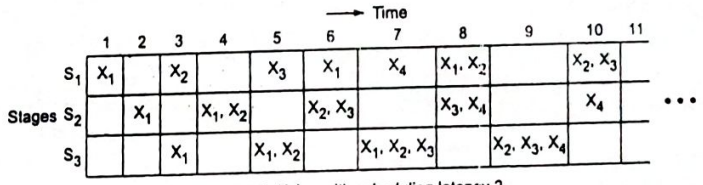
The number of columns in a reservation table is called the *evaluation time* of a given function. For example, the function X requires eight clock cycles to evaluate, and function Y requires six cycles, as shown in Figs. 6.3b and 6.3c, respectively.

A pipeline *initiation* table corresponds to each function evaluation. All initiations to a static pipeline use the same reservation table. On the other hand, a dynamic pipeline may allow different initiations to follow a mix of reservation tables. The checkmarks in each row of the reservation table correspond to the time instants (cycles) that a particular stage will be used.

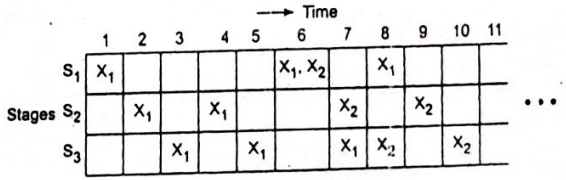
There may be multiple checkmarks in a row, which means repeated usage of the same stage in different cycles. Contiguous checkmarks in a row simply imply the extended usage of a stage over more than one cycle. Multiple checkmarks in a column mean that multiple stages need to be used in parallel during a particular clock cycle.

Latency Analysis The number of time units (clock cycles) between two initiations of a pipeline is *the latency* between them. Latency values must be nonnegative integers. A latency of k means that two initiations are separated by k clock cycles. Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.

A collision implies resource conflicts between two initiations in the pipeline. Therefore, all collisions must be avoided in scheduling a sequence of pipeline initiations. Some latencies will cause collisions, and some will not. Latencies that cause collisions are called *forbidden latencies*. In using the pipeline in Fig. 6.3 to evaluate the function X, latencies 2 and 5 are forbidden, as illustrated in Fig. 6.4.



(a) Collision with scheduling latency 2



(b) Collision with scheduling latency 5

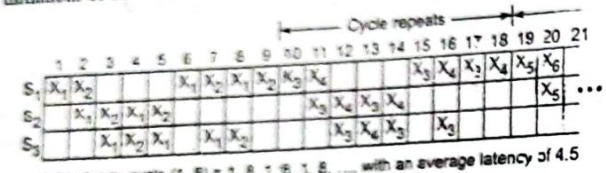
The ith initiation is denoted as X_i in Fig. 6.4. With latency 2, initiations X_1 and X_2 collide in stage 2 at time 4. At time 7, these initiations collide in stage 3. Similarly, other collisions are shown at times 5, 6, 8, ..., etc.

The collision patterns for latency 5 are shown in Fig. 6.4b, where X_1 and X_2 are scheduled 5 clock cycles apart. Their first collision occurs at time 6.

To detect a forbidden latency, one needs simply to check the distance between any two checkmarks in the same row of the reservation table. For example, the distance between the first mark and the second mark in row S_1 in Fig. 6.3b is 5, implying that 5 is a forbidden latency.

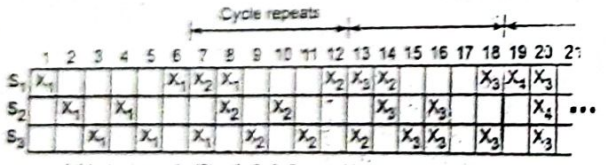
Similarly, latencies 2, 4, 5, and 7 are all seen to be forbidden from inspecting the same reservation table. From the reservation table in Fig. 6.3c, we discover the forbidden latencies 2 and 4 for function Y. A *latency sequence* is a sequence of permissible nonforbidden latencies between successive task initiations.

A *latency cycle* is a latency sequence which repeats the same subsequence (cycle) indefinitely. Figure 6.5 illustrates latency cycles in using the pipeline in Fig. 6.3a to evaluate the function X without causing a collision. For example, the latency cycle (1, 8) represents the infinite latency sequence 1, 8, 1, 8, 1, 8, ... This implies that successive initiations of new tasks are separated by one cycle and eight cycles alternately.



(a) Latency cycle (1, 8) = 1, 8, 1, 8, 1, 8, ... with an average latency of 4.5

(b) Latency cycle (3) = 3, 3, 3, 3, ... with an average latency of 3



(c) Latency cycle (6) = 6, 6, 6, 6, ... with an average latency of 6

Fig. 6.5 Three valid latency cycles for the evaluation of function X

The *average latency* of a latency cycle is obtained by dividing the sum of all latencies by the number of latencies along the cycle. The latency cycle (1, 8) thus has an average latency of $(1 + 8)/2 = 4.5$. A *constant cycle* is a latency cycle which contains only one latency value. Cycles (3) and (6) in Figs. 6.5b and 6.5c are both constant cycles. The average latency of a constant cycle is simply the latency itself. In the next section, we describe how to obtain these latency cycles systematically.

6.2.2 Collision-Free Scheduling

When scheduling events in a nonlinear pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions. In what follows, we present a systematic method for finding such collision-free scheduling.

We study below *collision vectors, state diagrams, single cycles, greedy cycles,* and *minimal average latency* (MAL). This pipeline design theory was originally developed by Davidson (1971) and Kogge (1981).

Collision Vectors By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies. For a reservation table with n columns, the *maximum forbidden latency* $m \le n - 1$. The permissible latency p should be as small as possible. The choice is made in the range $1 \le p \le m - 1$.

A permissible latency of $p = 1$ corresponds to the ideal case. In theory, a latency of 1 can be achieved in a static pipeline which follows a linear (diagonal or streamlined) reservation table as in Fig. 6.1c.

The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \ldots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision, and $C_i = 0$ if latency i is permissible. Note that it is always true that $C_m = 1$, corresponding to the maximum forbidden latency.

For the two reservation tables in Fig. 6.3, the collision vector $C_X = (1011010)$ is obtained for function X and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies 7, 5, 4, and 2 are forbidden and latencies 6, 3, and 1 are permissible. Similarly, 4 and 2 are forbidden latencies and 3 and 1 are permissible latencies for function Y.

State Diagrams From the above collision vector, one can construct a *state diagram* that shows the permissible state transitions among successive initiations. The collision vector, like C_X above, corresponds to the *initial state* of the pipeline at time 1 and thus is called an *initial collision vector*. Let p be a permissible latency within the range $1 \le p \le m - 1$.

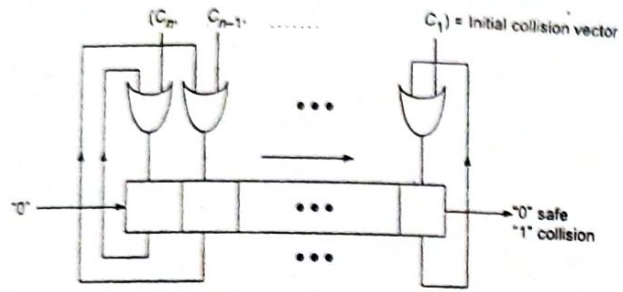
The *next state* of the pipeline at time $t + p$ is obtained with the assistance of an m-bit right shift register in Fig. 6.6a. The initial collision vector C is initially loaded into the register. The register is then shifted right. Each 1-bit shift corresponds to an increase in the latency by 1. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency. Likewise, a 1 bit being shifted out means a collision and thus the corresponding latency should be forbidden.

Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents. For example, from the initial state $C_X = (1011010)$, the next state (1111111) is reached after one right shift of the register, and the state (1011011) is reached after three shifts or six shifts.

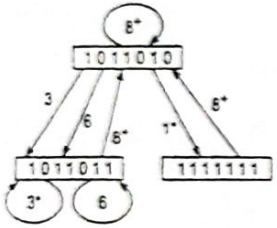
Example 6.2 The state transition diagram for a pipeline

A *state diagram* is obtained in Fig. 6.6b for function X. From the initial state (1011010), only three transitions are possible, corresponding to the three permissible latencies 6, 3, and 1 in the collision vector. Similarly, from state (1011011), one reaches the same state after either three shifts or six

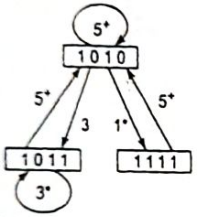
When the number of shifts is $m + 1$ or greater, all transitions are redirected back to the initial state. For example, after eight or more (denoted as 8^+) shifts, the next state must be the initial state, regardless of which state the transition starts from. In Fig. 6.6c, a state diagram is obtained for the reservation table in Fig. 6.3c using a 4-bit shift register. Once the initial collision vector is determined, the corresponding state diagram is uniquely determined.



