Slide 2

A superscalar implementation of the 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.

Slide 3

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.

Slide 4

Figure 16.1 compares, in general terms, the scalar and superscalar approaches. In a traditional scalar organization, there is a single pipelined functional unit for integer operations and one for floating-point operations. Parallelism is achieved by enabling multiple instructions to be at different stages of the pipeline at one time. In the superscalar organization, there are multiple functional units, each of which is implemented as a pipeline. Each individual functional unit provides a degree of parallelism by virtue of its pipelined structure. The use of multiple functional units enables the processor to execute streams of instructions in parallel, one stream for each pipeline. It is the responsibility of the hardware, in conjunction with the compiler, to assure that the parallel execution does not violate the intent of the program.

Slide 6

An alternative approach to achieving greater performance is referred to as super-pipelining, 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 16.2 compares the two approaches. The upper part of the diagram illustrates an ordinary pipeline, used as a base for comparison. The base pipeline 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 non-overlapping 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 super- scalar implementations are of course possible. Both the superpipeline and the superscalar implementations depicted in Figure 16.2 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.

Slide 7

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.

Fundamental Limitations to Parallelism

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

Slide 8

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; 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.

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 16.3). 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.

Slide 9
Instruction-level parallelism exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping.

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, i.e., 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.

Slide 10

The term instruction issue refers 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 the instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline.

Superscalar Instruction Issue Policies

- **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 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**
- **Order in which instructions are executed**
- **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.

Slide 11

The simplest instruction issue policy is to issue instructions in the exact order that would be achieved by sequential execution, i.e., in-order issue, and to write results in that same order, i.e., in-order completion. Not even scalar pipelines follow such a simple-minded policy. USE this policy as a baseline for comparing more sophisticated approaches.

Figure 16.4a 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, i.e., two integer arithmetic and one floating-point arithmetic functional units, and having two instances of the write-back pipeline stage.

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 16.4 Superscalar Instruction Issue and Completion Policies

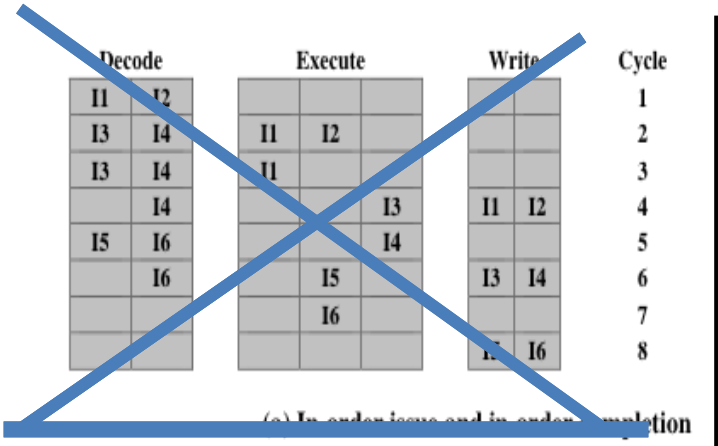
16.4 (a)

In-order issue & in-order completion

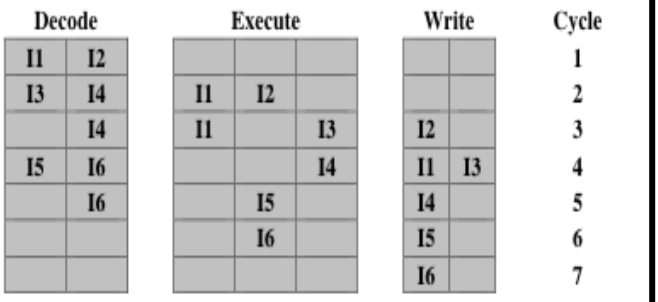
Constraints on a six-instruction code fragment:
I1 requires two cycles to execute.
I3 and I4 conflict for the same functional unit.
I5 depends on the value produced by I4.
I5 and I6 conflict for a functional unit.

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.

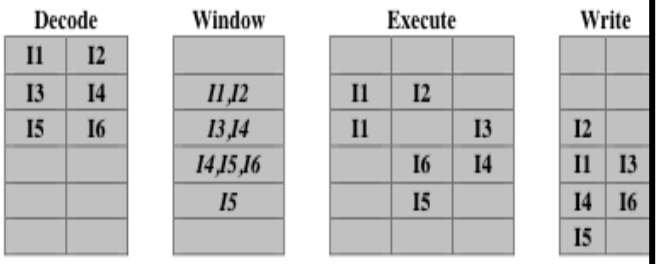
To guarantee in-order completion, when there is a conflict for a functional unit or when a functional unit requires more than one cycle to generate a result, the issuing of instructions temporarily stalls. In this example, the elapsed time from decoding the first instruction to writing the last results is eight cycles.



