3.2 INSTRUCTION PIPELINE

In a von Neumann architecture, the process of executing an instruction involves several steps. First, the control unit of a processor fetches the instruction from the cache (or from memory). Then the control unit decodes the instruction to determine the type of operation to be performed. When the operation requires operands, the control unit also determines the address of each operand and fetches them from cache (or memory). Next, the operation is performed on the operands and, finally, the result is stored in the specified location.

An instruction pipeline increases the performance of a processor by overlapping the processing of several different instructions. Often, this is done by dividing the instruction execution process into several stages. As shown in Figure 3.3, an instruction pipeline often consists of five stages, as follows:

1. **Instruction fetch (IF)**. Retrieval of instructions from cache (or main memory).
2. **Instruction decoding (ID)**. Identification of the operation to be performed.
3. **Operand fetch (OF)**. Decoding and retrieval of any required operands.
4. **Execution (EX)**. Performing the operation on the operands.
5. **Write-back (WB)**. Updating the destination operands.

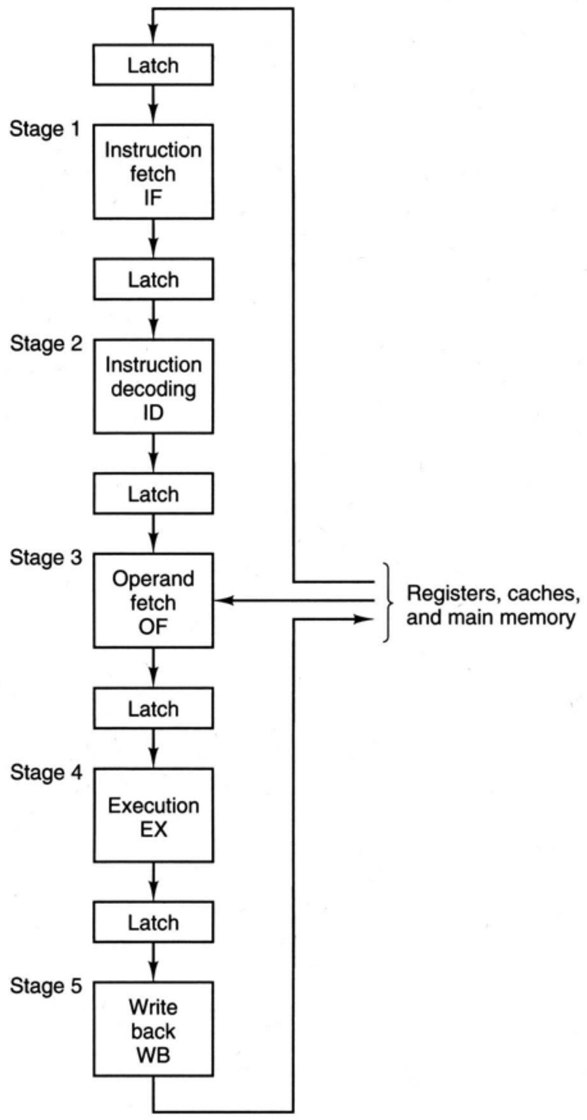


Figure 3.3 Stages of an instruction pipeline.

An instruction pipeline overlaps the process of the preceding stages for different instructions to achieve a much lower total completion time, on average, for a series of instructions. As an example, consider Figure 3.4, which shows the execution of four instructions in an instruction pipeline. During the first cycle, or clock pulse, instruction i_1 is fetched from memory. Within the second cycle, instruction i_1 is decoded while instruction i_2 is fetched. This process continues until all the instructions are executed. The last instruction finishes the write-back stage after the eighth clock cycle. Therefore, it takes 80 nanoseconds (ns) to complete execution of all the four instructions when assuming the clock period to be 10 ns. The total completion time can also be obtained using equation (3.1); that is,

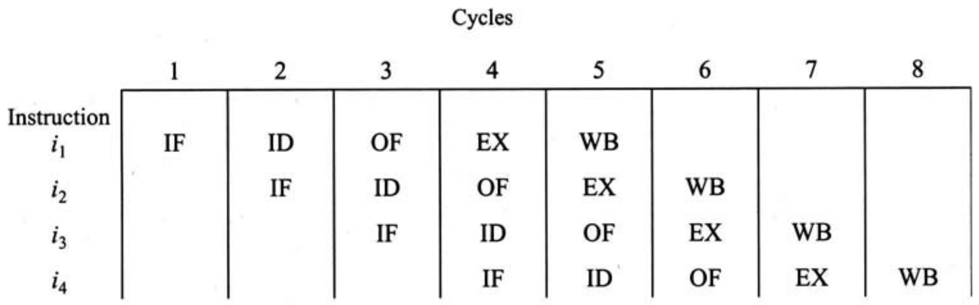
$$\begin{aligned} T_{pipe} &= m*P+(n-1)*P \\ &=5*10+(4-1)*10=80 \text{ ns.} \end{aligned}$$

Note that in a nonpipelined design the completion time will be much higher. Using equation (3.2),

$$T_{seq} = n*m*P = 4*5*10 = 200 \text{ ns.}$$

It is worth noting that a similar execution path will occur for an instruction whether a pipelined architecture is used or not; a pipeline simply takes advantage of these naturally occurring stages to improve processing efficiency. Henry Ford made the same connection when he realized that all cars were built in stages and invented the assembly line in the early 1900s. Some ideas have an enduring quality and can be apllied in many different ways!

Even though pipelining speeds up the execution of instructions, it does pose potential problems. Some of these problems and possible solutions are discussed next.



	Cycles							
Instruction	1	2	3	4	5	6	7	8
i_1	IF	ID	OF	EX	WB			
i_2		IF	ID	OF	EX	WB		
i_3			IF	ID	OF	EX	WB	
i_4				IF	ID	OF	EX	WB

Figure 3.4 Execution cycles of four consecutive instructions in an instruction pipeline.

3.2.1 Improving the Throughput of an Instruction Pipeline

Three sources of architectural problems may affect the throughput of an instruction pipeline. They are fetching, bottleneck, and issuing problems. Some solutions are given for each.

The fetching problem. In general, supplying instructions rapidly through a pipeline is costly in terms of chip area. Buffering the data to be sent to the pipeline is one simple way of improving the overall utilization of a pipeline. The utilization of a pipeline is defined as the percentage of time that the stages of the pipeline are used over a sufficiently long period of time. A pipeline is utilized 100% of the time when every stage is used (utilized) during each clock cycle.

Occasionally, the pipeline has to be drained and refilled, for example, whenever an interrupt or a branch occurs. The time spent refilling the pipeline can be minimized by having instructions and data loaded ahead of time into various geographically close buffers (like on-chip caches) for immediate transfer into the pipeline. If instructions and data for normal execution can be fetched before they are needed and stored in buffers, the pipeline will have a continuous source of information with which to work. Prefetch algorithms are used to make sure potentially needed instructions are available most of the time. Delays from memory-access conflicts can thereby be reduced if these algorithms are used, since the time required to transfer data from main memory is far greater than the time required to transfer data from a buffer.

The bottleneck problem. The bottleneck problem relates to the amount of load (work) assigned to a stage in the pipeline. If too much work is applied to one stage, the time taken to complete an operation at that stage can become unacceptably long. This relatively long time spent by the instruction at one stage will inevitably create a bottleneck in the pipeline system. In such a system, it is better to remove the bottleneck that is the source of congestion. One solution to this problem is to further subdivide the stage. Another solution is to build multiple copies of this stage into the pipeline.