(a) State transition using an n-bit right shift register, where n is the maximum forbidden latency



(b) State diagram for function X



(c) State diagram for function Y

Figure 6.6 [illegible]

The 0's and 1's in the present state, say at time t, of a state diagram indicate the permissible and forbidden latencies, respectively, at time t. The bitwise ORing of the shifted version of the present state with the initial collision vector is meant to prevent collisions from future initiations starting at time $t + 1$ and onward.

Thus the state diagram covers all permissible state transitions that avoid collisions. All latencies equal to or greater than m are permissible. This implies that collisions can always be avoided if events are scheduled far apart (with latencies of m^+). However, such long latencies are not tolerable from the viewpoint of pipeline throughput.

Greedy Cycles From the state diagram, we can determine optimal latency cycles which result in the MAL. There are infinitely many latency cycles one can trace from the state diagram. For example, (1, 8), (1, 8, 6, 8), (3), (6), (3, 8), (3, 6, 3) ..., are legitimate cycles traced from the state diagram in Fig. 6.6b. Among these cycles, only *simple cycles* are of interest.

A simple cycle is a latency cycle in which each state appears only once. In the state diagram in Fig. 6.6b, only (3), (6), (8), (1, 8), (3, 8), and (6, 8) are simple cycles. The cycle (1, 8, 6, 8) is not simple because it travels through the state (1011010) twice. Similarly, the cycle (3, 6, 3, 8, 6) is not simple because it repeats the state (1011011) three times.

Some of the simple cycles are *greedy cycles*. A greedy cycle is one whose edges are all made with minimum latencies from their respective starting states. For example, in Fig. 6.6b the cycles (1, 8) and (3) are greedy cycles. Greedy cycles in Fig. 6.6c are (1, 5) and (3). Such cycles must first be simple, and their average latencies must be lower than those of other simple cycles. The greedy cycle (1, 8) in Fig. 6.6b has an average latency of $(1 + 8)/2 = 4.5$, which is lower than that of the simple cycle $(6, 8) = (6 + 8)/2 = 7$. The greedy cycle (3) has a constant latency which equals the MAL for evaluating function X without causing a collision.

The MAL in Fig. 6.6c is 3, corresponding to either of the two greedy cycles. The minimum-latency edges in the state diagrams are marked with asterisks.

At least one of the greedy cycles will lead to the MAL. The collision-free scheduling of pipeline events is thus reduced to finding greedy cycles from the set of simple cycles. The greedy cycle yielding the MAL is the final choice.

6.2.3 Pipeline Schedule Optimization

An optimization technique based on the MAL is given below. The idea is to insert noncompute delay stages into the original pipeline. This will modify the reservation table, resulting in a new collision vector and an improved state diagram. The purpose is to yield an optimal latency cycle, which is absolutely the shortest.

Bounds on the MAL In 1972, Shar determined the following bounds on the *minimal average latency* (MAL) achievable by any control strategy on a statically reconfigured pipeline executing a given reservation table:

(1) The MAL is lower-bounded by the maximum number of checkmarks in any row of the reservation table.
(2) The MAL is lower than or equal to the average latency of any greedy cycle in the state diagram.
(3) The average latency of any greedy cycle is upper-bounded by the number of 1's in the initial collision vector plus 1. This is also an upper bound on the MAL.

Interested readers may refer to Shar (1972) or find proofs of these bounds in Kogge (1981). These results suggest that the optimal latency cycle must be selected from one of the lowest greedy cycles. However, a greedy cycle is not sufficient to guarantee the optimality of the MAL. The lower bound guarantees the optimality. For example, the MAL = 3 for both function X and function Y and has met the lower bound of 3 from their respective reservation tables.

From Fig. 6.6b, the upper bound on the MAL for function X is equal to $4 + 1 = 5$, a rather loose bound. On the other hand, Fig. 6.6c shows a rather tight upper bound of $2 + 1 = 3$ on the MAL. Therefore, all greedy cycles for function Y lead to the optimal latency value of 3, which cannot be lowered further.

To optimize the MAL, one needs to find the lower bound by modifying the reservation table. The approach is to reduce the maximum number of checkmarks in any row. The modified reservation table must preserve the original function being evaluated. Patel and Davidson (1976) have suggested the use of noncompute delay stages to increase pipeline performance with a shorter MAL. Their technique is described below.

a contrary conclusion. The relationship between the two measures is a function of the reservation table and of the initiation cycle adopted.

At least one stage of the pipeline should be fully (100%) utilized at the steady state in any acceptable initiation cycle; otherwise, the pipeline capability has not been fully explored. In such cases, the initiation cycle may not be optimal and another initiation cycle should be examined for improvement.

6.3 INSTRUCTION PIPELINE DESIGN

A stream of instructions can be executed by a pipeline in an overlapped manner. We describe below instruction pipelines for CISC and RISC scalar processors. Topics to be studied include instruction prefetching, data forwarding, hazard avoidance, interlocking for resolving data dependences, dynamic instruction scheduling, and branch handling techniques for improving pipelined processor performance. Further discussion on instruction level parallelism will be found in Chapter 12.

6.3.1 Instruction Execution Phases

A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and write-back phases. These phases are ideal for overlapped execution on a linear pipeline.

Pipelined Instruction Processing A typical instruction pipeline is depicted in Fig. 6.9. The *fetch stage* (F) fetches instructions from a cache memory, ideally one per cycle. The *decode stage* (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The *issue stage* (I) reserves resources. The operands are also read from registers during the issue stage.

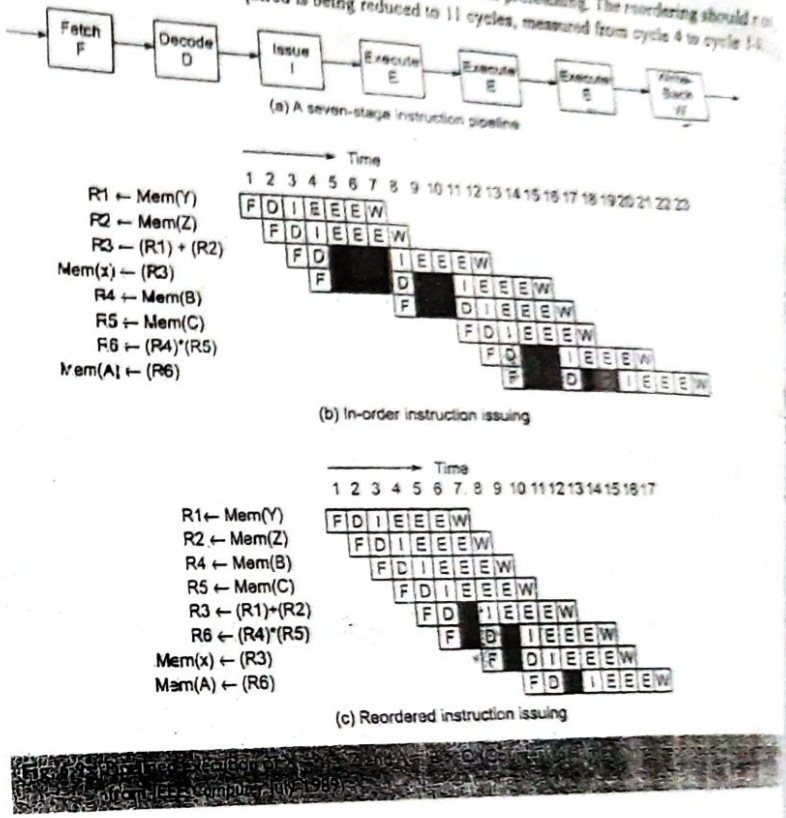
The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 6.9a. The last *writeback stage* (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 6.9 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements $X = Y + Z$ and $A = B \times C$. Here we have assumed that *load* and *store* instructions take four execution clock cycles, while floating-point *add* and *multiply* operations take three cycles.

The above timing assumptions represent typical values found in an older CISC processor. In many RISC processors, fewer clock cycles are needed. On the other hand, Cray 1 required 11 cycles for a load and a floating-point addition took six. With in-order instruction issuing, if an instruction is blocked from issuing due to a data or resource dependence, all instructions following it are blocked.

Figure 6.9b illustrates the issue of instructions following the original program order. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resource latency or conflicts or due to data dependences. The first two *load* instructions issue on consecutive cycles. The *add* is dependent on both *loads* and must wait three cycles before the data (Y and Z) are loaded in.

