CHAPTER 21

The IA-64 Architecture

KEY POINTS

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

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

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

21.1 MOTIVATION

The basic concepts underlying IA-64 are:

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

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

Table 21.1 summarizes key differences between IA-64 and a traditional superscalar approach.

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

Table 21.1 Traditional Superscalar versus IA-64 Architecture

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

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

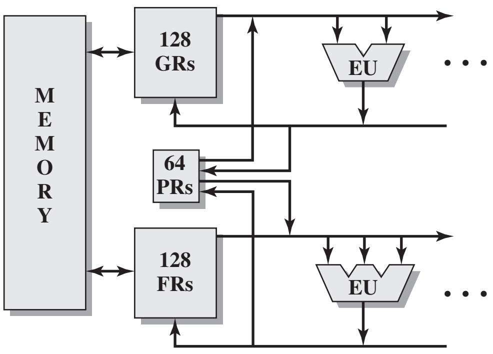
Processor designers have few choices in how to use this glut of transistors. One approach is to dump those extra transistors into bigger on-chip caches. Bigger caches can improve performance to a degree but eventually reach a point of diminishing returns, in which larger caches result in tiny improvements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

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

21.2 GENERAL ORGANIZATION

As with any processor architecture, IA-64 can be implemented in a variety of organizations. Figure 21.1 suggests in general terms the organization of an IA-64 machine. The key features are:

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



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

Figure 21.1 General Organization for IA-64 Architecture

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

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

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

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

Each IA-64 instruction is categorized into one of six types. Table 21.2 lists the instruction types and the execution unit types on which they may be executed. The extended (X) instruction type includes instructions in which two slots in a bundle are used to encode the instruction, allowing for more information than fits into a 41-bit instruction (slots and bundles are explained in the next section).

Table 21.2 Relationship Between Instruction Type and Execution Unit Type

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

21.3 PREDICATION, SPECULATION, AND SOFTWARE PIPELINING

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

Instruction Format

IA-64 defines a 128-bit **bundle** that contains three instructions, called **syllables**, and a template field (Figure 21.2a). The processor can fetch instructions one or more bundles at a time; each bundle fetch brings in three instructions. The template field

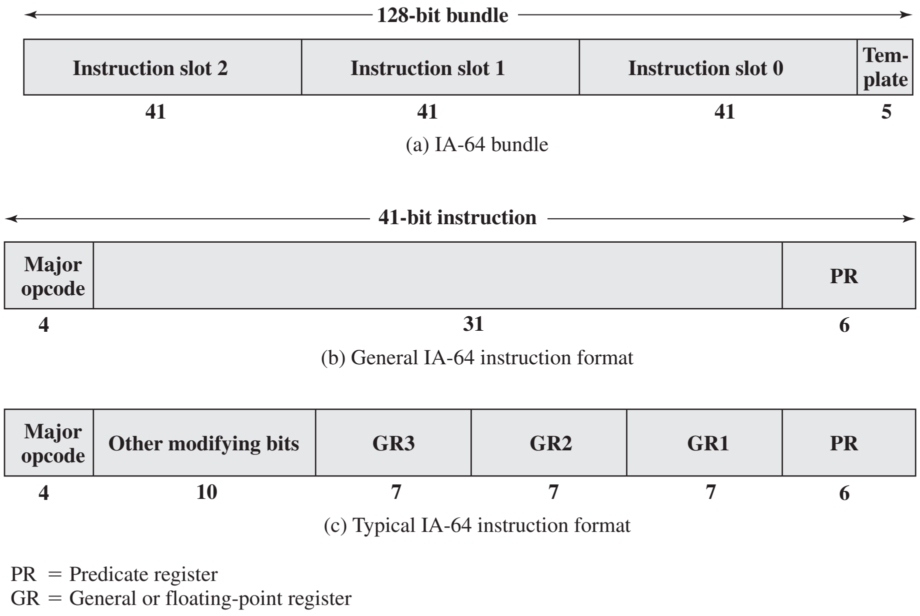


Figure 21.2 IA-64 Instruction Format

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

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

Table 21.3 shows the interpretation of the possible values for the 5-bit template field (some values are reserved and not in current use). The template value accomplishes two purposes:

Table 21.3 Template Field Encoding and Instruction Set Mapping

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

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

Each instruction has a fixed-length 41-bit format (Figure 21.2b). This is somewhat longer than the traditional 32-bit length found on RISC and RISC superscalar machines (although it is much shorter than the 118-bit micro-operation of the Pentium 4). Two factors lead to the additional bits. First, IA-64 makes use of more registers than a typical RISC machine: 128 integer and 128 floating-point registers. Second, to accommodate the predicated execution technique, an IA-64 machine includes 64 predicate registers. Their use is explained subsequently.