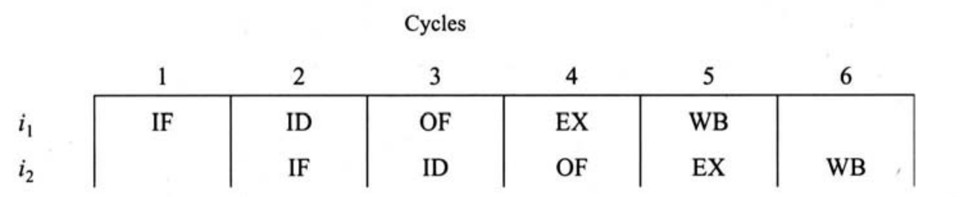
The issuing problem. If an instruction is available, but cannot be executed for some reason, a hazard exists for that instruction. These hazards create issuing problems; they prevent issuing an instruction for execution. Three types of hazard are discussed here. They are called *structural hazard, data hazard,* and *control hazard.* A structural hazard refers to a situation in which a required resource is not available (or is busy) for executing an instruction. A data hazard refers to a situation in which there exists a data dependency (operand conflict) with a prior instruction. A control hazard refers to a situation in which an instruction, such as branch, causes a change in the program flow. Each of these hazards is explained next.

Structural Hazard. A structural hazard occurs as a result of resource conflicts between instructions. One type of structural hazard that may occur is due to the design of execution units. If an execution unit that requires more than one clock cycle (such as multiply) is not fully pipelined or is not replicated, then a sequence of instructions that uses the unit cannot be subsequently (one per clock cycle) issued for execution. Replicating and/or pipelining execution units increases the number of instructions that can be issued simultaneously. Another type of structural hazard that may occur is due to the design of register files. If a register file does not have multiple write (read) ports, multiple writes (reads) to (from) registers cannot be performed simultaneously. For example, under certain situations the instruction pipeline might want to perform two register writes in a clock cycle. This may not be possible when the register file has only one write port.

The effect of a structural hazard can be reduced fairly simply by implementing multiple execution units and using register files with multiple input/output ports.

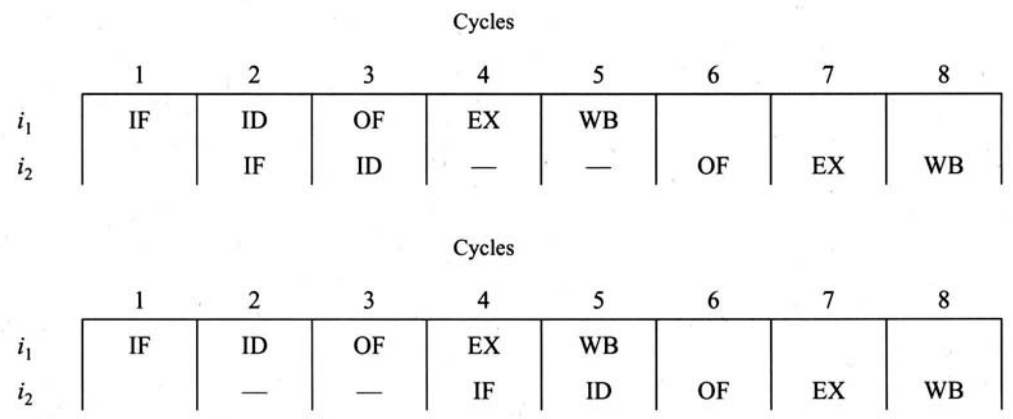
Data Hazard. In a nonpipelined processor, the instructions are executed one by one, and the execution of an instruction is completed before the next instruction is started. In this way, the instructions are executed in the same order as the program. However, this may not be true in a pipelined processor, where instruction executions are overlapped. An instruction may be started and completed before the previous instruction is completed. The data hazard, which is also referred to as the data dependency problem, comes about as a result of overlapping (or changing the order of) the execution of data-dependent instructions. For example, in Figure 3.5 instruction i_2 has a data dependency on i_1 because it uses the result of i_1 (i.e., the contents of register R_2) as input data. If the instructions were sent to a pipeline in the normal manner, i_2 would be in the OF stage before i_1 passed through the WB stage. This would result in using the old contents of R_2 for computing a new value for R_5, leading to an invalid result. To have a valid result, i_2 must not enter the OF stage until i_1 has passed through the WB stage. In this way, as is shown in Figure 3.6, the execution of i_2 will be delayed for two clock cycles. In other words, instruction i_2 is said to be *stalled* for two clock cycles. Often, when an instruction is stalled, the instructions that are positioned after the stalled instruction will also be stalled. However, the instructions before the stalled instruction can continue execution.

*i*1	Add	R_2,	R_3,	R_4	-- $R_2=R_3+R_4$
i_2	Add	R_5,	R_2,	R_1	-- $R_5=R_2+R_1$



	Cycles 1	2	3	4	5	6
i_1	IF	ID	OF	EX	WB	
i_2		IF	ID	OF	EX	WB

Figure 3.5 Instruction i_2 has data dependency on i_1.



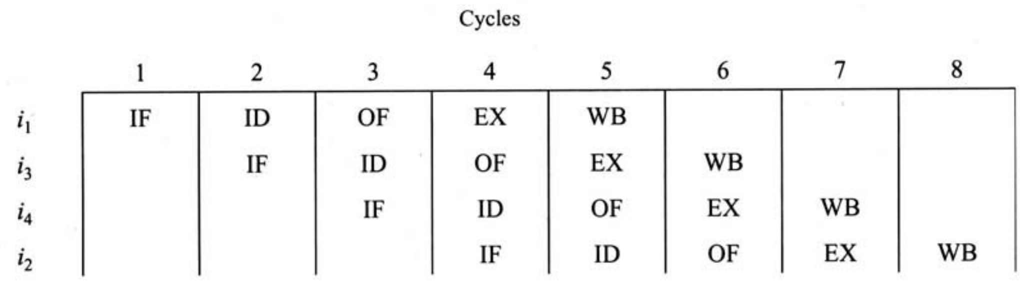
	Cycles 1	2	3	4	5	6	7	8
i_1	IF	ID	OF	EX	WB			
i_2		IF	ID	—	—	OF	EX	WB

	Cycles 1	2	3	4	5	6	7	8
i_1	IF	ID	OF	EX	WB			
i_2		—	—	IF	ID	OF	EX	WB

Figure 3.6 Two ways of executing data dependent instructions.

The delaying of execution can be accomplished in two ways. One way is to delay the OF or IF stages of i_2 for two clock cycles. To insert a delay, an extra hardware component called a pipeline interlock can be added to the pipeline. A *pipeline interlock* detects the dependency and delays the dependent instructions until the conflict is resolved. Another way is to let the compiler solve the dependency problem. During compilation, the compiler detects the dependency between data and instructions. It then rearranges these instructions so that the dependency is not hazardous to the system. If it is not possible to rearrange the instructions, NOP (no operation) instructions are inserted to create delays. For example, consider the four instructions in Figure 3.7. These instructions may be reordered so that i_3 and i_4, which are not dependent on i_1 and i_2, are inserted between i_1 and i_2.

i_1	Add	R_2,	R_3,	R_4	-- $R_2=R_3+R_4$
i_2	Add	R_5,	R_2,	R_1	-- $R_5=R_2+R_1$
i_3	Add	R_6,	R_6,	R_7	-- $R_6=R_6+R_7$
i_4	Add	R_8,	R_8,	R_7	-- $R_8=R_8+R_7$



	Cycles 1	2	3	4	5	6	7	8
i_1	IF	ID	OF	EX	WB			
i_3		IF	ID	OF	EX	WB		
i_4			IF	ID	OF	EX	WB	
i_2				IF	ID	OF	EX	WB

Figure 3.7 Rearranging the order of instruction execution.