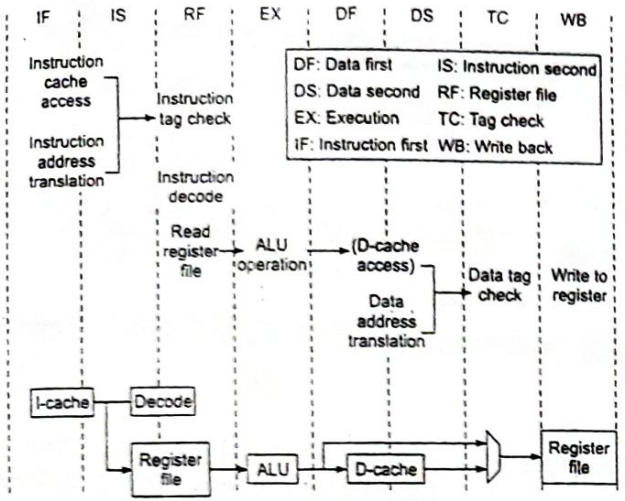
Similarly, the *store* of the sum to memory location X must wait three cycles for the *add* to finish due to a flow dependence. There are similar blockages during the calculation of A. The total time required is 17 clock cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20 when the last instruction starts execution. This timing measure eliminates the undue effects of the pipeline "startup" or "draining" delays.

Figure 6.9c shows an improved timing after the instruction issuing order is changed to eliminate unnecessary delays due to dependence. The idea is to issue all four *load* operations in the beginning. Both the *add* and *multiply* instructions are blocked fewer cycles due to this data prefetching. The reordering should not change the end results. The time required is being reduced to 11 cycles, measured from cycle 4 to cycle 14.

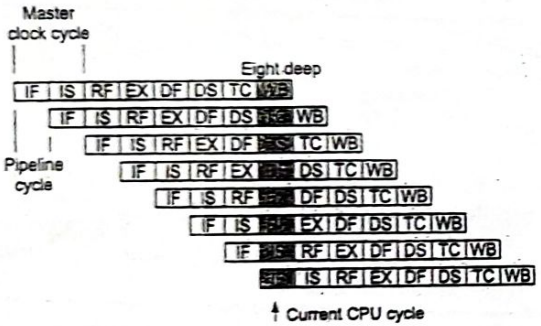


Example 6.4 The MIPS R4000 instruction pipeline

The MIPS R4000 was a pipelined 64-bit processor using separate instruction and data caches in an eight-stage pipeline for executing register-based instructions. As illustrated in Fig. 6.10, the processor design was targeted to achieve an execution rate approaching one instruction per cycle.



(a) R4000 pipeline stages



(b) R4000 instruction overlapping in pipeline

Fig. 6.10 [illegible]

The execution of each R4000 instruction consisted of eight major steps as summarized in Fig. 6.10a. Each of these steps required approximately one clock cycle. The instruction and data memory references are split across two stages. The single-cycle ALU stage took slightly more time than each of the cache access stages.

The overlapped execution of successive instructions is shown in Fig. 6.10b. This pipeline operated efficiently because different CPU resources, such as address and bus access, ALU operations, register accesses, and so on, were utilized simultaneously on a noninterfering basis.

The internal pipeline clock rate (100 MHz) of the R4000 was twice the external input or master clock

frequency. Figure 6.10b shows the optimal pipeline movement, completing one instruction every internal clock cycle. Load and branch instructions introduce extra delays.

6.3.2 Mechanisms for Instruction Pipelining

We introduce instruction buffers and describe the use of cacheing, collision avoidance, multiple functional units, register tagging, and internal forwarding to smooth pipeline flow and to remove bottlenecks and unnecessary memory access operations.

Prefetch Buffers Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate. In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer as illustrated in Fig. 6.11. The block access can be achieved using interleaved memory modules or using a cache to shorten the effective memory-access time as demonstrated in the MIPS R4000.

prefetch Buffers

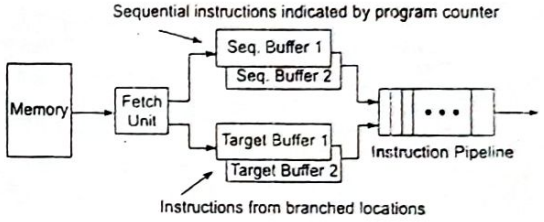


Fig. 6.11 The use of sequential and target buffers

Sequential instructions are loaded into a pair of *sequential buffers* for in-sequence pipelining. Instructions from a branch target are loaded into a pair of *target buffers* for out-of-sequence pipelining. Both buffers operate in a first-in-first-out fashion. These buffers become part of the pipeline as additional stages.

A conditional branch instruction causes both sequential buffers and target buffers to fill with instructions. After the branch condition is checked, appropriate instructions are taken from one of the two buffers, and instructions in the other buffer are discarded. Within each pair, one can use one buffer to load instructions from memory and use another buffer to feed instructions into the pipeline. The two buffers in each pair alternate to prevent a collision between instructions flowing into and out of the pipeline.

A third type of prefetch buffer is known as a *loop buffer*. This buffer holds sequential instructions contained in a small loop. The loop buffers are maintained by the fetch stage of the pipeline. Prefetched instructions in the loop body will be executed repeatedly until all iterations complete execution. The loop buffer operates in two steps. First, it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes when the target of a branch falls within the loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer. The CDC 6600 and Cray 1 made use of loop buffers.

Multiple Functional Units Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 6.12).

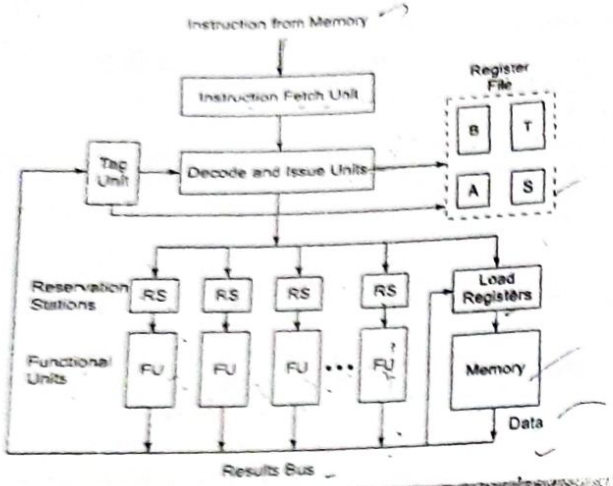


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging. (Courtesy of G. Sohi, reprinted with permission from IEEE Transactions on Computers, March 1990)

Sohi (1990) used a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resource dependences among the successive instructions entering the pipeline, the *reservation stations* (RS) are used with each functional unit. Operations wait in the RS until their data dependences have been resolved. Each RS is uniquely identified by a *tag*, which is monitored by a *tag unit*.

The tag unit keeps checking the tags from all currently used registers or RSs. This register tagging technique allows the hardware to resolve conflicts between source and destination registers assigned for multiple instructions. Besides resolving conflicts, the RSs also serve as buffers to interface the pipelined functional units with the decode and issue units. The multiple functional units operate in parallel, once the dependences are resolved. This alleviates the bottleneck in the execution stages of the instruction pipeline.

Internal Data Forwarding The throughput of a pipelined processor can be further improved with internal data forwarding among multiple functional units. In some cases, some memory-access operations can be replaced by register transfer operations. The idea is described in Fig. 6.13.

A *store-load forwarding* is shown in Fig. 6.13a in which the *load operation* (LD R2, M) from memory to register R2 can be replaced by the *move* operation (MOVE R2, R1) from register R1 to register R2. Since register transfer is faster than memory access, this data forwarding will reduce memory traffic and thus results in a shorter execution time. Similarly, *load-load forwarding* (Fig. 6.13b) eliminates the second

load operation (LD R2, M) and replaces it with the *move* operation (MOVE R2, R1). Further discussion of operand forwarding will be continued in Chapter 12.

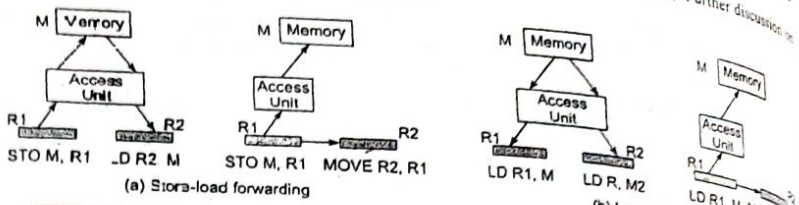


Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

Example 6.5 Implementing the dot-product operation with internal data forwarding between a multiply unit and an add unit

One can feed the output of a multiplier directly to the input of an adder (Fig. 6.14) for implementing the following dot-product operation:

$$s = \sum_{i=1}^{n} a_i \times b_i$$

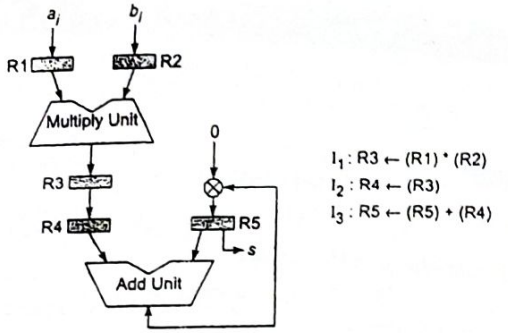
Without internal data forwarding between the two functional units, the three instructions must be sequentially executed in a looping structure (Fig. 6.14a). With data forwarding, the output of the multiplier is fed directly into the input register R4 of the adder (Fig. 6.14b). At the same time, the output of the multiplier is also routed to register R3. Internal data forwarding between the two functional units thus reduces the execution time through the pipelined processor.