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



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



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

Figure 16.4 Superscalar Instruction Issue and Completi

16.4 (b) In-Order Issue & Out-of-Order Completion

- **scalar RISC processors use out-of-order completion to improve the performance of instructions that require multiple cycles.**
- **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.**
- **an output dependency, Write After Write, i.e., [WAW], arises.**
  I1: R3 ← R3 op R5
  I2: R4 ← R3 + 1
  I3: R3 ← R5 + 1
  I4: R7 ← R3 op R4

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 16.1.

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 & Out-of-Order Completion

In-Order Issue ➔ processor only decodes instructions up to a dependency or conflict.
No additional instructions are decoded until the problem is resolved. Processor can not look beyond the point of conflict to instructions independent of those in the pipeline.

Decouple the Decode and Execute stages of the pipeline ←➔ Instruction Window Buffer
Fetch ➔ Decode ➔ Window (repeat until window is full)
Execute Unit Available ➔ select an instruction such that
(1) it needs the available unit, &
(2) no conflicts nor dependencies block the instruction
➔ Processor has Look-Head Capability: it identifies independent instructions that can be provided to the execute units.
Instructions are issued from the Instruction Window without regard to their original program order.
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.

I6 can execute before I5 which saves one cycle from 16.4(b)

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 16.4 Superscalar Instruction Issue and Completion Policies

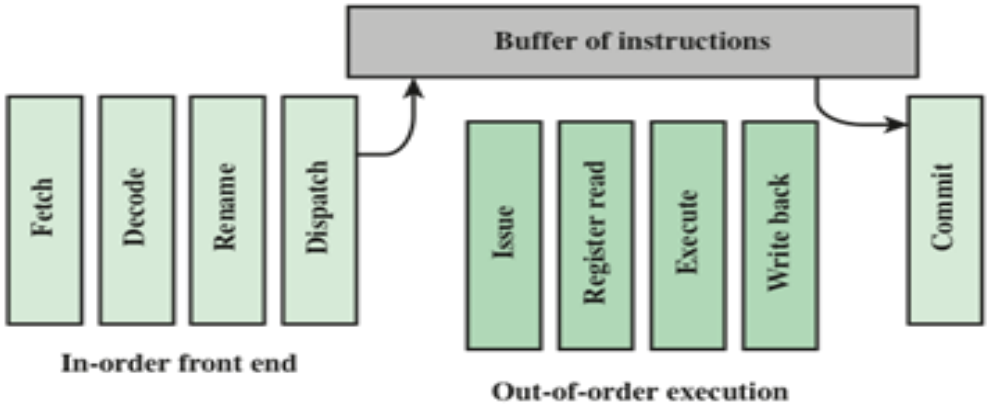


Figure 16.5 Organization for Out-of-Order Issue with Out-of-Order Completion

Write After Read [WAR] Dependency
I1: R3 ← R3 op R5 I2: R4 ← R3 + 1 I3: R3 ← R5 + 1 I4: R7 ← R3 op R4
I3 cannot complete execution before I2 has fetched its operands
Antidepedency == second instruction destroys a value that the first instruction requires.

Slide 12

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. Figure 16.5 suggests this organization.

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 16.4c 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 16.4b, is one cycle.

The instruction window is depicted in Figure 16.4c 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 write after read [WAR] dependency), arises. 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.

Slide 13
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 WAW dependencies and WAR dependencies. These dependencies differ from RAW data dependencies and resource conflicts, which reflect the flow of data through a program and the sequence of execution.

WAW dependencies and WAR dependencies, 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 15), 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.

I1: $R3_b \leftarrow R3_a$ op $R5_a$
I2: $R4_b \leftarrow R3_b + 1$
I3: $R3_c \leftarrow R5_a + 1$
I4: $R7_b \leftarrow R3_c$ op $R4_b$

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).

Slide 14

Three hardware techniques that can be used in a superscalar processor to enhance performance:

1. **duplication of resources,**
2. **out-of-order issue, and**
3. **renaming**

[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 16.6 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 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.

Slide 15

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.

Slide 16

Overview of superscalar execution of programs; see Figure 16.7.
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 stage, 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. 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 executes 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, i.e., committing, or retiring, the instruction.

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.

