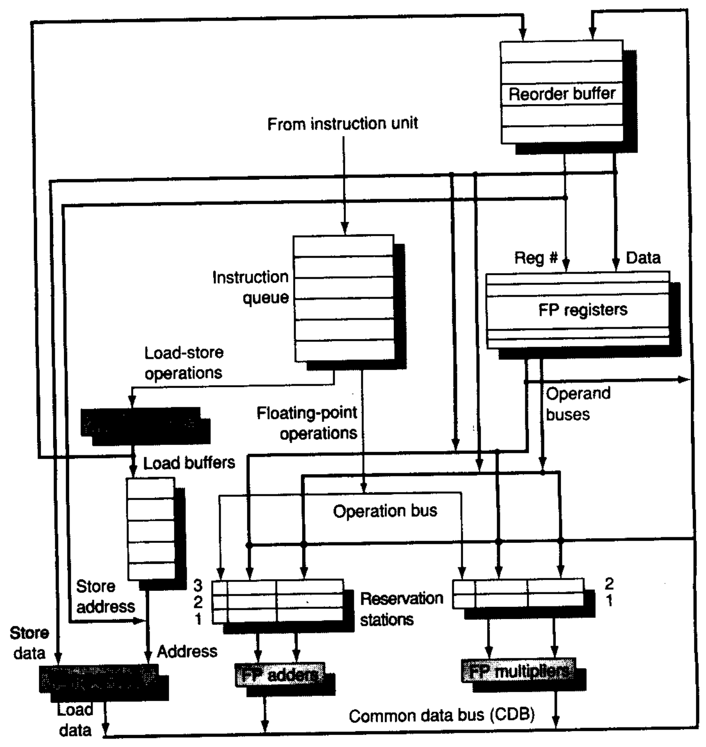


Figure 2.14 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 2.9 on page 94, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB entry allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries.

2. *Execute*—If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation. Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation.

3. *Write result*—When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity we assume that this occurs during the Write Results stage of a store; we discuss relaxing this requirement later.
4. *Commit*—This is the final stage of completing an instruction, after which only its result remains. (Some processors call this commit phase "completion" or "graduation.") There are three different sequences of actions at commit depending on whether the committing instruction is a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB. Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished.

Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free. Now, let's examine how this scheme would work with the same example we used for Tomasulo's algorithm.

Example Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the code segment below, the same one we used to generate Figure 2.11, show what the status tables look like when the MUL.D is ready to go to commit.

```
L.D       F6,32(R2)
L.D       F2,44(R3)
MUL.D     F0,F2,F4
SUB.D     F8,F6,F2
DIV.D     F10,F0,F6
ADD.D     F6,F8,F2
```

Answer Figure 2.15 shows the result in the three tables. Notice that although the SUB.D instruction has completed execution, it does not commit until the MUL.D commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo's algorithm (see page 97 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB entry that is the destination for the result produced by this reservation station entry.

The above example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling. Compare the content of Figure 2.15 with that of Figure 2.11 on page 100, which shows the same code sequence in operation on a processor with Tomasulo's algorithm. The key difference is that, in the example above, no instruction after the earliest uncompleted instruction (MUL.D above) is allowed to complete. In contrast, in Figure 2.11 the SUB.D and ADD.D instructions have also completed.

One implication of this difference is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model. For example, if the MUL.D instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions from the ROB. Because instruction commit happens in order, this yields a precise exception.

By contrast, in the example using Tomasulo's algorithm, the SUB.D and ADD.D instructions could both complete before the MUL.D raised the exception. The result is that the registers F8 and F6 (destinations of the SUB.D and ADD.D instructions) could be overwritten, and the interrupt would be imprecise.

Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors, since the program will likely terminate; see Appendix G for further discussion of this topic. Other types of exceptions, such as page faults, are much more difficult to accommodate if they are imprecise, since the program must transparently resume execution after handling such an exception.

The use of a ROB with in-order instruction commit provides precise exceptions, in addition to supporting speculative execution, as the next example shows.

Example Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 2.13 in execution:

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop      ;branches if R1≠R2
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in

	Reorder buffer					
Entry	**Busy**	**Instruction**		**State**	**Destination**	**Value**
1	no	`L.D`	`F6,32(R2)`	Commit	F6	Mem[34 + Regs[R2]]
2	no	`L.D`	`F2,44(R3)`	Commit	F2	Mem[45 + Regs[R3]]
3	yes	`MUL.D`	`F0,F2,F4`	Write result	F0	#2 × Regs[F4]
4	yes	`SUB.D`	`F8,F2,F6`	Write result	F8	#2 – #1
5	yes	`DIV.D`	`F10,F0,F6`	Execute	F10	
6	yes	`ADD.D`	`F6,F8,F2`	Write result	F6	#4 + #2

	Reservation stations							
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**Dest**	**A**
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	`MUL.D`	Mem[45 + Regs[R3]]	Regs[F4]			#3	
Mult2	yes	`DIV.D`		Mem[34 + Regs[R2]]	#3		#5	

	FP register status									
Field	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F10**
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	. . .	yes	yes

Figure 2.15 At the time the `MUL.D` is ready to commit, only the two `L.D` instructions have committed, although several others have completed execution. The `MUL.D` is at the head of the ROB, and the two `L.D` instructions are there only to ease understanding. The `SUB.D` and `ADD.D` instructions will not commit until the `MUL.D` instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The `DIV.D` is in execution, but has not completed solely due to its longer latency than `MUL.D`. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed, but are shown for informational purposes. We do not show the entries for the load-store queue, but these entries are kept in order.

the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer Figure 2.16 shows the result in two tables.

	Reorder buffer				
Entry	**Busy**	**Instruction**	**State**	**Destination**	**Value**
1	no	`L.D F0,0(R1)`	Commit	F0	Mem[0 + Regs[R1]]
2	no	`MUL.D F4,F0,F2`	Commit	F4	#1 × Regs[F2]
3	yes	`S.D F4,0(R1)`	Write result	0 + Regs[R1]	#2
4	yes	`DADDIU R1,R1,#-8`	Write result	R1	Regs[R1] – 8
5	yes	`BNE R1,R2,Loop`	Write result		
6	yes	`L.D F0,0(R1)`	Write result	F0	Mem[#4]
7	yes	`MUL.D F4,F0,F2`	Write result	F4	#6 × Regs[F2]
8	yes	`S.D F4,0(R1)`	Write result	0 + #4	#7
9	yes	`DADDIU R1,R1,#-8`	Write result	R1	#4 – 8
10	yes	`BNE R1,R2,Loop`	Write result		

	FP register status								
Field	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	...	no

Figure 2.16 Only the `L.D` and `MUL.D` instructions have committed, although all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that the branch `BNE` is not taken the first time in Figure 2.16. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.

In practice, processors that speculate try to recover as early as possible after a branch is mispredicted. This recovery can be done by clearing the ROB for all entries that appear after the mispredicted branch, allowing those that are before the branch in the ROB to continue, and restarting the fetch at the correct branch successor. In speculative processors, performance is more sensitive to the branch prediction, since the impact of a misprediction will be higher. Thus, all the aspects of handling branches—prediction accuracy, latency of misprediction detection, and misprediction recovery time—increase in importance.

Exceptions are handled by not recognizing the exception until it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the ROB. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the ROB is cleared. If the instruction reaches the head of the ROB, then we know it is no longer speculative and the exception should really be taken. We can also try to handle exceptions as soon as they arise and all earlier branches are resolved, but this is more challenging in the case of exceptions than for branch mispredict and, because it occurs less frequently, not as critical.

Figure 2.17 shows the steps of execution for an instruction, as well as the conditions that must be satisfied to proceed to the step and the actions taken. We show the case where mispredicted branches are not resolved until commit. Although speculation seems like a simple addition to dynamic scheduling, a comparison of Figure 2.17 with the comparable figure for Tomasulo's algorithm in Figure 2.12 shows that speculation adds significant complications to the control. In addition, remember that branch mispredictions are somewhat more complex as well.

There is an important difference in how stores are handled in a speculative processor versus in Tomasulo's algorithm. In Tomasulo's algorithm, a store can update memory when it reaches Write Result (which ensures that the effective address has been calculated) and the data value to store is available. In a speculative processor, a store updates memory only when it reaches the head of the ROB. This difference ensures that memory is not updated until an instruction is no longer speculative.

Figure 2.17 has one significant simplification for stores, which is unneeded in practice. Figure 2.17 requires stores to wait in the Write Result stage for the register source operand whose value is to be stored; the value is then moved from the Vk field of the store's reservation station to the Value field of the store's ROB entry. In reality, however, the value to be stored need not arrive until *just before* the store commits and can be placed directly into the store's ROB entry by the sourcing instruction. This is accomplished by having the hardware track when the source value to be stored is available in the store's ROB entry and searching the ROB on every instruction completion to look for dependent stores.

This addition is not complicated, but adding it has two effects: We would need to add a field to the ROB, and Figure 2.17, which is already in a small font, would be even longer! Although Figure 2.17 makes this simplification, in our examples, we will allow the store to pass through the Write Result stage and simply wait for the value to be ready when it commits.

Like Tomasulo's algorithm, we must avoid hazards through memory. WAW and WAR hazards through memory are eliminated with speculation because the actual updating of memory occurs in order, when a store is at the head of the ROB, and hence, no earlier loads or stores can still be pending. RAW hazards through memory are maintained by two restrictions:

1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and

Status	Wait until	Action or bookkeeping

Issue all instructions	Reservation station (r) and ROB (b) both available	if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/
		  {h ← RegisterStat[rs].Reorder;
		  if (ROB[h].Ready)/* Instr completed already */
		    {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;}
		  else {RS[r].Qj ← h;} /* wait for instruction */
		} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};
		RS[r].Busy ← yes; RS[r].Dest ← b;
		ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;

FP operations and stores		if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/
		  {h ← RegisterStat[rt].Reorder;
		  if (ROB[h].Ready)/* Instr completed already */
		    {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;}
		  else {RS[r].Qk ← h;} /* wait for instruction */
		} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};

FP operations		RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes;
		ROB[b].Dest ← rd;

Loads		RS[r].A ← imm; RegisterStat[rt].Reorder ← b;
		RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;

Stores		RS[r].A ← imm;
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	ROB[h].Address ← RS[r].Vj + RS[r].A;

Write result all but store	Execution done at *r* and CDB available	b ← RS[r].Dest; RS[r].Busy ← no;
		∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0});
		∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0});
		ROB[b].Value ← result; ROB[b].Ready ← yes;

Store	Execution done at *r* and (RS[r].Qk == 0)	ROB[h].Value ← RS[r].Vk;

Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	d ← ROB[h].Dest; /* register dest, if exists */
		if (ROB[h].Instruction==Branch)
		  {if (branch is mispredicted)
		    {clear ROB[h], RegisterStat; fetch branch dest;};}
		else if (ROB[h].Instruction==Store)
		    {Mem[ROB[h].Destination] ← ROB[h].Value;}
		else /* put the result in the register destination */
		  {Regs[d] ← ROB[h].Value;};
		ROB[h].Busy ← no; /* free up ROB entry */
		/* free up dest register if no one else writing it */
		if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};

Figure 2.17 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the sources, r is the reservation station allocated, b is the assigned ROB entry, and h is the head entry of the ROB. RS is the reservation station data structure. The value returned by a reservation station is called the result. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure.

