KEY POINTS

- A **superscalar processor** is one in which multiple independent instruction pipelines are used. Each pipeline consists of multiple stages, so that each pipeline can handle multiple instructions at a time. Multiple pipelines introduce a new level of parallelism, enabling multiple streams of instructions to be processed at a time. A superscalar processor exploits what is known as **instruction-level parallelism**, which refers to the degree to which the instructions of a program can be executed in parallel.
- A superscalar processor typically fetches multiple instructions at a time and then attempts to find nearby instructions that are independent of one another and can therefore be executed in parallel. If the input to one instruction depends on the output of a preceding instruction, then the latter instruction cannot complete execution at the same time or before the former instruction. Once such dependencies have been identified, the processor may issue and complete instructions in an order that differs from that of the original machine code.
- The processor may eliminate some unnecessary dependencies by the use of additional registers and the renaming of register references in the original code.
- Whereas pure RISC processors often employ delayed branches to maximize the utilization of the instruction pipeline, this method is less appropriate to a superscalar machine. Instead, most superscalar machines use traditional branch prediction methods to improve efficiency.

A superscalar implementation of a processor architecture is one in which common instructions—integer and floating-point arithmetic, loads, stores, and conditional branches—can be initiated simultaneously and executed independently. Such implementations raise a number of complex design issues related to the instruction pipeline.

Superscalar design arrived on the scene hard on the heels of RISC architecture. Although the simplified instruction set architecture of a RISC machine lends itself readily to superscalar techniques, the superscalar approach can be used on either a RISC or CISC architecture.

Whereas the gestation period for the arrival of commercial RISC machines from the beginning of true RISC research with the IBM 801 and the Berkeley RISC I was seven or eight years, the first superscalar machines became commercially available within just a year or two of the coining of the term *superscalar*. The superscalar approach has now become the standard method for implementing high-performance microprocessors.

In this chapter, we begin with an overview of the superscalar approach, contrasting it with superpipelining. Next, we present the key design issues associated with superscalar implementation. Then we look at several important examples of superscalar architecture.

14.1 OVERVIEW

The term *superscalar*, first coined in 1987 [AGER87], refers to a machine that is designed to improve the performance of the execution of scalar instructions. In most applications, the bulk of the operations are on scalar quantities. Accordingly, the superscalar approach represents the next step in the evolution of high-performance general-purpose processors.

The essence of the superscalar approach is the ability to execute instructions independently and concurrently in different pipelines. The concept can be further exploited by allowing instructions to be executed in an order different from the program order. Figure 14.1 shows, in general terms, the superscalar approach. There are multiple functional units, each of which is implemented as a pipeline, which support parallel execution of several instructions. In this example, two integer, two floating-point, and one memory (either load or store) operations can be executing at the same time.

Many researchers have investigated superscalar-like processors, and their research indicates that some degree of performance improvement is possible. Table 14.1 presents the reported performance advantages. The differences in the results arise from differences both in the hardware of the simulated machine and in the applications being simulated.

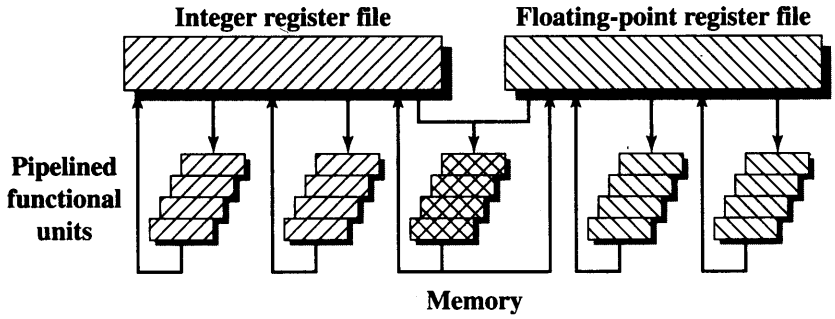


Figure 14.1 General Superscalar Organization [COME95]

Table 14.1 Reported Speedups of Superscalar-Like Machines

Reference	Speedup
[TJAD70]	1.8
[KUCK72]	8
[WEIS84]	1.58
[ACOS86]	2.7
[SOHI90]	1.8
[SMIT89]	2.3
[JOUP89b]	2.2
[LEE91]	7

Superscalar versus Superpipelined

An alternative approach to achieving greater performance is referred to as superpipelining, a term first coined in 1988 [JOUP88]. Superpipelining exploits the fact that many pipeline stages perform tasks that require less than half a clock cycle. Thus, a doubled internal clock speed allows the performance of two tasks in one external clock cycle. We have seen one example of this approach with the MIPS R4000.

Figure 14.2 compares the two approaches. The upper part of the diagram illustrates an ordinary pipeline, used as a base for comparison. The base pipeline

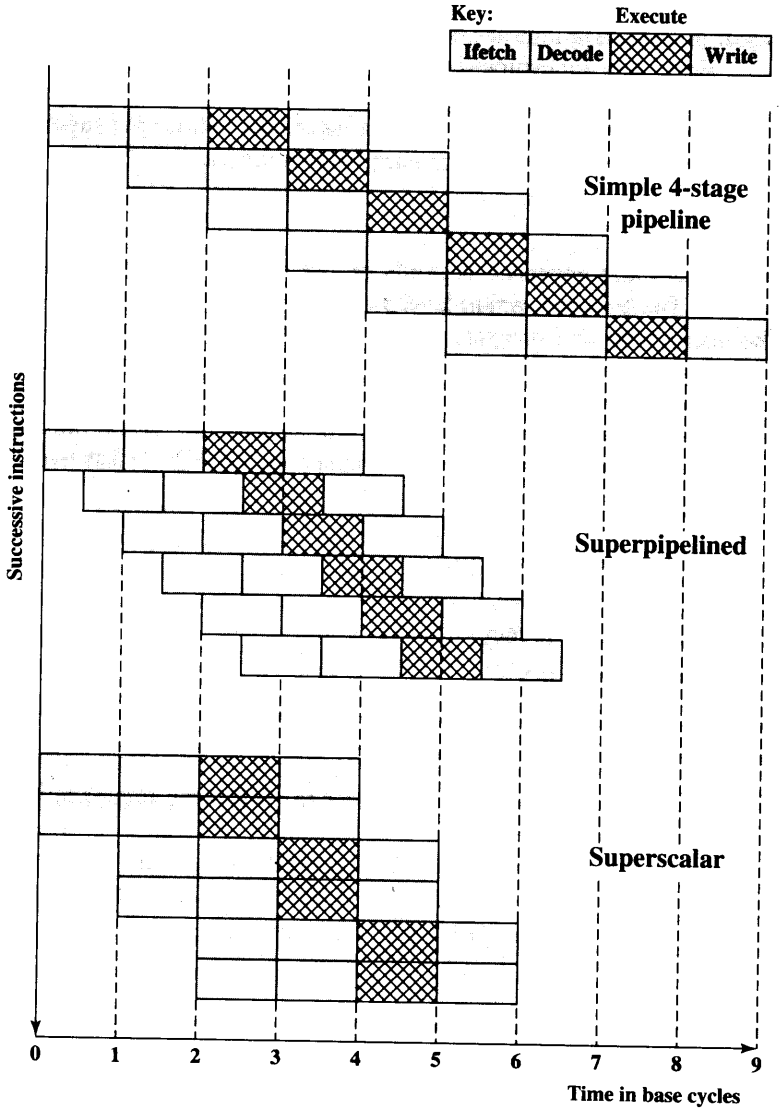


Figure 14.2 Comparison of Superscalar and Superpipeline Approaches

issues one instruction per clock cycle and can perform one pipeline stage per clock cycle. The pipeline has four stages: instruction fetch, operation decode, operation execution, and result write back. The execution stage is crosshatched for clarity. Note that although several instructions are executing concurrently, only one instruction is in its execution stage at any one time.

The next part of the diagram shows a superpipelined implementation that is capable of performing two pipeline stages per clock cycle. An alternative way of looking at this is that the functions performed in each stage can be split into two nonoverlapping parts and each can execute in half a clock cycle. A superpipeline implementation that behaves in this fashion is said to be of degree 2. Finally, the lowest part of the diagram shows a superscalar implementation capable of executing two instances of each stage in parallel. Higher-degree superpipeline and superscalar implementations are of course possible.

Both the superpipeline and the superscalar implementations depicted in Figure 14.2 have the same number of instructions executing at the same time in the steady state. The superpipelined processor falls behind the superscalar processor at the start of the program and at each branch target.

Limitations

The superscalar approach depends on the ability to execute multiple instructions in parallel. The term **instruction-level parallelism** refers to the degree to which, on average, the instructions of a program can be executed in parallel. A combination of compiler-based optimization and hardware techniques can be used to maximize instruction-level parallelism. Before examining the design techniques used in superscalar machines to increase instruction-level parallelism, we need to look at the fundamental limitations to parallelism with which the system must cope. [JOHN91] lists five limitations:

- True data dependency
- Procedural dependency
- Resource conflicts
- Output dependency
- Antidependency

We examine the first three of these limitations in the remainder of this section. A discussion of the last two must await some of the developments in the next section.

True Data Dependency Consider the following sequence:[1]

```
add    r1, r2    ;load register r1 with the contents
                  of r2 plus the contents of r1
move   r3, r1    ;load register r3 with the contents
                  of r1
```

The second instruction can be fetched and decoded but cannot execute until the first instruction executes. The reason is that the second instruction needs data

[1]For the Intel 80x86 and Pentium assembly language, a comment is indicated by a semicolon. The semicolon and all characters following the semicolon on the same line are ignored by the assembler.

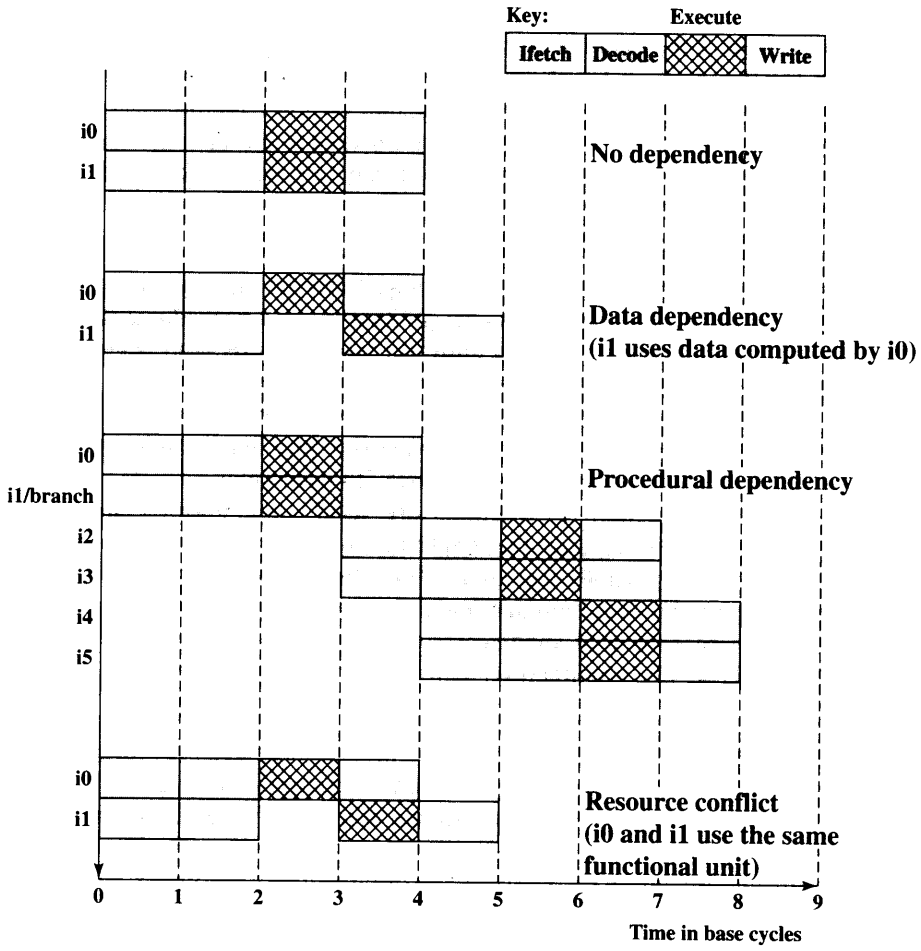


Figure 14.3 Effect of Dependencies

produced by the first instruction. This situation is referred to as a **true data dependency** (also called **flow dependency** or **write-read dependency**).

Figure 14.3 illustrates this dependency in a superscalar machine of degree 2. With no dependency, two instructions can be fetched and executed in parallel. If there is a data dependency between the first and second instructions, then the second instruction is delayed as many clock cycles as required to remove the dependency. In general, any instruction must be delayed until all of its input values have been produced.

In a simple pipeline, such as illustrated in the upper part of Figure 14.2, the aforementioned sequence of instructions would cause no delay. However, consider the following, in which one of the loads is from memory rather than from a register:

```
load   r1, eff   ;load register r1 with the contents
                  of effective memory address eff
move   r3, r1    ;load register r3 with the contents
                  of r1
```

A typical RISC processor takes two or more cycles to perform a load from memory because of the delay of an off-chip memory or cache access. One way to compensate for this delay is for the compiler to reorder instructions so that one or more subsequent instructions that do not depend on the memory load can begin flowing through the pipeline. This scheme is less effective in the case of a superscalar pipeline: The independent instructions executed during the load are likely to be executed on the first cycle of the load, leaving the processor with nothing to do until the load completes.

Procedural Dependencies As was discussed in Chapter 12, the presence of branches in an instruction sequence complicates the pipeline operation. The instructions following a branch (taken or not taken) have a procedural dependency on the branch and cannot be executed until the branch is executed. Figure 14.3 illustrates the effect of a branch on a superscalar pipeline of degree 2.

As we have seen, this type of procedural dependency also affects a scalar pipeline. The consequence for a superscalar pipeline is more severe, because a greater magnitude of opportunity is lost with each delay.

If variable-length instructions are used, then another sort of procedural dependency arises. Because the length of any particular instruction is not known, it must be at least partially decoded before the following instruction can be fetched. This prevents the simultaneous fetching required in a superscalar pipeline. This is one of the reasons that superscalar techniques are more readily applicable to a RISC or RISC-like architecture, with its fixed instruction length.

Resource Conflict A resource conflict is a competition of two or more instructions for the same resource at the same time. Examples of resources include memories, caches, buses, register-file ports, and functional units (e.g., ALU adder).

In terms of the pipeline, a resource conflict exhibits similar behavior to a data dependency (Figure 14.3). There are some differences, however. For one thing, resource conflicts can be overcome by duplication of resources, whereas a true data dependency cannot be eliminated. Also, when an operation takes a long time to complete, resource conflicts can be minimized by pipelining the appropriate functional unit.

14.2 DESIGN ISSUES

Instruction-Level Parallelism and Machine Parallelism

[JOUP89a] makes an important distinction between the two related concepts of instruction-level parallelism and machine parallelism. **Instruction-level parallelism** exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping.