In the previous type of data hazard, an instruction uses the result of a previous instruction as input data. In addition to this type of data hazard, other types may occur in designs that allow concurrent execution of instructions. Note that the type of pipeline design considered so far preserves the execution order of instructions in the program. Later in this section we will consider architectures that allow concurrent execution of independent instructions.

There are three primary types of data hazards: RAW (read after write), WAR (write after read), and WAW (write after write). The hazard names denote the execution ordering of the instructions that must be maintained to produce a valid result; otherwise, an invalid result might occur. Each of these hazards is explained in the following discussion. In each explanation, it is assumed that there are two instructions i_1 and i_2, and i_2 should be executed after i_1.

RAW: This type of data hazard was discussed previously; it refers to the situation in which i_2 reads a data source before i_1 writes to it. This may produce an invalid result since the read must be performed after the write in order to obtain a valid result. For example, in the sequence

i_1:	Add	R_2,	R_3,	R_4	--R_2=R_3+R_4
i_2:	Add	R_5,	R_2,	R_1	--R_5=R_2+R_1

an invalid result may be produced if i_2 reads R_2 before i_1 writes to it.

WAR: This refers to the situation in which i_2 writes to a location before i_1 reads it. For example, in the sequence

i_1:	Add	R_2,	R_3,	R_4	--R_2=R_3+R_4
i_2:	Add	R_4,	R_5,	R_6	--R_4=R_5+R_6

an invalid result may be produced if i_2 writes to R_4 before i_1 reads it; that is, the instruction i_1 might use the wrong value of R_4.

WAW: This refers to the situation in which i_2 writes to a location before i_1 writes to it. For example, in the sequence

i_1:	Add	R_2,	R_3,	R_4	--R_2=R_3+R_4
i_2:	Add	R_2,	R_5,	R_6	--R_2=R_5+R_6

the value of R_2 is recomputed by i_2. If the order of execution were reversed, that is, i_2 writes to R_2 before i_1 writes to it, an invalid value for R_2 might be produced.

Note that the WAR and WAW types of hazards cannot happen when the order of completion of instructions execution in the program is preserved. However, one way to enhance the architecture of an instruction pipeline is to increase concurrent execution of the instructions by dispatching several independent instructions to different functional units, such as adders/subtractors, multipliers, and dividers. That is, the instructions can be executed out of order, and so their execution may be completed out of order too. Hence, in such architectures all types of data hazards are possible.

In today's architectures, the dependencies between instructions are checked statically by the compiler and/or dynamically by the hardware at run time. This preserves the execution order for dependent instructions, which ensures valid results. Many different static dependency checking techniques have been developed to exploit parallelism in a loop [LIL 94, WOL 91]. These techniques have the advantage of being able to look ahead at the entire program and are able to detect most dependencies.

Unfortunately, certain dependencies cannot be detected at compile time. For example, it is not always possible to determine the actual memory addresses of load and store instructions in order to resolve a possible dependency between them. However, during the run time the actual memory addresses are known, and thereby dependencies between instructions can be determined by dynamically checking the dependency. In general, dynamic dependency checking has the advantage of being able to determine dependencies that are either impossible or hard to detect at compile time. However, it may not be able to exploit all the parallelism available in a loop because of the limited lookahead ability that can be supported by the hardware. In practice, a combined static-dynamic dependency checking is often used to take advantage of both approaches.

Here we will discuss the techniques for dynamic dependency checking. Two of the most commonly used techniques are called *Tomasulo's method* [TOM 67] and the *scoreboard method* [THO 64, THO 70]. The basic concept behind these methods is to use a mechanism for identifying the availability of operands and functional units in successive computations.

Tomasulo's Method. Tomasulo's method was developed by R. Tomasulo to overcome the long memory access delays in the IBM 360/91 processor. Tomasulo's method increases concurrent execution of the instructions with minimal (or no) effort by the compiler or the programmer. In this method, a busy bit and a tag register are associated with registers. The busy bit of a particular register is set when an issued instruction designates that register as a destination. (The destination register, or sink register, is the register that the result of the instruction will be written to.) The busy bit is cleared when the result of the execution is written back to the register. The tag of a register identifies the unit whose result will be sent to the register (this will be made clear shortly).

Each functional unit may have more than one set (source_1 and source_2) of input registers. Each such set is called a reservation station and is used to keep the operands of an issued instruction. A tag register is also associated with each register of a reservation station. In addition, a common data bus (CDB) connects the output of the functional units to their inputs and the registers. Such a common data bus structure, called a *forwarding* technique (also referred to as *feed-forwarding*), plays a very important role in organizing the order in which various instructions are presented to the pipeline for execution. The CDB makes it possible for the result of an operation to become available to all functional units without first going through a register. It allows a direct copy of the result of an operation to be given to all the functional units waiting for that result. In other words, a currently executing instruction can have access to the result of a previous instruction before the result of the previous instruction has actually been written to an output register.

Figure 3.8 represents a simple architecture for such a method. In this architecture there are nine units communicating through a common data bus. The units include five registers, two add reservation stations called A_1 and A_2 (virtually two adders), and two multiply reservation stations called M_1 and M_2 (virtually two multipliers). The binary-coded tags 1 to 5 are associated with registers in the register file, 6 and 7 are associated to add stations, and 8 and 9 are associated with multiply stations. The tags are used to direct the result of an instruction to the next instruction through the CDB. For example, consider the execution of the following two instructions.

i_1	Add	R_2,	R_3,	R_4	-- R_2=R_3+R_4
i_2	Add	R_2,	R_2,	R_1	-- R_2=R_2+R_1

After issuing the instruction i_1 to the add station A_1, the busy bit of the register R_2 is set to 1, the contents of the registers R_3 and R_4 are sent to source_1 and source_2 of the add station A_1, respectively, and the tag of R_2 is set to 6 (i.e., 110), which is the tag of A_1. Then the adder unit starts execution of i_1. In the meantime, during the process of operand fetch for the instruction i_2, it becomes known that the register R_2 is busy. This means that instruction i_2 depends on the result of instruction i_1. To let the execution of i_2 start as soon as possible, the contents of tag of R_2 (i.e., 110) are sent to the tag of the source_1 of the add station A_2; therefore, tag of source_1 of A_2 is set to 6. At this time the tag of R_2 is changed to 7, which means that the result of A_2 must be transferred to R_2. Also, the contents of R_1 are sent to source_2 of A_2. Right before the adder finishes the execution of i_1 and produces the result, it sends a request signal to the CDB for sending the result. (Since CDB is shared with many units, its time sharing can be controlled by a central priority circuit.) When the CDB acknowledges the request, the adder A_1 sends the result to the CDB. The CDB broadcasts the result together with the tag of A_1 (i.e., 6) to all the units. Each reservation station, while waiting for data, compares its source register tags with the tag on the CDB. If they match, the data are copied to the proper register(s). Similarly, at the same time, each register whose busy bit is set to 1 compares its tag with the tag on the CDB. If they match, the register updates its data and clears the busy bit. In this case the data are copied to source_1 of A_2. Next, A_2 starts execution and the result is sent to R_2.

