Appendix H
Hardware and Software for VLIW and EPIC

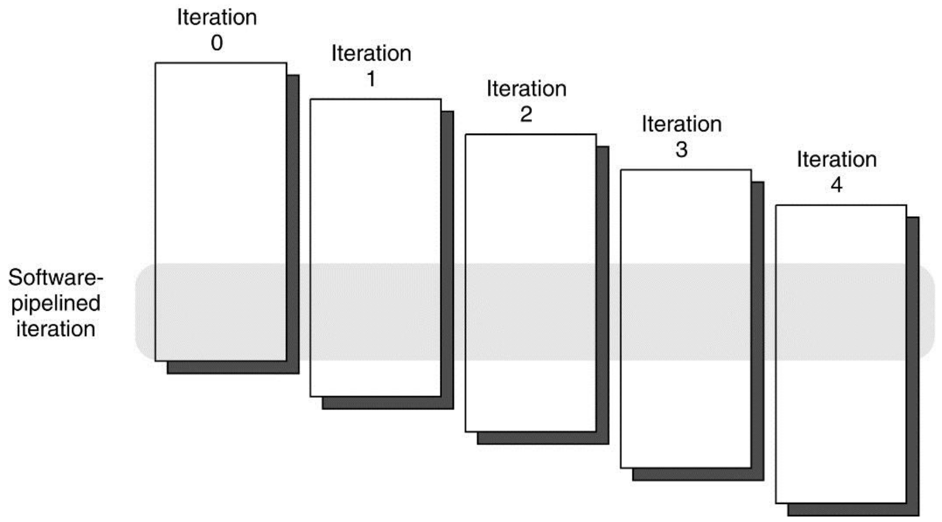


Figure H.1 A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.

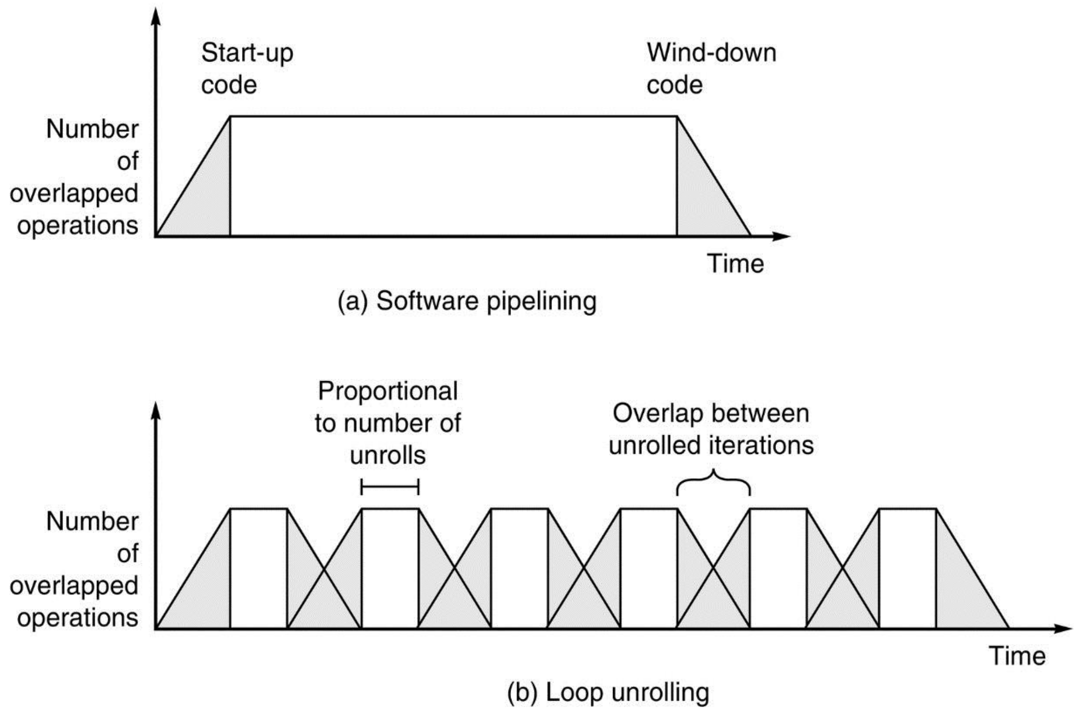

Start-up code
Wind-down code
Number of overlapped operations
Time
(a) Software pipelining
Proportional to number of unrolls
Overlap between unrolled iterations
Number of overlapped operations
Time
(b) Loop unrolling

