4.10 Parallelism via instructions

This section has been set as optional by your instructor.

Be forewarned: this section is a brief overview of fascinating but complex topics. If you want to learn more details, you should consult our more advanced book, *Computer Architecture: A Quantitative Approach*, fifth edition, where the material covered in this section is expanded to almost 200 pages (including appendices)!

Pipelining exploits the potential **parallelism** among instructions. This parallelism is called, naturally enough, *instruction-level parallelism* (ILP). There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Using our laundry analogy and assuming that the washer cycle was longer than the others were, we could divide our washer into three machines that perform the wash, rinse, and spin steps of a traditional washer. We would then move from a four-stage to a six-stage pipeline. To get the full speed-up, we need to rebalance the remaining steps so they are the same length, in processors or in laundry. The amount of parallelism being exploited is higher, since there are more operations being overlapped. Performance is potentially greater since the clock cycle can be shorter.

Instruction-level parallelism: The parallelism among instructions.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is *multiple issue*. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

Multiple issue: A scheme whereby multiple instructions are launched in one clock cycle.

Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, the CPI to be less than 1. As mentioned in COD Chapter 1 (Computer Abstractions and Technology), it is sometimes useful to flip the metric and use *IPC*, or *instructions per clock cycle*. Hence, a 3-GHz four-way multiple-issue microprocessor can execute a peak rate of 12 billion instructions per second and have a best-case CPI of 0.33, or an IPC of 3. Assuming a five-stage pipeline, such a processor would have 15 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle. Even moderate designs will aim at a peak IPC of 2. There are typically, however, many constraints on what types of instructions may be executed simultaneously, and what happens when dependences arise.

There are two main ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called *static multiple issue* and *dynamic multiple issue*. As we will see, both approaches have other, more commonly used names, which may be less precise or more restrictive.

Static multiple issue: An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

Dynamic multiple issue: An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

Two primary and distinct responsibilities must be dealt with in a multiple-issue pipeline:

1. Packaging instructions into *issue slots*: how does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.
2. Dealing with data and control hazards: in static issue processors, the compiler handles some or all of the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

Although we describe these as distinct approaches, in reality, one approach often borrows techniques from the other, and neither approach can claim to be perfectly pure.

Issue slots: The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.

PARTICIPATION ACTIVITY 4.10.1: Pipeline parallelism.

multiple issue | static multiple issue | issue slots | single issue | dynamic multiple issue | ILP

	When one instruction is launched per clock cycle.
	The parallelism between instructions.
	When multiple instructions are launched per clock cycle.
	A multiple issue implementation where decisions are made during execution by the processor.

The positions available to issue instructions in a given clock cycle.

A multiple issue implementation where decisions are made by the compiler before execution.

Reset

The concept of speculation

One of the most important methods for finding and exploiting more ILP is speculation. Based on the great idea of **prediction**, *speculation* is an approach that allows the compiler or the processor to "guess" about the properties of an instruction, to enable execution to begin for other instructions that may depend on the speculated instruction. For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier.

Speculation: An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

Another example is that we might speculate that a store that precedes a load does not refer to the same address, which would allow the load to be executed before the store. The difficulty with speculation is that it may be wrong. So, any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively. The implementation of this back-out capability adds complexity.

Speculation may be done in the compiler or by the hardware. For example, the compiler can use speculation to reorder instructions, moving an instruction across a branch or a load across a store. The processor hardware can perform the same transformation at runtime using techniques we discuss later in this section.

The recovery mechanisms used for incorrect speculation are rather different. In the case of speculation in software, the compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to use when the speculation is wrong. In hardware speculation, the processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation is correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation is incorrect, the hardware flushes the buffers and re-executes the correct instruction sequence. Misspeculation typically requires the pipeline to be flushed, or at least stalled, and thus further reduces performance.

Speculation introduces one other possible problem: speculating on certain instructions may introduce exceptions that were formerly not present. For example, suppose a load instruction is moved in a speculative manner, but the address it uses is not within bounds when the speculation is incorrect. The result would be that an exception that should not have occurred would occur. The problem is complicated by the fact that if the load instruction were not speculative, then the exception must occur! In compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point, the exception is raised, and normal exception handling proceeds.

Since speculation can improve performance when done properly and decrease performance when done carelessly, significant effort goes into deciding when it is appropriate to speculate. Later in this section, we will examine both static and dynamic techniques for speculation.