Figure 21.2c shows in more detail the typical instruction format. All instructions include a 4-bit major opcode and a reference to a predicate register. Although the major opcode field can only discriminate among 16 possibilities, the interpretation of the major opcode field depends on the template value and the location of the instruction within a bundle (Table 21.3), thus affording more possible opcodes. Typical instructions also include three fields to reference registers, leaving 10 bits for other information needed to fully specify the instruction.

Assembly-Language Format

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

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

where

qp	Specifies a 1-bit predicate register used to qualify the instruction. If the value of the register is 1 (true) at execution time, the instruction executes and the result is committed in hardware. If the value is false, the result of the instruction is not committed but is discarded. Most IA-64 instructions may be qualified by a predicate but need not be. To account for an instruction that is not predicated, the qp value is set to 0 and predicate register zero always has the constant value of 1.
mnemonic	Specifies the name of an IA-64 instruction.
comp	Specifies one or more instruction completers, separated by periods, which are used to qualify the mnemonic. Not all instructions require the use of a completer.
dest	Specifies one or more destination operands, with the typical case being a single destination.
srcs	Specifies one or more source operands. Most instructions have two or more source operands.

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

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

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

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

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

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

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

- Predication
- Control speculation
- Data speculation
- Software pipelining

Figure 21.3, based on a figure in [HALF97], illustrates the first two of these techniques, which are discussed in this subsection and the next.

Predicated Execution

Predication is a technique whereby the compiler determines which instructions may execute in parallel. In the process, the compiler eliminates branches from the program by using conditional execution. A typical example in a high-level language is an **if-then-else** instruction. A traditional compiler inserts a conditional branch at the **if** point of this construct. If the condition has one logical outcome, the branch is not taken and the next block of instructions is executed, representing the **then** path; at the end of this path is an unconditional branch around the next block, representing the **else** path. If the condition has the other logical outcome, the branch is taken around the **then** block of instructions and execution continues at the **else** block of instructions. The two instruction streams join together after the end of the **else** block. An IA-64 compiler instead does the following (Figure 21.3a):