Figure H.2 The execution pattern for (a) a software-pipelined loop and (b) an unrolled loop. The shaded areas are the times when the loop is not running with maximum overlap or parallelism among instructions. This occurs once at the beginning and once at the end for the software-pipelined loop. For the unrolled loop it occurs *m/n* times if the loop has a total of *m* iterations and is unrolled *n* times. Each block represents an unroll of *n* iterations. Increasing the number of unrollings will reduce the start-up and clean-up overhead. The overhead of one iteration overlaps with the overhead of the next, thereby reducing the impact. The total area under the polygonal region in each case will be the same, since the total number of operations is just the execution rate multiplied by the time.

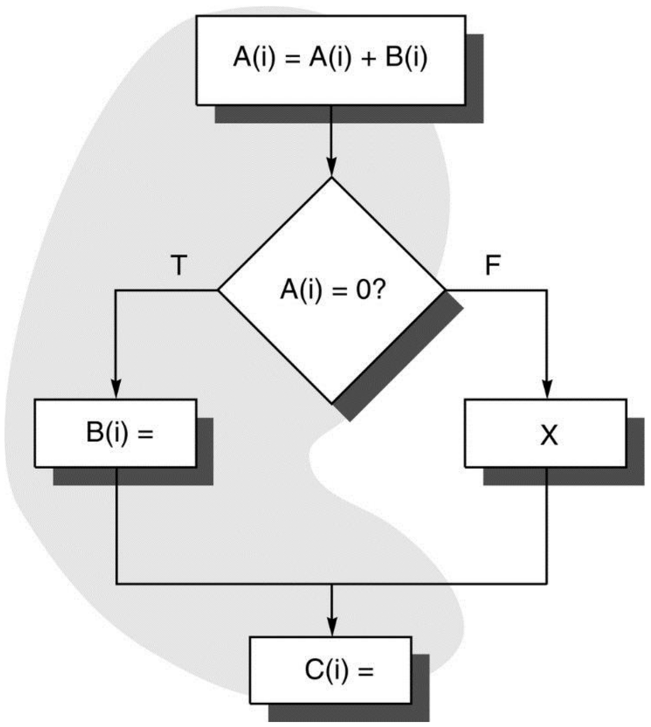


Figure H.3 A code fragment and the common path shaded with gray. Moving the assignments to B or C requires a more complex analysis than for straight-line code. In this section we focus on scheduling this code segment efficiently without hardware assistance. Predication or conditional instructions, which we discuss in the next section, provide another way to schedule this code.