As an example of the concept of instruction-level parallelism, consider the following two code fragments [JOUP89b]:

```
Load R1 ← R2          Add R3 ← R3, "1"
Add R3 ← R3, "1"      Add R4 ← R3, R2
Add R4 ← R4, R2       Store [R4] ← R0
```

The three instructions on the left are independent, and in theory all three could be executed in parallel. In contrast, the three instructions on the right cannot be executed in parallel because the second instruction uses the result of the first, and the third instruction uses the result of the second.

The degree of instruction-level parallelism is determined by the frequency of true data dependencies and procedural dependencies in the code. These factors, in turn, are dependent on the instruction set architecture and on the application. Instruction-level parallelism is also determined by what [JOUP89a] refers to as operation latency: the time until the result of an instruction is available for use as an operand in a subsequent instruction. The latency determines how much of a delay a data or procedural dependency will cause.

Machine parallelism is a measure of the ability of the processor to take advantage of instruction-level parallelism. Machine parallelism is determined by the number of instructions that can be fetched and executed at the same time (the number of parallel pipelines) and by the speed and sophistication of the mechanisms that the processor uses to find independent instructions.

Both instruction-level and machine parallelism are important factors in enhancing performance. A program may not have enough instruction-level parallelism to take full advantage of machine parallelism. The use of a fixed-length instruction set architecture, as in a RISC, enhances instruction-level parallelism. On the other hand, limited machine parallelism will limit performance no matter what the nature of the program.

Instruction Issue Policy

As was mentioned, machine parallelism is not simply a matter of having multiple instances of each pipeline stage. The processor must also be able to identify instruction-level parallelism and orchestrate the fetching, decoding, and execution of instructions in parallel. [JOHN91] uses the term **instruction issue** to refer to the process of initiating instruction execution in the processor's functional units and the term **instruction issue policy** to refer to the protocol used to issue instructions. In general, we can say that instruction issue occurs when instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline.

In essence, the processor is trying to look ahead of the current point of execution to locate instructions that can be brought into the pipeline and executed. Three types of orderings are important in this regard:

- The order in which instructions are fetched
- The order in which instructions are executed
- The order in which instructions update the contents of register and memory locations

The more sophisticated the processor, the less it is bound by a strict relationship between these orderings. To optimize utilization of the various pipeline elements, the processor will need to alter one or more of these orderings with respect to the ordering to be found in a strict sequential execution. The one constraint on the processor is that the result must be correct. Thus, the processor must accommodate the various dependencies and conflicts discussed earlier.

In general terms, we can group superscalar instruction issue policies into the following categories:

- In-order issue with in-order completion
- In-order issue with out-of-order completion
- Out-of-order issue with out-of-order completion

In-Order Issue with In-Order Completion The simplest instruction issue policy is to issue instructions in the exact order that would be achieved by sequential execution (in-order issue) and to write results in that same order (in-order completion). Not even scalar pipelines follow such a simple-minded policy. However, it is useful to consider this policy as a baseline for comparing more sophisticated approaches.

Figure 14.4a gives an example of this policy. We assume a superscalar pipeline capable of fetching and decoding two instructions at a time, having three separate functional units (e.g., two integer arithmetic and one floating-point arithmetic), and having two instances of the write-back pipeline stage. The example assumes the following constraints on a six-instruction code fragment:

- I1 requires two cycles to execute.
- I3 and I4 conflict for the same functional unit.
- I5 depends on the value produced by I4.
- I5 and I6 conflict for a functional unit.

Instructions are fetched two at a time and passed to the decode unit. Because instructions are fetched in pairs, the next two instructions must wait until the pair of decode pipeline stages has cleared. To guarantee in-order completion, when there is a conflict for a functional unit or when a functional unit requires more than one cycle to generate a result, the issuing of instructions temporarily stalls.

In this example, the elapsed time from decoding the first instruction to writing the last results is eight cycles.

In-Order Issue with Out-of-Order Completion Out-of-order completion is used in scalar RISC processors to improve the performance of instructions that require multiple cycles. Figure 14.4b illustrates its use on a superscalar processor. Instruction I2 is allowed to run to completion prior to I1. This allows I3 to be completed earlier, with the net result of a savings of one cycle.

With out-of-order completion, any number of instructions may be in the execution stage at any one time, up to the maximum degree of machine parallelism across all functional units. Instruction issuing is stalled by a resource conflict, a data dependency, or a procedural dependency.

In addition to the aforementioned limitations, a new dependency, which we referred to earlier as an **output dependency** (also called **write-write dependency**), arises. The following code fragment illustrates this dependency (*op* represents any operation):

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

(a) In-order issue and in-order completion

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

(b) In-order issue and out-of-order completion

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1, I2*	I1	I2				2
I5	I6	*I3, I4*	I1		I3	I2		3
		I4, I5, I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

(c) Out-of-order issue and out-of-order completion

Figure 14.4 Superscalar Instruction Issue and Completion Policies

Instruction I2 cannot execute before instruction I1, because it needs the result in register R3 produced in I1; this is an example of a true data dependency, as described in Section 14.1. Similarly, I4 must wait for I3, because it uses a result produced by I3. What about the relationship between I1 and I3? There is no data dependency here, as we have defined it. However, if I3 executes to completion prior to I1, then the wrong value of the contents of R3 will be fetched for the execution of I4. Consequently, I3 must complete after I1 to produce the correct output values. To ensure this, the issuing of the third instruction must be stalled if its result might later be overwritten by an older instruction that takes longer to complete.

Out-of-order completion requires more complex instruction issue logic than in-order completion. In addition, it is more difficult to deal with instruction interrupts and exceptions. When an interrupt occurs, instruction execution at the current point is suspended, to be resumed later. The processor must assure that the resumption takes into account that, at the time of interruption, instructions ahead of the instruction that caused the interrupt may already have completed.

Out-of-Order Issue with Out-of-Order Completion With in-order issue, the processor will only decode instructions up to the point of a dependency or conflict. No additional instructions are decoded until the conflict is resolved. As a result, the processor cannot look ahead of the point of conflict to subsequent instructions that may be independent of those already in the pipeline and that may be usefully introduced into the pipeline.

To allow out-of-order issue, it is necessary to decouple the decode and execute stages of the pipeline. This is done with a buffer referred to as an **instruction window**. With this organization, after a processor has finished decoding an instruction, it is placed in the instruction window. As long as this buffer is not full, the processor can continue to fetch and decode new instructions. When a functional unit becomes available in the execute stage, an instruction from the instruction window may be issued to the execute stage. Any instruction may be issued, provided that (1) it needs the particular functional unit that is available and (2) no conflicts or dependencies block this instruction.

The result of this organization is that the processor has a lookahead capability, allowing it to identify independent instructions that can be brought into the execute stage. Instructions are issued from the instruction window with little regard for their original program order. As before, the only constraint is that the program execution behaves correctly.

Figures 14.4c illustrates this policy. During each of the first three cycles, two instructions are fetched into the decode stage. During each cycle, subject to the constraint of the buffer size, two instructions move from the decode stage to the instruction window. In this example, it is possible to issue instruction I6 ahead of I5 (recall that I5 depends on I4, but I6 does not). Thus, one cycle is saved in both the execute and write-back stages, and the end-to-end savings, compared with Figure 14.4b, is one cycle.

The instruction window is depicted in Figure 14.4c to illustrate its role. However, this window is not an additional pipeline stage. An instruction being in the window simply implies that the processor has sufficient information about that instruction to decide when it can be issued.

The out-of-order issue, out-of-order completion policy is subject to the same constraints described earlier. An instruction cannot be issued if it violates a dependency or conflict. The difference is that more instructions are available for issuing, reducing the probability that a pipeline stage will have to stall. In addition, a new dependency, which we referred to earlier as an **antidependency** (also called **read-write dependency**), arises. The code fragment considered earlier illustrates this dependency:

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```

Instruction I3 cannot complete execution before instruction I2 begins execution and has fetched its operands. This is so because I3 updates register R3, which is a source operand for I2. The term *antidependency* is used because the constraint is similar to that of a true data dependency, but reversed: Instead of the first instruction producing a value that the second instruction uses, the second instruction destroys a value that the first instruction uses.

Register Renaming

When out-of-order instruction issuing and/or out-of-order instruction completion are allowed, we have seen that this gives rise to the possibility of output dependencies and antidependencies. These dependencies differ from true data dependencies and resource conflicts, which reflect the flow of data through a program and the sequence of execution. Output dependencies and antidependencies, on the other hand, arise because the values in registers may no longer reflect the sequence of values dictated by the program flow.

When instructions are issued in sequence and complete in sequence, it is possible to specify the contents of each register at each point in the execution. When out-of-order techniques are used, the values in registers cannot be fully known at each point in time just from a consideration of the sequence of instructions dictated by the program. In effect, values are in conflict for the use of registers, and the processor must resolve those conflicts by occasionally stalling a pipeline stage.

Antidependencies and output dependencies are both examples of storage conflicts. Multiple instructions are competing for the use of the same register locations, generating pipeline constraints that retard performance. The problem is made more acute when register optimization techniques are used (as discussed in Chapter 13), because these compiler techniques attempt to maximize the use of registers, hence maximizing the number of storage conflicts.

One method for coping with these types of storage conflicts is based on a traditional resource-conflict solution: duplication of resources. In this context, the technique is referred to as **register renaming**. In essence, registers are allocated dynamically by the processor hardware, and they are associated with the values needed by instructions at various points in time. When a new register value is created (i.e., when an instruction executes that has a register as a destination operand), a new register is allocated for that value. Subsequent instructions that access that value as a source operand in that register must go through a renaming process: the register references in those instructions must be revised to refer to the register containing the needed value. Thus, the same original register reference in several different instructions may refer to different actual registers, if different values are intended.

Let us consider how register renaming could be used on the code fragment we have been examining:

$$
\begin{aligned}
&\text{I1: } R3_b \leftarrow R3_a \text{ op } R5_a \\
&\text{I2: } R4_b \leftarrow R3_b + 1 \\
&\text{I3: } R3_c \leftarrow R5_a + 1 \\
&\text{I4: } R7_b \leftarrow R3_c \text{ op } R4_b
\end{aligned}
$$

The register reference without the subscript refers to the logical register reference found in the instruction. The register reference with the subscript refers to a hardware register allocated to hold a new value. When a new allocation is made for a particular logical register, subsequent instruction references to that logical register as a source operand are made to refer to the most recently allocated hardware register (recent in terms of the program sequence of instructions).

In this example, the creation of register $R3_c$ in instruction I3 avoids the antidependency on the second instruction and the output dependency on the first instruction, and it does not interfere with the correct value being accessed by I4. The result is that I3 can be issued immediately; without renaming, I3 cannot be issued until the first instruction is complete and the second instruction is issued.

Machine Parallelism

In the preceding, we have looked at three hardware techniques that can be used in a superscalar processor to enhance performance: duplication of resources, out-of-order issue, and renaming. One study that illuminates the relationship among these techniques was reported in [SMIT89]. The study made use of a simulation that modeled a machine with the characteristics of the MIPS R2000, augmented with various superscalar features. A number of different program sequences were simulated.

Figure 14.5 shows the results. In each of the graphs, the vertical axis corresponds to the mean speedup of the superscalar machine over the scalar machine. The horizontal axis shows the results for four alternative processor organizations. The base

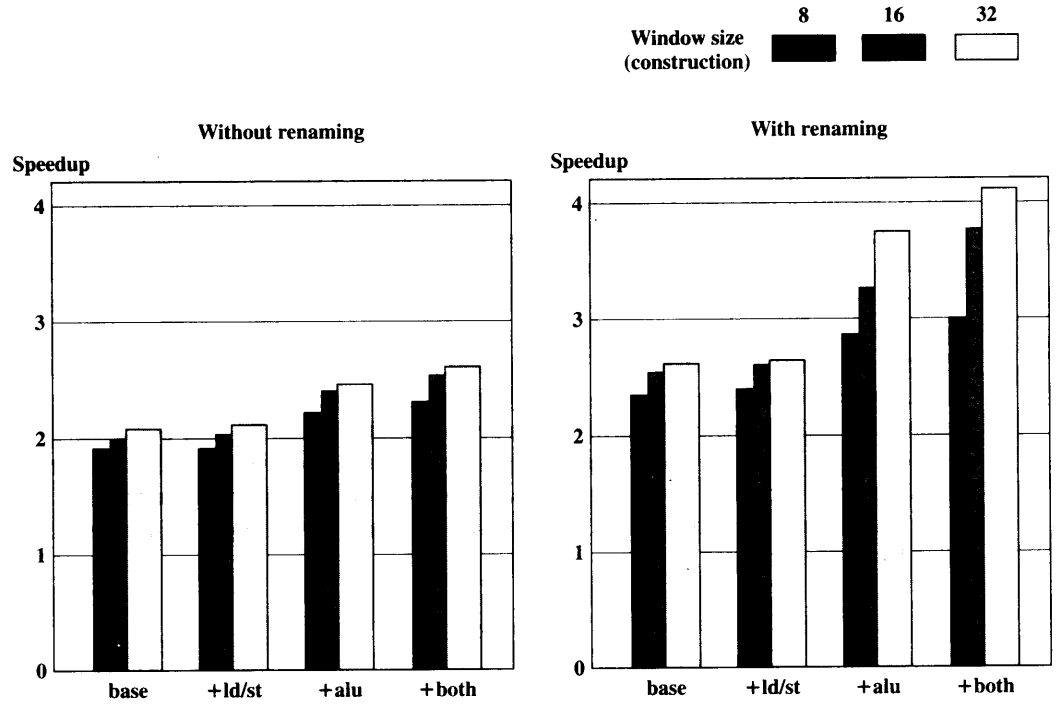


Figure 14.5 Speedups of Various Machine Organizations Without Procedural Dependencies

machine does not duplicate any of the functional units, but it can issue instructions out of order. The second configuration duplicates the load/store functional unit that accesses a data cache. The third configuration duplicates the ALU, and the fourth configuration duplicates both load/store and ALU. In each graph, results are shown for instruction window sizes of 8, 16, and 32 instructions, which dictates the amount of lookahead the processor can do. The difference between the two graphs is that, in the second, register renaming is allowed. This is equivalent to saying that the first graph reflects a machine that is limited by all dependencies, whereas the second graph corresponds to a machine that is limited only by true dependencies.

The two graphs, combined, yield some important conclusions. The first is that it is probably not worthwhile to add functional units without register renaming. There is some slight improvement in performance, but at the cost of increased hardware complexity. With register renaming, which eliminates antidependencies and output dependencies, noticeable gains are achieved by adding more functional units. Note, however, that there is a significant difference in the amount of gain achievable between using an instruction window of 8 versus a larger instruction window. This indicates that if the instruction window is too small, data dependencies will prevent effective utilization of the extra functional units; the processor must be able to look quite far ahead to find independent instructions to utilize the hardware more fully.

Branch Prediction

Any high-performance pipelined machine must address the issue of dealing with branches. For example, the Intel 80486 addressed the problem by fetching both the next sequential instruction after a branch and speculatively fetching the branch target instruction. However, because there are two pipeline stages between prefetch and execution, this strategy incurs a two-cycle delay when the branch gets taken.