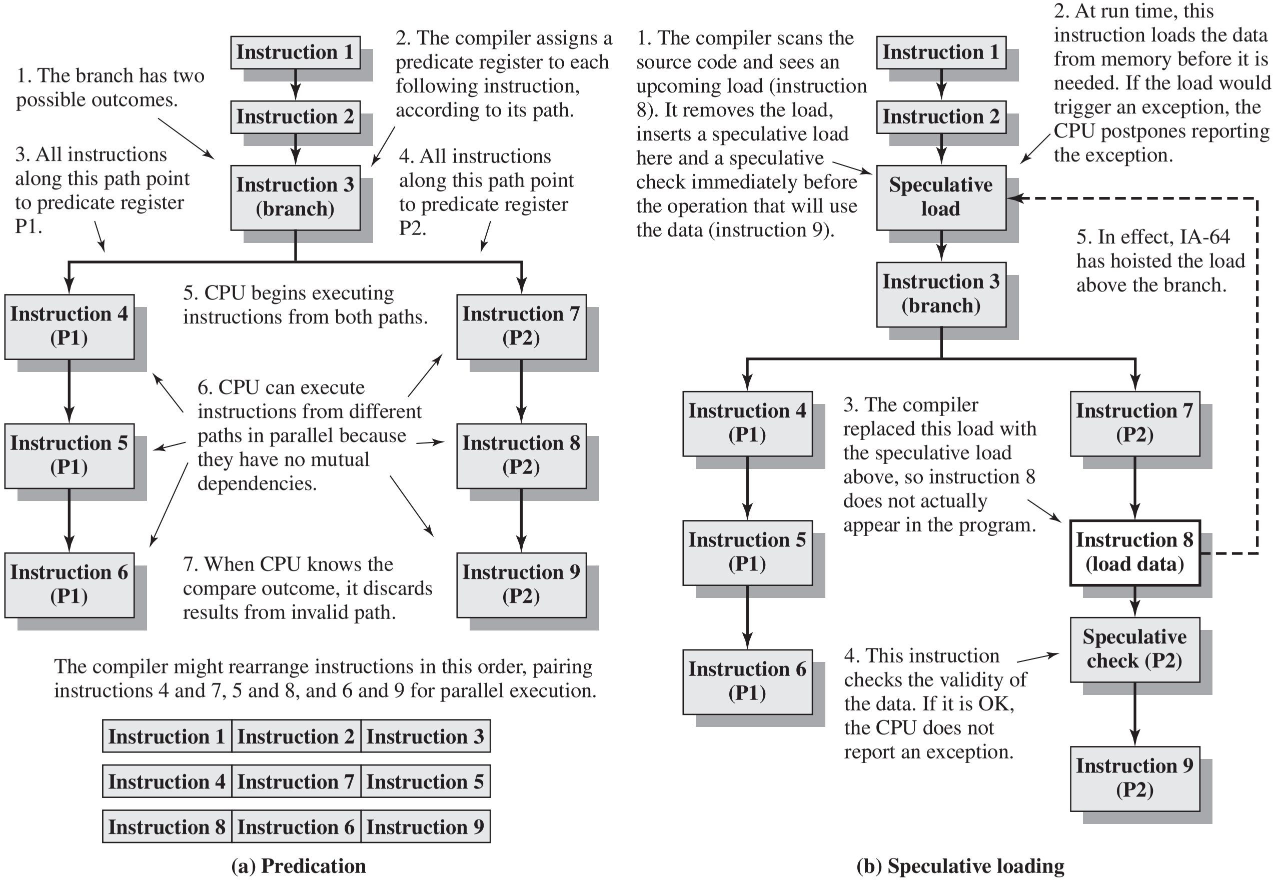

1. The branch has two possible outcomes.
Instruction 1
Instruction 2
Instruction 3 (branch)
2. The compiler assigns a predicate register to each following instruction, according to its path.
3. All instructions along this path point to predicate register P1.
4. All instructions along this path point to predicate register P2.
Instruction 4 (P1)
Instruction 5 (P1)
Instruction 6 (P1)
5. CPU begins executing instructions from both paths.
6. CPU can execute instructions from different paths in parallel because they have no mutual dependencies.
7. When CPU knows the compare outcome, it discards results from invalid path.
Instruction 7 (P2)
Instruction 8 (P2)
Instruction 9 (P2)
The compiler might rearrange instructions in this order, pairing instructions 4 and 7, 5 and 8, and 6 and 9 for parallel execution.
Instruction 1 | Instruction 2 | Instruction 3
Instruction 4 | Instruction 7 | Instruction 5
Instruction 8 | Instruction 6 | Instruction 9
(a) Predication
1. The compiler scans the source code and sees an upcoming load (instruction 8). It removes the load, inserts a speculative load here and a speculative check immediately before the operation that will use the data (instruction 9).
Instruction 1
Instruction 2
Speculative load
Instruction 3 (branch)
2. At run time, this instruction loads the data from memory before it is needed. If the load would trigger an exception, the CPU postpones reporting the exception.
5. In effect, IA-64 has hoisted the load above the branch.
Instruction 4 (P1)
Instruction 5 (P1)
Instruction 6 (P1)
3. The compiler replaced this load with the speculative load above, so instruction 8 does not actually appear in the program.
4. This instruction checks the validity of the data. If it is OK, the CPU does not report an exception.
Instruction 7 (P2)
Instruction 8 (load data)
Speculative check (P2)
Instruction 9 (P2)
(b) Speculative loading

Figure 21.3 IA-64 Predication and Speculative Loading

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

As an example, consider the following source code:

Source Code:

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

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

Assembly Code:

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

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

Figure 21.4 shows a flow diagram of this assembly code. This diagram breaks the assembly language program into separate blocks of code. For each block that

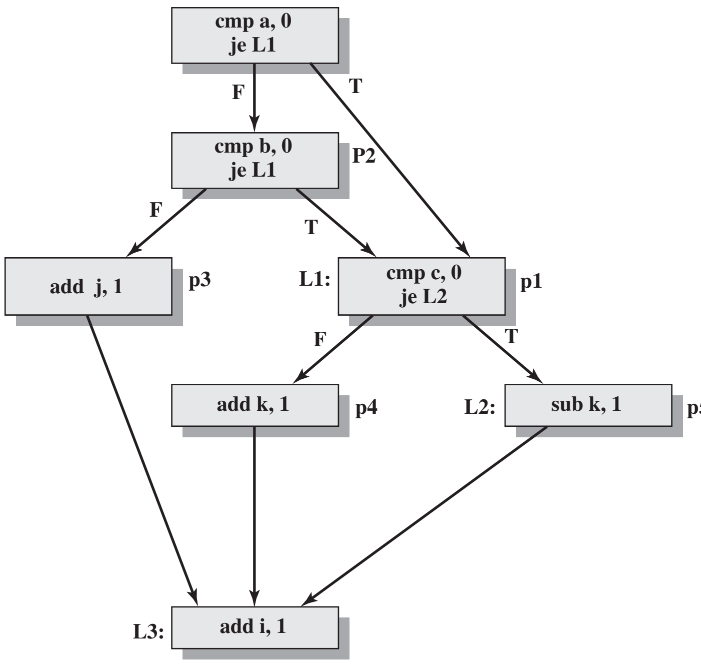


Figure 21.4 Example of Predication

executes conditionally, the compiler can assign a predicate. These predicates are indicated in Figure 21.4. Assuming that all of these predicates have been initialized to false, the resulting IA-64 assembly code is as follows:

Predicated Code

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

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

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

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

Control Speculation

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

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

How then, can we move the load above the branch? The solution specified in IA-64 is the control speculation, which separates the load behavior (delivering the value) from the exception behavior (Figure 21.3b). A load instruction in the original program is replaced by two instructions:

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

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

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

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

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

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

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

We can't simply move the load instruction above the branch instruction, as is, because the load instruction may cause an exception (e.g., r5 may contain a null pointer). Instead, we convert the load to a speculative load, ld8.s, and then move it. The speculative load doesn't immediately signal an exception when detected; it just records that fact by setting the NaT bit for the target register (in this case, r1). The speculative load now executes unconditionally at least two cycles prior to the branch. The chk.s instruction then checks to see if the NaT bit is set on r1. If not, execution simply falls through to the next instruction. If so, a branch is taken to a recovery program. Note that the branch, check, and add instructions are all shown as being executed in the same clock cycle. However, the hardware ensures that the results produced by the speculative load do not update the application state (change the contents of r1 and r2) unless two conditions occur: the branch is not taken (p1 = 0) and the check does not detect a deferred exception (r1.NaT = 0).

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

Let us look at a more complex example, used by Intel and HP to benchmark predicated programs and to illustrate the use of speculative loads, known as the Eight Queens Problem. The objective is to arrange eight queens on a chessboard so that no queen threatens any other queen. Figure 21.5a shows one solution. The key line of source code, in an inner loop, is the following:

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

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

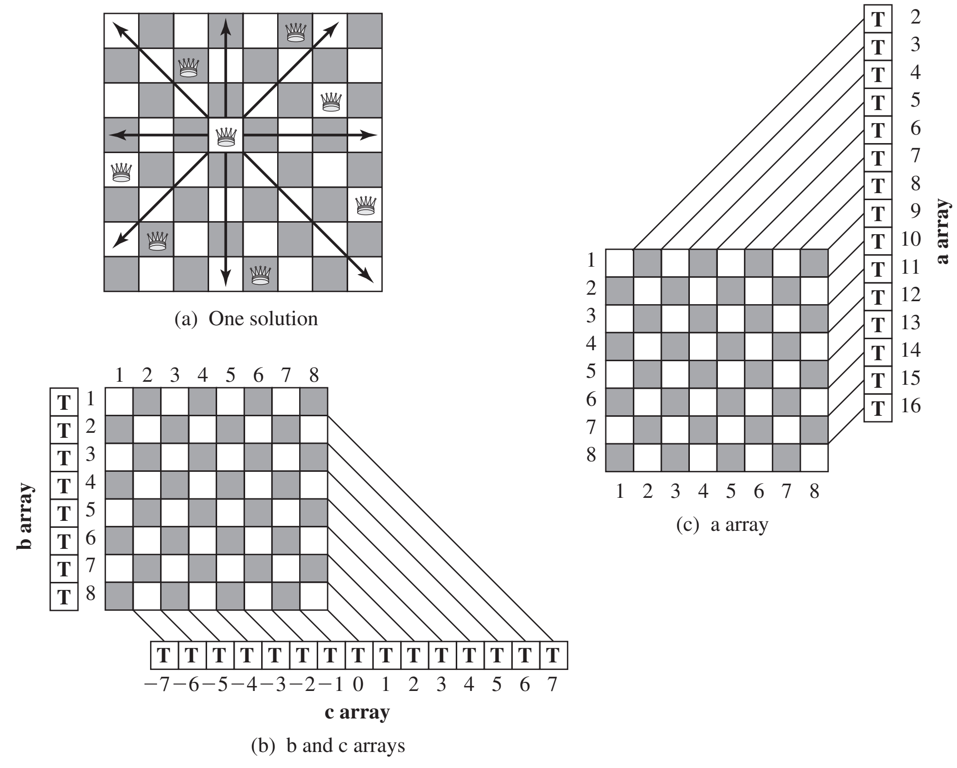


Figure 21.5 The Eight Queens Problem

The queen conflict tracking mechanism consists of three Boolean arrays that track queen status for each row and diagonal. TRUE means no queen is on that row or diagonal; FALSE means a queen is already there. Figures 21.5b and c show the mapping of the arrays to the chessboard. All array elements are initialized to TRUE. The B array elements 1 through 8 correspond to rows 1 through 8 on the board. A queen in row *n* sets b[*n*] to FALSE. C array elements are numbered from −7 to 7 and correspond to the difference between column and row numbers, which defines the diagonals that go down to the right. A queen at column 1, row 1 sets c[0] to FALSE. A queen at column 1, row 8 sets c[−7] to FALSE. The A array elements are numbered 2-16 and correspond to the sum of the column and row. A queen placed in column 1, row 1 sets a[2] to FALSE. A queen placed in column 3, row 5 sets a[8] to FALSE.