2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.

Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data. Some speculative processors will actually bypass the value from the store to the load directly, when such a RAW hazard occurs. Another approach is to predict potential collisions using a form of value prediction; we consider this in Section 2.9.

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units, as we will see in the "Putting It All Together" section. Indeed, speculation may be more useful in integer programs, since such programs tend to have code where the branch behavior is less predictable. Additionally, these techniques can be extended to work in a multiple-issue processor by allowing multiple instructions to issue and commit every clock. In fact, speculation is probably most interesting in such processors, since less ambitious techniques can probably exploit sufficient ILP within basic blocks when assisted by a compiler.

2.7 Exploiting ILP Using Multiple Issue and Static Scheduling

The techniques of the preceding sections can be used to eliminate data and control stalls and achieve an ideal CPI of one. To improve performance further we would like to decrease the CPI to less than one. But the CPI cannot be reduced below one if we issue only one instruction every clock cycle.

The goal of the *multiple-issue processors,* discussed in the next few sections, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in three major flavors:

1. statically scheduled superscalar processors,
2. VLIW (very long instruction word) processors, and
3. dynamically scheduled superscalar processors.

The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled.

VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction. VLIW processors are inherently statically scheduled by the compiler. When Intel and HP created the IA-64 architecture, described in Appendix G, they also introduced the name EPIC—explicitly parallel instruction computer—for this architectural style.

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	mostly in the embedded space: MIPS and ARM
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution, but no speculation	none at the present
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium 4, MIPS R12K, IBM Power5
VLIW/LIW	static	primarily software	static	all hazards determined and indicated by compiler (often implicitly)	most examples are in the embedded space, such as the TI C6x
EPIC	primarily static	primarily software	mostly static	all hazards determined and indicated explicitly by the compiler	Itanium

Figure 2.18 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix G focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

Although statically scheduled superscalars issue a varying rather than a fixed number of instructions per clock, they are actually closer in concept to VLIWs, since both approaches rely on the compiler to schedule code for the processor. Because of the diminishing advantages of a statically scheduled superscalar as the issue width grows, statically scheduled superscalars are used primarily for narrow issue widths, normally just two instructions. Beyond that width, most designers choose to implement either a VLIW or a dynamically scheduled superscalar. Because of the similarities in hardware and required compiler technology, we focus on VLIWs in this section. The insights of this section are easily extrapolated to a statically scheduled superscalar.

Figure 2.18 summarizes the basic approaches to multiple issue and their distinguishing characteristics and shows processors that use each approach.

The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. Since there is no fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach.

Since this advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider-issue processor. Indeed, for simple two-issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four-issue processor has manageable overhead, but as we will see in the next chapter, the growth in overhead is a major factor limiting wider-issue processors.

Let's consider a VLIW processor with instructions that contain five operations, including one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16–24 bits per unit, yielding an instruction length of between 80 and 120 bits. By comparison, the Intel Itanium 1 and 2 contain 6 operations per instruction packet.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straight-line code, then *local scheduling* techniques, which operate on a single basic block, can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex *global scheduling* algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they also must deal with significantly more complicated trade-offs in optimization, since moving code across branches is expensive.

In Appendix G, we will discuss *trace scheduling*, one of these global scheduling techniques developed specifically for VLIWs; we will also explore special hardware support that allows some conditional branches to be eliminated, extending the usefulness of local scheduling and enhancing the performance of global scheduling.

For now, we will rely on loop unrolling to generate long, straight-line code sequences, so that we can use local scheduling to build up VLIW instructions and focus on how well these processors operate.

Example Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop `x[i] = x[i] + s` (see page 76 for the MIPS code) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore delayed branches.

Answer Figure 2.19 shows the code. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 2.2 that used unrolled and scheduled code.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 2.19 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled.

For the original VLIW model, there were both technical and logistical problems that make the approach less efficient. The technical problems are the increase in code size and the limitations of lockstep operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as in earlier examples), thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Appendix G, we examine software scheduling approaches, such as software pipelining, that can achieve the benefits of unrolling without as much code expansion.

To combat this code size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded. In Appendix G, we show other techniques, as well as document the significant code expansion seen on IA-64.

Early VLIWs operated in lockstep; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause *all* the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable.

In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus, different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between successive implementations, or between implementations with different issue widths, more difficult than it is for a superscalar design. Of course, obtaining improved performance from a new superscalar design may require recompilation. Nonetheless, the ability to run old binary files is a practical advantage for the superscalar approach.

The EPIC approach, of which the IA-64 architecture is the primary example, provides solutions to many of the problems encountered in early VLIW designs, including extensions for more aggressive software speculation and methods to overcome the limitation of hardware dependence while preserving binary compatibility.

The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP programs, the original loop probably could have been run efficiently on a vector processor (described in Appendix F). It is not clear that a multiple-issue processor is preferred over a vector processor for such applications; the costs are similar, and the vector processor is typically the same speed or faster. The potential advantages of a multiple-issue processor versus a vector processor are their ability to extract some parallelism from less structured code and their ability to easily cache all forms of data. For these reasons multiple-issue approaches have become the primary method for taking advantage of instruction-level parallelism, and vectors have become primarily an extension to these processors.

2.8 Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

So far, we have seen how the individual mechanisms of dynamic scheduling, multiple issue, and speculation work. In this section, we put all three together, which yields a microarchitecture quite similar to those in modern microprocessors. For simplicity, we consider only an issue rate of two instructions per clock, but the concepts are no different from modern processors that issue three or more instructions per clock.

Let's assume we want to extend Tomasulo's algorithm to support a two-issue superscalar pipeline with a separate integer and floating-point unit, each of which can initiate an operation on every clock. We do not want to issue instructions to the reservation stations out of order, since this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit. Because the interaction of the integer and floating-point instructions is crucial, we also extend Tomasulo's scheme to deal with both the integer and floating-point functional units and registers, as well as incorporating speculative execution.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that the key is assigning a reservation station and updating the pipeline control tables. One approach is to run this step in half a clock cycle, so that two instructions can be processed in one clock cycle. A second alternative is to build the logic necessary to handle two instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock often include both approaches: They both pipeline and widen the issue logic.

Putting together speculative dynamic scheduling with multiple issue requires overcoming one additional challenge at the back end of the pipeline: we must be able to complete and commit multiple instructions per clock. Like the challenge of issuing multiple instructions, the concepts are simple, although the implementation may be challenging in the same manner as the issue and register renaming process. We can show how the concepts fit together with an example.

Example Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:   LD       R2,0(R1)       ;R2=array element
        DADDIU   R2,R2,#1       ;increment R2
        SD       R2,0(R1)       ;store result
        DADDIU   R1,R1,#8       ;increment pointer
        BNE      R2,R3,LOOP     ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

Answer Figures 2.20 and 2.21 show the performance for a two-issue dynamically scheduled processor, without and with speculation. In this case, where a branch can be a critical performance limiter, speculation helps significantly. The third branch in

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

Figure 2.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the LD following the BNE cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 2.21 shows this example with speculation,

the speculative processor executes in clock cycle 13, while it executes in clock cycle 19 on the nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to complete effective address calculation before a branch is decided, but unless speculative memory accesses are allowed, this improvement will gain only 1 clock per iteration.

This example clearly shows how speculation can be advantageous when there are data-dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. Incorrect speculation will not improve performance, but will, in fact, typically harm performance.

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	`LD R2,0(R1)`	1	2	3	4	5	First issue
1	`DADDIU R2,R2,#1`	1	5		6	7	Wait for `LW`
1	`SD R2,0(R1)`	2	3			7	Wait for `DADDIU`
1	`DADDIU R1,R1,#8`	2	3		4	8	Commit in order
1	`BNE R2,R3,LOOP`	3	7			8	Wait for `DADDIU`
2	`LD R2,0(R1)`	4	5	6	7	9	No execute delay
2	`DADDIU R2,R2,#1`	4	8		9	10	Wait for `LW`
2	`SD R2,0(R1)`	5	6			10	Wait for `DADDIU`
2	`DADDIU R1,R1,#8`	5	6		7	11	Commit in order
2	`BNE R2,R3,LOOP`	6	10			11	Wait for `DADDIU`
3	`LD R2,0(R1)`	7	8	9	10	12	Earliest possible
3	`DADDIU R2,R2,#1`	7	11		12	13	Wait for `LW`
3	`SD R2,0(R1)`	8	9			13	Wait for `DADDIU`
3	`DADDIU R1,R1,#8`	8	9		10	14	Executes earlier
3	`BNE R2,R3,LOOP`	9	13			14	Wait for `DADDIU`

Figure 2.21 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the `LD` following the `BNE` can start execution early because it is speculative.

2.9 Advanced Techniques for Instruction Delivery and Speculation

In a high-performance pipeline, especially one with multiple issue, predicting branches well is not enough; we actually have to be able to deliver a high-bandwidth instruction stream. In recent multiple-issue processors, this has meant delivering 4–8 instructions every clock cycle. We look at methods for increasing instruction delivery bandwidth first. We then turn to a set of key issues in implementing advanced speculation techniques, including the use of register renaming versus reorder buffers, the aggressiveness of speculation, and a technique called value prediction, which could further enhance ILP.

Increasing Instruction Fetch Bandwidth

A multiple issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput. Of course, fetching these instructions requires wide enough paths to the instruction cache, but the most difficult aspect is handling branches. In this section we look at two methods for dealing with branches and then discuss how modern processors integrate the instruction prediction and prefetch functions.

Branch-Target Buffers

To reduce the branch penalty for our simple five-stage pipeline, as well as for deeper pipelines, we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next PC should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a *branch-target buffer* or *branch-target cache.* Figure 2.22 shows a branch-target buffer.