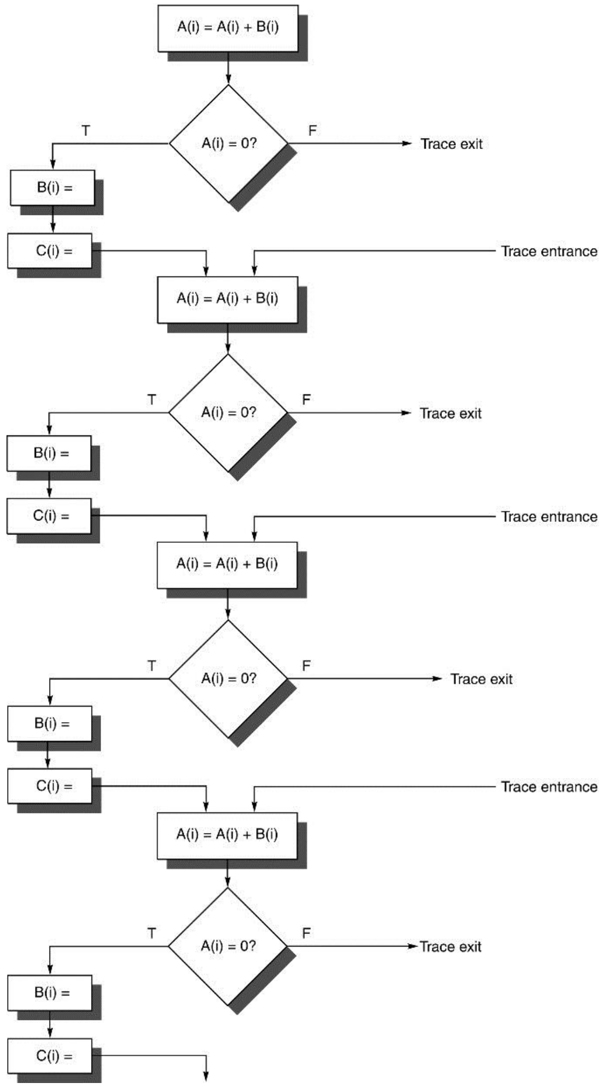


Figure H.4 This trace is obtained by assuming that the program fragment in Figure H.3 is the inner loop and unwinding it four times, treating the shaded portion in Figure H.3 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.

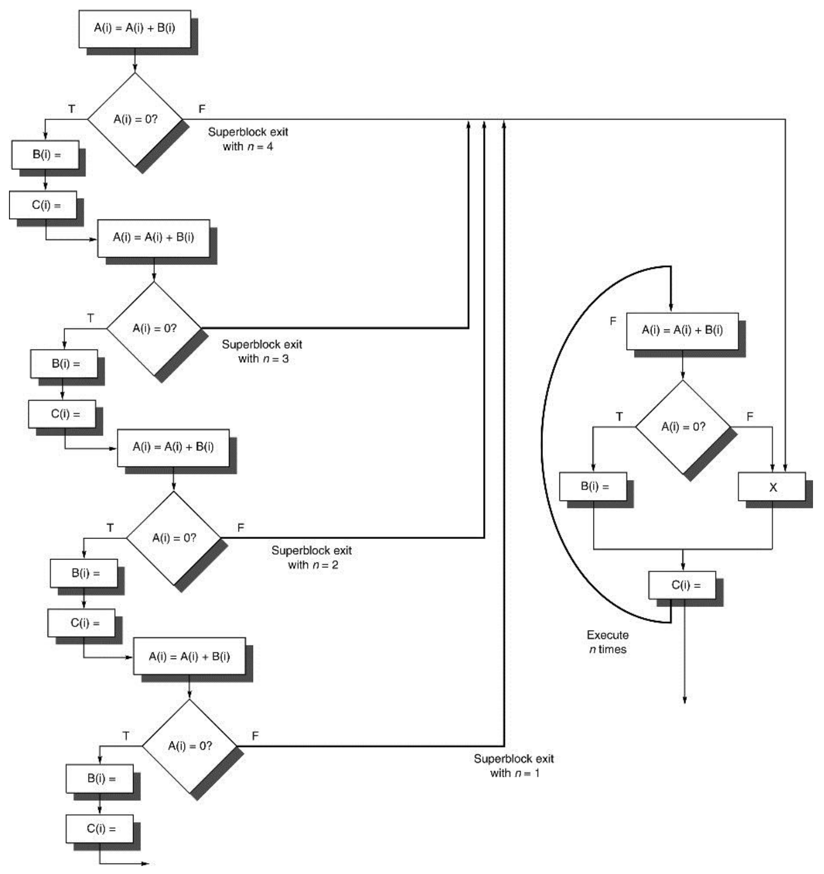
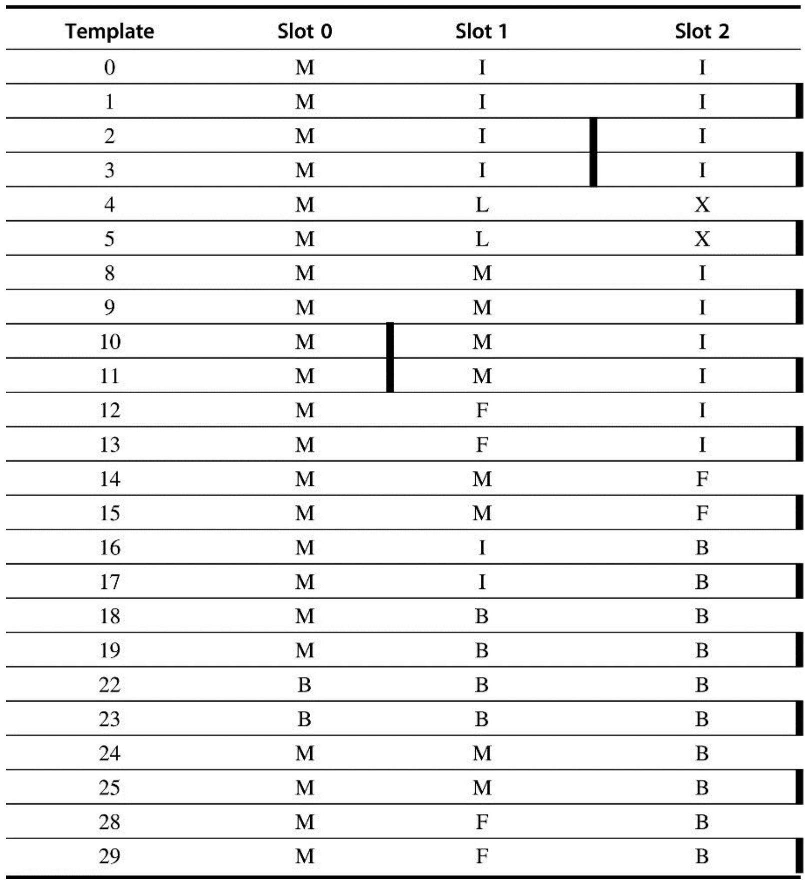