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

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

Assembly Code:

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

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

Using speculative loads and predicated execution yields the following:

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

Code with Speculation and Predication:

```
(7)       cmp.eq p1, p2 = 1, r2
(8)  (p2) br L2
(9)       chk.s r4, recovery_a   // fixup for
                                    loading a
(10)      cmp.eq p3, p4 = 1, r4
(11) (p4) br L2
(12)      chk.s r6, recovery_b   // fixup for
                                    loading b
(13)      cmp.eq p5, p6 = 1, r5
(14) (p6) br L2
(15) L1:  <code for then path>
(16) L2:  <code for else path>
```

The assembly program breaks down into three basic blocks of code, each of which is a load followed by a conditional branch. The address-setting instructions 4 and 7 in the Pentium assembly code are simple arithmetic calculations; these can be done anytime, so the compiler moves these up to the top. Then the compiler is faced with three simple blocks, each of which consists of a load, a condition calculation, and a conditional branch. There seems little hope of doing anything in parallel here. Furthermore, if we assume that the load takes two or more clock cycles, we have some wasted time before the conditional branch can be executed. What the compiler can do is hoist the second and third loads (instructions 5 and 8 in the Pentium code) above all the branches. This is done by putting a speculative load up top (IA-64 instructions 5 and 6) and leaving a check in the original code block (IA-64 instructions 9 and 12).

This transformation makes it possible to execute all three loads in parallel and to begin the loads early so as to minimize or avoid delays due to load latencies. The compiler can go further by more aggressive use of predication, and eliminate two of the three branches:

Revised Code with Speculation and Predication:

```
(1)               mov r1 = &b[j]
(2)               mov r3 = &a[i + j]
(3)               mov r5 = &c[i - j + 7]
(4)               ld8 r2 = [r1]
(5)               ld8.s r4 = [r3]
(6)               ld8.s r6 = [r5]
(7)               cmp.eq p1, p2 = 1, r2
(8)        (p1)   chk.s r4, recovery_a
(9)        (p1)   cmp.eq p3, p4 = 1, r4
(10)       (p3)   chk.s r6, recovery_b
(11)       (p3)   cmp.eq p5, p4 = 1, r5
(12)       (p6)   br L2
(13) L1:          <code for then path>
(14) L2:          <code for else path>
```

We already had a compare that generated two predicates. In the revised code, instead of branching on the false predicate, the compiler qualifies execution of both the check and the next compare on the true predicate. The elimination of two branches means the elimination of two potential mispredictions, so that the savings is more than just two instructions.

Data Speculation

In a control speculation, a load is moved earlier in a code sequence to compensate for load latency, and a check is made to assure that an exception doesn't occur if it subsequently turns out that the load was not taken. In data speculation, a load is moved before a store instruction that might alter the memory location that is the source of the load. A subsequent check is made to assure that the load receives the proper memory value. To explain the mechanism, we use an example taken from [INTE00a, Volume 1].

Consider the following program fragment:

```
st8  [r4] = r12       // Cycle 0
ld8  r6 = [r8] ;;     // Cycle 0
add  r5 = r6, r7 ;;   // Cycle 2
st8  [r18] = r5       // Cycle 3
```

As written, the code requires four instruction cycles to execute. If registers r4 and r8 do not contain the same memory address, then the store through r4 cannot affect the value at the address contained in r8; under this circumstance, it is safe to reorder the load and store to more quickly bring the value into r6, which is needed subsequently. However, because the addresses in r4 and r8 may be the same or overlap, such a swap is not safe. IA-64 overcomes this problem with the use of a technique known as advanced load.

```
ld8.a r6 = [r8] ;;    // Cycle -2 or earlier;
                         advanced load
// other instructions
st8 [r4] = r12        // Cycle 0
ld8.c r6 = [r8]       // Cycle 0; check load
add r5 = r6, r7 ;;    // Cycle 0
st8 [r18] = r5        // Cycle 1
```

Here we have moved the ld instruction earlier and converted it into an advanced load. In addition to performing the specified load, the ld8.a instruction writes its source address (address contained in r8) to a hardware data structure known as the Advanced Load Address Table (ALAT). Each IA-64 store instruction checks the ALAT for entries that overlap with its target address; if a match is found, the ALAT entry is removed. When the original ld8 is converted to an ld8.a instruction and moved, the original position of that instruction is replaced with a check load instruction, ld8.c. When the check load is executed, it checks the

ALAT for a matching address. If one is found, no store instruction between the advanced load and the check load has altered the source address of the load, and no action is taken. However, if the check load instruction does not find a matching ALAT entry, then the load operation is performed again to assure the correct result.

