Abhishek Singh

Chapter 6

Pipelining and Superscalar Techniques

This chapter deals with advanced pipelining and superscalar design in processor development. We begin with a discussion of conventional linear pipelines and analyze their performance. A generalized 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, internal data forwarding, software interlocking, hardware scoreboarding, hazard avoidance, branch handling, and instruction-issuing techniques. Both static and multifunctional arithmetic pipelines are designed. Superpipelining and superscalar design techniques are studied along with a 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.

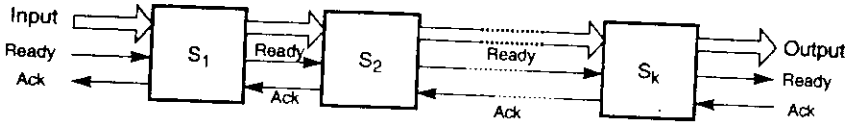
6.1.1 Asynchronous and Synchronous Models

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.

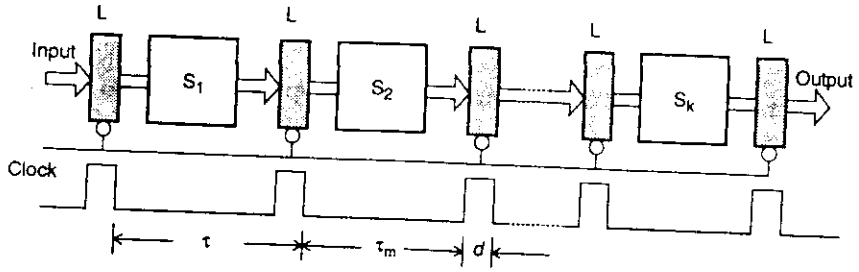
Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: *asynchronous* and *synchronous*.

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.

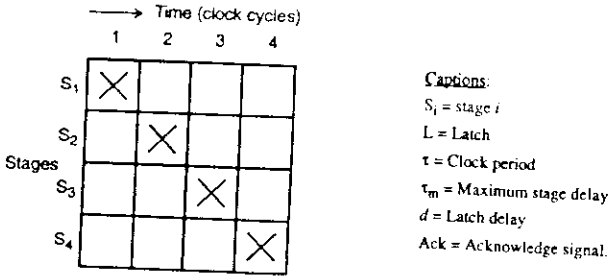
Asynchronous pipelines are useful in designing communication channels in message-passing multicomputers where pipelined wormhole routing is practiced (see Chapter 9). Asynchronous pipelines may have a variable throughput rate. Different amounts of delay may be experienced in different stages.



(a) An asynchronous pipeline model



(b) A synchronous pipeline model



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

Figure 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 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, $\tau_m >> d$ for 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

$$T_k = [k + (n - 1)]\tau \tag{6.4}$$

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 nonpipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n - 1)\tau} = \frac{nk}{k + (n - 1)} \tag{6.5}$$

Example 6.1 Pipeline speedup versus stream length

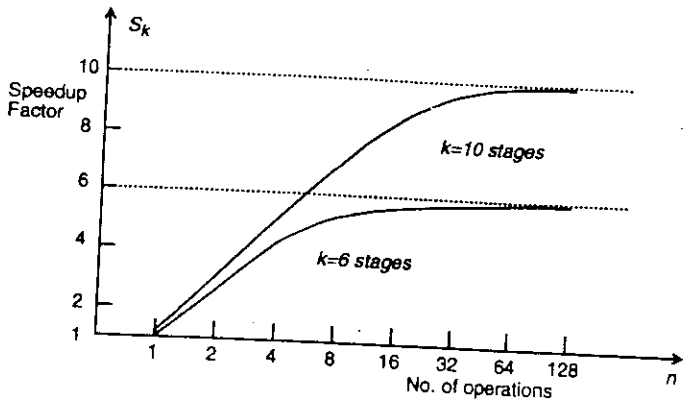
The maximum speedup is $S_k \rightarrow k$ as $n \rightarrow \infty$. This maximum speedup is very difficult to achieve because of data dependences between successive tasks (instructions), program branches, interrupts, and other factors to be studied in subsequent sections.