With the advent of RISC machines, the delayed branch strategy was explored. This allows the processor to calculate the result of conditional branch instructions before any unusable instructions have been prefetched. With this method, the processor always executes the single instruction that immediately follows the branch. This keeps the pipeline full while the processor fetches a new instruction stream.

With the development of superscalar machines, the delayed branch strategy has less appeal. The reason is that multiple instructions need to execute in the delay slot, raising several problems relating to instruction dependencies. Thus, superscalar machines have returned to pre-RISC techniques of branch prediction. Some, like the PowerPC 601, use a simple static branch prediction technique. More sophisticated processors, such as the PowerPC 620 and the Pentium 4, use dynamic branch prediction based on branch history analysis.

Superscalar Execution

We are now in a position to provide an overview of superscalar execution of programs; this is illustrated in Figure 14.6. The program to be executed consists of a linear sequence of instructions. This is the static program as written by the programmer or generated by the compiler. The instruction fetch process, which includes branch prediction, is used to form a dynamic stream of instructions. This stream is examined for dependencies, and the processor may remove artificial dependencies.

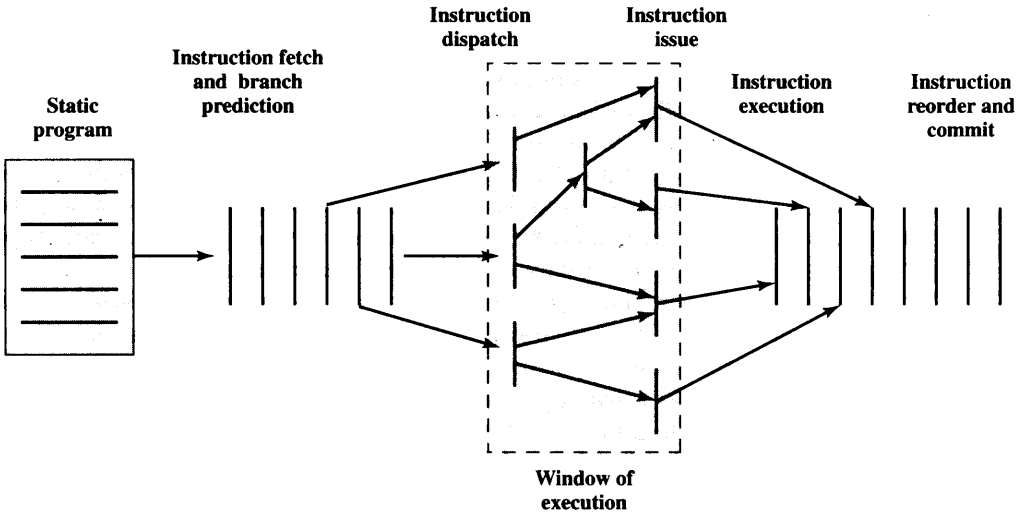


Figure 14.6 Conceptual Depiction of Superscalar Processing [SMIT95]

The processor then dispatches the instructions into a window of execution. In this window, instructions no longer form a sequential stream but are structured according to their true data dependencies. The processor performs the execution stage of each instruction in an order determined by the true data dependencies and hardware resource availability. Finally, instructions are conceptually put back into sequential order and their results are recorded.

The final step mentioned in the preceding paragraph is referred to as **committing**, or **retiring**, the instruction. This step is needed for the following reason. Because of the use of parallel, multiple pipelines, instructions may complete in an order different from that shown in the static program. Further, the use of branch prediction and speculative execution means that some instructions may complete execution and then must be abandoned because the branch they represent is not taken. Therefore, permanent storage and program-visible registers cannot be updated immediately when instructions complete execution. Results must be held in some sort of temporary storage that is usable by dependent instructions and then made permanent when it is determined that the sequential model would have executed the instruction.

Superscalar Implementation

Based on our discussion so far, we can make some general comments about the processor hardware required for the superscalar approach. [SMIT95] lists the following key elements:

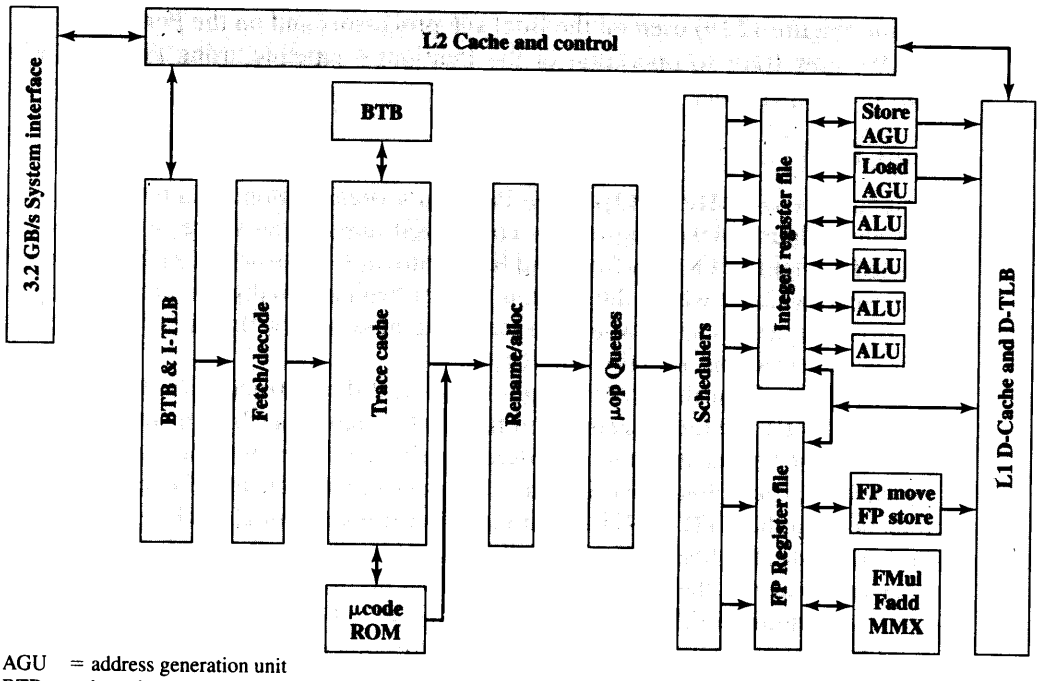
- Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions. These functions require the use of multiple pipeline fetch and decode stages, and branch prediction logic.
- Logic for determining true dependencies involving register values, and mechanisms for communicating these values to where they are needed during execution.

- Mechanisms for initiating, or issuing, multiple instructions in parallel.
- Resources for parallel execution of multiple instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references.
- Mechanisms for committing the process state in correct order.

14.3 PENTIUM 4

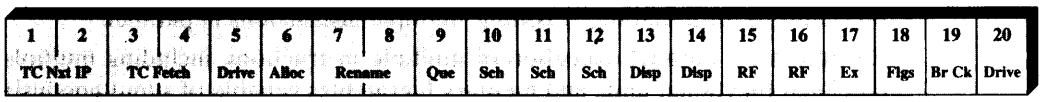
Although the concept of superscalar design is generally associated with the RISC architecture, the same superscalar principles can be applied to a CISC machine. Perhaps the most notable example of this is the Pentium. The evolution of superscalar concepts in the Intel line is interesting to note. The 80486 was a straightforward traditional CISC machine, with no superscalar elements. The original Pentium had a modest superscalar component, consisting of the use of two separate integer execution units. The Pentium Pro introduced a full-blown superscalar design. Subsequent Pentium models have refined and enhanced the superscalar design.

A general block diagram of the Pentium 4 was shown in Figure 4.13. Figure 14.7 depicts the same structure in a way more suitable for the pipeline discussion in this section. The operation of the Pentium 4 can be summarized as follows:



AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer

Figure 14.7 Pentium 4 Block Diagram



TC Next IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate
Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch
RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check

Figure 14.8 Pentium 4 Pipeline

1. The processor fetches instructions from memory in the order of the static program.
2. Each instruction is translated into one or more fixed-length RISC instructions, known as **micro-operations**, or **micro-ops**.
3. The processor executes the micro-ops on a superscalar pipeline organization, so that the micro-ops may be executed out of order.
4. The processor commits the results of each micro-op execution to the processor's register set in the order of the original program flow.

In effect, the Pentium 4 architecture consists of an outer CISC shell with an inner RISC core. The inner RISC micro-ops pass through a pipeline with at least 20 stages (Figure 14.8); in some cases, the micro-op requires multiple execution stages, resulting in an even longer pipeline. This contrasts with the five-stage pipeline (Figure 12.19) used on the Intel x86 processors and on the Pentium.

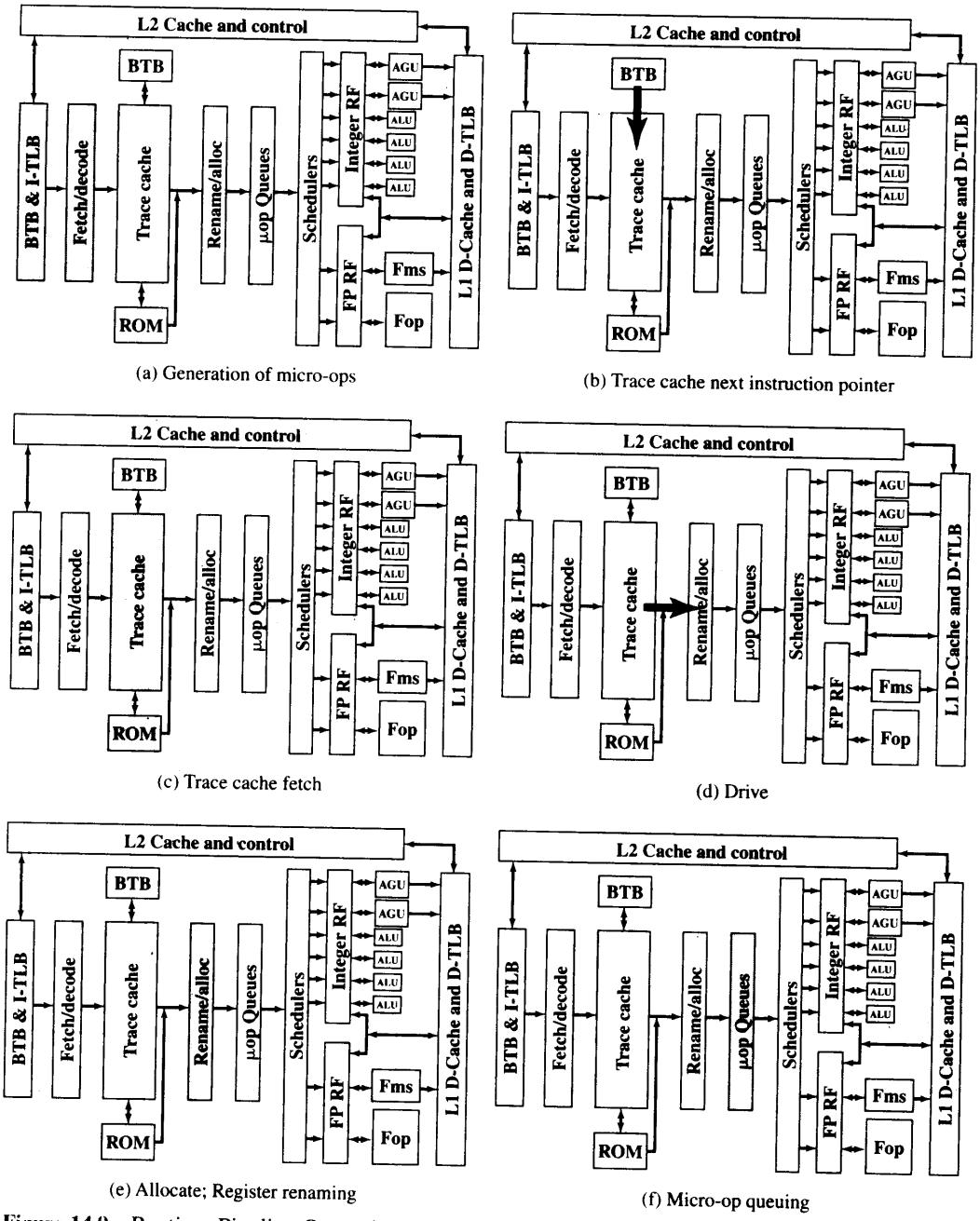
We now trace to operation of the Pentium 4 pipeline, using Figure 14.9 to illustrate its operation.

Front End

Generation of Micro-Ops The Pentium 4 organization includes an in-order front end (Figure 14.9a) that can be considered outside the scope of the pipeline depicted in Figure 14.8. This front end feeds into an L1 instruction cache, called the trace cache, which is where the pipeline proper begins. Usually, the processor operates from the trace cache; when a trace cache miss occurs, the in-order front end feeds new instructions into the trace cache.

With the aid of the branch target buffer and the instruction lookaside buffer (BTB & I-TLB), the fetch/decode unit fetches Pentium 4 machine instructions from the L2 cache 64 bytes at a time. As a default, instructions are fetched sequentially, so that each L2 cache line fetch includes the next instruction to be fetched. Branch prediction via the BTB & I-TLB unit may alter this sequential fetch operation. The ITLB translates the linear instruction pointer address given it into physical addresses needed to access the L2 cache. Static branch prediction in the front-end BTB is used to determine which instructions to fetch next.

Once instructions are fetched, the fetch/decode unit scans the bytes to determine instruction boundaries; this is a necessary operation because of the variable length of Pentium instructions. The decoder translates each machine instruction into from one to four micro-ops, each of which is a 118-bit RISC instruction. Note for comparison that most pure RISC machines have an instruction length of just 32 bits.