We may also want to speculatively execute instructions that are data dependent on a load instruction, together with the load itself. Starting with the same original program, suppose we move up both the load and the subsequent add instruction:

```
      ld8.a r6 = [r8] ;;        // Cycle -3 or earlier;
                                   advanced load
      // other instructions
      add r5 = r6, r7           // Cycle -1; add that uses r6
      // other instructions
      st8 [r4] = r12            // Cycle 0
      chk.a r6, recover         // Cycle 0; check
  back:                         // return point from jump to
                                   recover
      st8 [r18] = r5            // Cycle 0
```

Here we use a chk.a instruction rather than an ld8.c instruction to validate the advanced load. If the chk.a instruction determines that the load has failed, it cannot simply reexecute the load; instead, it branches to a recovery routine to clean up:

```
Recover:
    ld8 r6 = [r8] ;;       // reload r6 from [r8]
    add r5 = r6, r7 ;;     // re-execute the add
    br back                // jump back to main code
```

This technique is effective only if the loads and stores involved have little chance of overlapping.

Software Pipelining

Consider the following loop:

```
L1: ld4 r4 = [r5], 4 ;;  // Cycle 0; load postinc 4
    add r7 = r4, r9 ;;   // Cycle 2
    st4 [r6] = r7, 4     // Cycle 3; store postinc 4
    br.cloop L1 ;;       // Cycle 3
```

This loop adds a constant to one vector and stores the result in another vector (e.g. y[i] = x[i] + c). The ld4 instruction loads 4 bytes from memory. The qualifier ", 4" at the end of the instruction signals that this is the base update form of the load instruction; the address in r5 is incremented by 4 after the load takes place. Similarly, the st4 instruction stores four bytes in memory and the address in r6 is incremented by four after the store. The br.cloop instruction, known as a counted loop branch, uses the Loop Count (LC) application register. If the LC register is greater than zero, it is decremented and the branch is taken. The initial value in LC is the number of iterations of the loop.

Notice that in this program, there is virtually no opportunity for instruction-level parallelism within a loop. Further, the instructions in iteration x are all executed before iteration $x + 1$ begins. However, if there is no address conflict between the load and store (r5 and r6 point to nonoverlapping memory locations), then utilization could be improved by moving independent instructions from iteration $x + 1$ to iteration x. Another way of saying this is that if we unroll the loop code by actually writing out a new set of instructions for each iteration, then there is opportunity to increase parallelism. Let's see what could be done with five iterations:

```
ld4  r32  = [r5], 4 ;;  // Cycle 0
ld4  r33  = [r5], 4 ;;  // Cycle 1
ld4  r34  = [r5], 4     // Cycle 2
add  r36  = r32, r9 ;;  // Cycle 2
ld4  r35  = [r5], 4     // Cycle 3
add  r37  = r33, r9     // Cycle 3
st4  [r6] = r36, 4 ;;   // Cycle 3
ld4  r36  = [r5], 4     // Cycle 3
add  r38  = r34, r9     // Cycle 4
st4  [r6] = r37, 4 ;;   // Cycle 4
add  r39  = r35, r9     // Cycle 5
st4  [r6] = r38, 4 ;;   // Cycle 5
add  r40  = r36, r9     // Cycle 6
st4  [r6] = r39, 4 ;;   // Cycle 6
st4  [r6] = r40, 4 ;;   // Cycle 7
```

This program completes 5 iterations in 7 cycles, compared with 20 cycles in the original looped program. This assumes that there are two memory ports so that a load and a store can be executed in parallel. This is an example of software pipelining, analogous to hardware pipelining. Figure 21.6 illustrates the process. Parallelism is achieved by grouping together instructions from different iterations. For this to work, the temporary registers used inside the loop must be changed for each iteration to avoid register conflicts. In this case, two temporary registers are used (r4 and r7 in the original program). In the expanded program, the register number of each register is incremented for each iteration, and the register numbers are initialized sufficiently far apart to avoid overlap.

Figure 21.6 shows that the software pipeline has three phases. During the **prolog phase**, a new iteration is initiated with each clock cycle and the pipeline gradually fills

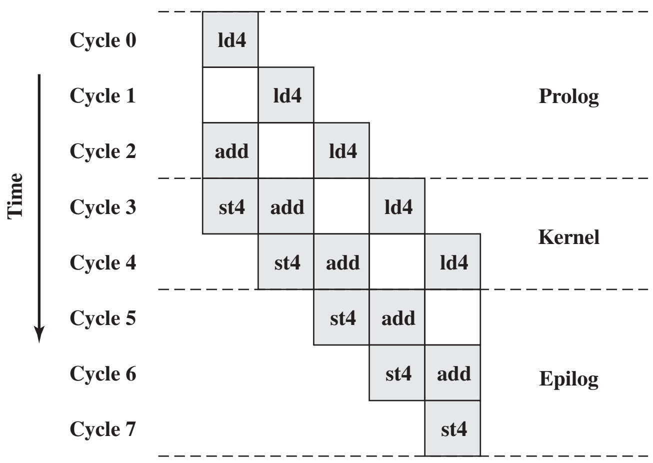


Figure 21.6 Software Pipelining Example

up. During the **kernel phase**, the pipeline is full, achieving maximum parallelism. For our example, three instructions are performed in parallel during the kernel phase, but the width of the pipeline is four. During the **epilog phase**, one iteration completes with each clock cycle.