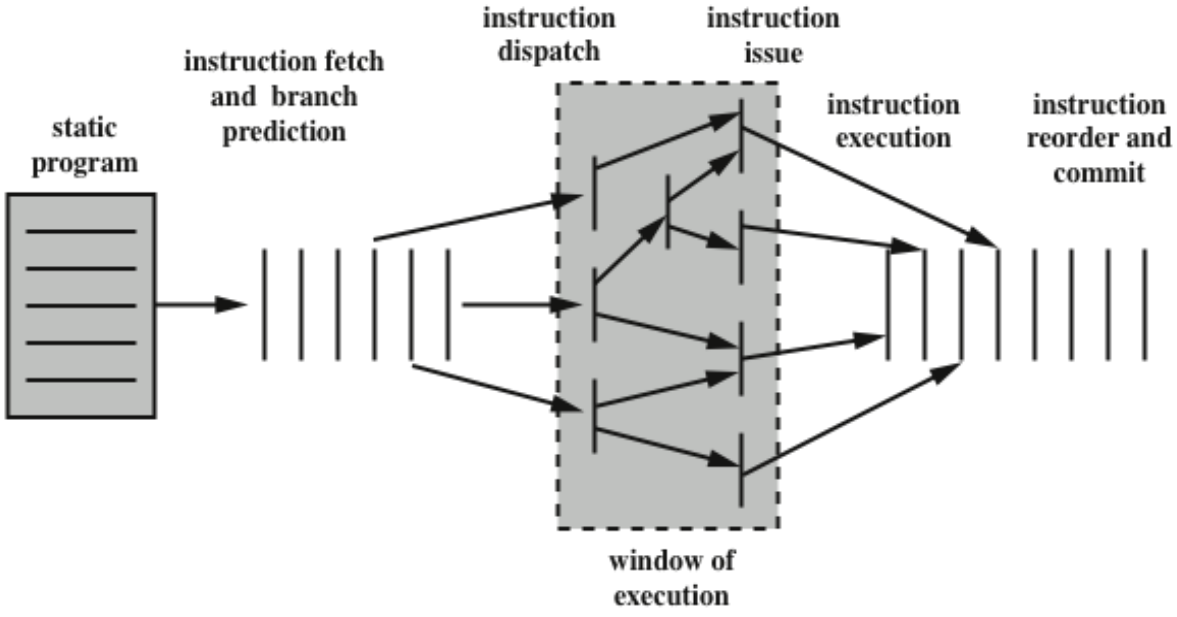


Figure 16.7 Conceptual Depiction of Superscalar Processing

Slide 17
Superscalar System Implementation

Processor Hardware required for the Superscalar System. [SMIT95]

- **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.**

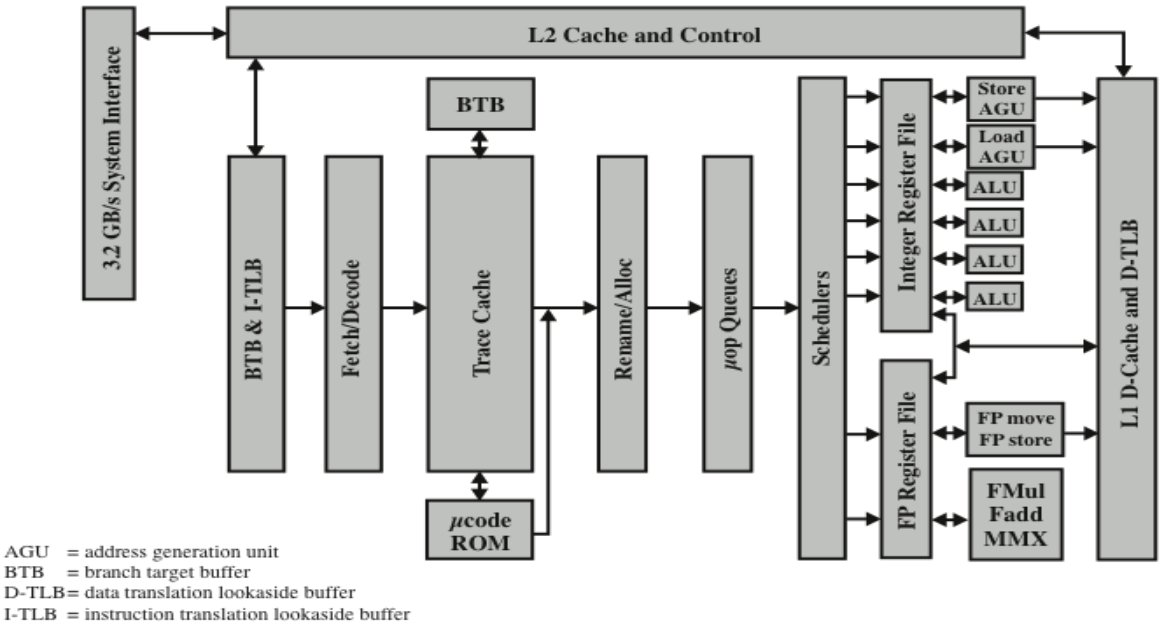


Figure 16.8 Pentium 4 Block Diagram

Slide 18

Although the concept of superscalar design is generally associated with the RISC architecture, the same superscalar principles can be applied to a CISC machine, e.g., The Pentium family.