Static multiple issue

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions issued in a given clock cycle, which is called an *issue packet*, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: *Very Long Instruction Word (VLIW)*.

Issue packet: The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Very Long Instruction Word (***VLIW***): A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards. Let's look at a simple static issue version of an LEGv8 processor, before we describe the use of these techniques in more aggressive processors.

An example: Static multiple issue with the LEGv8 ISA

To give a flavor of static multiple issue, we consider a simple two-issue LEGv8 processor, where one of the instructions can be an integer ALU operation or branch and the other can be a load or store. Such a design is like that used in some embedded processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue. Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a `nop`. Thus, the instructions always issue in pairs, possibly with a `nop` in one slot. The animation below shows how the instructions look as they go into the pipeline in pairs.

The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

PARTICIPATION ACTIVITY 4.10.2: Static two-issue pipeline in operation (COD Figure 4.66).

Start 2x speed

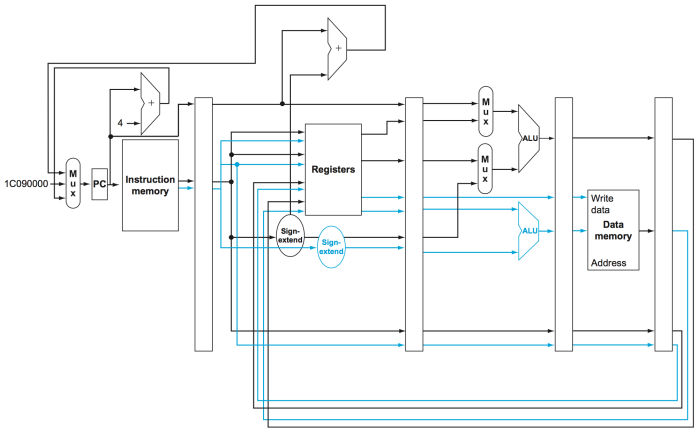
Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code, and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls. In others, the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction packet. Even so, a hazard generally forces the entire issue packet containing the dependent instruction to stall. Whether the software must handle all hazards or only try to reduce the fraction of hazards between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced. We will assume the second approach for this example.

To issue an ALU and a data transfer operation in parallel, the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see the figure below). In one clock cycle, we may need to read two registers for the ALU operation and two more for a store, and also one write port for an ALU operation and one write port for a load. Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

Figure 4.10.1: A static two-issue datapath (COD Figure 4.67).

The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.



Clearly, this two-issue processor can improve performance by up to a factor of two! Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards. For example, in our simple five-stage pipeline, loads have a *use latency* of one clock cycle, which prevents one instruction from using the result without stalling. In the two-issue, five-stage pipeline the result of a load instruction cannot be used on the next clock cycle. This means that the next two instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline now have a one-instruction use latency, since the results cannot be used in the paired load or store. To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler take on this role.

Use latency: Number of clock cycles between a load instruction and an instruction that can use the result of the load without stalling the pipeline.

Example 4.10.1: Simple multiple-issue code scheduling.