(a) Generation of micro-ops

(b) Trace cache next instruction pointer

(c) Trace cache fetch

(d) Drive

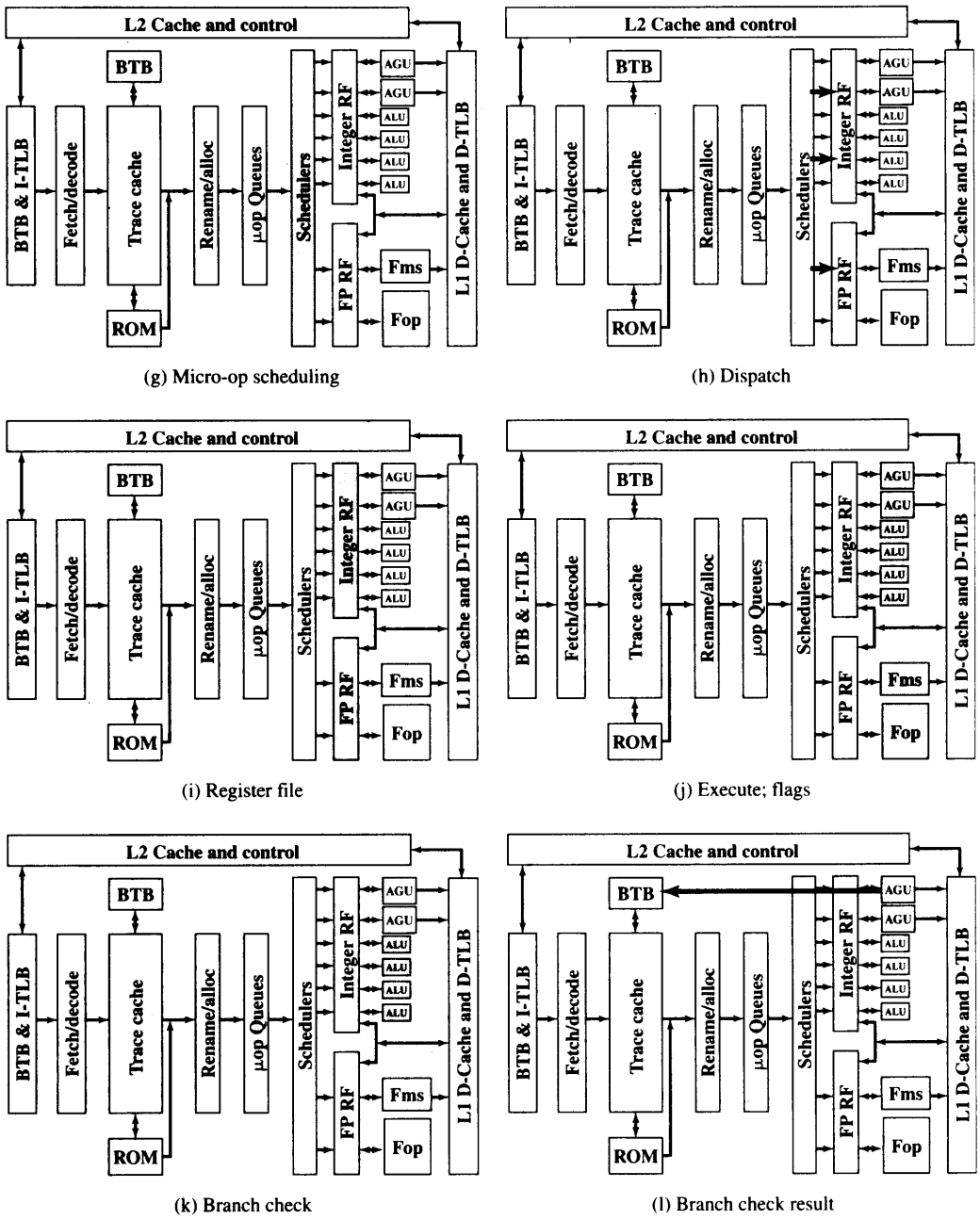
(e) Allocate; Register renaming

(f) Micro-op queuing

Figure 14.9 Pentium Pipeline Operation

The longer micro-op length is required to accommodate the more complex Pentium operations. Nevertheless, the micro-ops are easier to manage than the original instructions from which they derive.

The generated micro-ops are stored in the trace cache.



(g) Micro-op scheduling

(h) Dispatch

(i) Register file

(j) Execute; flags

(k) Branch check

(l) Branch check result

Figure 14.9 Continued

Trace Cache Next Instruction Pointer The first two pipeline stages (Figure 14.9b) deal with the selection of instructions in the trace cache and involve a separate branch prediction mechanism from that described in the previous section. The Pentium 4 uses a dynamic branch prediction strategy based on the history of recent

executions of branch instructions. A branch target buffer (BTB) is maintained that caches information about recently encountered branch instructions. Whenever a branch instruction is encountered in the instruction stream, the BTB is checked. If an entry already exists in the BTB, then the instruction unit is guided by the history information for that entry in determining whether to predict that the branch is taken. If a branch is predicted, then the branch destination address associated with this entry is used for prefetching the branch target instruction.

Once the instruction is executed, the history portion of the appropriate entry is updated to reflect the result of the branch instruction. If this instruction is not represented in the BTB, then the address of this instruction is loaded into an entry in the BTB; if necessary, an older entry is deleted.

The description of the preceding two paragraphs fits, in general terms, the branch prediction strategy used on the original Pentium model, as well as the later Pentium models, including Pentium 4. However, in the case of the Pentium, a relatively simple 2-bit history scheme is used. The later Pentium models have much longer pipelines (20 stages for the Pentium 4 compared with 5 stages for the Pentium) and therefore the penalty for misprediction is greater. Accordingly, the later Pentium models use a more elaborate branch prediction scheme with more history bits to reduce the misprediction rate.

The Pentium 4 BTB is organized as a four-way set-associative cache with 512 lines. Each entry uses the address of the branch as a tag. The entry also includes the branch destination address for the last time this branch was taken and a 4-bit history field. Thus use of four history bits contrasts with the 2 bits used in the original Pentium and used in most superscalar processors. With 4 bits, the Pentium 4 mechanism can take into account a longer history in predicting branches. The algorithm that is used is referred to as Yeh's algorithm [YEH91]. The developers of this algorithm have demonstrated that it provides a significant reduction in misprediction compared to algorithms that use only 2 bits of history [EVER98].

Conditional branches that do not have a history in the BTB are predicted using a static prediction algorithm, according to the following rules:

- For branch addresses that are not IP relative, predict taken if the branch is a return and not taken otherwise.
- For IP-relative backward conditional branches, predict taken. This rule reflects the typical behavior of loops.
- For IP-relative forward conditional branches, predict not taken.

Trace Cache Fetch The trace cache (Figure 14.9c) takes the already-decoded micro-ops from the instruction decoder and assembles them in to program-ordered sequences of micro-ops called traces. Micro-ops are fetched sequentially from the trace cache, subject to the branch prediction logic.

A few instructions require more than four micro-ops. These instructions are transferred to microcode ROM, which contains the series of micro-ops (five or more) associated with a complex machine instruction. For example, a string instruction may translate into a very large (even hundreds), repetitive sequence of micro-ops. Thus, the microcode ROM is a microprogrammed control unit in the sense discussed in Part Four. After the microcode ROM finishes sequencing micro-ops for the current Pentium instruction, fetching resumes from the trace cache.

Drive The fifth stage (Figure 14.9d) of the Pentium 4 pipeline delivers decoded instructions from the trace cache to the rename/allocator module.

Out-of-Order Execution Logic

This part of the processor reorders micro-ops to allow them to execute as quickly as their input operands are ready.

Allocate The allocate stage (Figure 14.9e) allocates resources required for execution. It performs the following functions:

- If a needed resource, such as a register, is unavailable for one of the three micro-ops arriving at the allocator during a clock cycle, the allocator stalls the pipeline.
- The allocator allocates a reorder buffer (ROB) entry, which tracks the completion status of one of the 126 micro-ops that could be in process at any time.
- The allocator allocates one of the 128 integer or floating-point register entries for the result data value of the micro-op, and possibly a load or store buffer used to track one of the 48 loads or 24 stores in the machine pipeline.
- The allocator allocates an entry in one of the two micro-op queues in front of the instruction schedulers.

The ROB is a circular buffer that can hold up to 126 micro-ops and also contains the 128 hardware registers. Each buffer entry consists of the following fields:

- **State:** Indicates whether this micro-op is scheduled for execution, has been dispatched for execution, or has completed execution and is ready for retirement.
- **Memory Address:** The address of the Pentium instruction that generated the micro-op.
- **Micro-op:** The actual operation.
- **Alias Register:** If the micro-op references one of the 16 architectural registers, this entry redirects that reference to one of the 128 hardware registers.

Micro-ops enter the ROB in order. Micro-ops are then dispatched from the ROB to the Dispatch/Execute unit out of order. The criterion for dispatch is that the appropriate execution unit and all necessary data items required for this micro-op are available. Finally, micro-ops are retired from the ROB in order. To accomplish in-order retirement, micro-ops are retired oldest first after each micro-op has been designated as ready for retirement.

Register Renaming The rename stage (Figure 14.9e) remaps references to the 16 architectural registers (8 floating-point registers, plus EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) into a set of 128 physical registers. The stage removes false dependencies caused by a limited number of architectural registers while preserving the true data dependencies (reads after writes).

Micro-Op Queuing After resource allocation and register renaming, micro-ops are placed in one of two micro-op queues (Figure 14.9f), where they are held until there is room in the schedulers. One of the two queues is for memory operations (loads and stores) and the other for micro-ops that do not involve memory

references. Each queue obeys a FIFO (first-in-first-out) discipline, but no order is maintained between queues. That is, a micro-op may be read out of one queue out of order with respect to micro-ops in the other queue. This provides greater flexibility to the schedulers.

Micro-Op Scheduling and Dispatching The schedulers (Figure 14.9g) are responsible for retrieving micro-ops from the micro-op queues and dispatching these for execution. Each scheduler looks for micro-ops in whose status indicates that the micro-op has all of its operands. If the execution unit needed by that micro-op is available, then the scheduler fetches the micro-op and dispatches it to the appropriate execution unit (Figure 14.9h). Up to six micro-ops can be dispatched in one cycle. If more than one micro-op is available for a given execution unit, then the scheduler dispatches them in sequence from the queue. This is a sort of FIFO discipline that favors in-order execution, but by this time the instruction stream has been so rearranged by dependencies and branches that it is substantially out of order.

Four ports attach the schedulers to the execution units. Port 0 is used for both integer and floating-point instructions, with the exception of simple integer operations and the handling of branch mispredictions, which are allocated to Port 1. In addition, MMX execution units are allocated between these two ports. The remaining ports are for memory loads and stores.

Integer and Floating-Point Execution Units

The integer and floating-point register files are the source for pending operations by the execution units (Figure 14.9i). The execution units retrieve values from the register files as well as from the L1 data cache (Figure 14.9j). A separate pipeline stage is used to compute flags (e.g., zero, negative); these are typically the input to a branch instruction.

A subsequent pipeline stage performs branch checking (Figure 14.9k). This function compares the actual branch result with the prediction. If a branch prediction turns out to have been wrong, then there are micro-operations in various stages of processing that must be removed from the pipeline. The proper branch destination is then provided to the Branch Predictor during a drive stage (Figure 14.9l), which restarts the whole pipeline from the new target address.

14.4 POWERPC

The PowerPC architecture is a direct descendant of the IBM 801, the RT PC, and the RS/6000, the last also referred to as an implementation of the POWER architecture. All of these are RISC machines, but the first in the series to exhibit superscalar features was the RS/6000. The first implementation of the PowerPC architecture, the 601, has a superscalar design quite similar to that of the RS/6000. Subsequent PowerPC models carry the superscalar concept further. In this section, we focus on the 601, which provides a good example of a RISC-based superscalar design. At the end of the section, we briefly consider the 620.

PowerPC 601

Figure 14.10 is a general view of the 601 organization. As with other superscalar machines, the 601 is broken up into independent functional units to enhance opportunities for overlapped execution. In particular, the core of the 601 consists of three independent pipelined execution units: integer, floating-point, and branch processing. Together, these units can execute three instructions at a time, yielding a superscalar design of degree 3.

Figure 14.11 shows a logical view of the 601 architecture, emphasizing the flow of instructions between functional units. The fetch unit can prefetch up to eight instructions at a time from the cache. The cache unit supports a combined instruction/data cache and is responsible for feeding instructions to the other units and data to the registers. Cache arbitration logic sends the address of the highest-priority access to the cache.

Dispatch Unit The dispatch unit takes instructions from the cache and loads them into the dispatch queue, which can hold eight instructions at a time. It processes this stream of instructions to feed a steady flow of instructions to the branch processing, integer, and floating-point units. The upper half of the queue simply acts as a buffer to hold instructions until they move into the lower half. Its purpose is to ensure that the dispatch unit is not delayed waiting for instructions from the cache. In the lower half, instructions are dispatched according to the following scheme:

- **Branch processing unit:** Handles all branch instructions. The lowest such instruction in the bottom half of the dispatch queue is issued to the branch processing unit if that unit can accept it.
- **Floating-point unit:** Handles all floating-point instructions. The lowest such instruction in the bottom half of the dispatch queue is issued to the floating-point unit if the instruction pipeline in that unit is not full.