Software pipelining by loop unrolling places a burden on the compiler or programmer to assign register names properly. Further, for long loops with many iterations, the unrolling results in a significant expansion in code size. For an indeterminate loop (total iterations unknown at compile time), the task is further complicated by the need to do a partial unroll and then to control the loop count. IA-64 provides hardware support to perform software pipelining with no code expansion and with minimal burden on the compiler. The key features that support software pipelining are:

- **Automatic register renaming:** A fixed-sized area of the predicate and floating-point register files (p16 to p63; fr32 to fr127) and a programmable-sized area of the general register file (maximum range of r32 to r127) are capable of rotation. This means that during each iteration of a software-pipeline loop, register references within these ranges are automatically incremented. Thus, if a loop makes use of general register r32 on the first iteration, it automatically makes use of r33 on the second iteration, and so on.
- **Predication:** Each instruction in the loop is predicated on a rotating predicate register. The purpose of this is to determine whether the pipeline is in prolog, kernel, or epilog phase, as explained subsequently.
- **Special loop terminating instructions:** These are branch instructions that cause the registers to rotate and the loop count to decrement.

This is a relatively complex topic; here, we present an example that illustrates some of the IA-64 software pipelining capabilities. We take the original loop program from this section and show how to program it for software pipelining, assuming a loop count of 200 and that there are two memory ports:

```
              mov lc = 199         // set    loop    count
                                      register to 199,
                                   // which equals loop
                                      count - 1
              mov ec = 4           // set   epilog   count
                                      register equal
                                   // to number of epilog
                                      stages + 1
              mov pr.rot = 1<<16;; // pr16 = 1; rest = 0
L1:  (p16)    ld4 r32 = [r5], 4    // Cycle 0
     (p17)    ---                  // Empty stage
     (p18)    add r35 = r34, r9    // Cycle 0
     (p19)    st4 [r6] = r36, 4    // Cycle 0
              br.ctop L1 ;;        // Cycle 0
```

We summarize the key points related to this program:

1. The loop body is partitioned into multiple *stages*, with zero or more instructions per stage.
2. Execution of the loop proceeds through three phases. During the prolog phase, a new loop iteration is started each time around, adding one stage to the pipeline. During the kernel phase, one loop iteration is started and one completed each time around; the pipeline is full, with the maximum number of stages active. During the epilog phase, no new iterations are started and one iteration is completed each time around, draining the software pipeline.
3. A predicate is assigned to each stage to control the activation of the instructions in that stage. During the prolog phase, p16 is true and p17, p18, and p19 are false for the first iteration. For the second iteration, p16 and p17 are true; during the third iteration p16, p17, and p18 are true. During the kernel phase, all predicates are true. During the epilog phase, the predicates are turned to false one by one, beginning with p16. The changes in predicate values are achieved by predicate register rotation.
4. All general registers with register numbers greater than 31 are rotated with each iteration. Registers are rotated toward larger register numbers in a wraparound fashion. For example, the value in register x will be located in register $x + 1$ after one rotation; this is achieved not by moving values but by hardware renaming of registers. Thus, in our example, the value that the load writes in r32 is read by the add two iterations (and two rotations) later as r34. Similarly the value that the add writes in r35 is read by the store one iteration later as r36.
5. For the br.ctop instruction, the branch is taken if either $LC > 0$ or $EC > 1$. Execution of br.ctop has the following additional effects: If $LC > 0$, then LC is decremented; this happens during the prolog and kernel phases. If $LC = 0$

Table 21.4 Loop Trace for Software Pipelining Example

Cycle	Execution Unit/Instruction				State before br.ctop					
	M	I	M	B	p16	p17	p18	p19	LC	EC
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
					0	0	0	0	0	0

and EC > 1, EC is decremented; this happens during the epilog phase. The instruction also control register rotation. If LC > 0, each execution of br.ctop places a 1 in p63. With rotation, p63 becomes p16, feeding a continuous sequence of ones into the predicate registers during the prolog and kernel phases. If LC $= 0$, then br.ctop sets p63 to 0, feeding zeros into the predicate registers during the epilog phase.

Table 21.4 shows a trace of the execution of this example.