Figure 6.2a plots the speedup factor as a function of n, the number of tasks (operations or instructions) performed by the pipeline. For small values of n, the speedup can be very poor. The smallest value of S_k is 1 when $n = 1$.

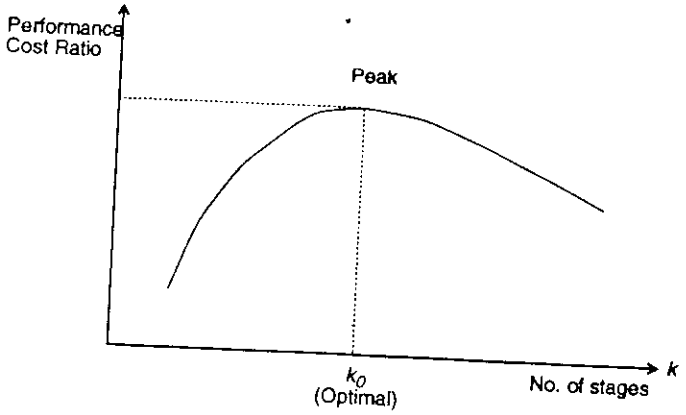
The larger the number k of subdivided pipeline stages, the higher the potential speedup performance. When $n = 64$, an eight-stage pipeline has a speedup value of 7.1 and a four-stage pipeline has a speedup of 3.7. However, the number of pipeline stages cannot increase indefinitely due to practical constraints on costs, control complexity, circuit implementation, and packaging limitations. Furthermore, the stream length n also affects the speedup; the longer the better in using a pipeline. ■

Optimal Number of Stages The finest level of pipelining is called *micropipelining*, with a subdivision of pipeline stages at the logic gate level. In practice, most pipelining is staged at the functional level with $2 \leq k \leq 15$. Very few pipelines are designed to exceed 10 stages in real computers.

On the other hand, the coarse level for pipeline stages can be conducted at the processor level, called *macropipelining*. The optimal choice of the number of pipeline stages should be able to maximize a performance/cost ratio.



(a) Speedup factor as a function of the number of operations (Eq. 6.5)



(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Figure 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit.

Let t be the total time required for a nonpipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $p = t/k + d$, where d is the latch delay. Thus, the pipeline has a maximum throughput of $f = 1/p = 1/(t/k+d)$. The total pipeline cost is roughly estimated by $c + kh$, where c covers the cost of all logic stages and h represents the cost of each latch. A pipeline *performance/cost ratio* (PCR) has been defined by Larson (1973):

$$PCR = \frac{f}{c + kh} = \frac{1}{(t/k + d)(c + kh)} \tag{6.6}$$

Figure 6.2b plots the PCR as a function of k. The peak of the PCR curve corre-

sponds to an optimal choice for the number of desired pipeline stages:

$$k_0 = \sqrt{\frac{t \cdot c}{d \cdot h}} \tag{6.7}$$

where t is the total flow-through delay of the pipeline. The total stage cost c, the latch delay d, and the latch cost h can be adjusted to achieve the optimal value k_0.

Efficiency and Throughput The *efficiency* E_k of a linear k-stage pipeline is defined as

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)} \tag{6.8}$$

Obviously, the efficiency approaches 1 when $n \to \infty$, and a lower bound on E_k is $1/k$ when $n = 1$. The *pipeline throughput* H_k is defined as the number of tasks (operations) performed per unit time:

$$H_k = \frac{n}{[k + (n-1)]\tau} = \frac{nf}{k + (n-1)} \tag{6.9}$$

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 $H_k = E_k \cdot f = E_k/\tau = S_k/k\tau$.

6.2 Nonlinear Pipeline Processors

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.