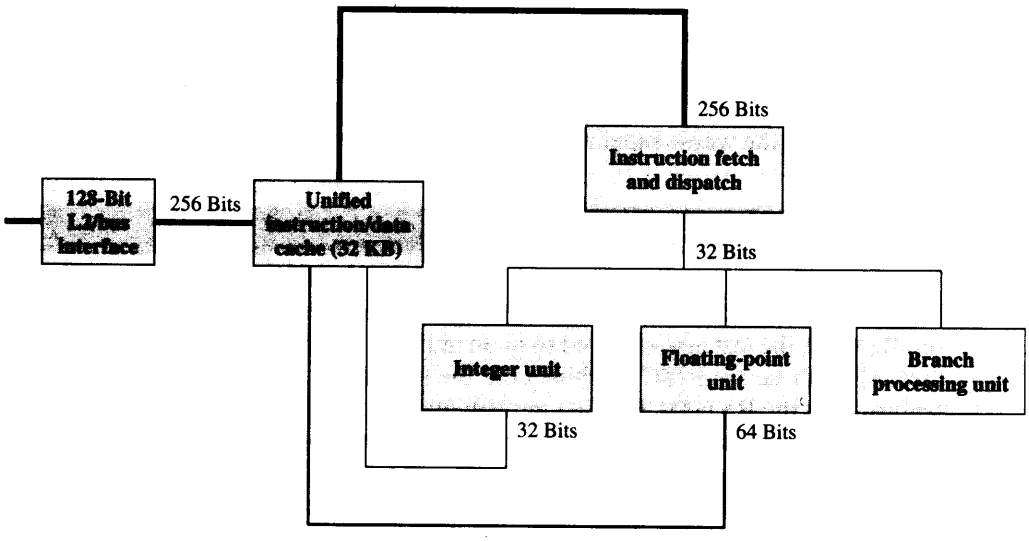


Figure 14.10 PowerPC 601 Block Diagram

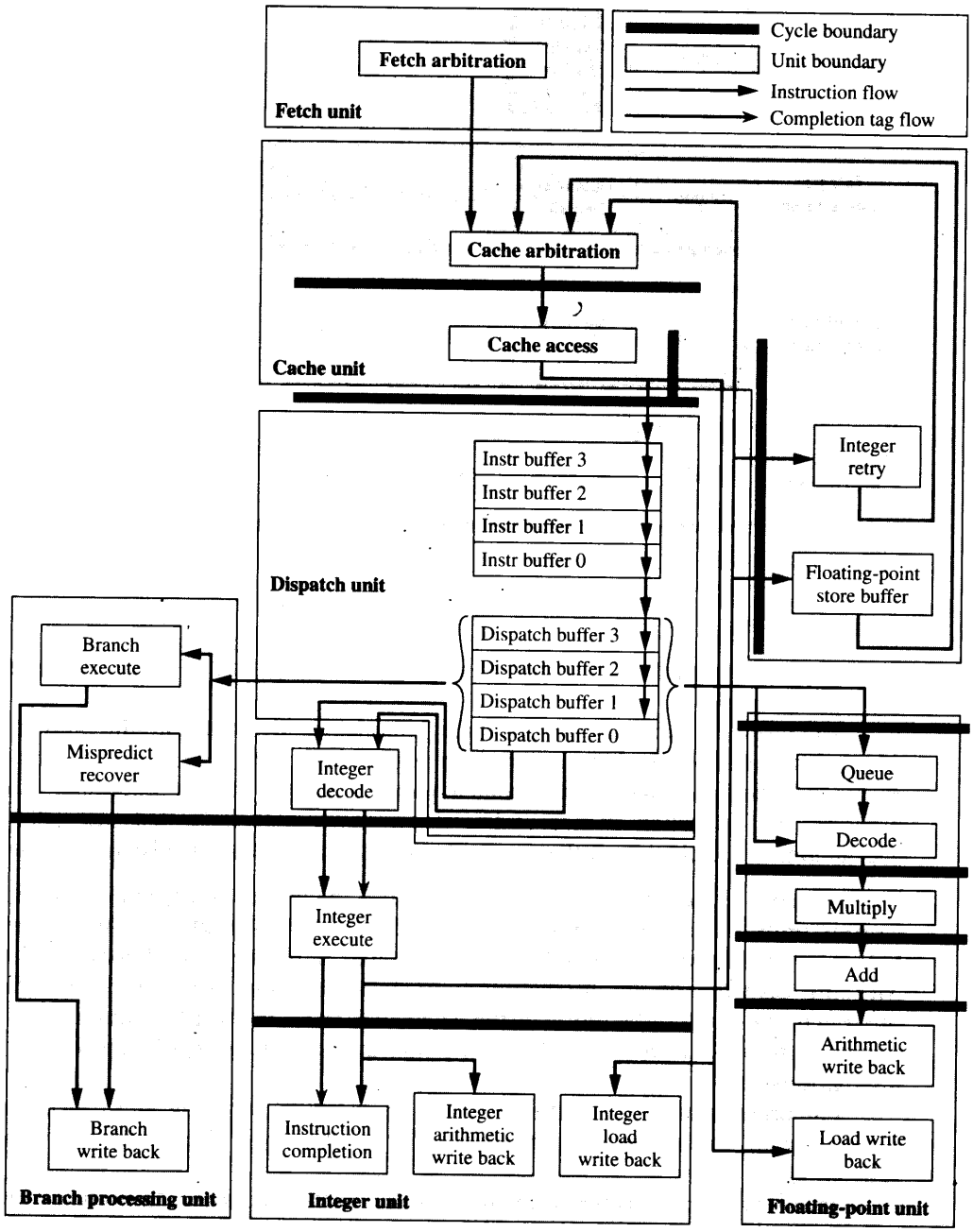


Figure 14.11 PowerPC 601 Pipeline Structure [POTT94]

- **Integer unit:** Handles integer instructions, load/stores between the register files and the cache, and integer compare instructions. An integer instruction is only issued after it has filtered to the bottom of the dispatch queue.

Allowing branch and floating-point instructions to be issued out of order from the dispatch queue helps keep the instruction pipelines in the branch processing and

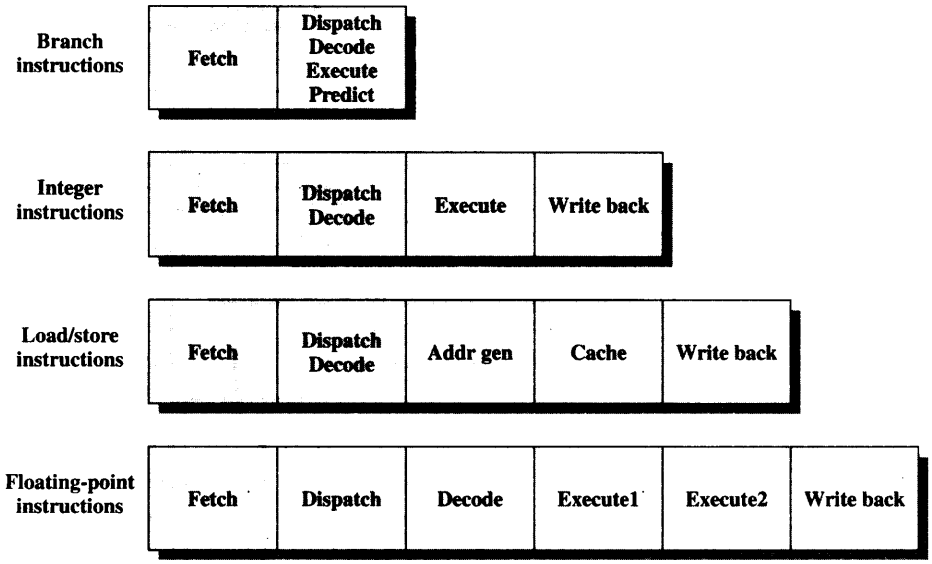


Figure 14.12 PowerPC 601 Pipeline

floating-point units full, and it moves instructions through the dispatch queue as rapidly as possible.

The dispatch unit also contains logic that enables it to calculate the prefetch address. It continues fetching instructions sequentially until a branch instruction moves into the lower half of the dispatch queue. When the branch processing unit processes an instruction, it may update the prefetch address so that succeeding instructions are fetched from the new address and entered into the dispatch queue.

Instruction Pipelines Figure 14.12 illustrates the instruction pipelines for the various units. There is a common fetch cycle for all instructions; this occurs before an instruction is dispatched to a particular unit. The second cycle begins with the dispatch of an instruction to a particular unit. This overlaps with other activities within the unit. During each clock cycle, the dispatch unit considers the bottom four entries of the instruction queue and dispatches up to three instructions.

For branch instructions, the second cycle involves decoding and executing instructions as well as predicting branches. The last activity is discussed in the next subsection.

The integer unit deals with instructions that cause a load/store operation with memory (including floating-point load/store), a register–register move, or an ALU operation. In the case of a load/store, there is an address generation cycle followed by sending the resulting address to the cache and, if necessary, a write-back cycle. For other instructions, the cache is not involved and there is an execute cycle followed by a write back to register.

Floating-point instructions follow a similar pipeline, but there are two execute cycles, reflecting the complexity of floating-point operations.

Several additional points are worth noting. The condition register contains eight independent 4-bit condition code fields. This allows multiple condition codes to be retained, which reduces the interlock or dependency between instructions. For example, the compiler can transform the sequence

```
compare
branch
compare
branch
•
•
•
```

to the sequence

```
compare
compare
•
•
•
branch
branch
•
•
•
```

Because each functional unit can send its condition codes to different fields in the condition register, interlocks between instructions caused by sharing of condition codes can be avoided.

The presence of the Save and Restore registers (SRRs) in the branch processor allows it to handle simple interrupts and software interrupts without involving logic in the other functional units. Thus, simple operating system services can be performed rapidly without complicated state manipulation or synchronization between the functional units.

Because the 601 can issue branch and floating-point instructions out of order, controls are needed to ensure proper execution. When a dependency exists (i.e., when an instruction needs an operand that has yet to be computed by a previous instruction), the pipeline in the corresponding unit stalls.

Branch Processing

The key to the high performance of a RISC or superscalar machine is its ability to optimize the use of the pipeline. Typically, the most critical element in the design is how branches are handled. In the PowerPC, branch processing is the responsibility of the branch unit. The unit is designed so that in many cases, branches have no effect on the pace of execution in the other units; these type of branches are referred

to as zero-cycle branches. To achieve zero-cycle branching, the following strategies are employed:

1. Logic is provided to scan through the dispatch buffer for branches. Branch target addresses are generated when a branch first appears in the lower half of the queue and no prior branches are pending execution.
2. An attempt is made to determine the outcome of conditional branches. If the condition code has been set sufficiently far in advance, this can be determined. In any case, as soon as a branch instruction is encountered, logic determines if the branch:
   a. Will be taken; this is the case for unconditional branches and for conditional branches whose condition code is known and indicates a branch.
   b. Will not be taken; this is the case for conditional branches whose condition code is known and indicates no branch.
   c. Outcome cannot yet be determined. In this case, the branch is guessed to be taken for backward branches (typical of loops) and guessed not to be taken for forward branches. Sequential instructions past the branch instruction are passed to the execution units in a conditional fashion. Once the condition code value is produced in the execution unit, the branch unit either cancels the instructions in the pipeline and proceeds with the fetched target if the branch is taken, or signals for the conditional instructions to be executed. The compiler can use a single bit in the instruction coding to reverse this default behavior.

The incorporation of a branch prediction strategy based on branch history was rejected because the designers felt that a minimal payoff would be achieved.

As an example of the branch prediction effect, consider the program of Figure 14.13 and assume that the branch processor predicts that the conditional branch instruction is not taken (the default case for a forward branch). Figure 14.14a shows the effect on the pipeline if in fact the branch is not taken. In the first cycle, the dispatch queue is loaded with eight instructions. The first six instructions are integer instructions and are dispatched one per cycle to the integer unit. The conditional branch instruction cannot be dispatched until it progresses to the lower half of the dispatch queue, which happens in cycle 5. The branch unit predicts that this branch will not be taken, and so the next instruction in sequence is conditionally dispatched (indicated by a D′). The branch cannot be resolved until the compare instruction executes in cycle 8. At that time, the branch processor confirms that its prediction was correct, and execution continues. There are no delays, and the pipeline is kept full.

Note that no instructions are fetched during cycles 4 through 8. This is because the cache is busy during those cycles with the cache access stage of the five load instructions. Even so, the instruction stream is not delayed, because the dispatch queue can hold eight instructions.

Figure 14.14b shows the result if the prediction is incorrect and the branch is taken. In this case, the three instructions starting at the IF must be flushed, and fetching resumes with instructions starting at ELSE. As a result, the execute stage of the integer pipeline is idle for cycles 9 and 10, resulting in a two-cycle loss because of the incorrect prediction.

```
if (a > 0)
        a = a + b + c + d + e;
else
        a = a - b - c - d - e;
```

(a) C code

```
                                    #r1 points to a,
                                    #r1+4 points to b,
                                    #r1+8 points to c,
                                    #r1+12 points to d,
                                    #r1+16 points to e.
        lwz     r8=a(r1)            #load a
        lwz     r12=b(r1,4)         #load b
        lwz     r9=c(r1,8)          #load c
        lwz     r10=d(r1,12)        #load d
        lwz     r11=e(r1,16)        #load e
        cmpi    cr0=r8,0            #compare immediate
        bc      ELSE,cr0/gt=false   #branch if bit false
IF:
        add     r12=r8,r12          #add
        add     r12=r12,r9          #add
        add     r12=r12,r10         #add
        add     r4=r12,r11          #add
        stw     a(r1)=r4            #store
        b       OUT                 #unconditional branch
ELSE:
        subf    r12=r12,r8          #subtract
        subf    r12=r9,r12          #subtract
        subf    r12=r10,r12         #subtract
        subf    r4=r12,r11          #subtract
        stw     a(r1)=r4            #store
OUT:
```

(b) Assembly code

Figure 14.13 Code Example with Conditional Branch [WEIS94]

PowerPC 620

The 620 is the first 64-bit implementation of the PowerPC architecture. A notable feature of this implementation is that it includes six independent execution units:

- Instruction unit
- Three integer units
- Load/store unit
- Floating-point unit

This organization enables the processor to dispatch up to four instructions simultaneously to the three integer units and one floating-point unit.

The 620 employs a high-performance branch prediction strategy that involves prediction logic, register rename buffers, and reservation stations inside the execution units. When an instruction is fetched, it is assigned a rename buffer to hold instruction results temporarily, such as register stores. Because of the use of rename buffers, the processor can *speculatively execute* instructions based on branch prediction; if the

```
                                1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
     lwz  r8=a(r1)              F  D  E  C  W
     lwz  r12=b(r1,4)           F  .  D  E  C  W
     lwz  r9=c(r1,8)            F  .  .  D  E  C  W
     lwz  r10=d(r1,12)          F  .  .  .  D  E  C  W
     lwz  r11=e(r1,16)          F  .  .  .  .  D  E  C  W
     cmpi cr0=r8,0              F  .  .  .  .  .  D  E
     bc   ELSE,cr0/gt=false     F  .  .  .  S
IF:  add  r12=r8,r12            F  .  .  .  .  .  .  D' E  W
     add  r12=r12,r9               F  .  .  .  .  .  D  E  W
     add  r12=r12,r10              F  .  .  .  .  .  .  D  E  W
     add  r4=r12,r11                                    F  .  D  E  W
     stw  a(r1)=r4                                      F  .  .  D  E  C
     b    OUT
ELSE:subf r12=r8,r12
     subf r12=r12,r9
     subf r12=r12,r10
     subf r4=r12,r11
     stw  a(r1)=r4
OUT:
```

(a) Correct prediction: Branch was not taken

```
                                1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
     lwz  r8=a(r1)              F  D  E  C  W
     lwz  r12=b(r1,4)           F  .  D  E  C  W
     lwz  r9=c(r1,8)            F  .  .  D  E  C  W
     lwz  r10=d(r1,12)          F  .  .  .  D  E  C  W
     lwz  r11=e(r1,16)          F  .  .  .  .  D  E  C  W
     cmpi cr0=r8,0              F  .  .  .  .  .  D  E
     bc   ELSE,cr0/gt=false     F  .  .  .  S
IF:  add  r12=r8,r12            F  .  .  .  .  .  .  D'
     add  r12=r12,r9               F  .  .  .  .  .  .
     add  r12=r12,r10              F  .  .  .  .  .  .
     add  r4=r12,r11
     stw  a(r1)=r4
     b    OUT
ELSE:subf r12=r8,r12                                    F  D  E  W
     subf r12=r12,r9                                    F  .  D  E  W
     subf r12=r12,r10                                   F  .  .  D  E  W
     subf r4=r12,r11                                    F  .  .  .  D  E  W
     stw  a(r1)=r4                                      F  .  .  .  .  D  E  C
OUT:
```

(b) Incorrect prediction: Branch was taken

F = fetch
D = dispatch/decode
E = execute/address
C = cache access
W = writeback
S = dispatch

Figure 14.14 Branch Prediction: Not Taken [WEIS94]

prediction turns out to be incorrect, then the results of the speculative instructions can be flushed without damaging the register file. Once the outcome of a branch is confirmed, temporary results can be written out permanently.

Each unit has two or more reservation stations, which store dispatched instructions that must be held up for the results of other instructions. This feature clears these instructions out of the instruction unit, enabling it to continue dispatching instructions to other execution units.

The 620 can speculatively execute up to four unresolved branch instructions (versus one for the 601). Branch prediction is based on the use of a branch history table with 2048 entries. Simulations run by the PowerPC designers show that the branch prediction success rate is 90% [THOM94].

14.5 RECOMMENDED READING

Two good book-length treatments of superscalar design are [SHEN05] and [OMON99]. Worthwhile survey articles on the subject are [SMIT95] and [SIMA97]. [JOUP89a] examines instruction-level parallelism, looks at various techniques for maximizing parallelism, and compares superscalar and superpipelined approaches using simulation. Recent papers that provide good coverage of superscalar design issues include [SIMA04], [PATT01], and [MOSH01].