A(i) = A(i) + B(i)
A(i) = 0?
T
F
B(i) =
C(i) =
Superblock exit with n = 4
Superblock exit with n = 3
Superblock exit with n = 2
Superblock exit with n = 1
X
Execute n times

Figure H.5 This superblock results from unrolling the code in Figure H.3 four times and creating a superblock.

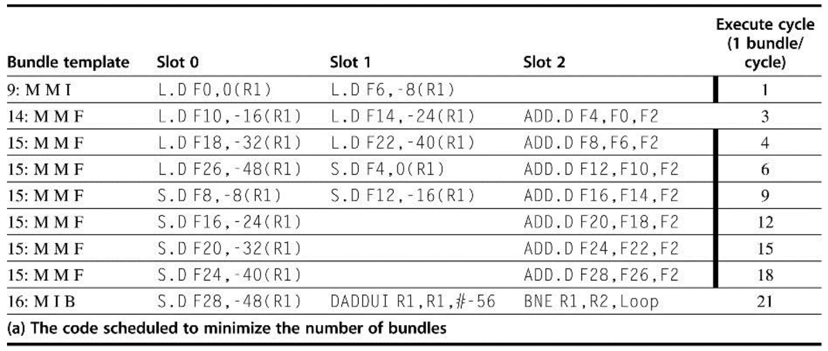
Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	Add, subtract, and, or, compare
	I	Non-ALU	integer Integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	Add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Figure H.6 The five execution unit slots in the IA-64 architecture and what instructions types they may hold are shown. A-type instructions, which correspond to integer ALU instructions, may be placed in either an I-unit or M-unit slot. L + X slots are special, as they occupy two instruction slots; L + X instructions are used to encode 64-bit immediates and a few special instructions. L + X instructions are executed either by the I-unit or the B-unit.