Hazard Avoidance The *read* and *write* of shared variables by different instructions in a pipeline may lead to different results if these instructions are executed out of order. As illustrated in Fig. 6.15, three types of logic *hazards* are possible.

Consider two instructions I and J. Instruction J is assumed to logically follow instruction I according to program order. If the actual execution order of these two instructions violates the program order, incorrect results may be read or written, thereby producing hazards.

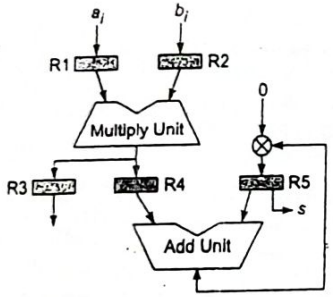
Hazards should be prevented before these instructions enter the pipeline, such as by holding instruction J until the dependence on instruction I is resolved. We use the notation D(I) and R(I) for the *domain* and *range* of an instruction I.

The domain contains the *input set* (such as operands in registers or in memory) to be used by instruction I. The range corresponds to the *output set* of instruction I. Listed below are conditions under which possible hazards can occur:



I_1 : R3 ← (R1) * (R2)
I_2 : R4 ← (R3)
I_3 : R5 ← (R5) + (R4)

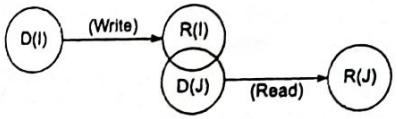
(a) Without data forwarding



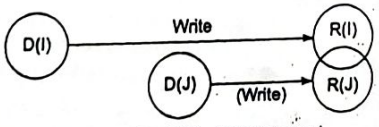
I'_1 : R3 ← (R1) * (R2)
I'_2 : R4 ← (R1) * (R2)
I'_3 : R5 ← (R4) + (R5)

I'_1 and I'_2 can be executed simultaneously with internal data forwarding.

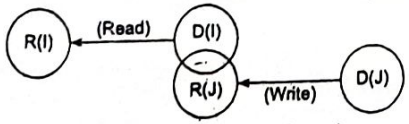
(b) With internal data forwarding



(a) Read-after-Write (RAW) hazard



(b) Write-after-Write (WAW) hazard



(c) Write-after-Read (WAR) hazard

$$R(I) \cap D(J) \neq \phi \text{ for RAW hazard}$$
$$R(I) \cap R(J) \neq \phi \text{ for WAW hazard} \quad (6.11)$$
$$D(I) \cap R(J) \neq \phi \text{ for WAR hazard}$$

These conditions are necessary but not sufficient. This means the hazard may not appear even if one or more of the conditions exist. The RAW hazard corresponds to the flow dependence, WAR to the antidependence, and WAW to the output dependence introduced in Section 2.1. The occurrence of a logic hazard depends on the order in which the two instructions are executed. Chapter 12 discusses techniques to handle such hazards.

6.3.3 Dynamic Instruction Scheduling

In this section, we describe three methods for scheduling instructions through an instruction pipeline. The *static scheduling* scheme is supported by an optimizing compiler. *Dynamic scheduling* is achieved using a technique such as Tomasulo's *register-tagging* scheme built in the IBM 360/91, or the *scoreboarding* scheme built in the CDC 6600 processor.

Static Scheduling Data dependences in a sequence of instructions create interlocked relationships among them. Interlocking can be resolved through a compiler-based static scheduling approach. A compiler or a postprocessor can be used to increase the separation between interlocked instructions.

Consider the execution of the following code fragment. The *multiply* instruction cannot be initiated until the preceding *load* is complete. This data dependence will stall the pipeline for three clock cycles since the two *loads* overlap by one cycle.

Instruction:	
Add	R0, R1
Move	R1, R5
Load	R2, M(α)
Load	R3, M(β)
Multiply	R2, R3

The two *loads*, since they are independent of the *add* and *move*, can be moved ahead to increase the spacing between them and the *multiply* instruction. The following program is obtained after this modification:

Load	R2, M(α)
Load	R3, M(β)
Add	R0, R1
Move	R1, R5
Multiply	R2, R3

Through this code rearrangement, the data dependences and program semantics are preserved, and the *multiply* can be initiated without delay. While the operands are being loaded from memory cells α and β into registers R2 and R3, the two instructions *add* and *move* consume three cycles and thus pipeline stalling is avoided.

The technique is similar to that used for software interlocking. NOPs can be used as fillers if needed. The probability of moving one instruction ($d = 2$ in Fig. 6.20a) into the delay slot is greater than 0.6, that of moving two instructions ($d = 3$ in Fig. 6.20b) is about 0.2, and that of moving three instructions ($d = 4$ in Fig. 6.20c) is less than 0.1, according to some program trace results.

Example 6.8 A delayed branch with code motion into a delay slot

Code motion across branches can be used to achieve a delayed branch, as illustrated in Fig. 6.21. Consider the execution of a code fragment in Fig. 6.21a. The original program is modified by moving the useful instruction I1 into the delay slot after the branch instruction I3.

```
I1.  LOAD   R1, A
I2.  Dec    R3, 1
I3.  BrZero R3, I5
I4.  Add    R2, R4
I5.  Sub    R5, R6
I6.  Store  R5, B
```

(a) Original program

```
I2.  Dec    R3, 1
I3.  BrZero R3, I5
I1.  Load   R1, A
I4.  Add    R2, R4
I5.  Sub    R5, R6
I6.  Store  R5, B
```

(b) Moving useful instructions into the delay slot

Fig. 6.21 Code motion across a branch to achieve a delayed branch with one slot for improved performance

In case the branch is not taken, the execution of the modified program produces the same results as the original program. In case the branch is taken in the modified program, execution of the delayed instructions I1 and I5 is needed anyway.

In general, data dependence between instructions moving across the branch and the remaining instructions being scheduled must be analyzed. Since instruction I1 is independent of the remaining instructions, leaving it in the delay slot will not create logic hazards or data dependences.

Sometimes NOP fillers can be inserted in the delay slot if no useful instructions can be found. However, inserting NOP fillers does not save any cycles in the delayed branch operation. From the above analysis one can conclude that delayed branching may be more effective in short instruction pipelines with about four stages. Delayed branching has been built into some RISC processors, including the MIPS R4000 and Motorola MC88110.

6.4 ARITHMETIC PIPELINE DESIGN

Pipelining techniques can be applied to speed up numerical arithmetic computations. We start with a review of arithmetic principles and standards. Then we consider arithmetic pipelines with fixed functions.

A fixed-point multiply pipeline design and the MC68040 floating-point unit are used as examples to illustrate the design techniques involved. A multifunction arithmetic pipeline is studied with the TI-ASC arithmetic processor as an example.

6.4.1 Computer Arithmetic Principles

In a digital computer, arithmetic is performed with *finite precision* due to the use of fixed-size memory words or registers. Fixed-point or integer arithmetic offers a fixed range of numbers that can be operated upon. Floating-point arithmetic operates over a much increased dynamic range of numbers.

In modern processors, fixed-point and floating-point arithmetic operations are very often performed by separate hardware on the same processor chip.

Finite precision implies that numbers exceeding the limit must be truncated or rounded to provide precision within the number of significant bits allowed. In the case of floating-point numbers, exceeding the exponent range means error conditions, called *overflow* or *underflow*. The Institute of Electrical and Electronics Engineers (IEEE) has developed standard formats for 32- and 64-bit floating numbers known as the *IEEE 754 Standard*. This standard has been adopted for most of today's computers.

Fixed-Point Operations Fixed-point numbers are represented internally in machines in *sign-magnitude*, *one's complement*, or *two's complement* notation. Most computers use the two's complement notation because of its unique representation of all numbers (including zero). One's complement notation introduces a second zero representation called *dirty zero*.

Add, *subtract*, *multiply*, and *divide* are the four primitive arithmetic operations. For fixed-point numbers the add or subtract of two n-bit integers (or fractions) produces an n-bit result with at most one carry-out.