[POPE91] provides a detailed look at a proposed superscalar machine. It also provides an excellent tutorial on the design issues related to out-of-order instruction policies. Another look at a proposed system is found in [KUGA91]; this article raises and considers most of the important design issues for superscalar implementation. [LEE91] examines software techniques that can be used to enhance superscalar performance. [WALL91] is an interesting study of the extent to which instruction-level parallelism can be exploited in a superscalar processor.

Volume I of [INTE04] provides general description of the Pentium 4 pipeline; more detail is provided in [INTE01a] and [INTE01b].

[POTT94] is a detailed examination of instruction pipelining on the PowerPC 601. [SHAN95] also provides good coverage.

HINT01 Hinton, G., et al. "The Microarchitecture of the Pentium 4 Processor." *Intel Technology Journal*, Q1 2001. **http://developer.intel.com/technology/itj/**

INTE04 Intel Corp. *IA-32 Intel Architecture Software Developer's Manual (4 volumes)*. Document 253665 through 253668. 2004. **http://developer.intel.com/design/Pentium4/documentation.htm.**

INTE01a Intel Corp. *Intel Pentium 4 Processor Optimization Reference Manual*. Document 248966-04 2001. **http://developer.intel.com/design/Pentium4/documentation.htm.**

INTE01b Intel Corp. *Desktop Performance and Optimization for Intel Pentium 4 Processor*. Document 248966-04 2001 **http://developer.intel.com/design/Pentium4/documentation.htm.**

JOUP89a Jouppi, N., and Wall, D. "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines." *Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

KUGA91 Kuga, M.; Murakami, K.; and Tomita, S. "DSNS (Dynamically-hazard resolved, Statically-code-scheduled, Nonuniform Superscalar): Yet Another Superscalar Processor Architecture." *Computer Architecture News*, June 1991.

LEE91 Lee, R.; Kwok, A.; and Briggs, F. "The Floating Point Performance of a Superscalar SPARC Processor." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

MOSH01 Moshovos, A., and Sohi, G. "Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling." *Proceedings of the IEEE*, November 2001.

OMON99 Omondi, A. *The Microarchitecture of Pipelined and Superscalar Computers*. Boston: Kluwer, 1999.

PATT01 Patt, Y. "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution." *Proceedings of the IEEE*, November 2001.

POPE91 Popescu, V., et al. "The Metaflow Architecture." *IEEE Micro*, June 1991.

POTT94 Potter, T., et al. "Resolution of Data and Control-Flow Dependencies in the PowerPC 601." *IEEE Micro*, October 1994.

SHAN95 Shanley, T. *PowerPC System Architecture*. Reading, MA: Addison-Wesley, 1995.

SHEN05 Shen, J., and Lipasti, M. *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, 2005.

SIMA97 Sima, D. "Superscalar Instruction Issue." *IEEE Micro*, September/October 1997.

SIMA04 Sima, D. "Decisive Aspects in the Evolution of Microprocessors." *Proceedings of the IEEE*, December 2004.

SMIT95 Smith, J., and Sohi, G. "The Microarchitecture of Superscalar Processors." *Proceedings of the IEEE*, December 1995.

WALL91 Wall, D. "Limits of Instruction-Level Parallelism." *Proceedings, Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

14.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

antidependency	machine parallelism	register renaming
branch prediction	micro-operations	resource conflict
commit	micro-ops	retire
flow dependency	out-of-order completion	superpipelined
in-order issue	out-of-order issue	superscalar
in-order completion	output dependency	true data dependency
instruction issue	procedural dependency	write-read dependency
instruction-level parallelism	read-write dependency	write-write dependency
instruction window		

Review Questions

14.1 What is the essential characteristic of the superscalar approach to processor design?

14.2 What is the difference between the superscalar and superpipelined approaches?

14.3 What is instruction-level parallelism?

14.4 Briefly define the following terms:

- True data dependency
- Procedural dependency
- Resource conflicts
- Output dependency
- Antidependency

14.5 What is the distinction between instruction-level parallelism and machine parallelism?

14.6 List and briefly define three types of superscalar instruction issue policies.

14.7 What is the purpose of an instruction window?

14.8 What is register renaming and what is its purpose?

14.9 What are the key elements of a superscalar processor organization?

Problems

14.1 When out-of-order completion is used in a superscalar processor, resumption of execution after interrupt processing is complicated, because the exceptional condition may have been detected as an instruction that produced its result out of order. The program cannot be restarted at the instruction following the exceptional instruction, because subsequent instructions have already completed, and doing so would cause these instructions to be executed twice. Suggest a mechanism or mechanisms for dealing with this situation.

14.2 Consider the following sequence of instructions, where the syntax consists of an opcode followed by the destination register followed by one or two source registers:

```
0     ADD     R3, R1, R2
1     LOAD    R6, [R3]
2     AND     R7, R5, 3
3     ADD     R1, R6, R0
4     SRL     R7, R0, 8
5     OR      R2, R4, R7
6     SUB     R5, R3, R4
7     ADD     R0, R1, R10
8     LOAD    R6, [R5]
9     SUB     R2, R1, R6
10    AND     R3, R7, 15
```

Assume the use of a four-stage pipeline: fetch, decode/issue, execute, write back. Assume that all pipeline stages take one clock cycle except for the execute stage. For simple integer arithmetic and logical instructions, the execute stage takes one cycle, but for a LOAD from memory, five cycles are consumed in the execute stage.

If we have a simple scalar pipeline but allow out-of-order execution, we can construct the following table for the execution of the first seven instructions:

Instruction	Fetch	Decode	Execute	Write Back
0	0	1	2	3
1	1	2	4	9
2	2	3	5	6
3	3	4	10	11
4	4	5	6	7
5	5	6	8	10
6	6	7	9	12

The entries under the four pipeline stages indicate the clock cycle at which each instruction begins each phase. In this program, the second ADD instruction (instruction 3) depends on the LOAD instruction (instruction 1) for one of its operands, rm. Because the LOAD instruction takes five clock cycles, and the issue logic encounters the dependent ADD instruction after two clocks, the issue logic must delay the ADD

instruction for three clock cycles. With an out-of-order capability, the processor can stall instruction 3 at clock cycle 4, and then move on to issue the following three independent instructions, which enter execution at clocks 6, 8, and 9. The LOAD finishes execution at clock 9, and so the dependent ADD can be launched into execution on clock 10.

a. Complete the preceding table.
b. Redo the table assuming no out-of-order capability. What is the savings using the capability?
c. Redo the table assuming a superscalar implementation that can handle two instructions at a time at each stage.

14.3 In the instruction queue in the dispatch unit of the PowerPC 601, instructions may be dispatched out of order to the branch processing and floating-point units, but instructions intended for the integer unit must be dispatched only from the bottom of the queue. Why this limitation?

14.4 Produce a figure similar to Figure 14.14 for the following cases:
a. Branch prediction: taken; correct prediction: branch was taken
b. Branch prediction: taken; incorrect prediction: branch was not taken

14.5 Consider the following assembly language program:

```
I1: Move R3, R7          /R3 ← (R7)/
I2: Load R8, (R3)        /R8 ← Memory (R3)/
I3: Add R3, R3, 4        /R3 ← (R3) + 4/
I4: Load R9, (R3)        /R9 ← Memory (R3)/
I5: BLE R8, R9, L3       /Branch if (R9) > (R8)/
```

This program includes write-write, read-write, and write-read dependencies. Show these.

14.6 Figure 14.15 shows an example of a superscalar processor organization. The processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines, with four processing stages (fetch, decode, execute, and store). Each pipeline has its own fetch decode and store unit. Four functional units (multiplier, adder, logic unit, and load unit) are available for use in the execute stage and are shared by the two pipelines on a dynamic basis. The two store units can be dynamically used by the two pipelines,

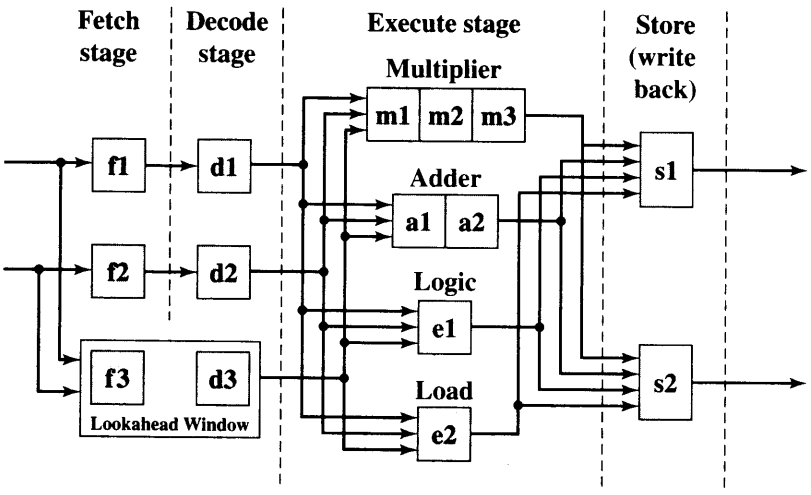


Figure 14.15 A Dual-Pipeline Superscalar Processor

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 for out-of-order instruction issue.

Consider the following program to be executed on this processor:

```
I1: Load R1, A      /R1 Memory (A)/
I2: Add R2, R1      /R2 ← (R2) + R(1)/
I3: Add R3, R4      /R3 ← (R3) + R(4)/
I4: Mul R4, R5      /R4 ← (R4) + R(5)/
I5: Comp R6         /R6 ← (R6)/
I6: Mul R6, R7      /R3 ← (R3) + R(4)/
```

a. What dependencies exist in the program?
b. Show the pipeline activity for this program on the processor of Figure 14.15 using in-order issue with in-order completion policies and using a presentation similar to Figure 14.2.
c. Repeat for in-order issue with out-of-order completion.
d. Repeat for out-of-order issue with out-of-order completion.

14.7 Figure 14.16 is from a paper on superscalar design. Explain the three parts of the figure, and define w, x, y, and z.

14.8 Yeh's dynamic branch prediction algorithm, used on the Pentium 4, is a two-level branch prediction algorithm. The first level is the history of the last n branches. The second level is the branch behavior of the last s occurrences of that unique pattern of the last n branches. It is implemented as possible. For each conditional branch instruction in a program, there is an entry in a Branch History Table (BHT). Each entry consists of n bits corresponding to the last n executions of the branch instruction, with a 1 if the branch was taken and a 0 if the branch was not. Each BHT entry indexes into a Pattern Table (PT) that has $2n$ entries, one for each possible pattern of n bits. Each PT entry consists of s bits that are used in branch prediction, as was described in Chapter 12 (e.g., Figure 12.17). When a conditional branch is encountered during instruction fetch and decode, the address of the instruction is used to retrieve the appropriate BHT entry, which shows the recent history of the instruction. Then, the BHT entry is used to retrieve the appropriate PT entry for

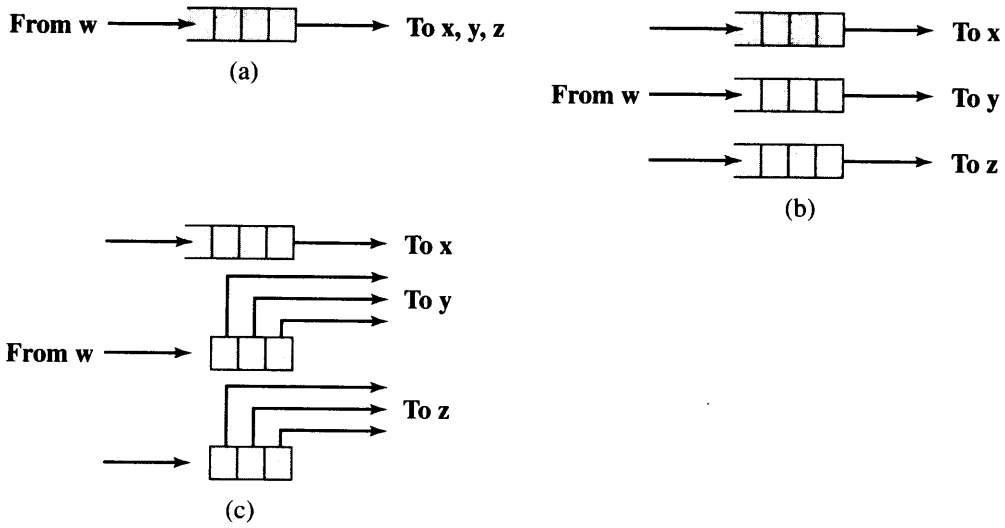


Figure 14.16 Figure for Problem 14.7

KEY POINTS

- The IA-64 instruction set architecture is a new approach to providing hardware support for instruction-level parallelism and is significantly different that the approach taken in superscalar architectures.
- The most noteworthy features of the IA-64 architecture are hardware support for predicated execution, control speculation, data speculation, and software pipelining.
- With **predicated execution**, every IA-64 instruction includes a reference to a 1-bit predicate register, and only executes if the predicate value is 1 (true). This enables the processor to speculatively execute both branches of an if statement and only commit after the condition is determined.
- With **control speculation**, a load instruction is moved earlier in the program and its original position replaced by a check instruction. The early load saves cycle time; if the load produces an exception, the exception is not activated until the check instruction determines if the load should have been taken.
- With **data speculation**, a load is moved before a store instruction that might alter the memory location that is the source of the load. A subsequent check is made to assure that the load receives the proper memory value.
- **Software pipelining** is a technique in which instructions from multiple iterations of a loop are enabled to execute in parallel.

With the Pentium 4, the microprocessor family that began with the 8086 and that has been the most successful computer product line ever appears to have come to an end. Intel has teamed up with Hewlett-Packard (HP) to develop a new 64-bit architecture, called IA-64. IA-64 is not a 64-bit extension of Intel's 32-bit x86 architecture, nor is it an adaptation of Hewlett-Packard's 64-bit PA-RISC architecture. Instead, IA-64 is a new architecture that builds on years of research at the two companies and at universities. The architecture exploits the vast circuitry and high speeds available on the newest generations of microchips by a systematic use of parallelism. IA-64 architecture represents a significant departure from the trend to superscalar schemes that have dominated recent processor development.

We begin this chapter with a discussion of the motivating factors for the new architecture. Next, we look at the general organization to support the architecture. We then examine in some detail the key features of the IA-64 architecture that promote instruction-level parallelism. Finally, we look at the IA-64 instruction set architecture and the Itanium organization.

15.1 MOTIVATION

The basic concepts underlying IA-64 are as follows:

- Instruction-level parallelism that is explicit in the machine instructions rather than being determined at run time by the processor
- Long or very long instruction words (LIW/VLIW)
- Branch predication (not the same thing as branch prediction)
- Speculative loading

Intel and HP refer to this combination of concepts as explicitly parallel instruction computing (EPIC). Intel and HP use the term **EPIC** to refer to the technology, or collection of techniques. **IA-64** is an actual instruction set architecture that is intended for implementation using the EPIC technology. The first Intel product based on this architecture is referred to as **Itanium**. Other products will follow, based on the same IA-64 architecture.

Table 15.1 summarizes key differences between, IA-64 and a traditional superscalar approach.