Because a branch-target buffer predicts the next instruction address and will send it out *before* decoding the instruction, we *must* know whether the fetched instruction is predicted as a taken branch. If the PC of the fetched instruction matches a PC in the prediction buffer, then the corresponding predicted PC is used as the next PC. The hardware for this branch-target buffer is essentially identical to the hardware for a cache.

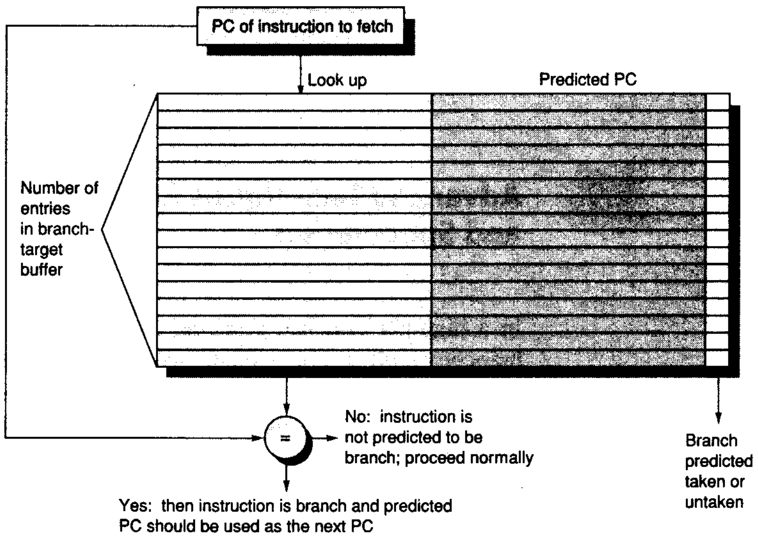

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Figure 2.22 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that unlike a branch-prediction buffer, the predictive entry must be matched to this instruction because the predicted PC will be sent out before it is known whether this instruction is even a branch. If the processor did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch should simply fetch the next sequential instruction, as if it were not a branch.

Figure 2.23 shows the detailed steps when using a branch-target buffer for a simple five-stage pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and the prediction is correct. Otherwise, there will be a penalty of at least 2 clock cycles. Dealing with the mispredictions and misses is a significant challenge, since we typically will have to halt instruction fetch while we rewrite the buffer entry. Thus, we would like to make this process fast to minimize the penalty.

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases. Figure 2.24 contains this information for the simple five-stage pipeline.

Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 2.24. Make the following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

Answer We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of 2 cycles.

$$\begin{aligned}
\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate} \times \text{Percent incorrect predictions} \\
&= 90\% \times 10\% = 0.09 \\
\text{Probability (branch not in buffer, but actually taken)} &= 10\% \\
\text{Branch penalty} &= (0.09 + 0.10) \times 2 \\
\text{Branch penalty} &= 0.38
\end{aligned}$$

This penalty compares with a branch penalty for delayed branches, which we evaluate in Appendix A, of about 0.5 clock cycles per branch. Remember, though, that the improvement from dynamic branch prediction will grow as the pipeline length and, hence, the branch delay grows; in addition, better predictors will yield a larger performance advantage.

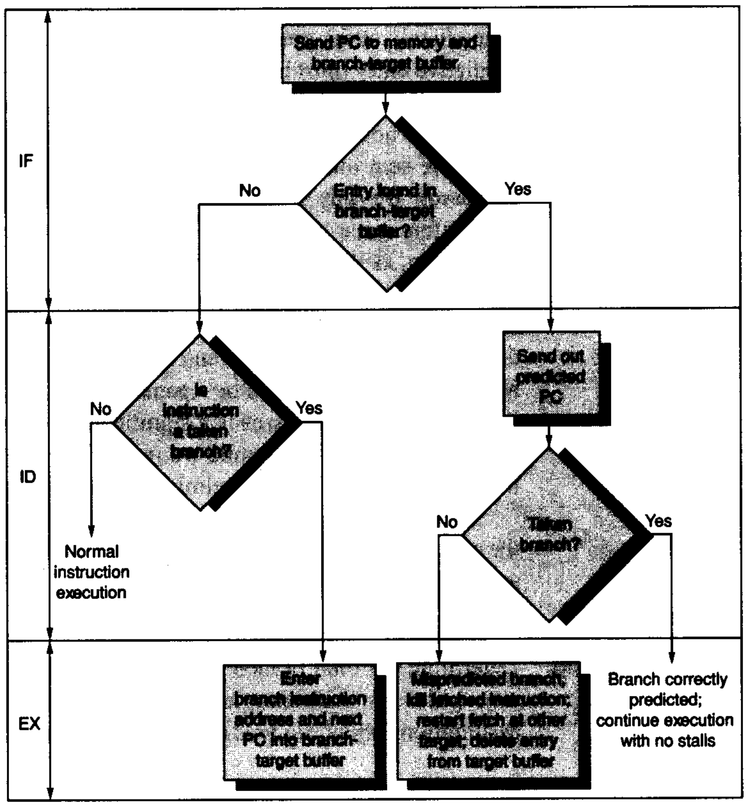

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Figure 2.23 The steps involved in handling an instruction with a branch-target buffer.

Instruction in buffer	Prediction	Actual branch	Penalty cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

Figure 2.24 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.

One variation on the branch-target buffer is to store one or more *target instructions* instead of, or in addition to, the predicted *target address*. This variation has two potential advantages. First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer. Second, buffering the actual target instructions allows us to perform an optimization called *branch folding*. Branch folding can be used to obtain 0-cycle unconditional branches, and sometimes 0-cycle conditional branches. Consider a branch-target buffer that buffers instructions from the predicted path and is being accessed with the address of an unconditional branch. The only function of the unconditional branch is to change the PC. Thus, when the branch-target buffer signals a hit and indicates that the branch is unconditional, the pipeline can simply substitute the instruction from the branch-target buffer in place of the instruction that is returned from the cache (which is the unconditional branch). If the processor is issuing multiple instructions per cycle, then the buffer will need to supply multiple instructions to obtain the maximum benefit. In some cases, it may be possible to eliminate the cost of a conditional branch when the condition codes are preset.

Return Address Predictors

As we try to increase the opportunity and accuracy of speculation we face the challenge of predicting indirect jumps, that is, jumps whose destination address varies at run time. Although high-level language programs will generate such jumps for indirect procedure calls, select or case statements, and FORTRAN-computed gotos, the vast majority of the indirect jumps come from procedure returns. For example, for the SPEC95 benchmarks, procedure returns account for more than 15% of the branches and the vast majority of the indirect jumps on average. For object-oriented languages like C++ and Java, procedure returns are even more frequent. Thus, focusing on procedure returns seems appropriate.

Though procedure returns can be predicted with a branch-target buffer, the accuracy of such a prediction technique can be low if the procedure is called from multiple sites and the calls from one site are not clustered in time. For example, in SPEC CPU95, an aggressive branch predictor achieves an accuracy of less than 60% for such return branches. To overcome this problem, some designs use a small buffer of return addresses operating as a stack. This structure caches the most recent return addresses: pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large (i.e., as large as the maximum call depth), it will predict the returns perfectly. Figure 2.25 shows the performance of such a return buffer with 0–16 elements for a number of the SPEC CPU95 benchmarks. We will use a similar return predictor when we examine the studies of ILP in Section 3.2.

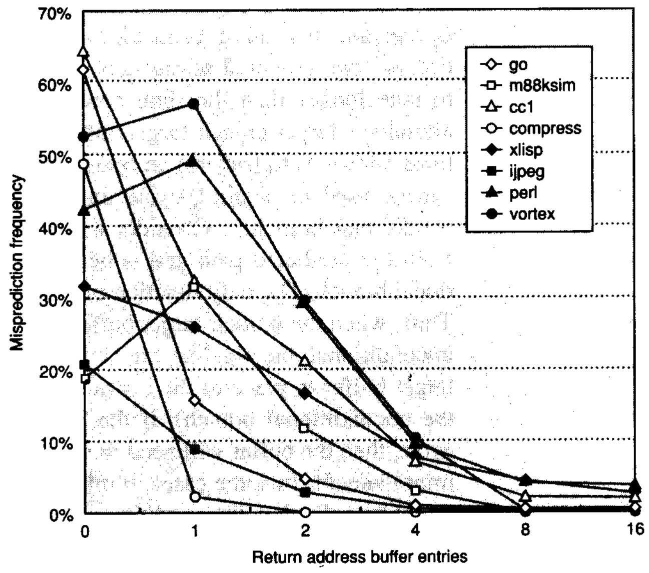


Figure 2.25 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Since call depths are typically not large, with some exceptions, a modest buffer works well. This data comes from Skadron et al. (1999), and uses a fix-up mechanism to prevent corruption of the cached return addresses.

Integrated Instruction Fetch Units

To meet the demands of multiple-issue processors, many recent designers have chosen to implement an integrated instruction fetch unit, as a separate autonomous unit that feeds instructions to the rest of the pipeline. Essentially, this amounts to recognizing that characterizing instruction fetch as a simple single pipe stage given the complexities of multiple issue is no longer valid.

Instead, recent designs have used an integrated instruction fetch unit that integrates several functions:

1. *Integrated branch prediction*—The branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so as to drive the fetch pipeline.
2. *Instruction prefetch*—To deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions (see Chapter 5 for a discussion of techniques for doing this), integrating it with branch prediction.

3. *Instruction memory access and buffering*—When fetching multiple instructions per cycle a variety of complexities are encountered, including the difficulty that fetching multiple instructions may require accessing multiple cache lines. The instruction fetch unit encapsulates this complexity, using prefetch to try to hide the cost of crossing cache blocks. The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

As designers try to increase the number of instructions executed per clock, instruction fetch will become an ever more significant bottleneck, and clever new ideas will be needed to deliver instructions at the necessary rate. One of the newer ideas, called *trace caches* and used in the Pentium 4, is discussed in Appendix C.

Speculation: Implementation Issues and Extensions

In this section we explore three issues that involve the implementation of speculation, starting with the use of register renaming, the approach that has almost totally replaced the use of a reorder buffer. We then discuss one important possible extension to speculation on control flow: an idea called value prediction.

Speculation Support: Register Renaming versus Reorder Buffers

One alternative to the use of a reorder buffer (ROB) is the explicit use of a larger physical set of registers combined with register renaming. This approach builds on the concept of renaming used in Tomasulo's algorithm and extends it. In Tomasulo's algorithm, the values of the *architecturally visible registers* (R0, . . . , R31 and F0, . . . , F31) are contained, at any point in execution, in some combination of the register set and the reservation stations. With the addition of speculation, register values may also temporarily reside in the ROB. In either case, if the processor does not issue new instructions for a period of time, all existing instructions will commit, and the register values will appear in the register file, which directly corresponds to the architecturally visible registers.