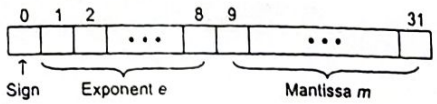
The multiplication of two n-bit numbers produces a $2n$-bit result which requires the use of two memory words or two registers to hold the full-precision result.

The division of an n-bit number by another may create an arbitrarily long quotient and a remainder. Only an approximate result is expected in fixed-point division with rounding or truncation. However, one can expand the precision by using a $2n$-bit dividend and an n-bit divisor to yield an n-bit quotient.

Floating-Point Numbers A floating-point number X is represented by a pair (m, e), where m is the mantissa (or *fraction*) and e is the *exponent* with an implied *base* (or *radix*). The algebraic value is represented as $m \times r^e$. The sign of X can be embedded in the mantissa.

Example 6.9 The IEEE 754 floating-point standard

A 32-bit floating-point number is specified in the IEEE 754 Standard as follows:



A binary base is assumed with $r = 2$. The 8-bit exponent e field uses an *excess-127* code. The dynamic range of e is (−127, 128), internally represented as (0, 255). The sign s and the 23-bit mantissa field m form a 25-bit sign-magnitude fraction, including an implicit or "hidden" 1 bit to the left of the binary point. Thus the complete mantissa actually represents the value 1.m in binary.

This hidden bit is not stored with the number. If $0 < e < 255$, then a nonzero normalized number represents the following algebraic value:

$$X = (-1)^s \times 2^{e-127} \times (1.m) \tag{6.15}$$

When $e = 255$ and $m \neq 0$, a *not-a-number* (NaN) is represented. NaNs can be caused by dividing a zero by a zero or taking the square root of a negative number, among many other nondeterminate cases. When $e = 255$ and $m = 0$, an infinite number $X = (-1)^s \infty$ is represented. Note that $+\infty$ and $-\infty$ are represented differently.

When $e = 0$ and $m \neq 0$, the number represented is $X = (-1)^s 2^{-126}(0.m)$. When $e = 0$ and $m = 0$, a zero is represented as $X = (-1)^s 0$. Again, $+0$ and -0 are possible.

The 64-bit (double-precision) floating point can be defined similarly using an excess-1023 code in the exponent field and a 52-bit mantissa field. A number which is nonzero, finite, non-NaN, and normalized, has the following value:

$$X = (-1)^s \times 2^{e-1023} \times (1.m) \tag{6.16}$$

Special rules are given in the standard to handle overflow or underflow conditions. Interested readers may check the published IEEE standards for details.

Floating-Point Operations The four primitive arithmetic operations are defined below for a pair of floating-point numbers represented by $X = (m_x, e_x)$ and $Y = (m_y, e_y)$. For clarity, we assume $e_x \leq e_y$ and base $r = 2$.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y} \tag{6.17}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y} \tag{6.18}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y} \tag{6.19}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y} \tag{6.20}$$

The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa operations, including four types of fixed-point operations.

Floating-point units are ideal for pipelined implementation. The two halves of the operations demand almost twice as much hardware as that required in a fixed-point unit. Arithmetic shifting operations are needed for equalizing the two exponents before their mantissas can be added or subtracted.

Shifting a binary fraction m to the right k places corresponds to the weighting $m \times 2^{-k}$, and shifting k places to the left corresponds to $m \times 2^k$. In addition, normalization of a floating-point number also requires left shifts to be performed.

Elementary Functions Elementary functions include trigonometric, exponential, logarithmic, and other transcendental functions. Truncated polynomials or power series can be used to evaluate the elementary functions, such as $\sin x$, $\ln x$, e^x, $\cosh x$, $\tan^{-1} y$, $\sqrt{x}$, x^3, etc. Interested readers may refer to the book by Hwang (1979) for details of computer arithmetic functions and their hardware implementation.

It should be noted that computer arithmetic can be implemented by hardwired logic circuitry as well as by table lookup using fast memory. Frequently used constants and special function values can also be generated by table lookup.

6.4.2 Static Arithmetic Pipelines

Most of today's arithmetic pipelines are designed to perform fixed functions. These *arithmetic/logic units* (ALUs) perform fixed-point and floating-point operations separately. The fixed-point unit is also called the integer unit. The floating-point unit can be built either as part of the central processor or on a separate coprocessor.

These arithmetic units perform scalar operations involving one pair of operands at a time. The pipelining in scalar arithmetic pipelines is controlled by software loops. Vector arithmetic units can be designed with pipeline hardware directly under firmware or hardwired control.

Scalar and vector arithmetic pipelines differ mainly in the areas of register files and control mechanisms involved. Vector hardware pipelines are often built as add-on options to a scalar processor or as an attached processor driven by a control processor. Both scalar and vector processors are used in modern supercomputers.

Arithmetic Pipeline Stages Depending on the function to be implemented, different pipeline stages in an arithmetic unit require different hardware logic. Since all arithmetic operations (such as *add, subtract, multiply, divide, squaring, square rooting, logarithm,* etc.) can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add and to shift.

For example, a typical three-stage floating-point adder includes a first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic; a second stage for fraction addition using a high-speed carry lookahead adder; and a third stage for fraction normalization and exponent readjustment using a shifter and another addition logic.

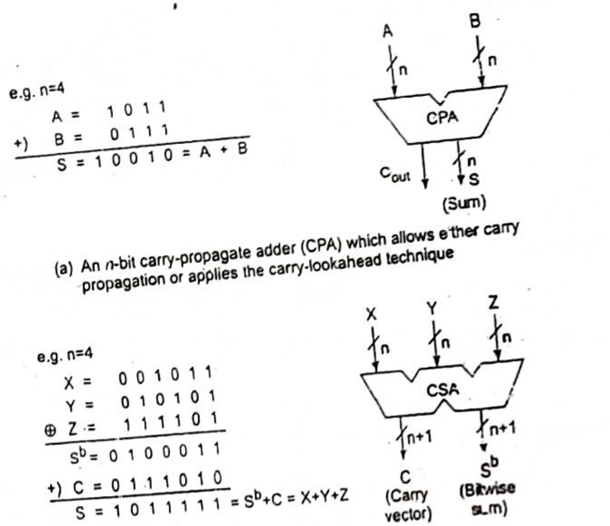
Arithmetic or logical shifts can be easily implemented with *shift registers*. High-speed addition requires either the use of a *carry-propagation adder* (CPA) which adds two numbers and produces an arithmetic sum as shown in Fig. 6.22a, or the use of a *carry-save adder* (CSA) to "add" three input numbers and produce one sum output and a carry output as exemplified in Fig. 6.22b.

In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry looka-head technique.

In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector. In general, an n-bit CSA is specified as follows: Let X, Y, and Z be three n-bit input numbers, expressed as $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ and so on. The CSA performs bitwise operations simultaneously on all columns of digits to produce two n-bit output numbers, denoted as $S^b = (0, S_{n-1}, S_{n-2}, \ldots, S_1, S_0)$ and $C = (C_n, C_{n-1}, \ldots, C_1, 0)$.

Note that the leading bit of the *bitwise sum* S^b is always a 0, and the tail bit of the *carry vector* C is always a 0. The input-output relationships are expressed below:

$$S_i = x_i \oplus y_i \oplus z_i$$
$$C_{i+1} = x_i y_i \vee y_i z_i \vee z_i x_i \tag{6.21}$$



(a) An n-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

(b) An n-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

Fig. 6.22 Distinction between a carry-propagate adder (CPA) and a carry-save adder (CSA)

for $i = 0, 1, 2, \ldots, n-1$, where ⊕ is the exclusive OR and ∨ is the logical OR operation. Note that the arithmetic sum of three input numbers, i.e., $S = X + Y + Z$, is obtained by adding the two output numbers, i.e., $S = S^b + C$, using a CPA. We use the CPA and CSAs to implement the pipeline stages of a fixed-point multiply unit as follows.

Multiply Pipeline Design Consider as an example the multiplication of two 8-bit integers $A \times B = P$, where P is the 16-bit product. This fixed-point multiplication can be written as the summation of eight partial products as shown below: $P = A \times B = P_0 + P_1 + P_2 + \cdots + P_7$, where × and + are arithmetic multiply and add operations, respectively.

```
                     1 0 1 1 0 1 0 1 = A
                 ×)  1 0 0 1 0 0 1 1 = B
    ---------------------------------------
                     1 0 1 1 0 1 0 1 = P0
                   1 0 1 1 0 1 0 1 0 = P1
                 0 0 0 0 0 0 0 0 0 0 = P2
               0 0 0 0 0 0 0 0 0 0 0 = P3
           1 0 1 1 0 1 0 1 0 0 0 0 0 = P4
         0 0 0 0 0 0 0 0 0 0 0 0 0 0 = P5
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = P6
  +) 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 = P7
    ---------------------------------------
     0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 = P
```


Note that the partial product P_j is obtained by multiplying the multiplicand A by the jth bit of B and then shifting the result j bits to the left for $j = 0, 1, 2, \ldots, 7$. Thus P_j is $(8 + j)$ bits long with j trailing zeros. The summation of the eight partial products is done with a *Wallace tree* of CSAs plus a CPA at the final stage, as shown in Fig. 6.23.