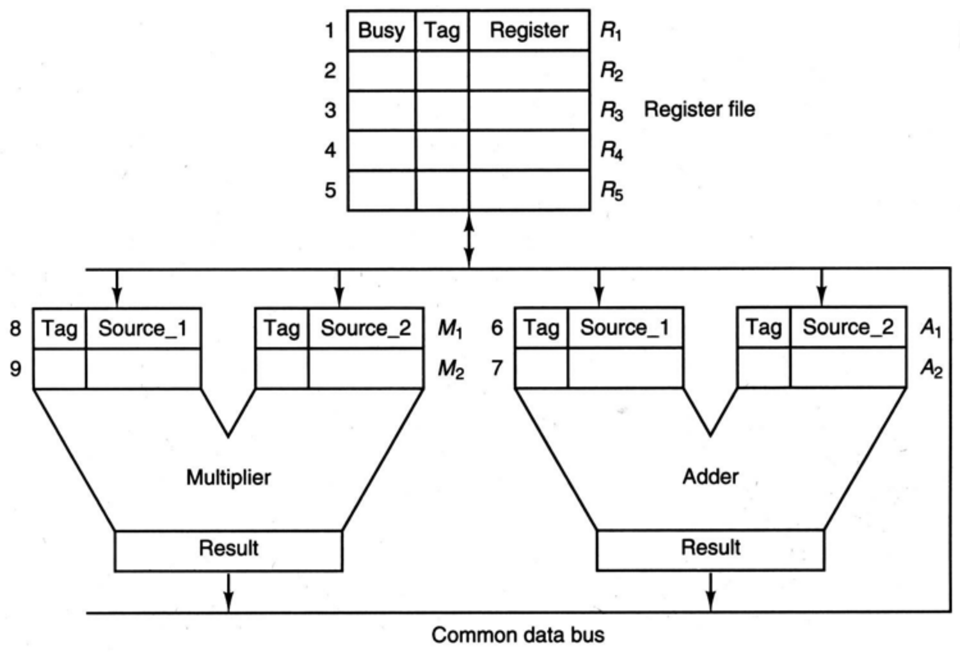


Figure 8. Common data bus architecture.

As demonstrated in the preceding example, the main concepts in Tomasulo's method are the addition of reservation stations, the innovation of the CDB, and the development of a simple tagging scheme. The reservation stations do the waiting for operands and hence free up the functional units from such a task. The CDB utilizes the reservation stations by providing them the result of an operation directly from the output of the functional unit. The tagging scheme preserves dependencies between successive operations while encouraging concurrency.

Although the extra hardware suggested by the Tomasulo's method encourages concurrent execution of instructions, the programmer and/or compiler still has substantial influence on the degree of concurrency. The following two programs for computing $(A*B)+(C+D)$ illustrate this.

Load	R_1,	A		
Load	R_2,	B		
Load	R_3,	C		
Load	R_4,	D		
Mul	R_5,	R_1,	R_2	-- $R_5 = R_1*R_2$
Add	R_5,	R_5,	R_3	-- $R_5 = R_5+R_3$
Add	R_4,	R_5,	R_4	-- $R_4 = R_5+R_4$

An alternative to this program that allows more concurrency is:

Load	R_1,	A		
Load	R_2,	B		
Load	R_3,	C		
Load	R_4,	D		
Mul	R_5,	R_1,	R_2	-- $R_5 = R_1*R_2$
Add	R_4,	R_3,	R_4	-- $R_4 = R_3+R_4$
Add	R_4,	R_4,	R_5	-- $R_4 = R_4+R_5$

In the second set of instructions, the multiply instruction and the first add instruction can be executed simultaneously, an impossibility in the first set of instructions. Often, in practice, a combination of hardware and software techniques is used to increase concurrency.

Scoreboard Method. The scoreboard method was first used in the high-performance CDC 6600 computer, in which multiple functional units allow instructions to be completed out of the original program order. This scheme maintains information about the status of each issued instruction, each register, and each functional unit in some buffers (or hardware mechanism) known as the *scoreboard*. When a new instruction is issued for execution, its influence on the registers and the functional units is added to the scoreboard. By considering a snapshot of the scoreboard, it can be determined if waiting is required for the new instruction. If no waiting is required, the proper functional unit immediately starts the execution of the instruction. If waiting is required (for example, one of the input operands is not yet available), execution of the new instruction is delayed until the waiting conditions are removed.

As described in [HEN 90], a scoreboard may consist of three tables: *instruction status*, *functional unit status*, and *destination register status*. Figure 3.9 represents a snapshot of the contents of these tables for the following program:

Load	R_1,____	A		
Load	R_2,____	B		
Load	R_3,____	C		
Load	R_4,____	D		
Mul	R_5,____	R_1,	R_2	-- $R_5 = R_1{*}R_2$
Add	R_2,____	R_3,	R_4	-- $R_2 = R_3+R_4$
Add	R_2,____	R_2,	R_5	-- $R_2 = R_2+R_5$

The instruction status table indicates whether or not an instruction is issued for execution. If the instruction is issued, the table shows which stage the instruction is in. After an instruction is brought in and decoded, the scoreboard will attempt to issue an instruction to the proper functional unit. An instruction will be issued if the functional unit is free and there is no other active instruction using the same destination register; otherwise, the issuing is delayed. In other words, an instruction is issued when WAW hazards and structural hazards do not exist. When such hazards exist, the issuing of the instruction and the instructions following are delayed until the hazards are removed. In this way the instructions are issued in order, while independent instructions are allowed to be executed out of order.