The Pentium 386 is a traditional CISC non-pipelined machine.

The Pentium 486 introduced the first pipelined x86 processor, reducing the average latency of integer operations from between two and four cycles to one cycle, but still limited to executing a single instruction each cycle, with no superscalar elements.

The Pentium Pro introduced a full-blown superscalar design with out-of-order execution.

Subsequent x86 models have refined and enhanced the superscalar design.

A general block diagram of the Pentium 4 is shown in both Figure 4.18 and Figure 16.8

The operation of the Pentium 4 can be summarized as follows:
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.

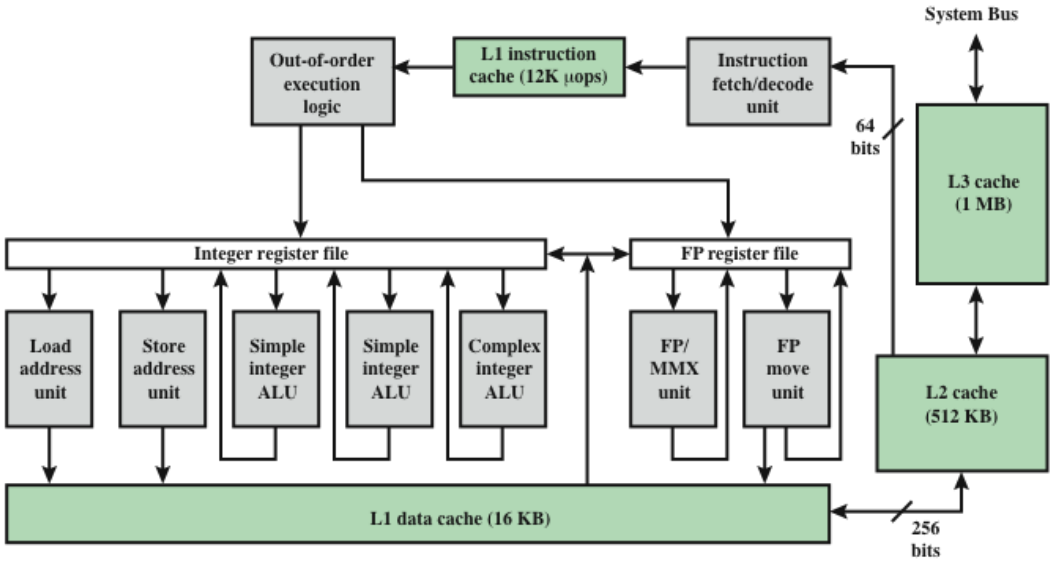
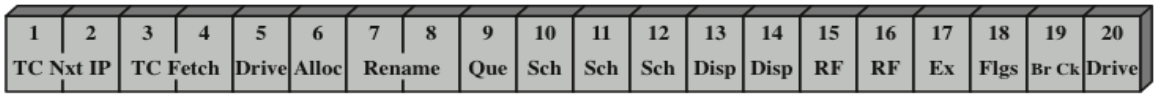


Figure 4.18 Pentium 4 Block Diagram

Slide 19

In effect, the Pentium 4 architecture implements a CISC instruction set architecture on a RISC micro-architecture. The inner RISC micro-ops pass through a pipeline with at least 20 stages (Figure 16.9); in some cases, the micro-op requires multiple execution stages, resulting in an even longer pipeline. This contrasts with the five-stage pipeline (Figure 14.21) used on the earlier Intel x86 processors and on the Pentium.



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 16.9 Pentium 4 Pipeline

Slide 20

The Pentium 4 organization includes an in-order front end (Figure 16.10a) that can be considered outside the scope of the pipeline depicted in Figure 16.9. 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 x86 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 x86 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. The longer micro-op length is required to accommodate the more complex x86 instructions. 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.

The first two pipeline stages (Figure 16.10b) 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.**

The trace cache (Figure 16.10c) 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.