For Intel, the move to a new architecture that is not hardware compatible with the x86 instruction architecture, was a momentous decision. But it was driven by the dictates of the technology. When the x86 family began, back in the late 1970s, the processor chip had tens of thousands of transistors and was an essentially scalar device. That is, instructions were processed one at a time, with little or no pipelining. As the number of transistors increased into the hundreds of thousands in the mid-1980s, Intel introduced pipelining (e.g., Figure 12.19). Meanwhile, other manufacturers were attempting to take advantage of the increased transistor count and increased speed by means of the RISC approach, which enabled more effective pipelining, and later the superscalar/RISC combination, which involved multiple execution units. With the Pentium, Intel made a modest attempt to use superscalar techniques, allowing two CISC instructions to execute at a time. Then, the Pentium Pro and Pentium II through Pentium 4 incorporated a mapping from CISC instructions to RISC-like micro-operations and the more aggressive use of superscalar techniques. This approach

Table 15.1 Traditional Superscalar versus IA-64 Architecture

Superscalar	IA-64
RISC-like instructions, one per word	RISC-like instructions bundled into groups of three
Multiple parallel execution units	Multiple parallel execution units
Reorders and optimizes instruction stream at run time	Reorders and optimizes instruction stream at compile time
Branch prediction with speculative execution of one path	Speculative execution along both paths of a branch
Loads data from memory only when needed, and tries to find the data in the caches first	Speculatively loads data before its needed, and still tries to find data in the caches first

enabled the effective use of a chip with millions of transistors. But for the next generation processor, the one beyond Pentium, Intel and other manufacturers are faced with the need to use effectively tens of millions of transistors on a single processor chip.

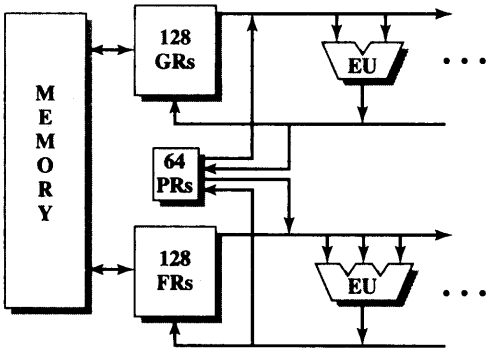
Processor designers have few choices in how to use this glut of transistors. One approach is to dump those extra transistors into bigger on-chip caches. Bigger caches can improve performance to a degree but eventually reach a point of diminishing returns, in which larger caches result in tiny improvements in hit rates. Another approach is to provide for multiple processors on a single chip. This approach is discussed in Chapters 2 and 16. Yet another alternative is to increase the degree of superscaling by adding more execution units. The problem with this approach is that designers are, in effect, hitting a complexity wall. As more and more execution units are added, making the processor "wider," more logic is needed to orchestrate these units. Branch prediction must be improved, out-of-order processing must be used, and longer pipelines must be employed. But with more and longer pipelines, there is a greater penalty for misprediction. Out-of-order execution requires a large number of renaming registers and complex interlock circuitry to account for dependencies. As a result, today's best processors can manage at most to retire six instructions per cycle, and usually less.

To address these problems, Intel and HP have come up with an overall design approach that enables the effective use of a processor with many parallel execution units. The heart of this new approach is the concept of explicit parallelism. With this approach, the compiler statically schedules the instructions at compile time, rather than having the processor dynamically schedule them at run time. The compiler determines which instructions can execute in parallel and includes this information with the machine instruction. The processor uses this information to perform parallel execution. One advantage of this approach is that the EPIC processor does not need as much complex circuitry as an out-of-order superscalar processor. Further, whereas the processor has only a matter of nanoseconds to determine potential parallel execution opportunities, the compiler has orders of magnitude more time to examine the code at leisure and see the program as a whole.

15.2 GENERAL ORGANIZATION

As with any processor architecture, IA-64 can be implemented in a variety of organizations. Figure 15.1 suggests in general terms the organization of an IA-64 machine. The key features are as follows:

- **Large number of registers:** The IA-64 instruction format assumes the use of 256 registers: 128 64-bit registers for integer, logical, and general-purpose use, and 128 82-bit registers for floating-point and graphic use. There are also 64 1-bit predicate registers used for predicated execution, as explained subsequently.
- **Multiple execution units:** A typical commercial superscalar machine today may support four parallel pipelines, using four parallel execution units in both the integer and floating-point portions of the processor. It is expected that IA-64 will be implemented on systems with eight or more parallel units.



GR = General-purpose or integer register
FR = Floating-point or graphics register
PR = One-bit predicate register
EU = Execution unit

Figure 15.1 General Organization for IA-64 Architecture

The register file is quite large compared with most RISC and superscalar machines. The reason for this is that a large number of registers is needed to support a high degree of parallelism. In a traditional superscalar machine, the machine language (and the assembly language) employs a small number of visible registers, and the processor maps these onto a larger number of registers using register renaming techniques and dependency analysis. Because we wish to make parallelism explicit and relieve the processor of the burden of register renaming and dependency analysis, we need a large number of explicit registers.

The number of execution units is a function of the number of transistors available in a particular implementation. The processor will exploit parallelism to the extent that it can. For example, if the machine language instruction stream indicates that eight integer instructions may be executed in parallel, a processor with four integer pipelines will execute these in two chunks. A processor with eight pipelines will execute all eight instructions simultaneously.

Four types of execution unit are defined in the IA-64 architecture:

- **I-unit:** For integer arithmetic, shift-and-add, logical, compare, and integer multimedia instructions
- **M-unit:** Load and store between register and memory plus some integer ALU operations
- **B-unit:** Branch instructions
- **F-unit:** Floating-point instructions

Each IA-64 instruction is categorized into one of six types. Table 15.2 lists the instruction types and the execution unit types on which they may be executed. The extended (X) instruction type includes instructions in which two slots in a bundle are used to encode the instruction, allowing for more information than fits into a 41-bit instruction (slots and bundles are explained in the next section).

Table 15.2 Relationship between Instruction Type and Execution Unit Type

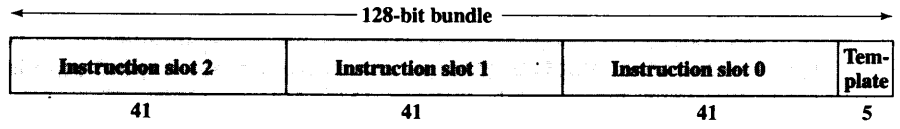
Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
X	Extended	I-unit/B-unit

15.3 PREDICATION, SPECULATION, AND SOFTWARE PIPELINING

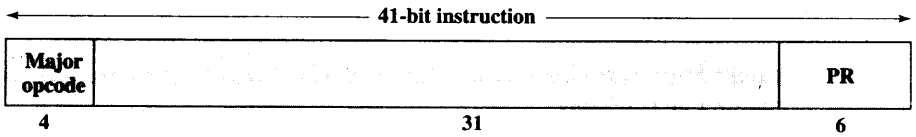
This section looks at the key features of the IA-64 architecture that support instruction-level parallelism. First, we need to provide an overview of the IA-64 instruction format and, to support the examples in this section, define the general format of IA-64 assembly language instructions.

Instruction Format

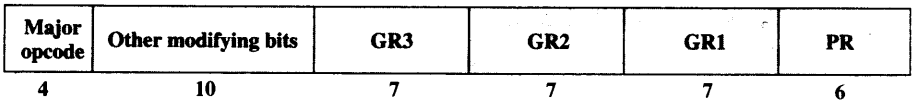
IA-64 defines a 128-bit **bundle** that contains three instructions, called **syllables**, and a template field (Figure 15.2a). The processor can fetch instructions one or more bundles at a time; each bundle fetch brings in three instructions. The



(a) IA-64 bundle



(b) General IA-64 instruction format



(c) Typical IA-64 instruction format

PR = Predicate register
GR = General or floating-point register

Figure 15.2 IA-64 Instruction Format

template field contains information that indicates which instructions can be executed in parallel. The interpretation of the template field is not confined to a single bundle. Rather, the processor can look at multiple bundles to determine which instructions may be executed in parallel. For example, the instruction stream may be such that eight instructions can be executed in parallel. The compiler will reorder instructions so that these eight instructions span contiguous bundles and set the template bits so that the processor knows that these eight instructions are independent.

The bundled instructions do not have to be in the original program order. Further, because of the flexibility of the template field, the compiler can mix independent and dependent instructions in the same bundle. Unlike some previous VLIW designs, IA-64 does not need to insert null-operation (NOP) instructions to fill in the bundles.

Table 15.3 shows the interpretation of the possible values for the 5-bit template field (some values are reserved and not in current use). The template value accomplishes two purposes:

Table 15.3 Template Field Encoding and Instruction Set Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit
05	M-unit	L-unit	X-unit
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit

1. The field specifies the mapping of instruction slots to execution unit types. Not all possible mappings of instructions to units are available.
2. The field indicates the presence of any **stops**. A stop indicates to the hardware that one or more instructions before the stop may have certain kinds of resource dependencies with one or more instructions after the stop. In the table, a heavy vertical line indicates a stop.

Each instruction has a fixed-length 41-bit format (Figure 15.2b). This is somewhat longer than the traditional 32-bit length found on RISC and RISC superscalar machines (although it is much shorter than the 118-bit micro-operation of the Pentium 4). Two factors lead to the additional bits. First, IA-64 makes use of more registers than a typical RISC machine: 128 integer and 128 floating-point registers. Second, to accommodate the predicated execution technique, an IA-64 machine includes 64 predicate registers. Their use is explained subsequently.

Figure 15.2c shows in more detail the typical instruction format. All instructions include a 4-bit major opcode and a reference to a predicate register. Although the major opcode field can only discriminate among 16 possibilities, the interpretation of the major opcode field depends on the template value and the location of the instruction within a bundle (Table 15.3), thus affording more possible opcodes. Typical instructions also include three fields to reference registers, leaving 10 bits for other information needed to fully specify the instruction.

Assembly-Language Format

As with any machine instruction set, an assembly language is provided for the convenience of the programmer. The assembler or compiler then translates each assembly language instruction into a 41-bit IA-64 instruction. The general format of an assembly language instruction is

$$[qp]\ mnemonic[.comp]\ dest\ =\ srcs$$

where

qp — Specifies a 1-bit predicate register used to qualify the instruction. If the value of the register is 1 (true) at execution time, the instruction executes and the result is committed in hardware. If the value is false, the result of the instruction is not committed but is discarded. Most IA-64 instructions may be qualified by a predicate but need not be. To account for an instruction that is not predicated, the qp value is set to 0 and predicate register zero always has the constant value of 1.

mnemonic — Specifies the name of an IA-64 instruction.

comp — Specifies one or more instruction completers, separated by periods, which are used to qualify the mnemonic. Not all instructions require the use of a completer.

dest — Specifies one or more destination operands, with the typical case being a single destination.

srcs — Specifies one or more source operands. Most instructions have two or more source operands.

On any line, any characters to the right of a double slash "//" are treated as a comment. Instruction groups and stops are indicated by a double semicolon ";;". An **instruction group** is defined as a sequence of instructions that have no read after write or write after write dependencies. The processor can issue these without hardware checks for register dependencies. Here is a simple example:

```
ld8  r1 = [r5] ;;     // First group
add  r3 = r1, r4      // Second group
```

The first instruction reads an 8-byte value from the memory location whose address is in register r5 and then places that value in register r1. The second instruction adds the contents of r1 and r4 and places the result in r3. Because the second instruction depends on the value in r1, which is changed by the first instruction, the two instructions cannot be in the same group for parallel execution.

Here is a more complex example, with multiple register flow dependencies:

```
ld8  r1 = [r5]        // First group
sub  r6 = r8, r9 ;;   // First group
add  r3 = r1, r4      // Second group
st8  [r6] = r12       // Second group
```

The last instruction stores the contents of r12 in the memory location whose address is in r6.

We are now ready to look at the four key mechanisms in the IA-64 architecture to support instruction-level parallelism:

- Predication
- Control speculation
- Data speculation
- Software pipelining

Figure 15.3, based on a figure in [HALF97], illustrates the first two of these techniques, which are discussed in this subsection and the next.

Predicated Execution

Predication is a technique whereby the compiler determines which instructions may execute in parallel. In the process, the compiler eliminates branches from the program by using conditional execution. A typical example in a high-level language is an **if-then-else** instruction. A traditional compiler inserts a conditional branch at the **if** point of this construct. If the condition has one logical outcome, the branch is not taken and the next block of instructions is executed, representing the **then** path; at the end of this path is an unconditional branch around the next block, representing the **else** path. If the condition has the other logical outcome, the branch is taken around the **then** block of instructions and execution continues at the **else** block of instructions. The two instruction streams join together after the end of the **else** block. An IA-64 compiler instead does the following (Figure 15.3a):

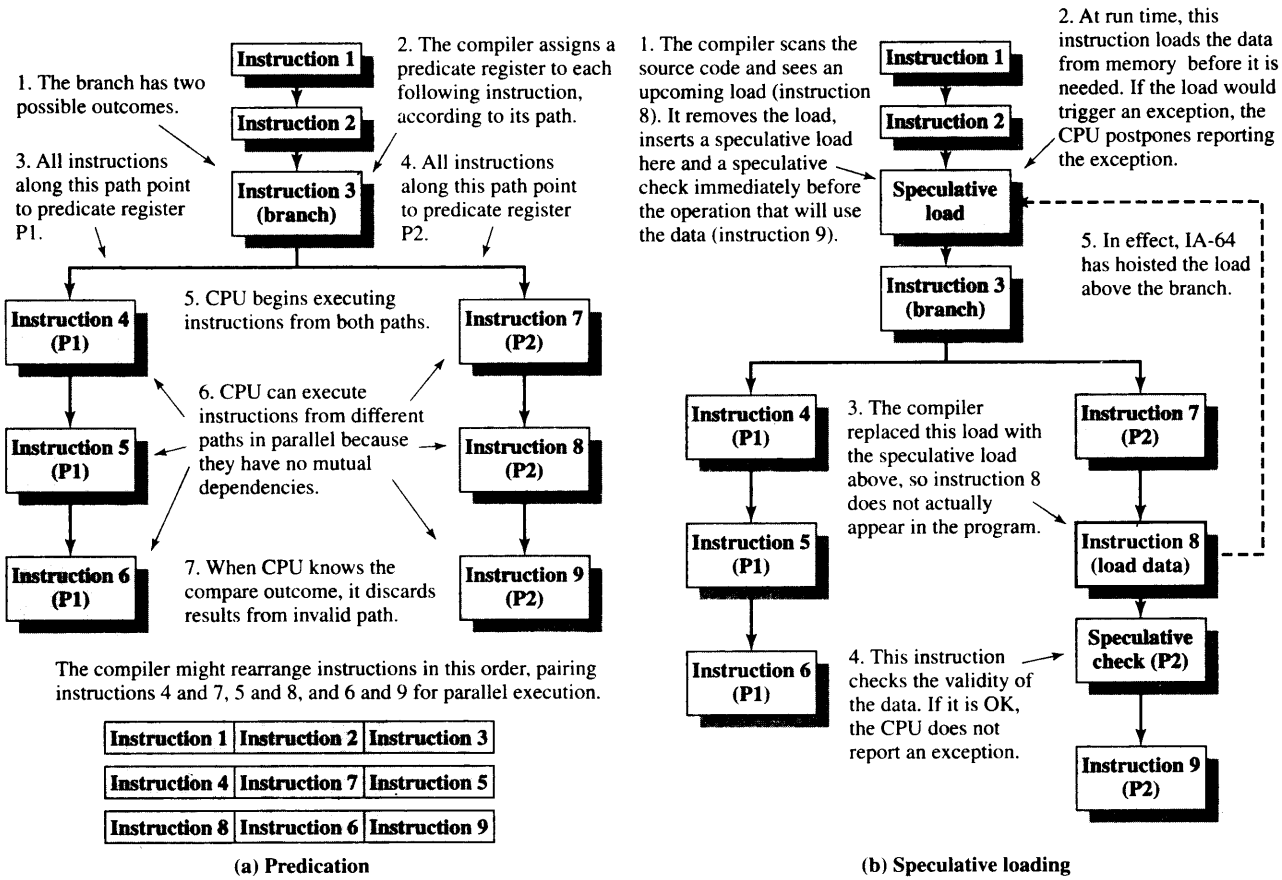


Figure 15.3 IA-64 Predication and Speculative Loading

1. At the **if** point in the program, insert a compare instruction that creates two predicates. If the compare is true, the first predicate is set to true and the second to false; if the compare is false, the first predicate is set to false and the second to true.
2. Augment each instruction in the **then** path with a reference to a predicate register that holds the value of the first predicate, and augment each instruction in the **else** path with a reference to a predicate register that holds the value of the second predicate.
3. The processor executes instructions along both paths. When the outcome of the compare is known, the processor discards the results along one path and commits the results along the other path. This enables the processor to feed instructions on both paths into the instruction pipeline without waiting for the compare operation to complete.