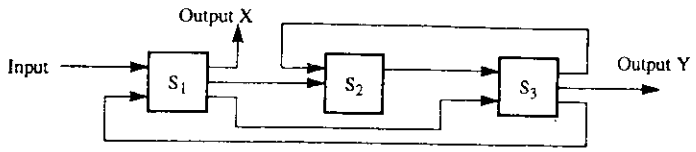
A dynamic pipeline allows feedforward and feedback connections in addition to the streamline connections. For this reason, some authors call such a structure a *nonlinear pipeline*.

6.2.1 Reservation and Latency Analysis

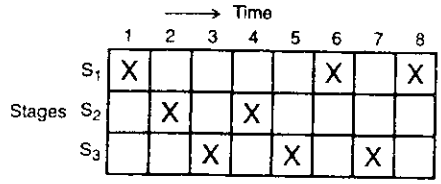
In a static pipeline, it is easy to partition a given function into a sequence of linearly ordered subfunctions. However, function partitioning in a dynamic pipeline becomes quite involved because the pipeline stages are interconnected with loops in addition to streamline connections.

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 *feedforward connection* from S_1 to S_3 and two *feedback connections* from S_3 to S_2 and from S_3 to S_1.

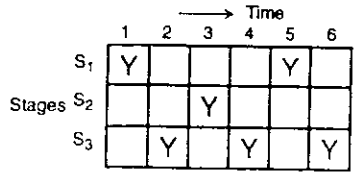
These feedforward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task. With these connections, the output of the pipeline is not necessarily from the last stage. In fact, following different dataflow patterns, one can use the same pipeline to evaluate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



(a) A three-stage pipeline



(b) Reservation table for function X



(c) Reservation table for function Y

Figure 6.3 A dynamic pipeline with feedforward and feedback connections for two different functions.

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 may follow different paths on the reservation table. A number of pipeline configurations may be represented by the same reservation table. There is a many-to-many mapping between various pipeline configurations and different reservation tables.

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.

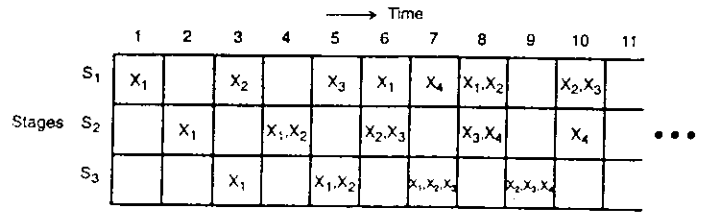
A pipeline *initiation* table corresponds to each function evaluation. All initiations

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.

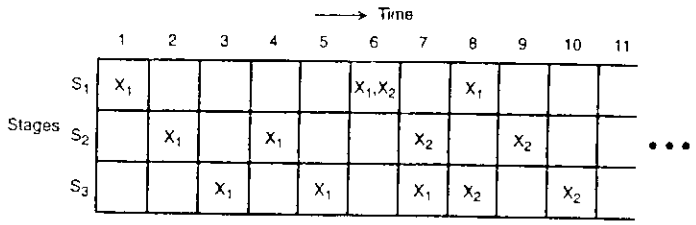
There may be multiple checkmarks in a row, which means repeated usage of the same stage in different cycles. Contiguous checkmarks in a row simply imply the extended usage of a stage over more than one cycle. Multiple checkmarks in a column mean that multiple stages are used in parallel during a particular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

Figure 6.4 Collisions with forbidden latencies 2 and 5 in using the pipeline in Fig. 6.3 to evaluate the function X.

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,

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 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 achieving such collision-free scheduling.

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

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* $m \leq n - 1$. The permissible latency p should be as small as possible. The choice is made in the range $1 \leq p \leq m - 1$.

A permissible latency of $p = 1$ corresponds to the ideal case. In theory, a latency of 1 can always be achieved in a static pipeline which follows a linear (diagonal or streamlined) reservation table as shown 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 $C = (C_m C_{m-1} \cdots C_2 C_1)$. 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.