In the register-renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values. Thus, the extended registers replace the function of both the ROB and the reservation stations. During instruction issue,.a renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination. WAW and WAR hazards are avoided by renaming of the destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become the architectural register until the instruction commits. The renaming map is a simple data structure that supplies the physical register number of the register that currently corresponds to the specified architectural register. This structure is similar in structure and function to the register status table in Tomasulo's algorithm. When an instruction commits, the renaming table is permanently updated to indicate that a physical register corresponds to the actual architectural register, thus effectively finalizing the update to the processor state.

An advantage of the renaming approach versus the ROB approach is that instruction commit is simplified, since it requires only two simple actions: record that the mapping between an architectural register number and physical register number is no longer speculative, and free up any physical registers being used to hold the "older" value of the architectural register. In a design with reservation stations, a station is freed up when the instruction using it completes execution, and a ROB entry is freed up when the corresponding instruction commits.

With register renaming, deallocating registers is more complex, since before we free up a physical register, we must know that it no longer corresponds to an architectural register, and that no further uses of the physical register are outstanding. A physical register corresponds to an architectural register until the architectural register is rewritten, causing the renaming table to point elsewhere. That is, if no renaming entry points to a particular physical register, then it no longer corresponds to an architectural register. There may, however, still be uses of the physical register outstanding. The processor can determine whether this is the case by examining the source register specifiers of all instructions in the functional unit queues. If a given physical register does not appear as a source and it is not designated as an architectural register, it may be reclaimed and reallocated.

Alternatively, the processor can simply wait until another instruction that writes the same architectural register commits. At that point, there can be no further uses of the older value outstanding. Although this method may tie up a physical register slightly longer than necessary, it is easy to implement and hence is used in several recent superscalars.

One question you may be asking is, How do we ever know which registers are the architectural registers if they are constantly changing? Most of the time when the program is executing it does not matter. There are clearly cases, however, where another process, such as the operating system, must be able to know exactly where the contents of a certain architectural register reside. To understand how this capability is provided, assume the processor does not issue instructions for some period of time. Eventually all instructions in the pipeline will commit, and the mapping between the architecturally visible registers and physical registers will become stable. At that point, a subset of the physical registers contains the architecturally visible registers, and the value of any physical register not associated with an architectural register is unneeded. It is then easy to move the architectural registers to a fixed subset of physical registers so that the values can be communicated to another process.

Within the past few years most high-end superscalar processors, including the Pentium series, the MIPS R12000, and the Power and PowerPC processors, have chosen to use register renaming, adding from 20 to 80 extra registers. Since all results are allocated a new virtual register until they commit, these extra registers replace a primary function of the ROB and largely determine how many instructions may be in execution (between issue and commit) at one time.

How Much to Speculate

One of the significant advantages of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This potential advantage, however, comes with a significant potential disadvantage. Speculation is not free: it takes time and energy, and the recovery of incorrect speculation further reduces performance. In addition, to support the higher instruction execution rate needed to benefit from speculation, the processor must have additional resources, which take silicon area and power. Finally, if speculation causes an exceptional event to occur, such as a cache or TLB miss, the potential for significant performance loss increases, if that event would not have occurred without speculation.

To maintain most of the advantage, while minimizing the disadvantages, most pipelines with speculation will allow only low-cost exceptional events (such as a first-level cache miss) to be handled in speculative mode. If an expensive exceptional event occurs, such as a second-level cache miss or a translation lookaside buffer (TLB) miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event. Although this may slightly degrade the performance of some programs, it avoids significant performance losses in others, especially those that suffer from a high frequency of such events coupled with less-than-excellent branch prediction.

In the 1990s, the potential downsides of speculation were less obvious. As processors have evolved, the real costs of speculation have become more apparent, and the limitations of wider issue and speculation have been obvious. We return to this issue in the next chapter.

Speculating through Multiple Branches

In the examples we have considered in this chapter, it has been possible to resolve a branch before having to speculate on another. Three different situations can benefit from speculating on multiple branches simultaneously: a very high branch frequency, significant clustering of branches, and long delays in functional units. In the first two cases, achieving high performance may mean that multiple branches are speculated, and it may even mean handling more than one branch per clock. Database programs, and other less structured integer computations, often exhibit these properties, making speculation on multiple branches important. Likewise, long delays in functional units can raise the importance of speculating on multiple branches as a way to avoid stalls from the longer pipeline delays.

Speculating on multiple branches slightly complicates the process of speculation recovery, but is straightforward otherwise. A more complex technique is predicting and speculating on more than one branch per cycle. The IBM Power2 could resolve two branches per cycle but did not speculate on any other instructions. As of 2005, no processor has yet combined full speculation with resolving multiple branches per cycle.

Value Prediction

One technique for increasing the amount of ILP available in a program is value prediction. *Value prediction* attempts to predict the value that will be produced by an instruction. Obviously, since most instructions produce a different value every time they are executed (or at least a different value from a set of values), value prediction can have only limited success. There are, however, certain instructions for which it is easier to predict the resulting value—for example, loads that load from a constant pool, or that load a value that changes infrequently. In addition, when an instruction produces a value chosen from a small set of potential values, it may be possible to predict the resulting value by correlating it without an instance.

Value prediction is useful if it significantly increases the amount of available ILP. This possibility is most likely when a value is used as the source of a chain of dependent computations, such as a load. Because value prediction is used to enhance speculations and incorrect speculation has detrimental performance impact, the accuracy of the prediction is critical.

Much of the focus of research on value prediction has been on loads. We can estimate the maximum accuracy of a load value predictor by examining how often a load returns a value that matches a value returned in a recent execution of the load. The simplest case to examine is when the load returns a value that matches the value on the last execution of the load. For a range of SPEC CPU2000 benchmarks, this redundancy occurs from less than 5% of the time to almost 80% of the time. If we allow the load to match any of the most recent 16 values returned, the frequency of a potential match increases, and many benchmarks show a 80% match rate. Of course, matching 1 of 16 recent values does not tell you what value to predict, but it does mean that even with additional information it is impossible for prediction accuracy to exceed 80%.

Because of the high costs of misprediction and the likely case that misprediction rates will be significant (20% to 50%), researchers have focused on assessing which loads are more predictable and only attempting to predict those. This leads to a lower misprediction rate, but also fewer candidates for accelerating through prediction. In the limit, if we attempt to predict only those loads that always return the same value, it is likely that only 10% to 15% of the loads can be predicted. Research on value prediction continues. The results to date, however, have not been sufficiently compelling that any commercial processor has included the capability.

One simple idea that has been adopted and is related to value prediction is address aliasing prediction. *Address aliasing prediction* is a simple technique that predicts whether two stores or a load and a store refer to the same memory address. If two such references do not refer to the same address, then they may be safely interchanged. Otherwise, we must wait until the memory addresses accessed by the instructions are known. Because we need not actually predict the address values, only whether such values conflict, the prediction is both more stable and simpler. Hence, this limited form of address value speculation has been used by a few processors.

2.10 Putting It All Together: The Intel Pentium 4

The Pentium 4 is a processor with a deep pipeline supporting multiple issue with speculation. In this section, we describe the highlights of the Pentium 4 microarchitecture and examine its performance for the SPEC CPU benchmarks. The Pentium 4 also supports multithreading, a topic we discuss in the next chapter.

The Pentium 4 uses an aggressive out-of-order speculative microarchitecture, called Netburst, that is deeply pipelined with the goal of achieving high instruction throughput by combining multiple issue and high clock rates. Like the microarchitecture used in the Pentium III, a front-end decoder translates each IA-32 instruction to a series of micro-operations (uops), which are similar to typical RISC instructions. The uops are than executed by a dynamically scheduled speculative pipeline.

The Pentium 4 uses a novel *execution trace cache* to generate the uop instruction stream, as opposed to a conventional instruction cache that would hold IA-32 instructions. A *trace cache* is a type of instruction cache that holds sequences of instructions to be executed including nonadjacent instructions separated by branches; a trace cache tries to exploit the temporal sequencing of instruction execution rather than the spatial locality exploited in a normal cache; trace caches are explained in detail in Appendix C.

The Pentium 4's execution trace cache is a trace cache of uops, corresponding to the decoded IA-32 instruction stream. By filling the pipeline from the execution trace cache, the Pentium 4 avoids the need to redecode IA-32 instructions whenever the trace cache hits. Only on trace cache misses are IA-32 instructions fetched from the L2 cache and decoded to refill the execution trace cache. Up to three IA-32 instructions may be decoded and translated every cycle, generating up to six uops; when a single IA-32 instruction requires more than three uops, the uop sequence is generated from the microcode ROM.

The execution trace cache has its own branch target buffer, which predicts the outcome of uop branches. The high hit rate in the execution trace cache (for example, the trace cache miss rate for the SPEC CPUINT2000 benchmarks is less than 0.15%), means that the IA-32 instruction fetch and decode is rarely needed.

After fetching from the execution trace cache, the uops are executed by an out-of-order speculative pipeline, similar to that in Section 2.6, but using register renaming rather than a reorder buffer. Up to three uops per clock can be renamed and dispatched to the functional unit queues, and three uops can be committed each clock cycle. There are four dispatch ports, which allow a total of six uops to be dispatched to the functional units every clock cycle. The load and store units each have their own dispatch port, another port covers basic ALU operations, and a fourth handles FP and integer operations. Figure 2.26 shows a diagram of the microarchitecture.