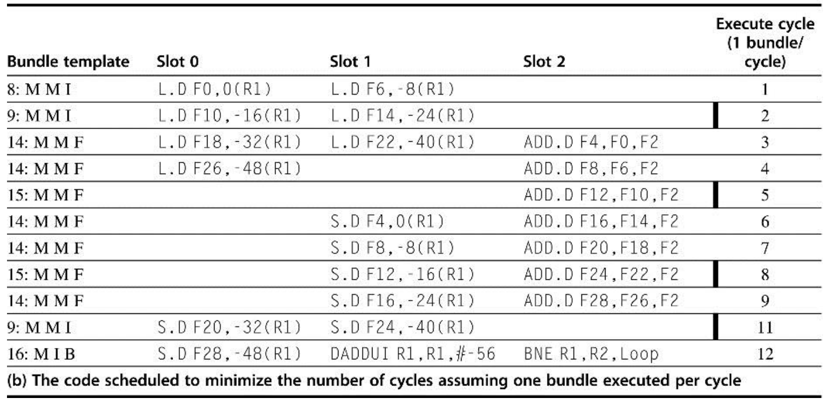
Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

Figure H.7 The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines and may appear within and/or at the end of the bundle. For example, template 9 specifies that the instruction slots are M, M, and I (in that order) and that the only stop is between this bundle and the next. Template 11 has the same type of instruction slots but also includes a stop after the first slot. The L + X format is used when slot 1 is L and slot 2 is X.



Bundle template	Slot 0	Slot 1	Slot 2		Execute cycle (1 bundle/ cycle)
9: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		▌	1
14: M M F	L.D F10,-16(R1)	L.D F14,-24(R1)	ADD.D F4,F0,F2		3
15: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F8,F6,F2	▌	4
15: M M F	L.D F26,-48(R1)	S.D F4,0(R1)	ADD.D F12,F10,F2	▌	6
15: M M F	S.D F8,-8(R1)	S.D F12,-16(R1)	ADD.D F16,F14,F2	▌	9
15: M M F	S.D F16,-24(R1)		ADD.D F20,F18,F2	▌	12
15: M M F	S.D F20,-32(R1)		ADD.D F24,F22,F2	▌	15
15: M M F	S.D F24,-40(R1)		ADD.D F28,F26,F2	▌	18
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop		21

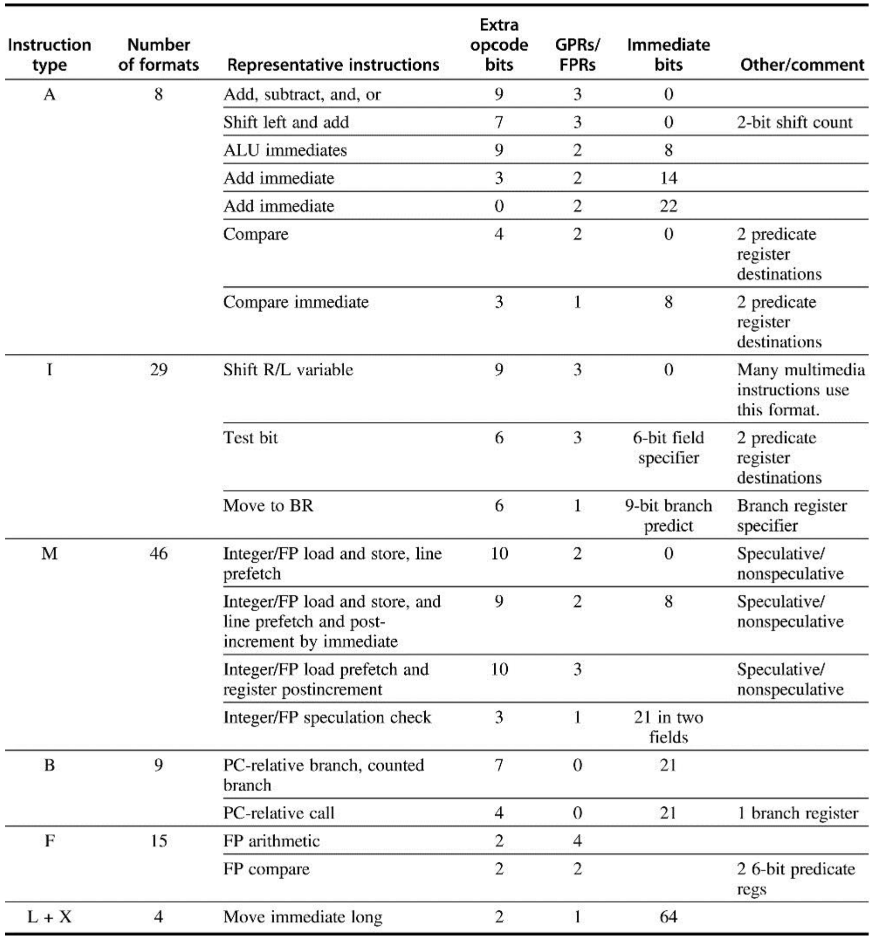
(a) The code scheduled to minimize the number of bundles