The first stage (S_1) generates all eight partial products, ranging from 8 bits to 15 bits, simultaneously. The second stage (S_2) is made up of two levels of four CSAs, and it essentially merges eight numbers into four numbers ranging from 13 to 15 bits. The third stage (S_3) consists of two CSAs, and it merges four numbers from S_2 into two 16-bit numbers. The final stage (S_4) is a CPA, which adds up the last two numbers to produce the final product P.

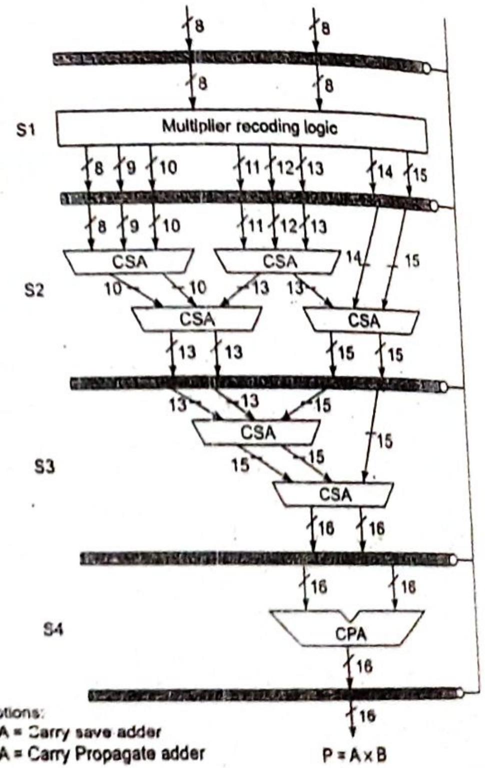


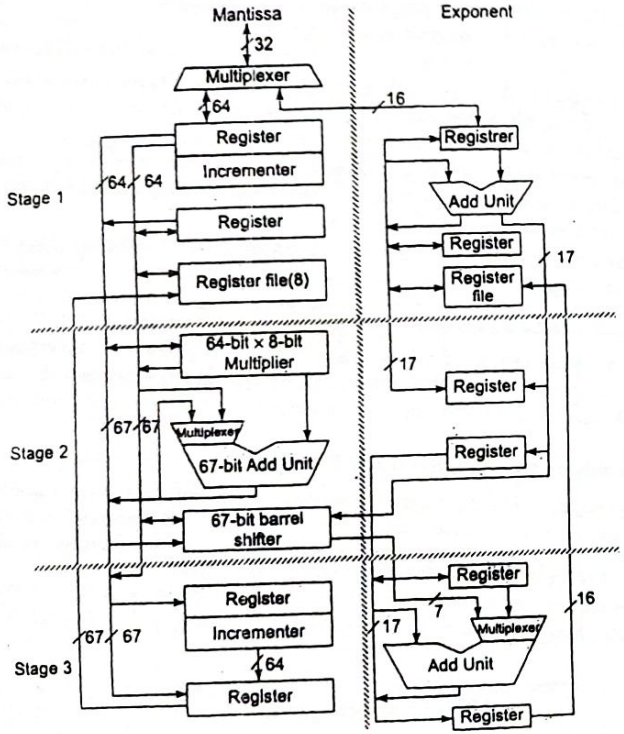
Fig. 6.23 [illegible]

For a maximum width of 16 bits, the CPA is estimated to need four gate levels of delay. Each [illegible] CSA can be implemented with a two-gate-level logic. The delay of the first stage (S_1) also involves [illegible] levels. Thus all the pipeline stages have an approximately equal amount of delay.

The matching of stage delays is crucial to the determination of the number of pipeline stages, as well as the clock period (Eq. 6.1). If the delay of the CPA stage can be further reduced to match that of a single CSA level, then the pipeline can be divided into six stages with a clock rate twice as fast. The basic concepts can be extended to operands with a larger number of bits, as we see in the example below.

Example 6.10 The floating-point unit in the Motorola MC68040

Figure 6.24 shows the design of a pipelined floating-point unit built as an on-chip feature in the Motorola M68040 processor.



This arithmetic pipeline has three stages. The mantissa section and exponent section are essentially two

separate pipelines. The mantissa section can perform floating-point add or multiply operations, either single-precision (32 bits) or double-precision (64 bits).

In the mantissa section, stage 1 receives input operands and returns with computation results; 64-bit registers are used in this stage. Note that all three stages are connected to two 64-bit data buses. Stage 2 contains the array multiplier (64 × 8) which must be repeatedly used to carry out a long multiplication of the two mantissas.

The 67-bit adder performs the addition/subtraction of two mantissas, the barrel shifter is used for normalization. Stage 3 contains registers for holding results before they are loaded into the register file in stage 1 for subsequent use by other instructions.

On the exponent side, a 16-bit bus is used between stages. Stage 1 has an exponent adder for comparing the relative magnitude of two exponents. The result of stage 1 is used to equalize the exponents before mantissa addition can be performed. Therefore, a shift count (from the output of the exponent adder) is sent to the barrel shifter for mantissa alignment.

After normalization of the final result (getting rid of leading zeros), the exponent needs to be readjusted in stage 3 using another adder. The final value of the resulting exponent is fed from the register in stage 3 to the register file in stage 1, ready for subsequent usage.

Convergence Division One technique for division involves repeated multiplications. Mantissa division is carried out by a *convergence method*. This convergence division obtains the quotient $Q = M/D$ of two normalized fractions $0.5 \le M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q = \frac{M \times R_1 \times R_2 \times \cdots \times R_k}{D \times R_1 \times R_2 \times \cdots \times R_k} \tag{6.22}$$

where the successive multipliers

$$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)} \quad \text{for } i = 1, 2, \ldots, k \quad \text{and} \quad D = 1 - \delta$$

The purpose is to choose R_i such that the denominator $D^{(k)} = D \times R_1 \times R_2 \times \cdots \times R_k \to 1$ for a sufficient number of k iterations, and then the resulting numerator $M \times R_1 \times R_2 \times \cdots \times R_k \to Q$.

Note that the multiplier R_i can be obtained by finding the two's complement of the previous chain product $D^{(i)} = D \times R_1 \times \cdots \times R_{i-1} = 1 - \delta^{2^{i-1}}$ because $2 - D^{(i)} = R_i$. The reason why $D^{(k)} \to 1$ for large k is that

$$\begin{aligned} D^{(i)} &= (1-\delta)(1+\delta)(1+\delta^2)(1+\delta^4)\cdots(1+\delta^{2^{i-1}}) \\ &= (1-\delta^2)(1+\delta^2)(1+\delta^4)\cdots(1+\delta^{2^{i-1}}) \\ &= (1-\delta^{2^i}) \quad \text{for } i = 1, 2, \ldots, k \end{aligned} \tag{6.23}$$

Since $0 < \delta = 1 - D \le 0.5$, $\delta^{2^i} \to 0$ as i becomes sufficiently large, say, $i = k$ for some k; thus $D^{(k)} = 1 - \delta^{2^k} = 1$ for large k. The end result is

$$Q = M \times (1+\delta) \times (1+\delta^2) \times \cdots \times (1+\delta^{2^{k-1}}) \tag{6.24}$$

The above two sequences of chain multiplications are carried out alternately between the numerator and denominator through the pipeline stages. To summarize, in this technique division is carried out by repeated multiplications. Thus divide and multiply can share the same hardware pipeline.

$$Z \leftarrow A_i \times B_i + Z \quad (6.25)$$

where the successive operands (A_i, B_i) were fed through the X- and Y-buffers, and the accumulated sums through the Z-buffer recursively.

The entire pipeline could perform the *multiply* (×) and the *add* (+) in a single flow through the pipeline. The two levels of buffer registers isolated the loading and fetching of operands to or from the PAU, respectively, as in the concept of using a pair in the prefetch buffers described in Fig. 6.11.

Even though the TI-ASC is no longer in production, the system provided a unique design for multifunction arithmetic pipelines. Today, most supercomputers implement arithmetic pipelines with dedicated functions for much simplified control circuitry and faster operations.

6.5 SUPERSCALAR PIPELINE DESIGN

Pipeline Design Parameters Some parameters used in designing the scalar base processor and superscalar processor are summarized in Table 6.1 for the pipeline processors to be studied below. All pipelines discussed are assumed to have k stages.