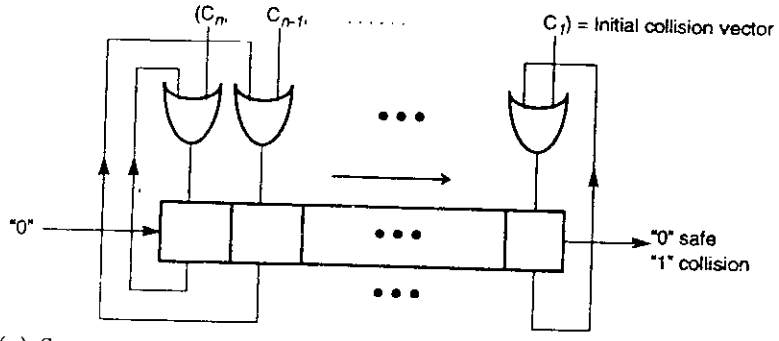
For the two reservation tables in Fig. 6.3, the collision vector $C_X = (1011010)$ is obtained for function X, and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies 7, 5, 4, and 2 are forbidden and latencies 6, 3, and 1 are permissible. Similarly, 4 and 2 are forbidden latencies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

State Diagrams From the above collision vector, one can construct a *state diagram* specifying 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 $1 \leq p \leq m - 1$.

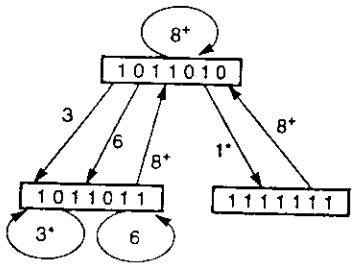
The *next state* of the pipeline at time $t + p$ is obtained with the assistance of an m-bit right shift register as in Fig. 6.6a. The initial collision vector C is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency 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

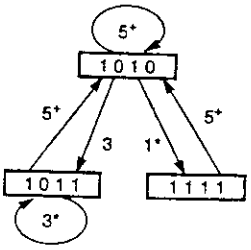
contents. For example, from the initial state $C_X = (1011010)$, the next state (1111111) is reached after one right shift of the register, and the next state (1011011) is reached after three shifts or six shifts.



(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 Two state diagrams obtained from the two reservation tables in Fig. 6.3, respectively.

Example 6.2 The state transition diagram for a pipeline unit

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

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. Different reservation tables

may result in the same or different initial collision vectors(s).

This implies that even different reservation tables may produce the same state diagram. However, different reservation tables may produce different collision vectors and thus different state diagrams.

■

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.

Delay Insertion The purpose of delay insertion is to modify the reservation table, yielding a new collision vector. This leads to a modified state diagram, which may produce greedy cycles meeting the lower bound on the MAL.

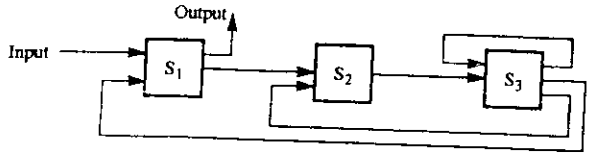
Before delay insertion, the three-stage pipeline in Fig. 6.7a is specified by the reservation table in Fig. 6.7b. This table leads to a collision vector $C = (1011)$, corresponding to forbidden latencies 1, 2, and 4. The corresponding state diagram (Fig. 6.7c) contains only one self-reflecting state with a greedy cycle of latency 3 equal to the MAL.

Based on the given reservation table, the maximum number of checkmarks in any row is 2. Therefore, the MAL = 3 so obtained in Fig. 6.7c is not optimal.

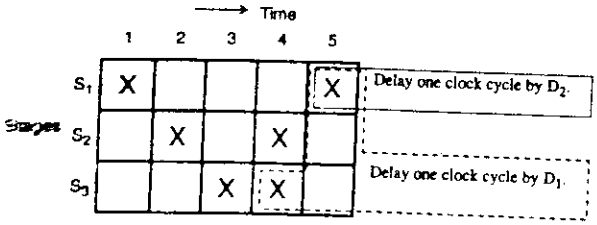
Example 6.3 Inserting noncompute delays to reduce the MAL

To insert a noncompute stage D_1 after stage S_3 will delay both X_1 and X_2 operations one cycle beyond time 4. To insert yet another noncompute stage D_2 after the second usage of S_1 will delay the operation X_2 by another cycle.