How would this loop be scheduled on a static two-issue pipeline for LEGv8?

```
Loop: LDUR X0, [X20,#0]  // X0=array element
      ADD  X0,X0,X21     // add scalar in X21
      STUR X0, [X20,#0]  // store result
      SUBI X20,X20,#8     // decrement pointer
      CMP  X20,X22       // compare to loop limit
      BGT  Loop          // branch if X20 > X22
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

Answer

The first three instructions have data dependences, as do the next two. The figure below shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes five clocks per loop iteration; at five clocks to execute six instructions, we get the disappointing CPI of 0.83 versus the best case of 0.5, or an IPC of 1.2 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

Figure 4.10.2: The scheduled code as it would look on a two-issue LEGv8 pipeline (COD Figure 4.68).

The empty slots are no-ops.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		LDUR X0,[X20,#0]	1
	SUBI X20, X20, #8		2
	ADD X0, X0, X21		3
	CMP X20, X22		4
	BGT Loop	STUR X0,[X20,#8]	5

An important compiler technique to get more performance from loops is *loop unrolling*, where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations.

Loop unrolling: A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

Example 4.10.2: Loop unrolling for multiple-issue pipelines.

See how well loop unrolling and scheduling work in the example above. For simplicity, assume that the loop index is a multiple of four.

Answer

To schedule the loop without any delays, it turns out that we need to make four copies of the loop body. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of `LDUR`, `ADD`, and `STUR`, plus one `SUBI`, one `CMP`, and one `CBZ`. the figure below shows the unrolled and scheduled code.

During the unrolling process, the compiler introduced additional registers `(X1, X2, X3)`. The goal of this process, called *register renaming*, is to eliminate dependences that are not true data dependences, but could either lead to potential hazards or prevent the compiler from flexibly scheduling the code. Consider how the unrolled code would look using only `X0`. There would be repeated instances of `LDUR X0, [X20,#0]`, `ADD X0, X0, X21` followed by `STUR X0, [X20,#8]`, but these sequences, despite using `X0`, are actually completely independent—no data values flow between one set of these instructions and the next set. This case is what is called an *antidependence* or *name dependence*, which is an ordering forced purely by the reuse of a name, rather than a real data dependence that is also called a true dependence.

Renaming the registers during the unrolling process allows the compiler to move these independent instructions subsequently to better schedule the code. The renaming process eliminates the name dependences, while preserving the true dependences.

Notice now that 14 of the 15 instructions in the loop execute as pairs. It takes eight clocks for four loop iterations, which yields an IPC of 15/8 = 1.88. Loop unrolling and scheduling more than doubled performance—8 versus 20 clock cycles for 4 iterations—partly from reducing the loop control instructions and partly from dual issue execution. The cost of this performance improvement is using four temporary registers rather than one, as well as more than doubling the code size.

Register renaming: The renaming of registers by the compiler or hardware to remove antidependences.

Antidependence: Also called ***name dependence***. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

Figure 4.10.3: The unrolled and scheduled code of the figure above as it would look on a static two-issue LEGv8 pipeline (COD Figure 4.69).

The empty slots are no-ops. Since the first instruction in the loop decrements `X20` by 32, the addresses loaded are the original value of `X20`, then that address minus 8, minus 16, and minus 24.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	SUBI X20, X20,#32	LDUR X0, [X20,#0]	1
		LDUR X1, [X20,#24]	2
	ADD X0, X0, X21	LDUR X2, [X20,#16]	3
	ADD X1, X1, X21	LDUR X3, [X20,#8]	4
	ADD X2, X2, X21	STUR X0, [X20,#32]	5
	ADD X3, X3, X21	STUR X1, [X20,#24]	6
	CMP X20, X22	STUR X2, [X20,#16]	7
	BGT Loop	STUR X3, [X20,#8]	8

PARTICIPATION ACTIVITY 4.10.3: Static multiple issue processors.

1) A very long instruction word (VLIW) architecture groups multiple operations together and then launches them like a single instruction.
   - ○ True
   - ○ False

2) In all static multiple issue processors, the compiler is responsible for removing all data hazards and avoiding all dependences.
   - ○ True
   - ○ False

3) If the use latency for a load instruction is one clock cycle, then an instruction can use the result from the load on the next clock cycle.
   - ○ True
   - ○ False

4) Both loop unrolling and register renaming allow a processor to better schedule instructions and improve performance.
   - ○ True
   - ○ False

5) Loop unrolling and register renaming can lead to an increase in code and the need for more resources.
   - ○ True
   - ○ False

Dynamic multiple-issue processors

Dynamic multiple-issue processors are also known as *superscalar* processors, or simply superscalars. In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

Superscalar: An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

Many superscalars extend the basic framework of dynamic issue decisions to include *dynamic pipeline scheduling*. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