The fifth stage (Figure 16.10d) of the Pentium 4 pipeline delivers decoded instructions from the trace cache to the rename/allocator module.

The allocate stage (Figure 16.10e) 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.

The rename stage (Figure 16.10e) 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).

After resource allocation and register renaming, micro-ops are placed in one of two micro-op queues (Figure 16.10f), 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.

Slide 21

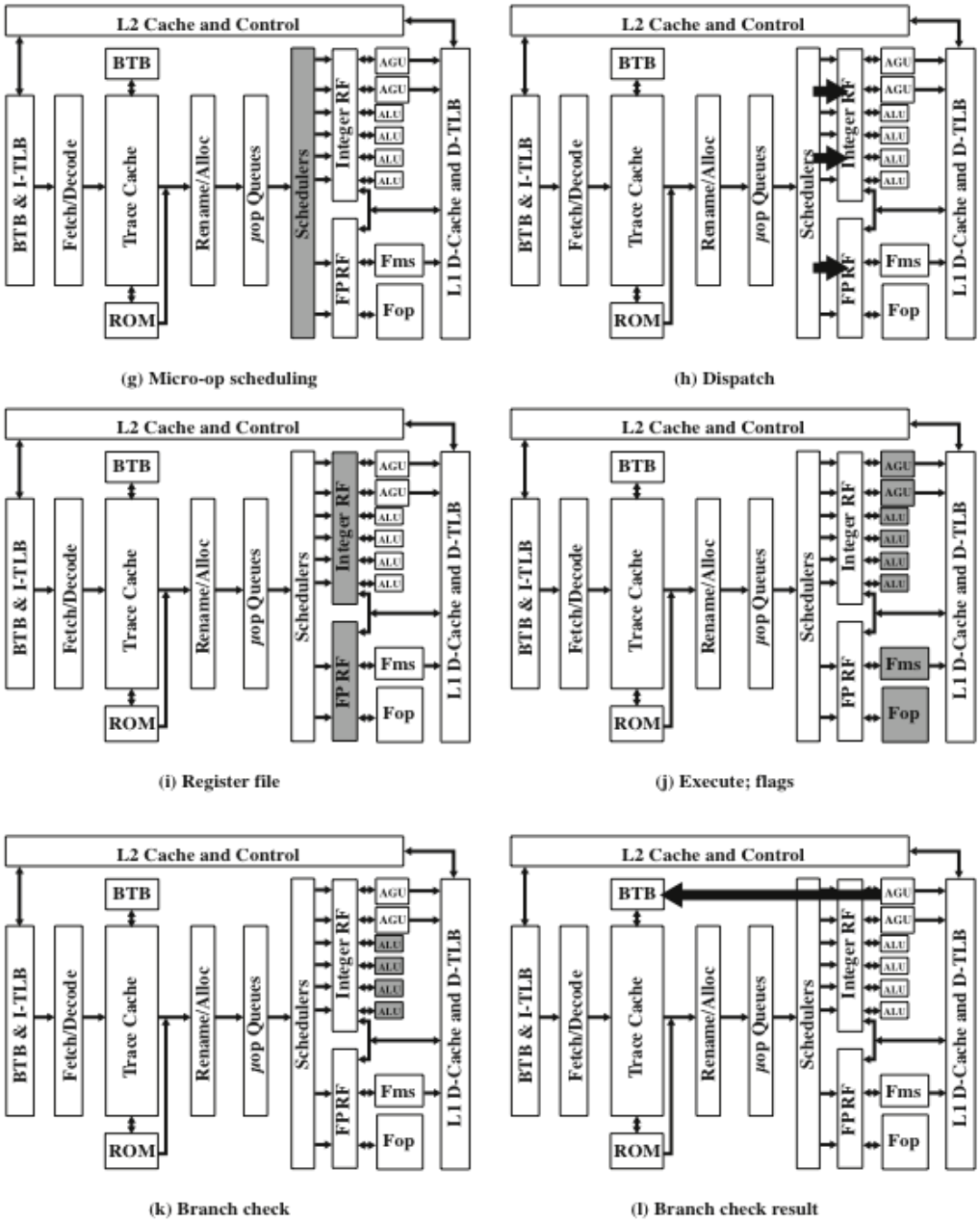


Figure 16.10 Pentium Pipeline Operation (page 2 of 2)

The schedulers (Figure 16.10g) 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 16.10h). 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.

The integer and floating-point register files are the source for pending operations by the execution units (Figure 16.10i). The execution units retrieve values from the register files as well as from the L1 data cache (Figure 16.10j). 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 16.10k). 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 16.10l), which restarts the whole pipeline from the new target address.