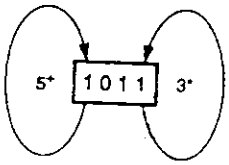
These delayed operations, as grouped in Fig. 6.7b, result in a new pipeline configuration in Fig. 6.8a. Both delay elements D_1 and D_2 are inserted as extra stages, as shown in Fig. 6.8b with an enlarged reservation table having $3 + 2 = 5$ rows and $5 + 2 = 7$ columns.



(a) A three-stage pipeline

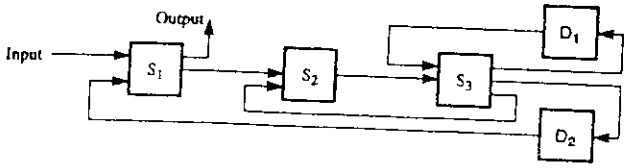


(b) Reservation table and operations being delayed

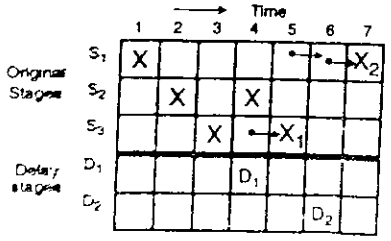


(c) New state transition diagram with MAL = 3

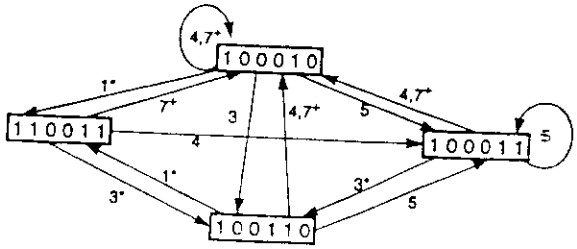
Figure 6.7 A pipeline with a minimum average latency of 3.



(a) Insertion of two noncompute delay stages



(b) Modified reservation table



(c) Modified state diagram with a reduced MAL = $(1 + 3)/2 = 2$

Figure 6.8 Insertion of two delay stages to obtain an optimal MAL for the pipeline in Fig. 6.7.

In total, the operation X_1 has been delayed one cycle from time 4 to time 5 and the operation X_2 has been delayed two cycles from time 5 to time 7. All remaining operations (marked as X in Fig. 6.8b) are unchanged. This new table leads to a new collision vector (100010) and a modified state diagram in Fig. 6.8c.

This diagram displays a greedy cycle (1, 3), resulting in a reduced MAL = $(1+3)/2 = 2$. The delay insertion thus improves the pipeline performance, yielding a lower bound for the MAL.

■

Pipeline Throughput This is essentially the initiation rate or the average number of task initiations per clock cycle. If N tasks are initiated within n pipeline cycles, then the *initiation rate* or *pipeline throughput* is measured as N/n. This rate is determined primarily by the inverse of the MAL adapted. Therefore, the scheduling strategy does affect the pipeline performance.

In general, the shorter the adapted MAL, the higher the throughput that can be expected. The highest achievable throughput is one task initiation per cycle, when the MAL equals 1 since $1 \leq$ MAL $\leq$ the shortest latency of any greedy cycle. Unless the MAL is reduced to 1, the pipeline throughput becomes a fraction.

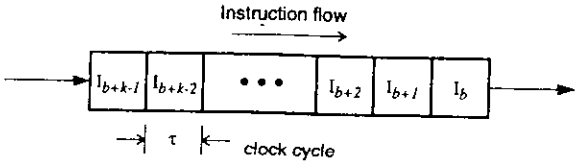
Pipeline Efficiency Another important measure is *pipeline efficiency*. The percentage of time that each pipeline stage is used over a sufficiently long series of task initiations is the *stage utilization*. The accumulated rate of all stage utilizations determines the pipeline efficiency.