```
LDUR  X0,  [X21,#20]
ADD   X1,  X0, X2
SUB   X23, X23, X3
ANDI  X5,  X23, 20
```

Dynamic pipeline scheduling: Hardware support for reordering the order of instruction execution to avoid stalls.

Even though the `SUB` instruction is ready to execute, it must wait for the `LDUR` and `ADD` to complete first, which might take many clock cycles if memory is slow. (COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) explains cache misses, the reason that memory accesses are sometimes very slow.) Dynamic **pipeline** scheduling allows such hazards to be avoided either fully or partially.

Dynamic pipeline scheduling

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units (a dozen or more in high-end designs in 2015), and a *commit unit*. The figure below shows the model. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called *reservation stations*, which hold the operands and the operation. (In the next section, we will discuss an alternative to reservation stations used by many recent processors.) As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the *reorder buffer*, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

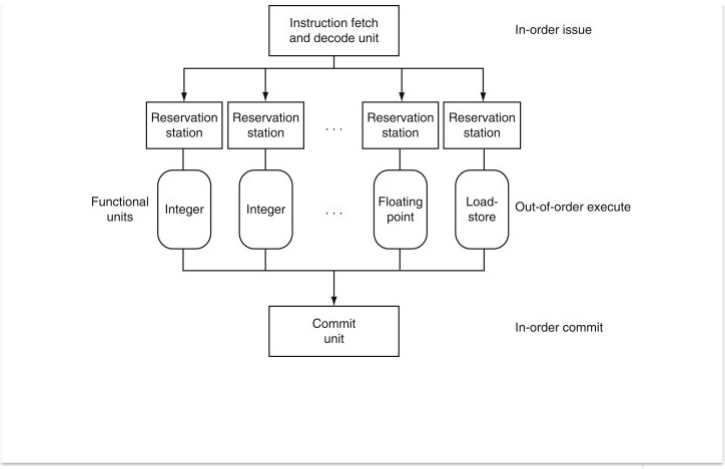
Commit unit: The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

Reservation station: A buffer within a functional unit that holds the operands and the operation.

Reorder buffer: The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

Figure 4.10.4: The three primary units of a dynamically scheduled pipeline (COD Figure 4.70).

The final step of updating the state is also called retirement or graduation.



The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming, just like that used by the compiler in our earlier loop-unrolling example. To see how this conceptually works, consider the following steps:

1. When an instruction issues, it is copied to a reservation station for the appropriate functional unit. Any operands that are available in the register file or reorder buffer are also immediately copied into the reservation station. The instruction is buffered in the reservation station until all the operands and the functional unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

Conceptually, you can think of a dynamically scheduled pipeline as analyzing the data flow structure of a program. The processor then executes the instructions in some order that preserves the data flow order of the program. This style of execution is called an *out-of-order execution*, since the instructions can be executed in a different order than the instructions were fetched.

Out-of-order execution: A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order. This conservative mode is called *in-order commit*. Hence, if an exception occurs, the computer can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception. Although the front end (fetch and issue) and the back end (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the data they need are available. Today, all dynamically scheduled pipelines use in-order commit.

In-order commit: A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path. Because the instructions are committed in order, we know whether the branch was correctly predicted before any instructions from the predicted path are committed. A speculative, dynamically scheduled pipeline can also support speculation on load addresses, allowing load-store reordering, and using the commit unit to avoid incorrect speculation. In the next section, we will look at the use of dynamic scheduling with speculation in the Intel Core i7 design.

PARTICIPATION ACTIVITY 4.10.4: Dynamic multiple issue processors.

1) In dynamic pipeline scheduling, a reservation station holds an instruction's result until it is safe to place the result in a register file or in memory.
   - ○ True
   - ○ False

2) Dynamic multiple issue processors can execute instructions in a different order than they were fetched.
   - ○ True
   - ○ False

3) Superscalar is another term for dynamic pipeline scheduling.
   - ○ True
   - ○ False

4) Results are committed in the order that those results are executed.
   - ○ True
   - ○ False

5) Dynamic pipeline scheduling uses the combination of reservation stations and

the reorder buffer to perform register renaming.

- ○ True
- ○ False

Understanding program performance

Given that compilers can also schedule code around data dependences, you might ask why a superscalar processor would use dynamic scheduling. There are three major reasons. First, not all stalls are predictable. In particular, cache misses (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)) in the **memory hierarchy** cause unpredictable stalls. Dynamic scheduling allows the processor to hide some of those stalls by continuing to execute instructions while waiting for the stall to end.

Second, if the processor speculates on branch outcomes using dynamic branch **prediction**, it cannot know the exact order of instructions at compile time, since it depends on the predicted and actual behavior of branches. Incorporating dynamic speculation to exploit more *instruction-level parallelism* (ILP) without incorporating dynamic scheduling would significantly restrict the benefits of speculation.

Third, as the pipeline latency and issue width change from one implementation to another, the best way to compile a code sequence also changes. For example, how to schedule a sequence of dependent instructions is affected by both issue width and latency. The pipeline structure affects both the number of times a loop must be unrolled to avoid stalls as well as the process of compiler-based register renaming. Dynamic scheduling allows the hardware to hide most of these details. Thus, users and software distributors do not need to worry about having multiple versions of a program for different implementations of the same instruction set. Similarly, old legacy code will get much of the benefit of a new implementation without the need for recompilation.