As an example, consider the following source code:

Source Code:

```
if (a&&b)
   j = j + 1;
else
   if (c)
      k = k + 1;
else
      k = k - 1;
i = i + 1;
```

Two if statements jointly select one of three possible execution paths. This can be compiled into the following code, using the Pentium assembly language. The program has three conditional branches and one unconditional branch instructions:

Assembly Code

```
      cmp   a,  0   ;  compare a with 0
      je    L1      ;  branch to L1 if a = 0
      cmp   b,  0
      je    L1
      add   j,  1   ;  j = j + 1
      jmp   L3
L1:   cmp   c,  0
      je    L2
      add   k,  1   ;  k = k + 1
      jmp   L3
L2:   sub   k,  1   ;  k = k - 1
L3:   add   i,  1   ;  i = i + 1
```

In the Pentium assembly language, a semicolon is used to delimit a comment.

Figure 15.4 shows a flow diagram of this assembly code. This diagram breaks the assembly language program into separate blocks of code. For each

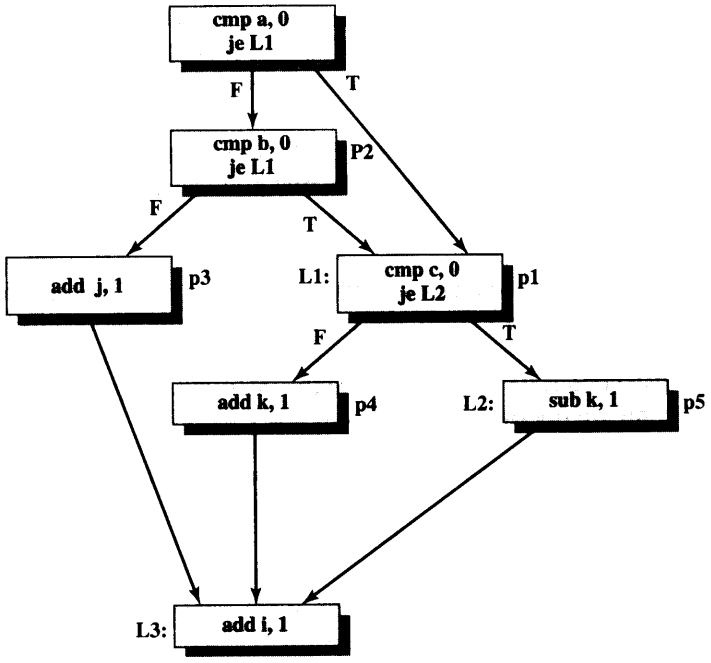


Figure 15.4 Example of Predication

block that executes conditionally, the compiler can assign a predicate. These predicates are indicated in Figure 15.4. Assuming that all of these predicates have been initialized to false, the resulting IA-64 assembly code is as follows:

Predicated Code:

```
(1)        cmp. eq p1, p2 = 0, a ;;
(2) (p2)   cmp. eq p1, p3 = 0, b
(3) (p3)   add  j = 1, j
(4) (p1)   cmp. ne p4, p5 = 0, c
(5) (p4)   add  k = 1, k
(6) (p5)   add  k = -1, k
(7)        add  i = 1, i
```

Instruction (1) compares the contents of symbolic register a with 0; it sets the value of predicate register p1 to 1 (true) and p2 to 0 (false) if the relation is true and will set the value of predicate p1 to 0 and p2 to 1 if the relation is false. Instruction (2) is to be executed only if the predicate p2 is true (i.e., if a is true, which is equivalent to a $\neq$ 0). The processor will fetch, decode, and begin executing this instruction, but only make a decision as to whether to commit the result after it determines whether the value of predicate register p1 is 1 or 0. Note that instruction (2) is a predicate-generating instruction and is itself predicated. This instruction requires three predicate register fields in its format.

Returning to our Pentium program, the first two conditional branches in the Pentium assembly code are translated into two IA-64 predicated compare instructions. If instruction (1) sets p2 to false, the instruction (2) is not executed. After instruction (2) in the IA-64 program, p3 is true only if the outer **if** statement in the source code is true. That is, predicate p3 is true only if the expression (a AND b) is true (i.e., a $\neq$ 0 AND b $\neq$ 0). The **then** part of the outer **if** statement is predicated on p3 for this reason. Instruction (4) of the IA-64 code decides whether the addition or subtraction instruction in the outer **else** part is performed. Finally, the increment of i is performed unconditionally. Looking at the source code and then at the predicated code, we see that only one of instructions (3), (5), and (6) is to be executed. In an ordinary superscalar processor, we would use branch prediction to guess which of the three is to be executed and go down that path. If the processor guesses wrong, the pipeline must be flushed. An IA-64 processor can begin execution of all three of these instructions and, once the values of the predicate registers are known, commit only the results of the valid instruction. Thus, we make use of additional parallel execution units to avoid the delays due to pipeline flushing.

Much of the original research on predicated execution was done at the University of Illinois. Their simulation studies indicate that the use of predication results in a substantial reduction in dynamic branches and branch mispredictions and a substantial performance improvement for processors with multiple parallel pipelines (e.g., [MAHL94], [MAHL95]).

Control Speculation

Another key innovation in IA-64 is control speculation, also known as speculative loading. This enables the processor to load data from memory before the program needs it, to avoid memory latency delays. Also, the processor postpones the reporting of exceptions until it becomes necessary to report the exception. The term *hoist* is used to refer to the movement of a load instruction to a point earlier in the instruction stream.

The minimization of load latencies is crucial to improving performance. Typically, early in a block of code, there are a number of load operations that bring data from memory to registers. Because memory, even augmented with one or two levels of cache, is slow compared with the processor, the delays in obtaining data from memory become a bottleneck. To minimize this, we would like to rearrange the code so that loads are done as early as possible. This can be done with any compiler, up to a point. The problem occurs if we attempt to move a load across a control flow. You cannot unconditionally move the load above a branch because the load may not actually occur. We could move the load conditionally, using predicates, so that the data could be retrieved from memory but not committed to an architectural register until the outcome of the predicate is known; or we can use branch prediction techniques of the type we saw in Chapter 14. The problem with this strategy is that the load can blow up. An exception due to invalid address or a page fault could be generated. If this happens, the processor would have to deal with the exception or fault, causing a delay.

How then, can we move the load above the branch? The solution specified in IA-64 is the control speculation, which separates the load behavior (delivering the value) from the exception behavior (Figure 15.3b). A load instruction in the original program is replaced by two instructions:

- A speculative load (ld.s) executes the memory fetch, performs exception detection, but does not deliver the exception (call the OS routine that handles the exception). This ld.s instruction is hoisted to an appropriate point earlier in the program.
- A checking instruction (chk.s) remains in the place of the original load and delivers exceptions. This chk.s instruction may be predicated so that it will only execute if the predicate is true.

If the ld.s detects an exception, it sets a token bit associated with the target register, known as the *Not a Thing* (NaT) bit. If the corresponding chk.s instruction is executed, and if the NaT bit is set, the chk.s instruction branches to an exception-handling routine.

Let us look at a simple example, taken from [INTE00a, Volume 1]. Here is the original program:

```
(p1)  br some_label          // Cycle 0
      ld8  r1 = [r5] ;;      // Cycle 1
      add  r2 = r1, r3       // Cycle 3
```

The first instruction branches if predicate p1 is true (register p1 has value 1). Note that the branch and load instructions are in the same instruction group, even though the load should not execute if the branch is taken. IA-64 guarantees that if a branch is taken, later instructions, even in the same instruction group, are not executed. IA-64 implementations may use branch prediction to try to improve efficiency but must assure against incorrect results. Finally, note that the add instruction is delayed by at least a clock period (one cycle) due to the memory latency of the load operation.

The compiler can rewrite this code using a control speculative load and a check:

```
      ld8.s r1 = [r5] ;;       // Cycle -2
      // Other instructions
(p1)  br some_label            // Cycle 0
      chk.s r1, recovery       // Cycle 0
      add r2 = r1, r3          // Cycle 0
```

We can't simply move the load instruction above the branch instruction, as is, because the load instruction may cause an exception (e.g., r5 may contain a null pointer). Instead, we convert the load to a speculative load, ld8.s, and then move it. The speculative load doesn't immediately signal an exception when detected; it just records that fact by setting the NaT bit for the target register (in this case, r1). The speculative load now executes unconditionally at least two cycles prior to the branch. The chk.s instruction then checks to see if the NaT bit is set on r1. If not, execution simply falls through to the next instruction. If so, a branch is taken to a

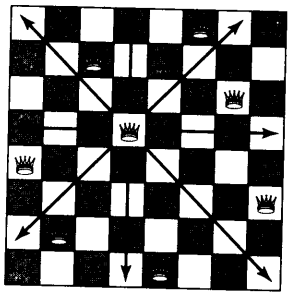
recovery program. Note that the branch, check, and add instructions are all shown as being executed in the same clock cycle. However, the hardware ensures that the results produced by the speculative load do not update the application state (change the contents of r1 and r2) unless two conditions occur: the branch is not taken (p1 = 0) and the check does not detect a deferred exception (r1.NaT = 0).

There is one other important point to note about this example. If there is no exception, then the speculative load is an actual load and takes place prior to the branch that it is supposed to follow. If the branch is taken, then a load has occurred that was not intended by the original program. The program, as written, assumes that r1 is not read on the taken-branch path. If r1 is read on the taken-branch path, then the compiler must use another register to hold the speculative result.

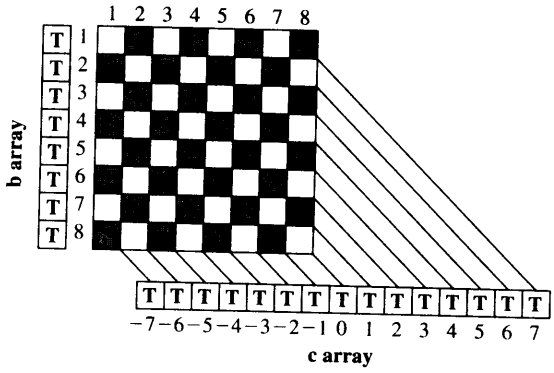
Let us look at a more complex example, used by Intel and HP to benchmark predicated programs and to illustrate the use of speculative loads, known as the Eight Queens Problem. The objective is to arrange eight queens on a chessboard so that no queen threatens any other queen. Figure 15.5a shows one solution. The key line of source code, in an inner loop, is the following:

```
if ((b[j] == true) && (a[i + j] == true) &&
                      (c[i - j] == true))
```

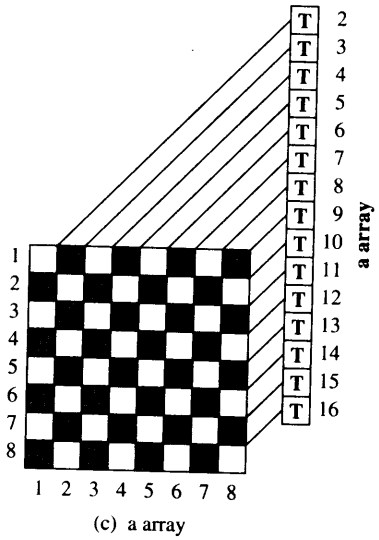
where $1 \leq i, j \leq 8$.



(a) One solution



(b) b and c arrays



(c) a array

Figure 15.5 The Eight Queens Problem

The queen conflict tracking mechanism consists of three Boolean arrays that track queen status for each row and diagonal. TRUE means no queen is on that row or diagonal; FALSE means a queen is already there. Figures 15.5b and c show the mapping of the arrays to the chess board. All array elements are initialized to TRUE. The B array elements 1 through 8 correspond to rows 1 through 8 on the board. A queen in row *n* sets b[*n*] to FALSE. C array elements are numbered from −7 to 7 and correspond to the difference between column and row numbers, which defines the diagonals that go down to the right. A queen at column 1, row 1 sets c[0] to FALSE. A queen at column 1, row 8 sets c[−7] to FALSE. The A array elements are numbered 2-16 and correspond to the sum of the column and row. A queen placed in column 1, row 1 sets a[2] to FALSE. A queen placed in column 3, row 5 sets a[8] to FALSE.

The overall program moves through the columns, placing a queen on each column such that the new queen is not attacked by a queen previously placed on either along a row or one of the two diagonals.

A straightforward Pentium assembly program includes three loads and three branches:

Assembly Code:

```
(1)        mov r2, &b[j] ; transfer contents
                         ; of location
                         ; b[j] to register r2
(2)        cmp r2, 1
(3)        jne L2
(4)        mov r4, &a[i + j]
(5)        cmp r4, 1
(6)        jne L2
(7)        mov r6, &c[i - j]
(8)        cmp r6, 1
(9)        jne L2
(10) L1:   <code for then path>
(11) L2:   <code for else path>
```

In the preceding program, the notation &x symbolizes an immediate address for location x.

Using speculative loads and predicated execution yields the following:

Code with Speculation and Predication:

```
(1)        mov r1 = &b[j] // transfer address of
                          // b[j] to r1
(2)        mov r3 = &a[i + j]
(3)        mov r5 = &c[i - j + 7]
(4)        ld8 r2 = [r1]  // load indirect via r1
(5)        ld8.s r4 = [r3]
(6)        ld8.s r6 = [r5]
(7)        cmp.eq p1, p2 = 1, r2
(8)  (p2)  br L2
```