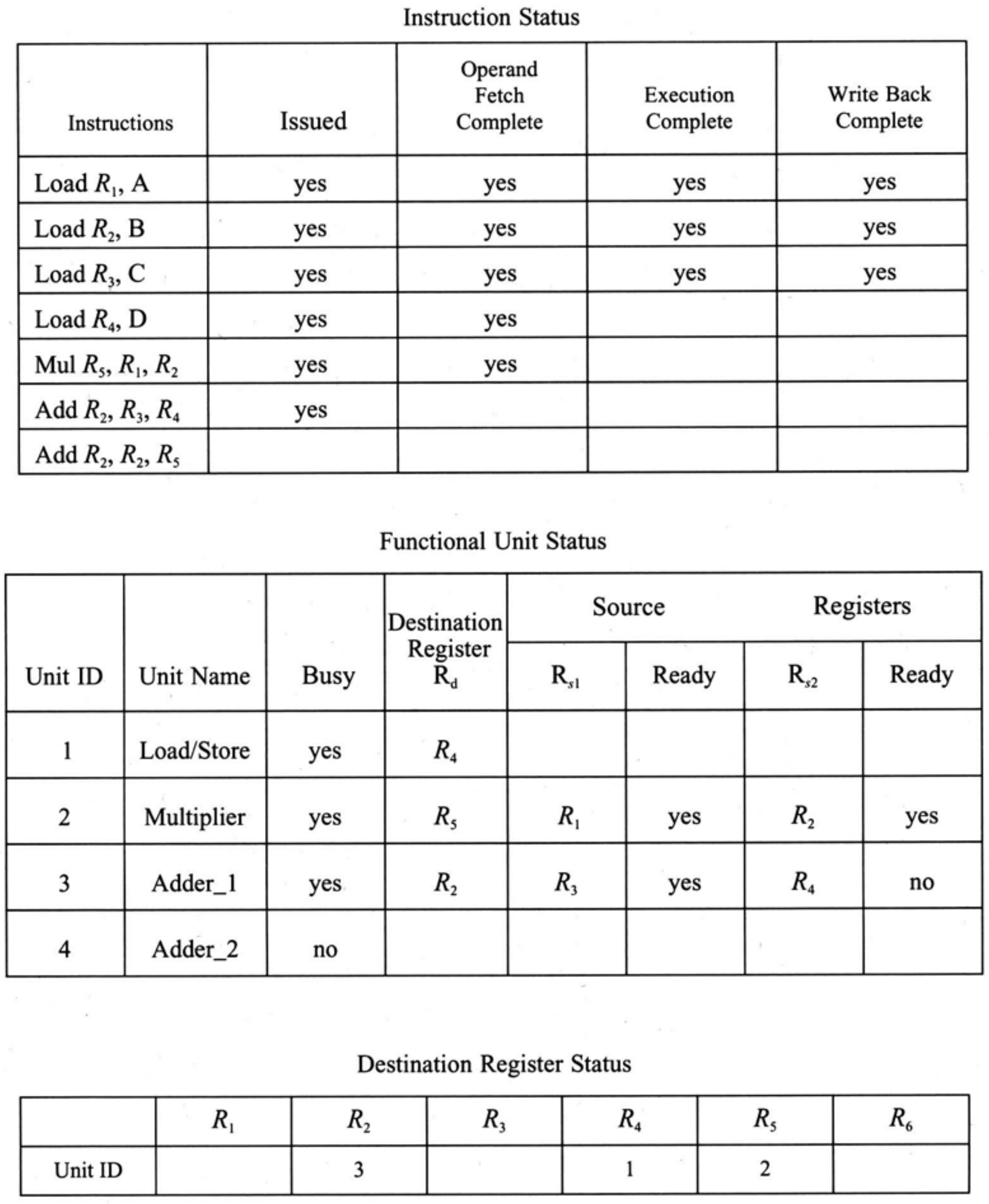
The functional unit status table indicates whether or not a functional unit is busy. A busy unit means that the execution of an issued instruction to that unit is not completed yet. For a busy unit, the table also identifies the destination register and the availability of the source registers. A source register for a unit is available if it does not appear as a destination for any other unit.

The destination register status table indicates the destination registers that have not yet been written to. For each such register the active functional unit that will write to the register is identified. The table has an entry for each register.

During the operand fetch stage, the scoreboard monitors the tables to determine whether or not the source registers are available to be read by an active functional unit. If none of the source registers is used as the destination register of other active functional units, the unit reads the operands from these registers and begins execution. After the execution is completed (i.e., at the end of execution stage), the scoreboard checks for WAR hazards before allowing the result to be written to the destination register. When no WAR hazard exists, the scoreboard tells the functional unit to go ahead and write the result to the destination register.

In Figure 3.9, the tables indicate that the first three load instructions have completed the execution and their operands are written to the destination registers. The last load instruction has been issued to the load/store unit (unit 1). This instruction has completed the fetch but has not yet written its operand to the register R_4. The multiplier unit is executing the instruction mul, and the first add instruction is issued to Adder_1 unit. The Adder_1 is waiting for R_4 to be written by load/store before it begins execution. This is because there is a RAW hazard between the last load and first add instructions. Note that at this time the second add cannot be issued because it uses R_2 as the source and destination register R_2, at this time, is busy with the first add. When R_4 is written by the load/store unit, Adder_1 begins execution.

At a later time, the scoreboard changes. As shown in Figure 3.10, if Adder_1 completes execution before the multiplier, it will write its result to R_2, and the second add instruction will be issued to the Adder_2 unit. Note that if the multiplier unit has not read R_2 before Adder_1 completes execution, Adder_1 will be prevented from writing the result to R_2 until the multiplier reads its operands; this is because there is a WAR hazard between mul and the first add instructions.



Instruction Status

Instructions	Issued	Operand Fetch Complete	Execution Complete	Write Back Complete
Load R_1, A	yes	yes	yes	yes
Load R_2, B	yes	yes	yes	yes
Load R_3, C	yes	yes	yes	yes
Load R_4, D	yes	yes		
Mul R_5, R_1, R_2	yes	yes		
Add R_2, R_3, R_4	yes			
Add R_2, R_2, R_5				