Since the Pentium 4 microarchitecture is dynamically scheduled, uops do not follow a simple static set of pipeline stages during their execution. Instead various stages of execution (instruction fetch, decode, uop issue, rename, schedule, execute, and retire) can take varying numbers of clock cycles. In the Pentium III,

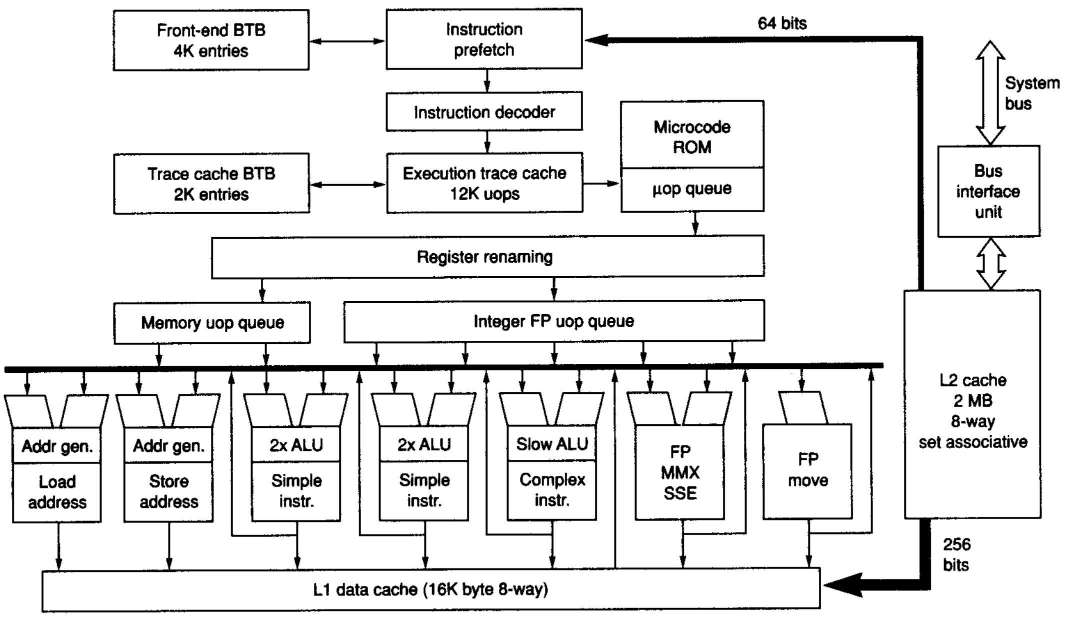


Figure 2.26 The Pentium 4 microarchitecture. The cache sizes represent the Pentium 4 640. Note that the instructions are usually coming from the trace cache; only when the trace cache misses is the front-end instruction prefetch unit consulted. This figure was adapted from Boggs et al. [2004].

the minimum time for an instruction to go from fetch to retire was 11 clock cycles, with instructions requiring multiple clock cycles in the execution stage taking longer. As in any dynamically scheduled pipeline, instructions could take much longer if they had to wait for operands. As stated earlier, the Pentium 4 introduced a much deeper pipeline, partitioning stages of the Pentium III pipeline so as to achieve a higher clock rate. In the initial Pentium 4 introduced in 1990, the minimum number of cycles to transit the pipeline was increased to 21, allowing for a 1.5 GHz clock rate. In 2004, Intel introduced a version of the Pentium 4 with a 3.2 GHz clock rate. To achieve this high clock rate, further pipelining was added so that a simple instruction takes 31 clock cycles to go from fetch to retire. This additional pipelining, together with improvements in transistor speed, allowed the clock rate to more than double over the first Pentium 4.

Obviously, with such deep pipelines and aggressive clock rates the cost of cache misses and branch mispredictions are both very high A two-level cache is used to minimize the frequency of DRAM accesses. Branch prediction is done with a branch-target buffer using a two-level predictor with both local and global branch histories; in the most recent Pentium 4, the size of the branch-target buffer was increased, and the static predictor, used when the branch-target buffer misses, was improved. Figure 2.27 summarizes key features of the microarchitecture, and the caption notes some of the changes since the first version of the Pentium 4 in 2000.

Feature	Size	Comments
Front-end branch-target buffer	4K entries	Predicts the next IA-32 instruction to fetch; used only when the execution trace cache misses.
Execution trace cache	12K uops	Trace cache used for uops.
Trace cache branch-target buffer	2K entries	Predicts the next uop.
Registers for renaming	128 total	128 uops can be in execution with up to 48 loads and 32 stores.
Functional units	7 total: 2 simple ALU, complex ALU, load, store, FP move, FP arithmetic	The simple ALU units run at twice the clock rate, accepting up to two simple ALU uops every clock cycle. This allows execution of two dependent ALU operations in a single clock cycle.
L1 data cache	16 KB; 8-way associative; 64-byte blocks write through	Integer load to use latency is 4 cycles; FP load to use latency is 12 cycles; up to 8 outstanding load misses.
L2 cache	2 MB; 8-way associative; 128-byte blocks write back	256 bits to L1, providing 108 GB/sec; 18-cycle access time; 64 bits to memory capable of 6.4 GB/sec. A miss in L2 does not cause an automatic update of L1.

Figure 2.27 Important characteristics of the recent Pentium 4 640 implementation in 90 nm technology (code named Prescott). The newer Pentium 4 uses larger caches and branch-prediction buffers, allows more loads and stores outstanding, and has higher bandwidth between levels in the memory system. Note the novel use of double-speed ALUs, which allow the execution of back-to-back dependent ALU operations in a single clock cycle; having twice as many ALUs, an alternative design point, would not allow this capability. The original Pentium 4 used a trace cache BTB with 512 entries, an L1 cache of 8 KB, and an L2 cache of 256 KB.

An Analysis of the Performance of the Pentium 4

The deep pipeline of the Pentium 4 makes the use of speculation, and its dependence on branch prediction, critical to achieving high performance. Likewise, performance is very dependent on the memory system. Although dynamic scheduling and the large number of outstanding loads and stores supports hiding the latency of cache misses, the aggressive 3.2 GHz clock rate means that L2 misses are likely to cause a stall as the queues fill up while awaiting the completion of the miss.

Because of the importance of branch prediction and cache misses, we focus our attention on these two areas. The charts in this section use five of the integer SPEC CPU2000 benchmarks and five of the FP benchmarks, and the data is captured using counters within the Pentium 4 designed for performance monitoring. The processor is a Pentium 4 640 running at 3.2 GHz with an 800 MHz system bus and 667 MHz DDR2 DRAMs for main memory.

Figure 2.28 shows the branch-misprediction rate in terms of mispredictions per 1000 instructions. Remember that in terms of pipeline performance, what matters is the number of mispredictions per instruction; the FP benchmarks generally have fewer branches per instruction (48 branches per 1000 instructions) versus the integer benchmarks (186 branches per 1000 instructions), as well as

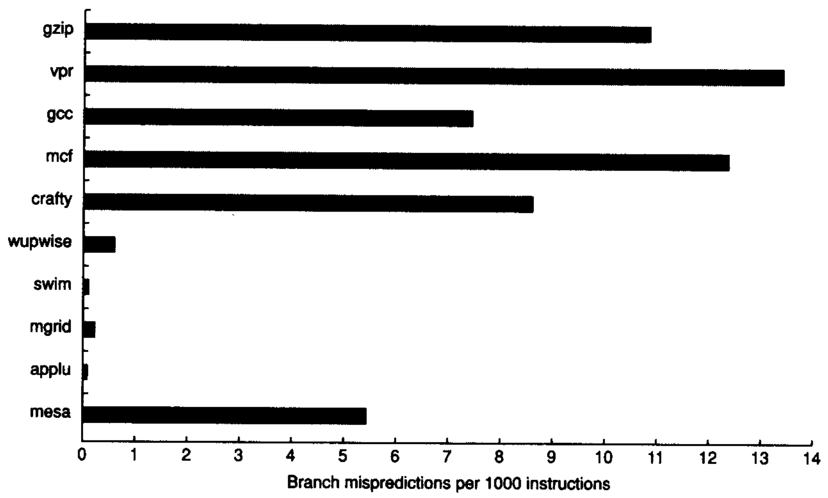


Figure 2.28 Branch misprediction rate per 1000 instructions for five integer and five floating-point benchmarks from the SPEC CPU2000 benchmark suite. This data and the rest of the data in this section were acquired by John Holm and Dileep Bhandarkar of Intel.

better prediction rates (98% versus 94%). The result, as Figure 2.28 shows, is that the misprediction rate per instruction for the integer benchmarks is more than 8 times higher than the rate for the FP benchmarks.

Branch-prediction accuracy is crucial in speculative processors, since incorrect speculation requires recovery time and wastes energy pursuing the wrong path. Figure 2.29 shows the fraction of executed uops that are the result of misspeculation. As we would suspect, the misspeculation rate results look almost identical to the misprediction rates.

How do the cache miss rates contribute to possible performance losses? The trace cache miss rate is almost negligible for this set of the SPEC benchmarks, with only one benchmark (186.craft) showing any significant misses (0.6%). The L1 and L2 miss rates are more significant. Figure 2.30 shows the L1 and L2 miss rates for these 10 benchmarks. Although the miss rate for L1 is about 14 times higher than the miss rate for L2, the miss penalty for L2 is comparably higher, and the inability of the microarchitecture to hide these very long misses means that L2 misses likely are responsible for an equal or greater performance loss than L1 misses, especially for benchmarks such as mcf and swim.

How do the effects of misspeculation and cache misses translate to actual performance? Figure 2.31 shows the effective CPI for the 10 SPEC CPU2000 benchmarks. There are three benchmarks whose performance stands out from the pack and are worth examining:

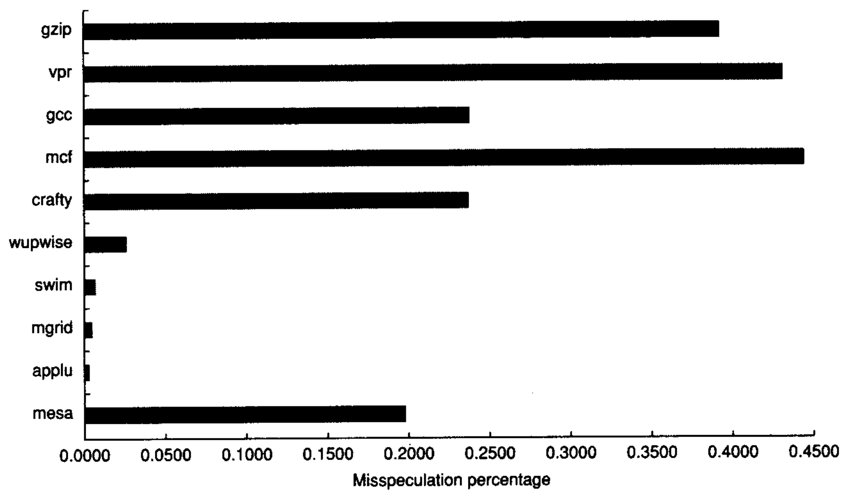


Figure 2.29 The percentage of uop instructions issued that are misspeculated.