Bundle template	Slot 0	Slot 1	Slot 2		Execute cycle (1 bundle/ cycle)
8: M M I	L.D F0,0(R1)	L.D F6,-8(R1)			1
9: M M I	L.D F10,-16(R1)	L.D F14,-24(R1)		▌	2
14: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2		3
14: M M F	L.D F26,-48(R1)		ADD.D F8,F6,F2		4
15: M M F			ADD.D F12,F10,F2	▌	5
14: M M F		S.D F4,0(R1)	ADD.D F16,F14,F2		6
14: M M F		S.D F8,-8(R1)	ADD.D F20,F18,F2		7
15: M M F		S.D F12,-16(R1)	ADD.D F24,F22,F2	▌	8
14: M M F		S.D F16,-24(R1)	ADD.D F28,F26,F2		9
9: M M I	S.D F20,-32(R1)	S.D F24,-40(R1)		▌	11
16: M I B	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop		12

(b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Figure H.8 The IA-64 instructions, including bundle bits and stops, for the unrolled version of x[i] = x[i] + s, when unrolled seven times and scheduled (a) to minimize the number of instruction bundles and (b) to minimize the number of cycles (assuming that a hazard stalls an entire bundle). Blank entries indicate unused slots, which are encoded as no-ops. The absence of stops indicates that some bundles could be executed in parallel. Minimizing the number of bundles yields 9 bundles versus the 11 needed to minimize the number of cycles. The scheduled version executes in just over half the number of cycles. Version (a) fills 85% of the instruction slots, while (b) fills 70%. The number of empty slots in the scheduled code and the use of bundles may lead to code sizes that are much larger than other RISC architectures. Note that the branch in the last bundle in both sequences depends on the DADD in the same bundle. In the IA-64 instruction set, this sequence would be coded as a setting of a predication register and a branch that would be predicated on that register. Normally, such dependent operations could not occur in the same bundle, but this case is one of the exceptions mentioned earlier.



Instruction type	Number of formats	Representative instructions	Extra opcode bits	GPRs/ FPRs	Immediate bits	Other/comment
A	8	Add, subtract, and, or	9	3	0	
		Shift left and add	7	3	0	2-bit shift count
		ALU immediates	9	2	8	
		Add immediate	3	2	14	
		Add immediate	0	2	22	
		Compare	4	2	0	2 predicate register destinations
		Compare immediate	3	1	8	2 predicate register destinations
I	29	Shift R/L variable	9	3	0	Many multimedia instructions use this format.
		Test bit	6	3	6-bit field specifier	2 predicate register destinations
		Move to BR	6	1	9-bit branch predict	Branch register specifier
M	46	Integer/FP load and store, line prefetch	10	2	0	Speculative/ nonspeculative
		Integer/FP load and store, and line prefetch and post-increment by immediate	9	2	8	Speculative/ nonspeculative
		Integer/FP load prefetch and register postincrement	10	3		Speculative/ nonspeculative
		Integer/FP speculation check	3	1	21 in two fields	
B	9	PC-relative branch, counted branch	7	0	21	
		PC-relative call	4	0	21	1 branch register
F	15	FP arithmetic	2	4		
		FP compare	2	2		2 6-bit predicate regs
L + X	4	Move immediate long	2	1	64	

Figure H.9 A summary of some of the instruction formats of the IA-64 ISA. The major opcode bits and the guarding predication register specifier add 10 bits to every instruction. The number of formats indicated for each instruction class in the second column (a total of 111) is a strict interpretation: A different use of a field, even of the same size, is considered a different format. The number of formats that actually have *different field sizes* is one-third to one-half as large. Some instructions have unused bits that are reserved; we have not included those in this table. Immediate bits include the sign bit. The branch instructions include prediction bits, which are used when the predictor does not have a valid prediction. Only one of the many formats for the multimedia instructions is shown in this table.