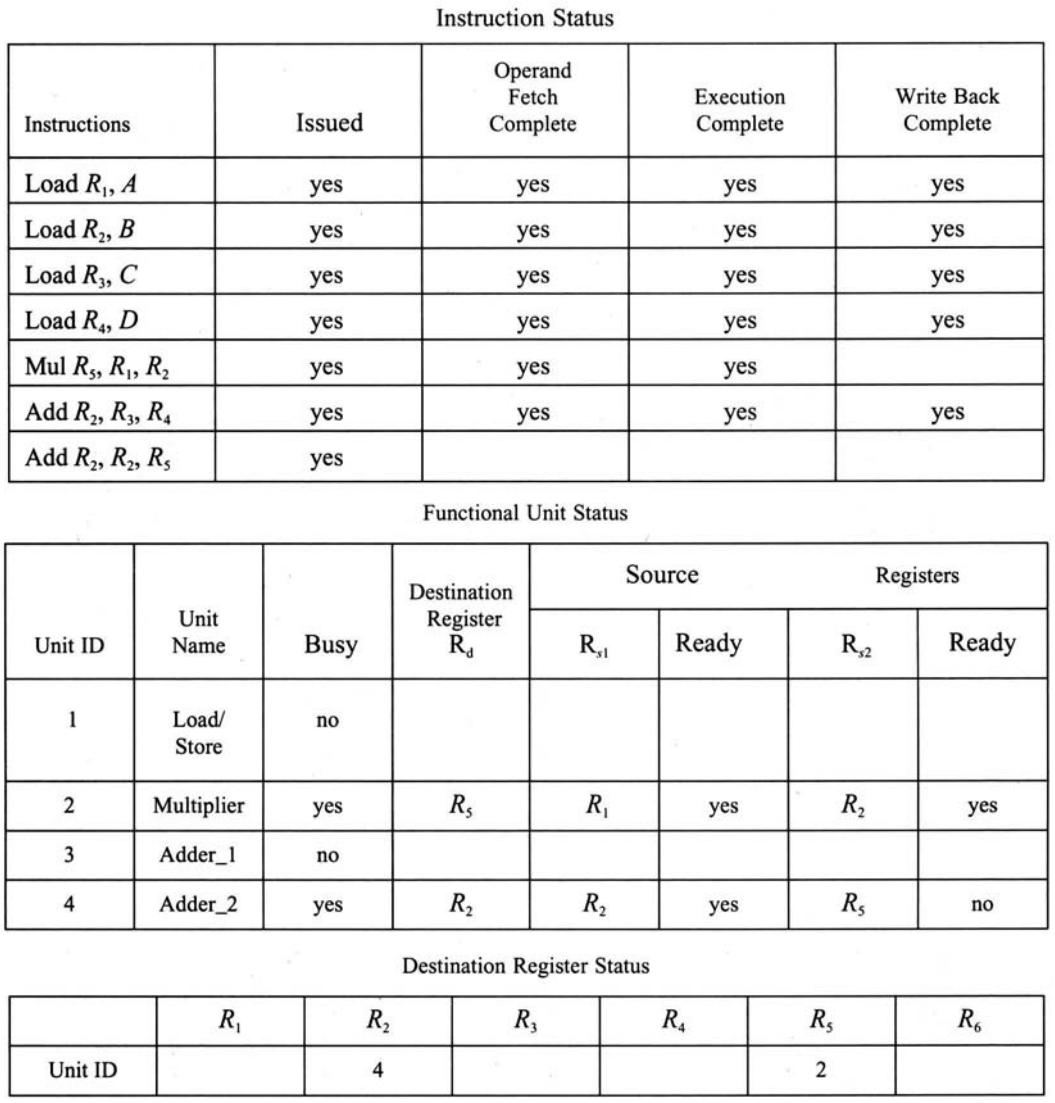
Functional Unit Status

Unit ID	Unit Name	Busy	Destination Register R_d	Source		Registers	
				R_{s1}	Ready	R_{s2}	Ready
1	Load/Store	yes	R_4				
2	Multiplier	yes	R_5	R_1	yes	R_2	yes
3	Adder_1	yes	R_2	R_3	yes	R_4	no
4	Adder_2	no					

Destination Register Status

	R_1	R_2	R_3	R_4	R_5	R_6
Unit ID		3		1	2	

Figure 3.9 A snapshot of the scoreboard after issuing the first add instruction.



Instruction Status

Instructions	Issued	Operand Fetch Complete	Execution Complete	Write Back Complete
Load R_1, A	yes	yes	yes	yes
Load R_2, B	yes	yes	yes	yes
Load R_3, C	yes	yes	yes	yes
Load R_4, D	yes	yes	yes	yes
Mul R_5, R_1, R_2	yes	yes	yes	
Add R_2, R_3, R_4	yes	yes	yes	yes
Add R_2, R_2, R_5	yes			

Functional Unit Status

Unit ID	Unit Name	Busy	Destination Register R_d	Source		Registers	
				R_{s1}	Ready	R_{s2}	Ready
1	Load/ Store	no					
2	Multiplier	yes	R_5	R_1	yes	R_2	yes
3	Adder_1	no					
4	Adder_2	yes	R_2	R_2	yes	R_5	no

Destination Register Status

	R_1	R_2	R_3	R_4	R_5	R_6
Unit ID		4			2	

Figure 3.10 A snapshot of the scoreboard after issuing the second add instruction.

The main component of the scoreboard approach is the destination register status table. This table is used to solve data hazards between instructions. Each time an instruction is issued for execution, the instruction's destination register is marked busy. The destination register stays busy until the instruction completes execution. When a new instruction is considered for execution, its operands are checked to ensure that there are no register conflicts with prior instructions still in execution.

Control Hazard. In any set of instructions, there is normally a need for some kind of statement that allows the flow of control to be something other than sequential. Instructions that do this are included in every programming language and are called *branches*. In general, about 30% of all instructions in a program are branches. This means that branch instructions in the pipeline can reduce the throughput tremendously if not handled properly. Whenever a branch is taken, the performance of the pipeline is seriously affected. Each such branch requires a new address to be loaded into the program counter, which may invalidate all the instructions that are either already in the pipeline or prefetched in the buffer. This draining and refilling of the pipeline for each branch degrade the throughput of the pipeline to that of a sequential processor. Note that the presence of a branch statement does not automatically cause the pipeline to drain and begin refilling. A branch not taken allows the continued sequential flow of uninterrupted instructions to the pipeline. Only when a branch is taken does the problem arise.

In general, branch instructions can be classified into three groups: (1) unconditional branch, (2) conditional branch, and (3) loop branch [LIL 88]. An unconditional branch always alters the sequential program flow. It sets a new target address in the program counter, rather than incrementing it by 1 to point to the next sequential instruction address, as is normally the case. A conditional branch sets a new target address in the program counter only when a certain condition, usually based on a condition code, is satisfied. Otherwise, the program counter is incremented by 1 as usual. In other words, a conditional branch selects a path of instructions based on a certain condition. If the condition is satisfied, the path starts from the target address and is called a *target path*. If it is not, the path starts from the next sequential instruction and is called a *sequential path*. Finally, a loop branch in a loop statement usually jumps back to the beginning of the loop and executes it either a fixed or a variable (data-dependent) number of times.

Among the preceding branch types, conditional branches are the hardest to handle. As an example, consider the following conditional branch instruction sequence:

i_1
i_2 (conditional branch to i_k)
i_3
.
.
.
i_k (target)
i_{k+1}

Figure 3.11 shows the execution of this sequence in our instruction pipeline when the target path is selected. In this figure, *c* denotes the branch penalty, that is, the number of cycles wasted whenever the target path is chosen.

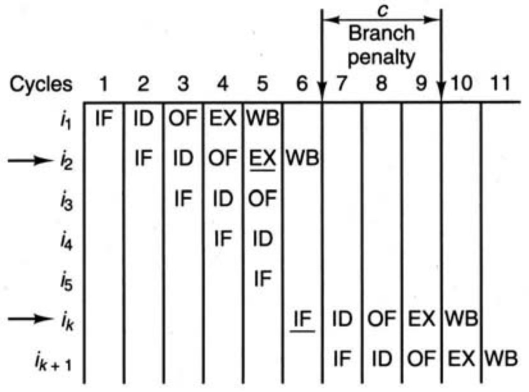


Figure 3.11 Branch Penalty.

To show the effect of the branch penalty on the overall pipeline performance, the average number of cycles per instruction must be determined. Let t_{ave} denote the average number of cycles required for execution of an instruction; then

$$t_{ave} = P_b * (\text{average number of cycles per branch instruction}) + (1 - P_b) * (\text{average number of cycles per nonbranch instruction}), \quad (3.3)$$

where P_b denotes the probability that a given instruction is a branch.

The average number of cycles per branch instruction can be determined by considering two cases. If the target path is chosen, 1+ c cycles (c = branch penalty) are needed for the execution; otherwise, there is no branch penalty and only one cycle is needed.

Thus average number of cycles per branch instruction = P_t (1+c) + (1- P_t) (1), where P_t denotes the probability that the *t* target path is chosen. The average number of cycles per nonbranch instruction is 1. After the pipeline becomes filled with instructions, a nonbranch instruction completes every cycle. Thus

$$t_{ave} = P_b\,[P_t\,(1+c) + (1-P_t\,)(1)] + (1- P_b)(1) = 1 + cP_b\,P_t.$$

After analyzing many practical programs, Lee and Smith [LEE 84] have shown the average P_b to be approximately 0.1 to 0.3 and the average P_t to be approximately 0.6 to 0.7. Assuming that P_b=0.2, P_t = 0.65, and c=3, then

$$t_{ave} = 1 + 3\,(0.2)(0.65) = 1.39.$$

In other words, the pipeline operates at 72% (100/1.39 = 72) of its maximum rate when branch instructions are considered.