The *pipeline cycle* for the scalar base processor is assumed to be 1 time unit, called the *base cycle*. We defined the instruction *issue rate*, *issue latency*, and *simple operation latency* in Section 4.1.1. The *instruction-level parallelism* (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.

For the base processor, all of these parameters have a value of 1. All processor types are designed relative to the base processor. The ILP is needed to fully utilize a given pipeline processor.

Table 6.1 *Design Parameters for Pipeline Processors*

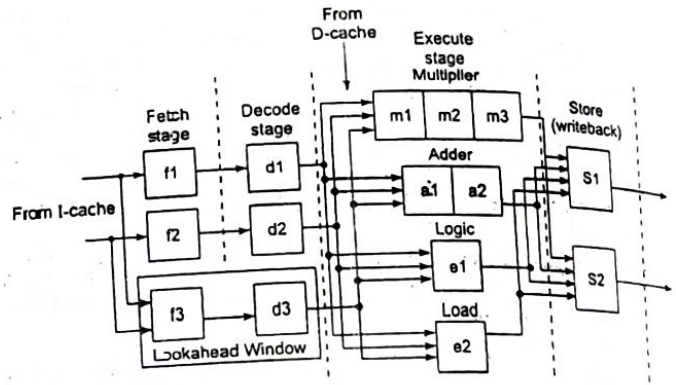
Machine type	*Scalar base machine of k pipeline stages*	*Superscalar machine of degree m*
Machine pipeline cycle	1 (base cycle)	m
Instruction issue rate	1	m
Instruction issue latency	1	1
Simple operation latency	1	m
ILP to fully utilize the pipeline	1	m

Note: All timing is relative to the base cycle for the scalar base machine. ILP: Instruction level parallelism

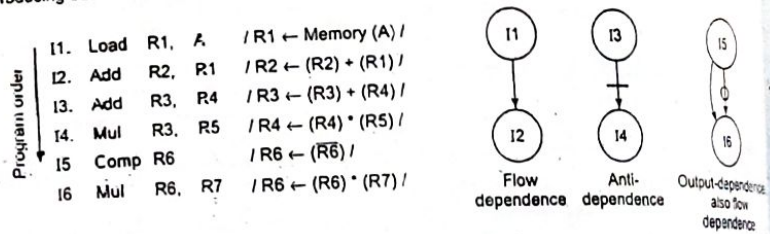
We study below the structure of superscalar pipelines, the data dependence problem, the factors causing pipeline stalling, and multi-instruction issuing mechanisms for achieving parallel pipelining operations. For a superscalar machine of degree m, m instructions are issued per cycle and the ILP should be m in order to fully utilize the pipeline. As a matter of fact, the scalar base processor can be considered a degenerate case of a superscalar processor of degree 1.

Superscalar Pipeline Structure In an m-issue superscalar processor, the instruction decoding and execution resources are increased to form effectively m pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines.

This resource-shared multiple-pipeline structure is illustrated by a design example in Fig. 6.28a. In this design, the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines in the design. Both pipelines have four processing stages labeled fetch, decode, execute, and store, respectively.



(a) A dual-pipeline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues



(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the multiplier

Fig. 6.28 A two-issue superscalar processor and a simple program for parallel execution

Each pipeline essentially has its own fetch unit, decode unit, and store unit. The two instruction streams flowing through the two pipelines are retrieved from a single source stream (the I-cache). The fan-out of a single instruction stream is subject to resource constraints and a data dependence relationship among successive instructions.

For simplicity, we assume that each pipeline stage requires one cycle, except the execute stage which may require a variable number of cycles. Four functional units, multiplier, adder, logic unit, and load unit, are available for use in the execute stage. These functional units are shared by the two pipelines on a dynamic basis. The multiplier itself has three pipeline stages, the adder has two stages, and the others each have only one stage.

The two store units (S1 and S2) can be dynamically used by the two pipelines, depending on availability at a particular cycle. There is a *lookahead window* with its own fetch and decoding logic. This window is used for instruction lookahead in case out-of-order instruction issue is desired to achieve better pipeline throughput.

It requires complex logic to schedule multiple pipelines simultaneously, especially when the instructions are retrieved from the same source. The aim is to avoid pipeline stalling and minimize pipeline idle time.

Data Dependences Consider the example program in Fig. 6.28b. A dependence graph is drawn to indicate the relationship among the instructions. Because the register content in R1 is loaded by I1 and then used by I2, we have flow dependence: I1→ I2.

Because the result in register R4 after executing I4 may affect the operand register R4 used by I3, we have antidependence: I3 ↤ I4. Since both I5 and I6 modify the register R6, and R6 supplies an operand for I6, we have both flow and output dependence: I5 → I6 and I5 ↤ I6 as shown in the dependence graph.

To schedule instructions through one or more pipelines, these data dependences must not be violated. Otherwise, erroneous results may be produced.

Pipeline Stalling This is a problem which may seriously lower pipeline utilization. Proper scheduling avoids pipeline stalling. The problem exists in both scalar and superscalar processors. However, it is more serious in a superscalar pipeline. Stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline. We use an example to illustrate the conditions causing pipeline stalling.

Consider the scheduling of two instruction pipelines in a two-issue superscalar processor. Figure 6.29a shows the case of no data dependence on the left and flow dependence (I1 → I2) on the right. Without data dependence, all pipeline stages are utilized without idling.

With dependence, instruction I2 entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages. This delay may also pass to the next instruction I4 entering the pipeline.

In Fig. 6.29b, we show the effect of branching (instruction I2). A delay slot of four cycles results from a branch taken by I2 at cycle 5. Therefore, both pipelines must be flushed before the target instructions I3 and I4 can enter the pipelines from cycle 6. Here, delayed branch or other amending actions are not taken.

In Fig. 6.29c, we show a combined problem involving both resource conflict and data dependence. Instructions I1 and I2 need to use the same functional unit, and I2 → I4 exists.

The net effect is that I2 must be scheduled one cycle behind because the two pipeline stages (e_1 and e_2) of the same functional unit must be used by I1 and I2 in an overlapped fashion. For the same reason, I3 is also delayed by one cycle. Instruction I4 is delayed by two cycles due to the flow dependence on I2. The shaded boxes in all the timing charts correspond to idle stages.

Superscalar Pipeline Scheduling Instruction issue and completion policies are critical to superscalar processor performance. Three scheduling policies are introduced below. When instructions are issued in program order, we call it *in-order issue*. When program order is violated, *out-of-order issue* is being practiced.

Similarly, if the instructions must be completed in program order, it is called *in-order completion*. Otherwise, *out-of-order completion* may result. In-order issue is easier to implement but may not yield the optimal performance. In-order issue may result in either in-order or out-of-order completion.

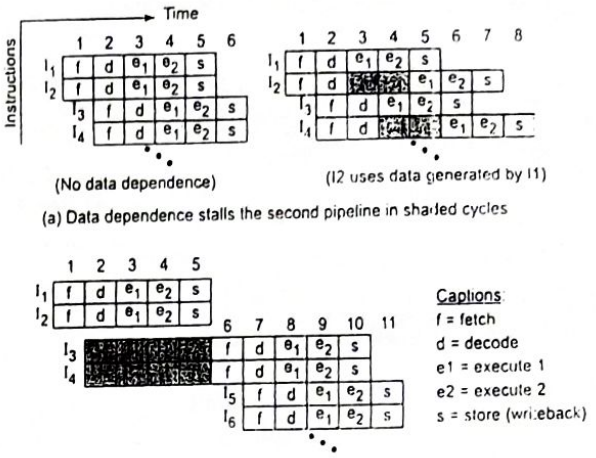


Fig. 6.29 Dependence, resource conflicts, and pipeline stalling in two-issue superscalar processors

Out-of-order issue usually ends up with out-of-order completion. The purpose of out-of-order issue and completion is to improve performance. These three scheduling policies are illustrated in Fig. 6.30 by execution of the example program in Fig. 6.28b on the dual-pipeline hardware in Fig. 6.28a.