The Big Picture

Both **pipelining** and multiple-issue execution increase peak instruction throughput and attempt to exploit instruction-level **parallelism** (ILP). Data and control dependences in programs, however, offer an upper limit on sustained performance because the processor must sometimes wait for a dependence to be resolved. Software-centric approaches to exploiting ILP rely on the ability of the compiler to find and reduce the effects of such dependences, while hardware-centric approaches rely on extensions to the pipeline and issue mechanisms. Speculation, performed by the compiler or the hardware, can increase the amount of ILP that can be exploited via **prediction**, although care must be taken since speculating incorrectly is likely to reduce performance.

Hardware/Software Interface

Modern, high-performance microprocessors are capable of issuing several instructions per clock; unfortunately, sustaining that issue rate is very difficult. For example, despite the existence of processors with four to six issues per clock, very few applications can sustain more than two instructions per clock. There are two primary reasons for this.

First, within the pipeline, the major performance bottlenecks arise from dependences that cannot be alleviated, thus reducing the parallelism among instructions and the sustained issue rate. Although little can be done about true data dependences, often the compiler or hardware does not know precisely whether a dependence exists or not, and so must conservatively assume the dependence exists. For example, code that makes use of pointers, particularly in ways that may lead to aliasing, will lead to more implied potential dependences. In contrast, the greater regularity of array accesses often allows a compiler to deduce that no dependences exist. Similarly, branches that cannot be accurately predicted whether at runtime or compile time will limit the ability to exploit ILP. Often, additional ILP is available, but the ability of the compiler or the hardware to find ILP that may be widely separated (sometimes by the execution of thousands of instructions) is limited.

Second, losses in the **memory hierarchy** (the topic of COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)) also limit the ability to keep the pipeline full. Some memory system stalls can be hidden, but limited amounts of ILP also limit the extent to which such stalls can be hidden.

Energy efficiency and advanced pipelining

The downside to the increasing exploitation of instruction-level parallelism via dynamic multiple issue and speculation is potential energy inefficiency. Each innovation was able to turn more transistors into performance, but they often did so very inefficiently. Now that we have collided with the power wall, we are seeing designs with multiple processors per chip where the processors are not as deeply pipelined or as aggressively speculative as its predecessors.

The belief is that while the simpler processors are not as fast as their sophisticated brethren, they deliver better performance per Joule, so that they can deliver more performance per chip when designs are constrained more by energy than they are by the number of transistors.

The figure below shows the number of pipeline stages, the issue width, speculation level, clock rate, cores per chip, and power of several past and recent Intel microprocessors. Note the drop in pipeline stages and power as companies switch to multicore designs.

Figure 4.10.5: Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power (COD Figure 4.71).

The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Elaboration

A commit unit controls updates to the register file and memory. Some dynamically scheduled processors update the register file immediately during execution, using extra registers to implement the renaming function and preserving the older copy of a register until the instruction updating the register is no longer speculative. Other processors buffer the result, which, as mentioned above, is typically in a structure called a reorder buffer, and the actual update to the register file occurs later as part of the commit. Stores to memory must be buffered until commit time either in a store buffer (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)) or in the reorder buffer. The commit unit allows the store to write to memory from the buffer when the buffer has a valid address and valid data, and when the store is no longer dependent on predicted branches.

Elaboration

Memory accesses benefit from nonblocking caches, which continue servicing cache accesses during a cache miss (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)). Out-of-order execution processors need the cache to allow instructions to execute during a miss.

PARTICIPATION ACTIVITY 4.10.5: Check Yourself: Instruction Level Parallelism.

State whether the following techniques or components are associated primarily with a software- or hardware-based approach to exploiting ILP.

1) Branch prediction
   - ○ Software
   - ○ Hardware
   - ○ Both software & hardware

2) Multiple issue
   - ○ Software
   - ○ Hardware
   - ○ Both software & hardware

3) VLIW
   - ○ Software
   - ○ Hardware
   - ○ Both software & hardware

4) Superscalar
   - ○ Software
   - ○ Hardware
   - ○

Both software & hardware

5) Dynamic scheduling
- Software
- Hardware
- Both software & hardware

6) Out of order execution
- Software
- Hardware
- Both software & hardware

7) Speculation
- Software
- Hardware
- Both software & hardware

8) Reorder buffer
- Software
- Hardware
- Both software & hardware

9) Register renaming
- Software
- Hardware
- Both software & hardware

Provide feedback on this section