Instruction	Latency
Integer load	1
Floating-point load	5–9
Correctly predicted taken branch	0–3
Mispredicted branch	6
Integer ALU operations	0
FP arithmetic	4

Figure H.10 The latency of some typical instructions on Itanium 2. The latency is defined as the smallest number of intervening instructions between two dependent instructions. Integer load latency assumes a hit in the first-level cache. FP loads always bypass the primary cache, so the latency is equal to the access time of the second-level cache. There are some minor restrictions for some of the functional units, but these primarily involve the execution of infrequent instructions.

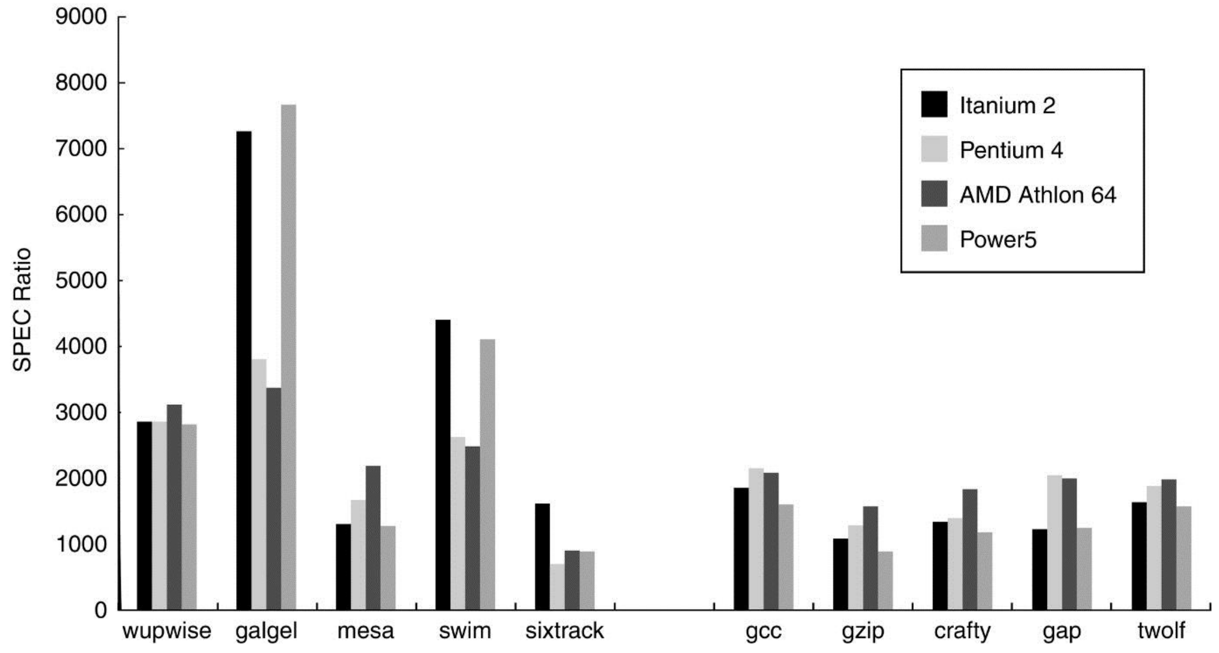


Figure H.11 The performance of four multiple-issue processors for five SPECfp and SPECint benchmarks. The clock rates of the four processors are Itanium 2 at 1.5 GHz, Pentium 4 Extreme Edition at 3.8 GHz, AMD Athlon 64 at 2.8 GHz, and the IBM Power5 at 1.9 GHz.