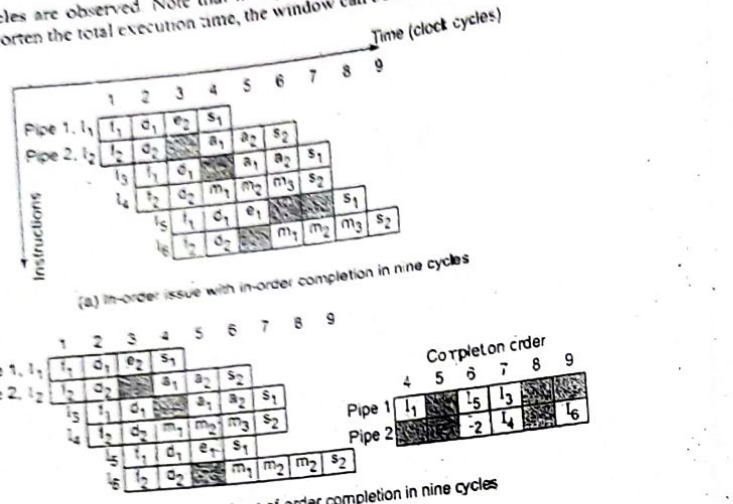
It is demonstrated that performance can be improved from an in-order to an out-of-order schedule. The performance is often indicated by the total execution time and the utilization rate of pipeline stages. Not all programs can be scheduled out of order. Data dependence and resource conflicts do impose constraints.

In-Order Issue Figure 6.30a shows a schedule for the six instructions being issued in program order I1, I2, ..., I6. Pipeline 1 receives I1, I3, and I5, and pipeline 2 receives I2, I4, and I6 in three consecutive cycles. Due to I1 → I2, I2 has to wait one cycle to use the data loaded in by I1.

I3 is delayed one cycle for the same adder used by I2. I6 has to wait for the result of I5 before it can enter the multiplier stages. In order to maintain in-order completion, I5 is forced to wait for two cycles to come out of pipeline 1. In total, nine cycles are needed and five idle cycles (shaded boxes) are observed.

In Fig. 6.30b, out-of-order completion is allowed even if in-order issue is practiced. The only difference between this out-of-order schedule and the in-order schedule is that I5 is allowed to complete ahead of I3 and I4, which are totally independent of I5. The total execution time does not improve. However, the pipeline utilization rate does.

Only three idle cycles are observed. Note that in Figs. 6.29a and 6.29b, we did not use the lookahead window. In order to shorten the total execution time, the window can be used to reorder the instruction issues.



(a) In-order issue with in-order completion in nine cycles

(b) In-order issue and out-of-order completion in nine cycles

(c) Out-of-order issue and out-of-order completion in seven cycles using an instruction lookahead window in the recoding process

Fig. 6.30 Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support (Timing charts correspond to parallel execution of the program in Fig. 6.26)

Out-of-Order Issue By using the lookahead window, instruction I5 can be decoded in advance because it is independent of all the other instructions. The six instructions are issued in three cycles as shown: I5 is fetched and decoded by the window, while I3 and I4 are decoded concurrently.

It is followed by issuing I6 and I1 at cycle 2, and I2 at cycle 3. Because the issue is out of order, the completion is also out of order as shown in Fig. 6.30c. Now, the total execution time has been reduced to seven cycles with no idle stages during the execution of these six instructions.

The in-order issue and completion is the simplest one to implement. It is rarely used today in conventional scalar processor due to some unnecessary delays in maintaining program order. However, in multiprocessor environment, this policy is still attractive. Allowing out-of-order completion can be used in both scalar and superscalar processors.

Some long-latency operations, such as loads and floating-point operations, can be hidden in out-of-order completion to achieve a better performance. Output dependence and antidependence are the two factors preventing out-of-order completion. Out-of-order issue gives the processor more freedom to exploit parallelism, and thus pipeline efficiency is enhanced.

The above example clearly demonstrates the advantages of instruction lookahead and of out-of-order issue and completion as far as pipeline optimization is concerned. It should be noted that instruction scheduling is an NP-complete problem. Optimal scheduling is very expensive to obtain.

Simple data dependence checking, a small lookahead window, and scoreboarding mechanisms are used along with an optimizing compiler, to exploit instruction parallelism in a superscalar processor.

Motorola 88110 Architecture The Motorola 88110 was an early superscalar RISC processor in the three-chip set, one CPU (88100) chip and two cache (88200) chips, in a single-chip implementation with additional improvements. The 88110 employed advanced techniques for exploiting instruction parallelism, including instruction issue, out-of-order instruction completion, speculative execution, instruction rescheduling, and two on-chip caches. The unit also supported demanding graphics and signal processing applications.

The 88110 employed a symmetrical superscalar instruction dispatch unit which dispatched two instructions each clock cycle into an array of 10 concurrent units. It allowed out-of-order instruction completion, out-of-order instruction issue, and branch prediction with speculative execution past branches.

The instruction set of the 88110 extended that of the 88100 in integer and floating-point operations. It added a new set of capabilities to support 3-D color graphics image rendering. The 88110 had two independent instruction and data paths, along with split caches for instructions and data. The instruction cache was 8K-byte, 2-way set-associative with 128 sets, two blocks for each set, and 32 bytes (8 words) per block. The data cache resembled that of the instruction set.

The 88110 employed the MESI cache coherence protocol. A write-invalidate procedure guaranteed that one processor on the bus had a modified copy of any cache block at any time. The 88110 was implemented with 1.3 million transistors in a 299-pin package and driven by a 50-MHz clock. Interested readers are referred to Diefendorff and Allen (1992) for details.

Superscalar Performance To compare the relative performance of a superscalar processor with a scalar base machine, we estimate the ideal execution time of N independent instructions through the pipeline.

The time required by the scalar base machine is

$$T(1, 1) = k + N - 1 \text{ (base cycles)}$$

The ideal execution time required by an m-issue superscalar machine is

$$T(m, 1) = k + \frac{N - m}{m} \text{ (base cycles)}$$

where k is the time required to execute the first m instructions through the m pipelines simultaneously, and the second term corresponds to the time required to execute the remaining $N - m$ instructions through m pipelines.

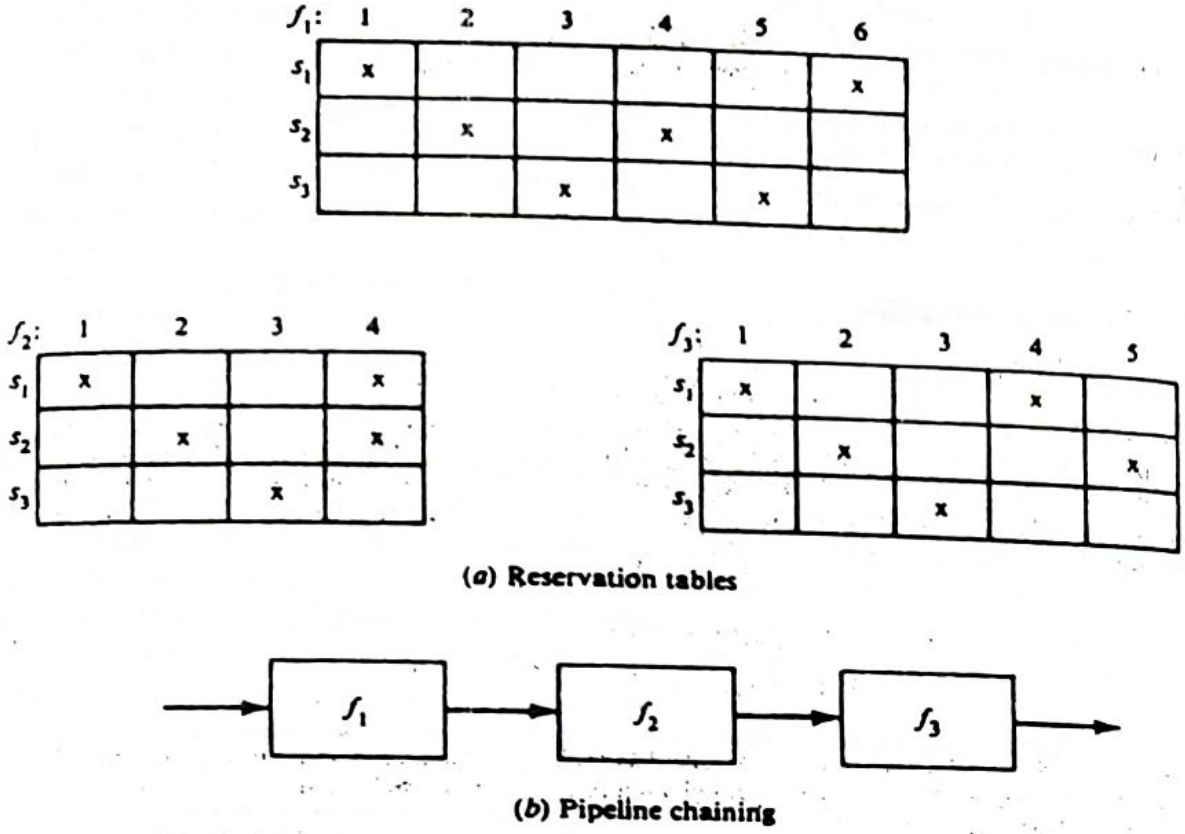


Figure 4.44 The chaining of three pipelines in Problem 4.10.

(*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.