Let us reexamine latency cycle (3) in Fig. 6.5b. Within each latency cycle of three clock cycles, there are two pipeline stages, S_1 and S_3, which are completely and continuously utilized after time 6. The pipeline stage S_2 is used for two cycles and is idle for one cycle.

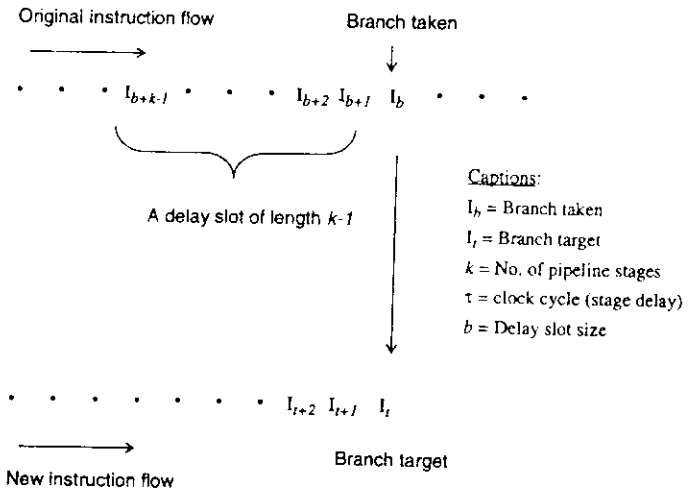
Therefore, the entire pipeline can be considered $8/9 = 88.8\%$ efficient for latency cycle (3). On the other hand, the pipeline is only $14/27 = 51.8\%$ efficient for latency cycle (1, 8) and $8/16 = 50\%$ efficient for latency cycle (6), as illustrated in Figs. 6.5a and 6.5c, respectively. Note that none of the three stages is fully utilized with respect to two initiation cycles.

The pipeline throughput and pipeline efficiency are related to each other. Higher throughput results from a shorter latency cycle. Higher efficiency implies less idle time for pipeline stages. The above example demonstrates that higher throughput also accompanies higher efficiency. Other examples may show 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.



(a) A k-stage pipeline



(b) An instruction stream containing a branch taken

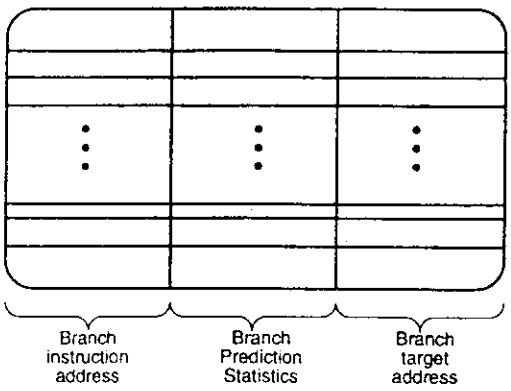
Figure 6.18 The decision of a branch taken at the last stage of an instruction pipeline causes $b \leq k-1$ previously loaded instructions to be drained from the pipeline.

Branch Prediction Branch can be predicted either based on branch code types statically or based on branch history during program execution. The probability of branch with respect to a particular branch instruction type can be used to predict branch. This requires collecting the frequency and probabilities of branch taken and branch types across a large number of program traces. Such a *static branch strategy* may not be always accurate.