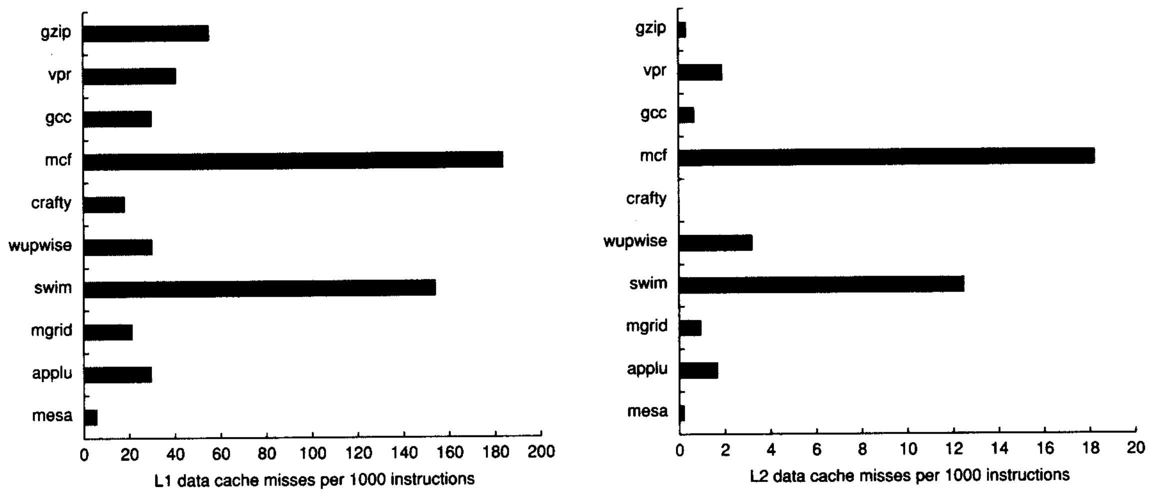


Figure 2.30 L1 data cache and L2 cache misses per 1000 instructions for 10 SPEC CPU2000 benchmarks. Note that the scale of the L1 misses is 10 times that of the L2 misses. Because the miss penalty for L2 is likely to be at least 10 times larger than for L1, the relative sizes of the bars are an indication of the relative performance penalty for the misses in each cache. The inability to hide long L2 misses with overlapping execution will further increase the stalls caused by L2 misses relative to L1 misses.

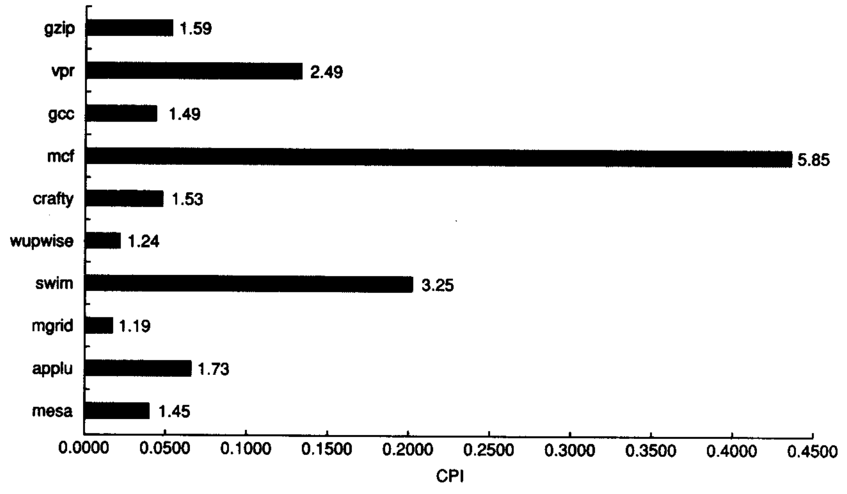


Figure 2.31 The CPI for the 10 SPEC CPU benchmarks. An increase in the CPI by a factor of 1.29 comes from the translation of IA-32 instructions into uops, which results in 1.29 uops per IA-32 instruction on average for these 10 benchmarks.

1. mcf has a CPI that is more than four times higher than that of the four other integer benchmarks. It has the worst misspeculation rate. Equally importantly, mcf has the worst L1 and the worst L2 miss rate among any benchmark, integer or floating point, in the SPEC suite. The high cache miss rates make it impossible for the processor to hide significant amounts of miss latency.
2. vpr achieves a CPI that is 1.6 times higher than three of the five integer benchmarks (excluding mcf). This appears to arise from a branch misprediction that is the worst among the integer benchmarks (although not much worse than the average) together with a high L2 miss rate, second only to mcf among the integer benchmarks.
3. swim is the lowest performing FP benchmark, with a CPI that is more than two times the average of the other four FP benchmarks. swim's problems are high L1 and L2 cache miss rates, second only to mcf. Notice that swim has excellent speculation results, but that success can probably not hide the high miss rates, especially in L2. In contrast, several benchmarks with reasonable L1 miss rates and low L2 miss rates (such as mgrid and gzip) perform well.

To close this section, let's look at the relative performance of the Pentium 4 and AMD Opteron for this subset of the SPEC benchmarks. The AMD Opteron and Intel Pentium 4 share a number of similarities:

- Both use a dynamically scheduled, speculative pipeline capable of issuing and committing three IA-32 instructions per clock.
- Both use a two-level on-chip cache structure, although the Pentium 4 uses a trace cache for the first-level instruction cache and recent Pentium 4 implementations have larger second-level caches.
- They have similar transistor counts, die size, and power, with the Pentium 4 being about 7% to 10% higher on all three measures at the highest clock rates available in 2005 for these two processors.

The most significant difference is the very deep pipeline of the Intel Netburst microarchitecture, which was designed to allow higher clock rates. Although compilers optimized for the two architectures produce slightly different code sequences, comparing CPI measures can provide important insights into how these two processors compare. Remember that differences in the memory hierarchy as well as differences in the pipeline structure will affect these measurements; we analyze the differences in memory system performance in Chapter 5. Figure 2.32 shows the CPI measures for a set of SPEC CPU2000 benchmarks for a 3.2 GHz Pentium 4 and a 2.6 GHz AMD Opteron. At these clock rates, the Opteron processor has an average improvement in CPI by 1.27 over the Pentium 4.

Of course, we should expect the Pentium 4, with its much deeper pipeline, to have a somewhat higher CPI than the AMD Opteron. The key question for the very deeply pipelined Netburst design is whether the increase in clock rate, which the deeper pipelining allows, overcomes the disadvantages of a higher

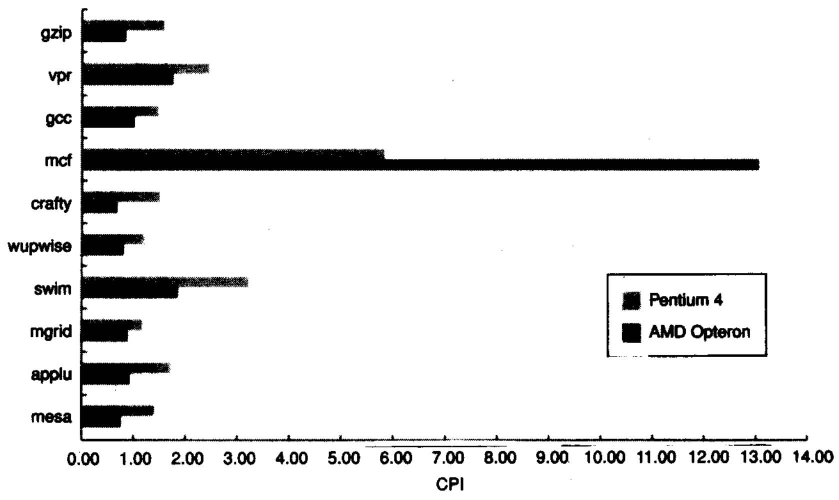


Figure 2.32 A 2.6 GHz AMD Opteron has a lower CPI by a factor of 1.27 versus a 3.2 GHz Pentium 4.

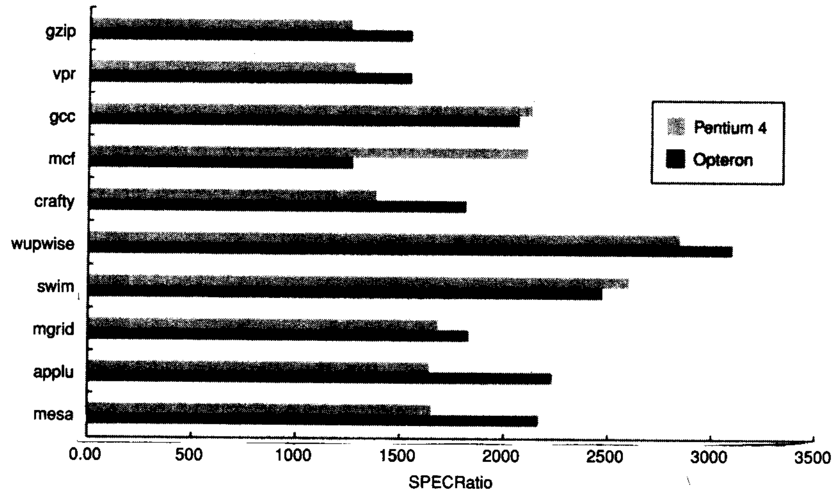


Figure 2.33 The performance of a 2.8 GHz AMD Opteron versus a 3.8 GHz Intel Pentium 4 shows a performance advantage for the Opteron of about 1.08.

CPI. We examine this by showing the SPEC CPU2000 performance for these two processors at their highest available clock rate of these processors in 2005: 2.8 GHz for the Opteron and 3.8 GHz for the Pentium 4. These higher clock rates will increase the effective CPI measurement versus those in Figure 2.32, since the cost of a cache miss will increase. Figure 2.33 shows the relative performance on the same subset of SPEC as Figure 2.32. The Opteron is slightly faster, meaning that the higher clock rate of the Pentium 4 is insufficient to overcome the higher CPI arising from more pipeline stalls.

Hence, while the Pentium 4 performs well, it is clear that the attempt to achieve both high clock rates via a deep pipeline and high instruction throughput via multiple issue is not as successful as the designers once believed it would be. We discuss this topic in depth in the next chapter.

2.11 Fallacies and Pitfalls

Our first fallacy has two parts: First, simple rules do not hold, and, second, the choice of benchmarks plays a major role.

Fallacy *Processors with lower CPIs will always be faster.*

Fallacy *Processors with faster clock rates will always be faster.*

Although a lower CPI is certainly better, sophisticated multiple-issue pipelines typically have slower clock rates than processors with simple pipelines. In applications with limited ILP or where the parallelism cannot be exploited by the hardware resources, the faster clock rate often wins. But, when significant ILP exists, a processor that exploits lots of ILP may be better.

The IBM Power5 processor is designed for high-performance integer and FP; it contains two processor cores each capable of sustaining four instructions per clock, including two FP and two load-store instructions. The highest clock rate for a Power5 processor in 2005 is 1.9 GHz. In comparison, the Pentium 4 offers a single processor with multithreading (see the next chapter). The processor can sustain three instructions per clock with a very deep pipeline, and the maximum available clock rate in 2005 is 3.8 GHz.