Sometimes, the performance of a pipeline is represented in terms of throughput. The throughput, *H*, of a pipeline can also be expressed as the average number of instructions executed per clock cycle. Thus

$$H = 1/t_{ave} = 1/(1+cP_bP_t).$$

To reduce the effect of branching on processor performance, several techniques have been proposed [LIL 88]. Some of the better known techniques are branch prediction, delayed branching, and multiple prefetching. Each of these techniques is explained next.

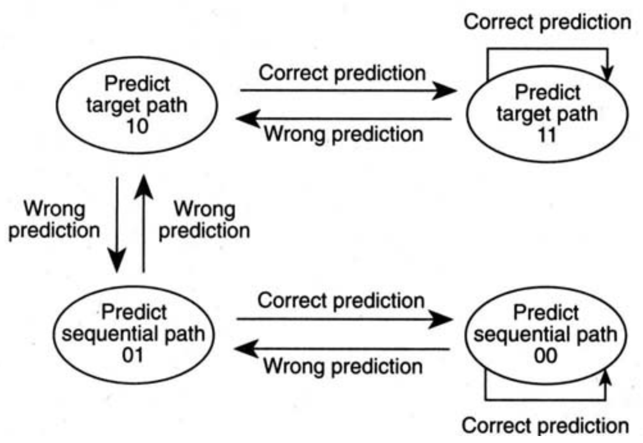
Branch Prediction. In this type of design, the outcome of a branch decision is predicted before the branch is actually executed. Therefore, based on a particular prediction, the sequential path or the target path is chosen for execution. Although the chosen path often reduces the branch penalty, it may increase the penalty in case of incorrect prediction.

There are two types of predictions, static and dynamic. In static prediction, a fixed decision for prefetching one of the two paths is made before the program runs. For example, a simple technique would be to always assume that the branch is taken. This technique simply loads the program counter with the target address when a branch is encountered. Another such technique is to automatically choose one path (sequential or target) for some branch types and another for the rest of the branch types. If the chosen path is wrong, the pipeline is drained and instructions corresponding to the correct path are fetched; the penalty is paid.

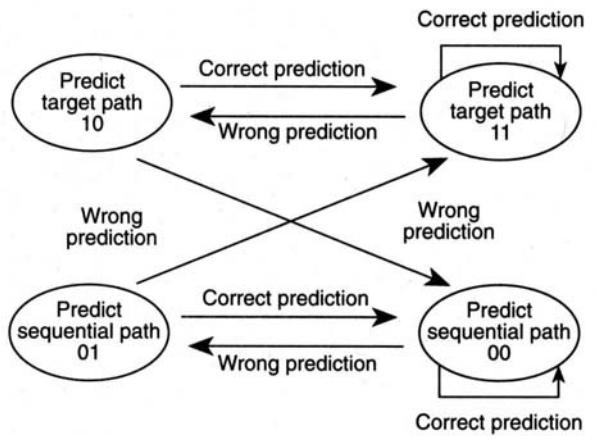
In dynamic prediction, during the execution of the program the processor makes a decision based on the past information of the previously executed branches. For example, a simple technique would be to record the history of the last two paths taken by each branch instruction. If the last two executions of a branch instruction have chosen the same path, that path will be chosen for the current execution of the branch instruction. If the two paths do not match, one of the paths will be chosen randomly.

A better approach is to associate an *n*-bit counter with each branch instruction. This is known as the counter-based branch prediction approach [PAN 92, HWU 89, LEE 84]. In this method, after executing a branch instruction for the first time, its counter, *C*, is set to a threshold, *T*, if the target path was taken, or to *T*-1 if the sequential path was taken. From then on, whenever the branch instruction is about to be executed, if $C \geq T$, then the target path is taken; otherwise, the sequential path is taken. The counter value *C* is updated after the branch is resolved. If the correct path is the target path, the counter is incremented by 1; if not, *C* is decremented by 1. If *C* ever reaches 2^n-1 (an upper bound), *C* is no longer incremented, even if the target path was correctly predicted and chosen. Likewise, *C* is never decremented to a value less than 0.

In practice, often *n* and *T* are chosen to be 2. Studies have shown that 2-bit predictors perform almost as well as predictors with more number of bits. The following diagram represents the possible states in a 2-bit predictor.



An alternative scheme to the preceding 2-bit predictor is to change the prediction only when the predicted path has been wrong for two consecutive times. The following diagram shows the possible states for such a scheme.



Most processors employ a small size cache memory called *branch target buffer* (BTB); sometimes referred to as *target instruction cache* (TIC). Often, each entry of this cache keeps a branch instruction's address with its target address and the history used by the prediction scheme. When a branch instruction is first executed, the processor allocates an entry in the BTB for this instruction. When a branch instruction is fetched, the processor searches the BTB to determine whether it holds an entry for the corresponding branch instruction. If there is a hit, the recorded history is used to determine whether the sequential or target path should be taken.

Static prediction methods usually require little hardware, but they may increase the complexity of the compiler. In contrast, dynamic prediction methods increase the hardware complexity, but they require less work at compile time. In general, dynamic prediction obtains better results than static prediction and also provides a greater degree of object code compatibility, since decisions are made after compile time.

To find the performance effect of branch prediction, we need to reevaluate the average number of cycles per branch instruction in equation (3.3). There are two possible cases: the predicted path is either correct or incorrect. In the case of a correctly predicted path, the penalty is *d* when the path is a target path (see Figure 3.12a), and the penalty is 0 when the path is a sequential path. (Note that, in Figure 3.12a, the address of target path is obtained after the decode stage. However, when a branch target buffer is used in the design, the target address can be obtained during or after the fetch stage.) In the case of an incorrectly predicted path for both target and sequential predicted paths, the penalty is *c* (See Figure 3.11 and Figure 3.12b). Putting it all together, we have average number of cycles per branch instruction =

$P_r\,[P_t(1+d) + (1- P_t)(1)] + (1- P_r)[\,P_t\,(1+c) + \;(1- P_t)(1+c)]$,

where P_r is the probability of a right prediction. Substituting this term in equation (3.3),

$$t_{ave} = P_b[P_r(P_t d + 1) +(1- P_r)(1+c)] + (1- P_b)(1)$$
$$= 1 + P_b c - P_b P_r c + P_b P_r P_t d.$$

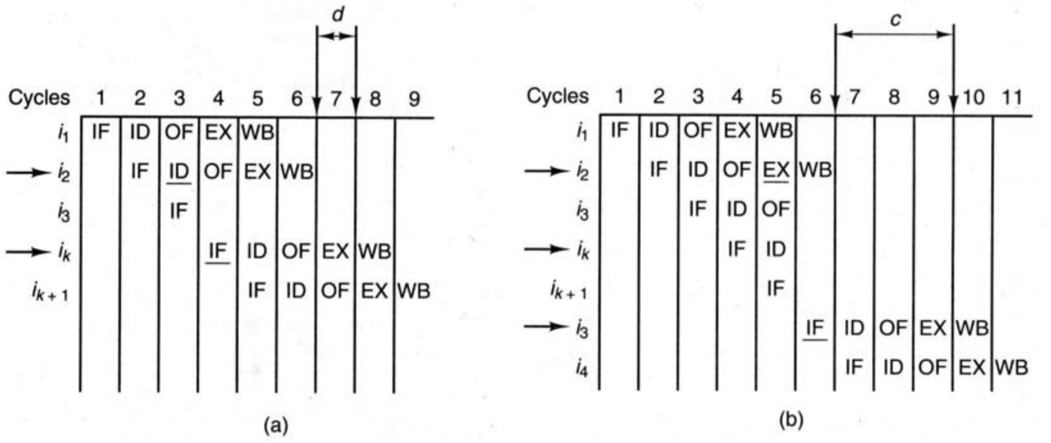


Figure 3.12 Branch penalties for when the target path is predicted: (a) The penalty for a correctly chosen target path. (b) The penalty for an incorrectly chosen target path.