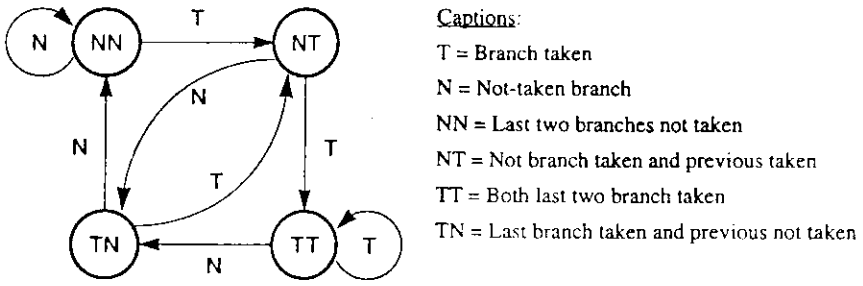
The static prediction direction (*taken* or *not taken*) is usually wired into the processor. According to past experience, the best performance is given by predicting *taken*. This results from the fact that most conditional branch instructions are taken in program execution. The wired-in static prediction cannot be changed once committed to the hardware. However, the scheme can be modified to allow the programmer or compiler to select the direction of each branch on a *semi-static* prediction basis.

A *dynamic branch strategy* uses recent branch history to predict whether or not the branch will be taken next time when it occurs. To be accurate, one may need

to use the entire history of the branch to predict the future choice. This is infeasible to implement. Therefore, most dynamic prediction is determined with limited recent history, as illustrated in Fig. 6.19.



(a) Branch target buffer organization



(b) A typical state diagram

Figure 6.19 Branch history buffer and a state transition diagram used in dynamic branch prediction. (Courtesy of Lee and Smith, *IEEE Computer*, 1984)

Cragon (1992) has classified dynamic branch strategies into three major classes: One class predicts the branch direction based upon information found at the decode stage. The second class uses a cache to store target addresses at the stage the effective address of the branch target is computed. The third scheme uses a cache to store target instructions at the fetch stage. All dynamic predictions are adjusted dynamically as a program is executed.

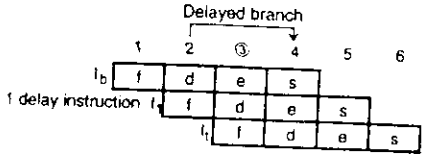
Dynamic prediction demands additional hardware to keep track of the past behavior of the branch instructions at run time. The amount of history recorded should be small. Otherwise, the prediction logic becomes too costly to implement.

Lee and Smith (1984) have shown the use of a *branch target buffer* (BTB) to

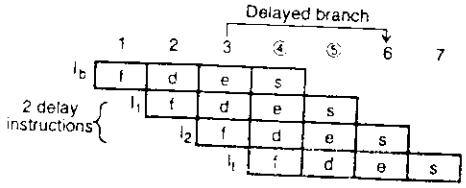
implement branch prediction (Fig. 6.19a). The BTB is used to hold recent branch information including the address of the branch target used. The address of the branch instruction locates its entry in the BTB.

For example, a state transition diagram (Fig. 6.19b) has been given by Lee and Smith for backtracking the last two branches in a given program. The BTB entry contains the backtracking information which will guide the prediction. Prediction information is updated upon completion of the current branch.

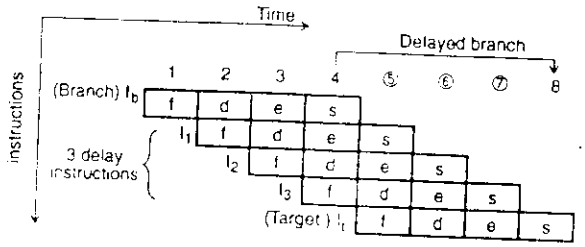
The BTB can be extended to store not only the branch target address but also the target instruction itself and a few of its successor instructions, in order to allow zero delay in converting conditional branches to unconditional branches. The *taken* (T) and *not-taken* (N) labels in the state diagram correspond to actual program behavior. Different programs may use different state diagrams which are modified dynamically according to historical program events.



(a) A delayed branch for 2 cycles when the branch condition is resolved at the decode stage



(b) A delayed branch for 3 cycles when the branch condition is resolved at the execute stage



(c) A delayed branch for 4 cycles when the branch condition is resolved at the store stage

Figure 6.20 The concept of delayed branch by moving independent instructions or NOP fillers into the delay slot of a four-stage pipeline.

Delayed Branches Examining the branch penalty, we realize that the branch penalty