Thus, the Power5 will be faster if the product of the instruction count and CPI is less than one-half the same product for the Pentium 4. As Figure 2.34 shows the CPI × instruction count advantages of the Power5 are significant for the FP programs, sometimes by more than a factor of 2, while for the integer programs the CPI × instruction count advantage of the Power5 is usually not enough to overcome the clock rate advantage of the Pentium 4. By comparing the SPEC numbers, we find that the product of instruction count and CPI advantage for the Power5 is 3.1 times on the floating-point programs but only 1.5 times on the integer programs. Because the maximum clock rate of the Pentium 4 in 2005 is exactly twice that of the Power5, the Power5 is faster by 1.5 on SPECfp2000 and the Pentium 4 will be faster by 1.3 on SPECint2000.

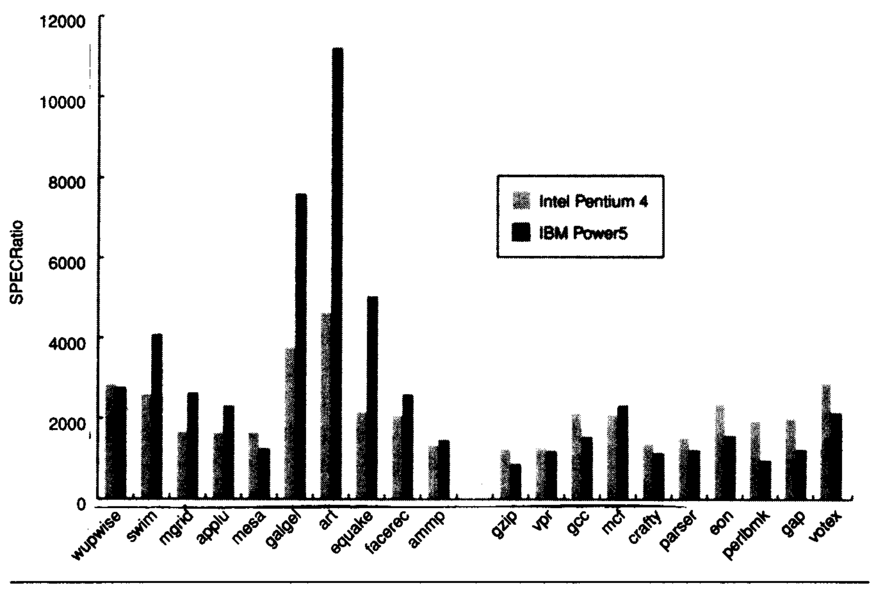


Figure 2.34 A comparison of the 1.9 GHZ IBM Power5 processor versus the 3.8 GHz Intel Pentium 4 for 20 SPEC benchmarks (10 integer on the left and 10 floating point on the right) shows that the higher clock Pentium 4 is generally faster for the integer workload, while the lower CPI Power5 is usually faster for the floating-point workload.

Pitfall *Sometimes bigger and dumber is better.*

Advanced pipelines have focused on novel and increasingly sophisticated schemes for improving CPI. The 21264 uses a sophisticated tournament predictor with a total of 29K bits (see page 88), while the earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4K bits). For the SPEC95 benchmarks, the more sophisticated branch predictor of the 21264 outperforms the simpler 2-bit scheme on all but one benchmark. On average, for SPECint95, the 21264 has 11.5 mispredictions per 1000 instructions committed, while the 21164 has about 16.5 mispredictions.

Somewhat surprisingly, the simpler 2-bit scheme works better for the transaction-processing workload than the sophisticated 21264 scheme (17 mispredictions versus 19 per 1000 completed instructions)! How can a predictor with less than 1/7 the number of bits and a much simpler scheme actually work better? The answer lies in the structure of the workload. The transaction-processing workload has a very large code size (more than an order of magnitude larger than any SPEC95 benchmark) with a large branch frequency. The ability of the 21164 predictor to hold twice as many branch predictions based on purely local behavior (2K versus the 1K local predictor in the 21264) seems to provide a slight advantage.

This pitfall also reminds us that different applications can produce different behaviors. As processors become more sophisticated, including specific microarchitectural features aimed at some particular program behavior, it is likely that different applications will see more divergent behavior.

2.12 Concluding Remarks

The tremendous interest in multiple-issue organizations came about because of an interest in improving performance without affecting the standard uniprocessor programming model. Although taking advantage of ILP is conceptually simple, the design problems are amazingly complex in practice. It is extremely difficult to achieve the performance you might expect from a simple first-level analysis.

Rather than embracing dramatic new approaches in microarchitecture, most of the last 10 years have focused on raising the clock rates of multiple-issue processors and narrowing the gap between peak and sustained performance. The dynamically scheduled, multiple-issue processors announced in the last five years (the Pentium 4, IBM Power5, and the AMD Athlon and Opteron) have the same basic structure and similar sustained issue rates (three to four instructions per clock) as the first dynamically scheduled, multiple-issue processors announced in 1995! But the clock rates are 10–20 times higher, the caches are 4–8 times bigger, there are 2–4 times as many renaming registers, and twice as many load-store units! The result is performance that is 8–16 times higher.

The trade-offs between increasing clock speed and decreasing CPI through multiple issue are extremely hard to quantify. In the 1995 edition of this book, we stated:

> Although you might expect that it is possible to build an advanced multiple-issue processor with a high clock rate, a factor of 1.5 to 2 in clock rate has consistently separated the highest clock rate processors and the most sophisticated multiple-issue processors. It is simply too early to tell whether this difference is due to fundamental implementation trade-offs, or to the difficulty of dealing with the complexities in multiple-issue processors, or simply a lack of experience in implementing such processors.

Given the availability of the Pentium 4 at 3.8 GHz, it has become clear that the limitation was primarily our understanding of how to build such processors. As we will see in the next chapter, however, it appears unclear that the initial success in achieving high-clock-rate processors that issue three to four instructions per clock can be carried much further due to limitations in available ILP, efficiency in exploiting that ILP, and power concerns. In addition, as we saw in the comparison of the Opteron and Pentium 4, it appears that the performance advantage in high clock rates achieved by very deep pipelines (20–30 stages) is largely lost by additional pipeline stalls. We analyze this behavior further in the next chapter.

One insight that was clear in 1995 and has become even more obvious in 2005 is that the peak-to-sustained performance ratios for multiple-issue processors are often quite large and typically grow as the issue rate grows. The lessons to be gleaned by comparing the Power5 and Pentium 4, or the Pentium 4 and Pentium III (which differ primarily in pipeline depth and hence clock rate, rather than issue rates), remind us that it is difficult to generalize about clock rate versus CPI, or about the underlying trade-offs in pipeline depth, issue rate, and other characteristics.

A change in approach is clearly upon us. Higher-clock-rate versions of the Pentium 4 have been abandoned. IBM has shifted to putting two processors on a single chip in the Power4 and Power5 series, and both Intel and AMD have delivered early versions of two-processor chips. We will return to this topic in the next chapter and indicate why the 20-year rapid pursuit of ILP seems to have reached its end.

2.13 Historical Perspective and References

Section K.4 on the companion CD features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics.

Case Studies with Exercises by Robert P. Colwell

Case Study 1: Exploring the Impact of Microarchitectural Techniques

Concepts illustrated by this case study

- Basic Instruction Scheduling, Reordering, Dispatch
- Multiple Issue and Hazards
- Register Renaming
- Out-of-Order and Speculative Execution
- Where to Spend Out-of-Order Resources

You are tasked with designing a new processor microarchitecture, and you are trying to figure out how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 2 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure 2.35 provides a sequence of instructions and list of latencies.

2.1 [10] <1.8, 2.1, 2.2> What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 2.35 if no new instruction execution could be initiated until the previous instruction execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a 1 cycle branch delay slot.

```
Loop: LD     F2,0(Rx)
I0:   MULTD  F2,F0,F2
I1:   DIVD   F8,F2,F0
I2:   LD     F4,0(Ry)
I3:   ADDD   F4,F0,F4
I4:   ADDD   F10,F8,F2
I5:   SD     F4,0(Ry)
I6:   ADDI   Rx,Rx,#8
I7:   ADDI   Ry,Ry,#8
I8:   SUB    R20,R4,Rx
I9:   BNZ    R20,Loop
```

Latencies beyond single cycle	
Memory LD	+3
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
ADDD	+2
MULTD	+4
DIVD	+10

Figure 2.35 Code and latencies for Exercises 2.1 through 2.6.

2.2 [10] <1.8, 2.1, 2.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 2.35 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with `<stall>` inserted where necessary to accommodate stated latencies. (*Hint:* An instruction with latency "+2" needs 2 `<stall>` cycles to be inserted into the code sequence. Think of it this way: a 1-cycle instruction has latency 1 + 0, meaning zero extra wait states. So latency 1 + 1 implies 1 stall cycle; latency 1 + *N* has *N* extra stall cycles.)

2.3 [15] <2.6, 2.7> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependence. Now how many cycles does the loop require?

2.4 [10] <2.6, 2.7> In the multiple-issue design of Exercise 2.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are not identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction *N* + 1 begins execution in Execution Pipe 1 at the same time that instruction *N* begins in Pipe 0, and *N* + 1 happens to require a shorter execution latency than *N*, then *N* + 1 will complete before *N* (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 2.35 that demonstrate this hazard.

2.5 [20] <2.7> Reorder the instructions to improve performance of the code in Figure 2.35. Assume the two-pipe machine in Exercise 2.3, and that the out-of-order completion issues of Exercise 2.4 have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?

2.6 [10/10] <2.1, 2.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not "living up to its potential."

a. [10] <2.1, 2.2> In your reordered code from Exercise 2.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?

b. [10] <2.1, 2.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance.

c. Hand-unroll two iterations of the loop in your reordered code from Exercise 2.5. What speedup did you obtain? (For this exercise, just color the $N + 1$ iteration's instructions green to distinguish them from the Nth iteration's; if you were actually unrolling the loop you would have to reassign registers to prevent collisions between the iterations.)

2.7 [15] <2.1> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep register usages multiple iterations from colliding, we rename their registers. Figure 2.36 shows example code that we would like our hardware to rename.

A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the source register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 2.36. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependences are maintained. Show the resulting code. (*Hint:* See Figure 2.37.)

```
Loop: LD     F2,0(Rx)
I0:   MULTD  F5,F0,F2
I1:   DIVD   F8,F0,F2
I2:   LD     F4,0(Ry)
I3:   ADDD   F6,F0,F4
I4:   ADDD   F10,F8,F2
I5:   SD     F4,0(Ry)
```

Figure 2.36 Sample code for register renaming practice.

```
I0: LD     T9,0(Rx)
I1: MULTD  T10,F0,T9
. . .
```

Figure 2.37 Expected output of register renaming.

```
I0:  MULTD  F5,F0,F2
I1:  ADDD   F9,F5,F4
I2:  ADDD   F5,F5,F2
I3:  DIVD   F2,F9,F0
```

Figure 2.38 Sample code for superscalar register renaming.