Assume that P_b = 0.2, P_t = 0.65, c = 3, and d = 1. Also assume that the predicted path is correct 70% of the time (i.e., P_r = 0.70). Then

$$t_{ave} = 1.27.$$

That is, the pipeline operates at 78% of its maximum rate due to the branch prediction.

Delayed Branching. The delayed branching scheme eliminates or significantly reduces the effect of the branch penalty. In this type of design, a certain number of instructions after the branch instruction is fetched and executed regardless of which path will be chosen for the branch. For example, a processor with a branch delay of *k* executes a path containing the next *k* sequential instructions and then either continues on the same path or starts a new path from a new target address. As often as possible, the compiler tries to fill the next *k* instruction slots after the branch with instructions that are independent from the branch instruction. NOP (no operation) instructions are placed in any remaining empty slots. As an example, consider the following code:

i_1:	Load	R_1,	A		
i_2:	Load	R_2,	B		
i_3:	BrZr	R_2,	i_7		-- branch to i_7 if R_2=0;
i_4:	Load	R_3,	C		
i_5:	Add	R_4,	R_2,	R_3	-- $R_4 = R_2+R_3$
i_6:	Mul	R_5,	R_1,	R_2	-- $R_5 = R_1$*R_2
i_7:	Add	R_4,	R_1,	R_2	-- $R_4 = R_1+R_2$.

Assuming that *k*=2, the compiler modifies this code by moving the instruction i_1 and inserting an NOP instruction after the branch instruction i_3. The modified code is

i_2:	Load	R_2,	B
i_3:	BrZr	R_2,	i_7

```
i1:    Load   R1,   A
       NOP
i4:    Load   R3,   C
i5:    Add    R4,   R2,   R3
i6:    Mul    R5,   R1,   R2
i7:    Add    R4,   R1,   R2.
```

As can be seen in the modified code, the instruction i_1 is executed regardless of the branch outcome.

Multiple Prefetching. In this type of design, the processor fetches both possible paths. Once the branch decision is made, the unwanted path is thrown away. By prefetching both possible paths, the fetch penalty is avoided in the case of an incorrect prediction.

To fetch both paths, two buffers are employed to service the pipeline. In normal execution, the first buffer is loaded with instructions from the next sequential address of the branch instruction. If a branch occurs, the contents of the first buffer are invalidated, and the secondary buffer, which has been loaded with instructions from the target address of the branch instruction, is used as the primary buffer.

This double buffering scheme ensures a constant flow of instructions and data to the pipeline and reduces the time delays caused by the draining and refilling of the pipeline. Some amount of performance degradation is unavoidable any time the pipeline is drained, however.

In summary, each of the preceding simple techniques reduces the degradation of pipeline throughput. However, the choice of any of these techniques for a particular design depends on factors such as throughput requirements and cost constraints. In practice, due to these factors, it is not unusual to see a mixture of these techniques implemented on a single processor.

3.2.2 Further Throughput Improvement of an Instruction Pipeline

One way to increase the throughput of an instruction pipeline is to exploit instruction-level parallelism. The common approaches to accomplish such parallelism are called *superscalar* [IBM 90, OEH 91], *superpipeline* [JOU 89, BAS 91], and *very long instruction word* (VLIW) [COL 88, FIS 83]. Each approach attempts to initiate several instructions per cycle.

Superscalar. The superscalar approach relies on spatial parallelism, that is, multiple operations running concurrently on separate hardware. This approach achieves the execution of multiple instructions per clock cycle by issuing several instructions to different functional units. A superscalar processor contains one or more instruction pipelines sharing a set of functional units. It often contains functional units, such as an add unit, multiply unit, divide unit, floating-point add unit, and graphic unit. A superscalar processor contains a control mechanism to preserve the execution order of dependent instructions for ensuring a valid result. The scoreboard method and Tomasulo's method (discussed in the previous section) can be used for implementing such mechanisms. In practice, most of the processors are based on the superscalar approach and employ a scoreboard method to ensure a valid result. Examples of such processors are given in Chapter 4.

Superpipeline. The superpipeline approach achieves high performance by overlapping the execution of multiple instructions on one instruction pipeline. A superpipeline processor often has an instruction pipeline with more stages than a typical instruction pipeline design. In other words, the execution process of an instruction is broken down into even finer steps. By increasing the number of stages in the instruction pipeline, each stage has less work to do. This allows the pipeline clock rate to increase (cycle time decreases), since the clock rate depends on the delay found in the slowest stage of the pipeline.

An example of such an architecture is the MIPS R4000 processor. The R4000 subdivides instruction fetching and data cache access to create an eight-stage pipeline. The stages are instruction fetch first half, instruction fetch second half, register fetch, instruction execute, data cache access first half, data cache access second half, tag check, and write back.

A superpipeline approach has certain benefits. The single functional unit requires less space and less logic on the chip than designs based on the superscalar approach. This extra space on the chip allows room for specialize circuitry to achieve higher speeds, room for large caches, and wide data paths.

Very Long Instruction Word (VLIW). The very long instruction word (VLIW) approach makes extensive use of the compiler by requiring it to incorporate several small independent operations into a long instruction word. The instruction is large enough to provide, in parallel, enough control bits over many functional units. In other words, a VLIW architecture provides many more functional units than a typical processor design, together with a compiler that finds parallelism across basic operations to keep the functional units as busy as possible. The compiler compacts ordinary sequential codes into long instruction words that make better use of resources. During execution, the control unit issues one long instruction per cycle. The issued instruction initiates many independent operations simultaneously.

A comparison of the three approaches will show a few interesting differences. For instance, the superscalar and VLIW approaches are more sensitive to resource conflicts than the superpipelined approach. In a superscalar or VLIW processor, a resource must be duplicated to reduce the chance of conflicts, while the superpipelined design avoids any resource conflicts.

To prevent the superpipelined processor from being slower than the superscalar, the technology used in the superpipelined must reduce the delay of the lengthy instruction pipeline. Therefore, in general, superpipelined designs require faster transistor technology such as GaAs (gallium arsinide), whereas superscalar designs require more transistors to account for the hardware resource duplication. The superscalar design often uses CMOS technology, since this technology provides good circuit density. Although superpipelining seems to be a more straightforward solution than superscaling, existing technology generally favors increasing circuit density over increasing circuit speed. Historically, circuit density has increased at a faster rate than transistor speed. This historical precedent suggests a general conclusion that the superscalar approach is more cost effective for industry to implement.

Technological advances have allowed superscalar and superpipelining techniques to be combined, providing good solutions to many current efficiency problems found in the computing industry. Such solutions, which attempt to take advantage of the positive attributes of each design, can be studied in existing processors. One example is the alpha microprocessor [DIG 92]. This microprocessor is described in detail in Chapter 4.