2.8 [20] <2.4> Exercise 2.7 explored simple register renaming: when the hardware register renamer sees a source register, it substitutes the destination T register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available T for it. But superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A simple scalar processor would therefore look up both src register mappings for each instruction, and allocate a new destination mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any dest-to-src relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 2.38. Assume that we would like to simultaneously rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups, and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any inter-instruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 2.39 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code. Assume the table starts out with every entry equal to its index (T0 = 0; T1 = 1, . . .).

2.9 [5] <2.4> If you ever get confused about what a register renamer has to do, go back to the assembly code you're executing, and ask yourself what has to happen for the right result to be obtained. For example, consider a three-way superscalar machine renaming these three instructions concurrently:

```
ADDI    R1, R1, R1
ADDI    R1, R1, R1
ADDI    R1, R1, R1
```

If the value of R1 starts out as 5, what should its value be when this sequence has executed?

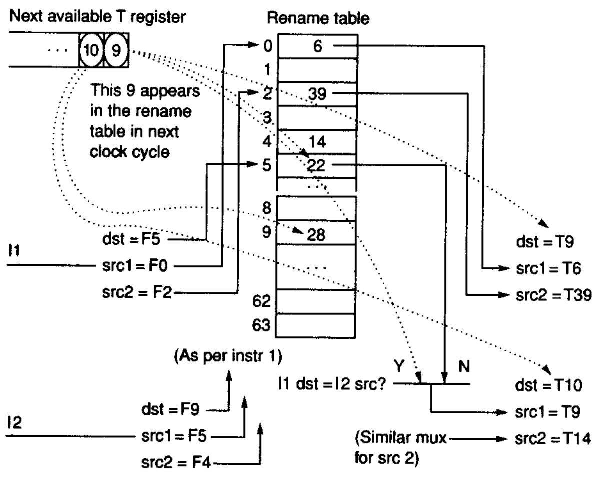


Figure 2.39 Rename table and on-the-fly register substitution logic for superscalar machines. (Note:"src" is source,"dst" is destination.)

```
Loop: LW     R1,0(R2)  ; LW     R3,8(R2)
      <stall>
      <stall>
      ADDI   R10,R1,#1; ADDI   R11,R3,#1
      SW     R1,0(R2)  ; SW     R3,8(R2)
      ADDI   R2,R2,#8
      SUB    R4,R3,R2
      BNZ    R4,Loop
```

Figure 2.40 Sample VLIW code with two adds, two loads, and two stalls.

2.10 [20] <2.4, 2.9> VLIW designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most L cycles later (where L is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider Figure 2.40. If loads have a 1 + 2 cycle latency, unroll this loop once, and show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls. Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

2.11 [10/10/10] <2.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write back) and the code in Figure 2.41. All ops are 1 cycle except LW and SW, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

a. [10] <2.3> How many clock cycles per loop iteration are lost to branch overhead?

b. [10] <2.3> Assume a static branch predictor, capable of recognizing a backwards branch in the decode stage. Now how many clock cycles are wasted on branch overhead?

c. [10] <2.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

2.12 [20/20/20/10/20] <2.4, 2.7, 2.10> Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 2.42. Assume that the ALUs can do all arithmetic ops (MULTD, DIVD, ADDD, ADDI, SUB) and branches, and that the Reservation Station (RS) can dispatch at most one operation to each functional unit per cycle (one op to each ALU plus one memory op to the LD/ST unit).

```
Loop: LW    R1,0(R2)
      ADDI  R1,R1,#1
      SW    R1,0(R2)
      ADDI  R2,R2,#4
      SUB   R4,R3,R2
      BNZ   R4,Loop
```

Figure 2.41 Code loop for Exercise 2.11.

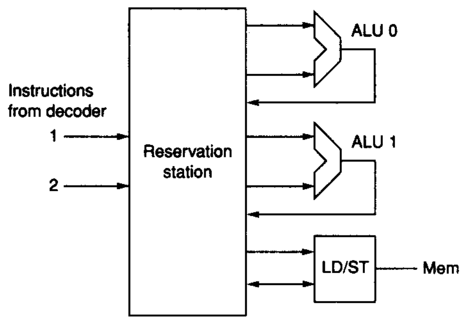


Figure 2.42 An out-of-order microarchitecture.

a. [15] <2.4> Suppose all of the instructions from the sequence in Figure 2.35 are present in the RS, with no renaming having been done. Highlight any instructions in the code where register renaming would improve performance. *Hint:* Look for RAW and WAW hazards. Assume the same functional unit latencies as in Figure 2.35.

b. [20] <2.4> Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle *N*, with latencies as given in Figure 2.35. Show how the RS should dispatch these instructions out-of-order, clock by clock, to obtain optimal performance on this code. (Assume the same RS restrictions as in part (a). Also assume that results must be written into the RS before they're available for use; i.e., no bypassing.) How many clock cycles does the code sequence take?

c. [20] <2.4> Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch what it has. Suppose that the RS is empty. In cycle 0 the first two register-renamed instructions of this sequence appear in the RS. Assume it takes 1 clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 2.2. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?

d. [10] <2.10> If you wanted to improve the results of part (c), which would have helped most: (1) another ALU; (2) another LD/ST unit; (3) full bypassing of ALU results to subsequent operations; (4) cutting the longest latency in half? What's the speedup?

e. [20] <2.7> Now let's consider speculation, the act of fetching, decoding, and executing beyond one or more conditional branches. Our motivation to do this is twofold: the dispatch schedule we came up with in part (c) had lots of nops, and we know computers spend most of their time executing loops (which implies the branch back to the top of the loop is pretty predictable.) Loops tell us where to find more work to do; our sparse dispatch schedule suggests we have opportunities to do some of that work earlier than before. In part (d) you found the critical path through the loop. Imagine folding a second copy of that path onto the schedule you got in part (b). How many more clock cycles would be required to do two loops' worth of work (assuming all instructions are resident in the RS)? (Assume all functional units are fully pipelined.)

Case Study 2: Modeling a Branch Predictor

Concept illustrated by this case study

- Modeling a Branch Predictor

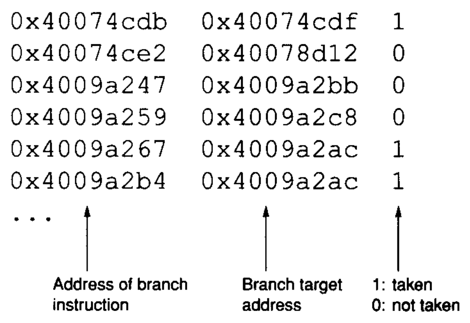
Besides studying microarchitecture techniques, to really understand computer architecture you must also program computers. Getting your hands dirty by directly modeling various microarchitectural ideas is better yet. Write a C or Java program to model a 2,1 branch predictor. Your program will read a series of lines from a file named history.txt (available on the companion CD—see Figure Figure 2.43).

Each line of that file has three data items, separated by tabs. The first datum on each line is the address of the branch instruction in hex. The second datum is the branch target address in hex. The third datum is a 1 or a 0; 1 indicates a taken branch, and 0 indicates not taken. The total number of branches your model will consider is, of course, equal to the number of lines in the file. Assume a direct-mapped BTB, and don't worry about instruction lengths or alignment (i.e., if your BTB has four entries, then branch instructions at 0x0, 0x1, 0x2, and 0x3 will reside in those four entries, but a branch instruction at 0x4 will overwrite BTB[0]). For each line in the input file, your model will read the pair of data values, adjust the various tables per the branch predictor being modeled, and collect key performance statistics. The final output of your program will look like that shown in Figure 2.44.

Make the number of BTB entries in your model a command-line option.

2.13 [20/10/10/10/10/10/10] <2.3> Write a model of a simple four-state branch target buffer with 64 entries.

a. [20] <2.3> What is the overall hit rate in the BTB (the fraction of times a branch was looked up in the BTB and found present)?



```
0x40074cdb   0x40074cdf   1
0x40074ce2   0x40078d12   0
0x4009a247   0x4009a2bb   0
0x4009a259   0x4009a2c8   0
0x4009a267   0x4009a2ac   1
0x4009a2b4   0x4009a2ac   1
. . .
```

Figure 2.43 Sample history.txt input file format.

b. [10] <2.3> What is the overall branch misprediction rate on a cold start (the fraction of times a branch was correctly predicted taken or not taken, regardless of whether that prediction "belonged to" the branch being predicted)?

c. [10] <2.3> Find the most common branch. What was its contribution to the overall number of correct predictions? (*Hint:* Count the number of times that branch occurs in the history.txt file, then track how each instance of that branch fares within the BTB model.)

d. [10] <2.3> How many capacity misses did your branch predictor suffer?

e. [10] <2.3> What is the effect of a cold start versus a warm start? To find out, run the same input data set once to initialize the history table, and then again to collect the new set of statistics.

f. [10] <2.3> Cold-start the BTB 4 more times, with BTB sizes 16, 32, and 64. Graph the resulting five misprediction rates. Also graph the five hit rates.

g. [10] Submit the well-written, commented source code for your branch target buffer model.

```
Exercise 2.13 (a)
        Number of hits BTB: 54390. Total brs: 55493. Hit rate: 99.8%
Exercise 2.13 (b)
        Incorrect predictions: 1562 of 55493, or 2.8%
Exercise 2.13 (c)
        <a simple unix command line shell script will give you the most
                common branch...show how you got it here.>
        Most signif. branch seen 15418 times, out of 55493 tot brs ;
                27.8%
        MS branch = 0x80484ef, correct predictions = 19151 (of 36342
                total correct preds) or 52.7%
Exercise 2.13 (d)
        Total unique branches (1 miss per br compulsory): 121
        Total misses seen: 104.
        So total capacity misses = total misses - compulsory misses = 17
Exercise 2.13 (e)
        Number of hits in BTB: 54390. Total brs: 55493. Hit rate: 99.8%
        Incorrect predictions: 1103 of 54493, or 2.0%
Exercise 2.13 (f)

         BTB Length   mispredict rate
                  1          32.91%
                  2           6.42%
                  4           0.28%
                  8           0.23%
                 16           0.21%
                 32           0.20%
                 64           0.20%
```

Figure 2.44 Sample program output format.