21.4 IA-64 INSTRUCTION SET ARCHITECTURE

Figure 21.7 shows the set of registers available to application programs. That is, these registers are visible to applications and may be read and, in most cases, written. The register sets include:

- **General registers:** 128 general-purpose 64-bit registers. Associated with each register is a NaT bit used to track deferred speculative exceptions, as explained in Section 21.3. Registers r0 through r31 are referred to as static; a program reference to any of these references is literally interpreted. Registers r32 through r127 can be used as rotating registers for software pipelining (discussed in Section 21.3) and for register stack implementation (discussed subsequently in this section). References to these registers are virtual, and the hardware my perform register renaming dynamically.
- **Floating-point registers:** 128 82-bit registers for floating-point numbers. This size is sufficient to hold IEEE 754 double extended format numbers (see Table 9.3). Registers fr0 through fr31 are static, and registers fr32 through fr127 can be used as rotating registers for software pipelining.

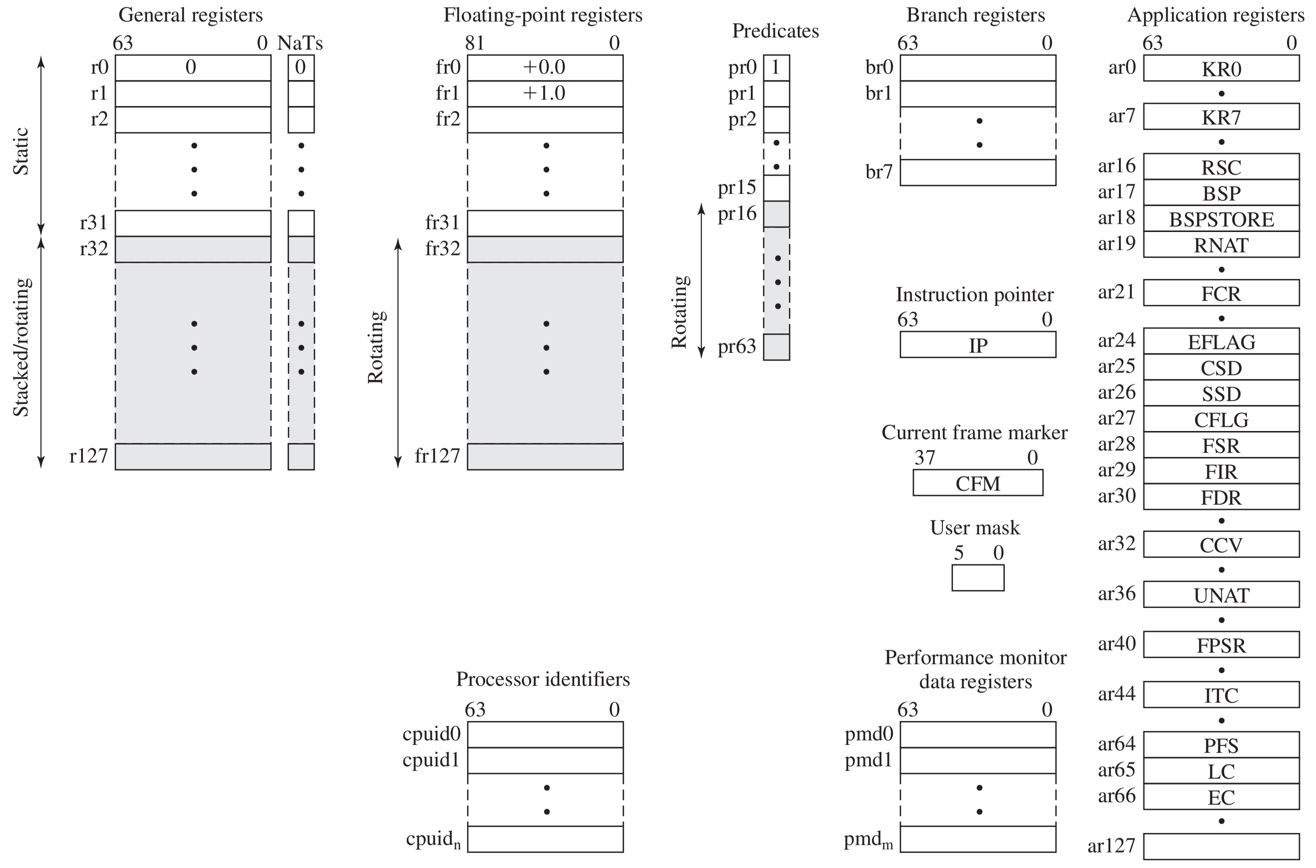


Figure 21.7 IA-64 Application Register Set

- **Predicate registers:** 64 1-bit registers used as predicates. Register pr0 is always set to 1 to enable unpredicated instructions. Registers pr0 through pr15 are static, and registers pr16 through pr63 can be used as rotating registers for software pipelining.
- **Branch registers:** 8 64-bit registers used for branches.
- **Instruction pointer:** Holds the bundle address of the currently executing IA-64 instruction.
- **Current frame marker:** Holds state information relating to the current general register stack frame and rotation information for fr and pr registers.
- **User mask:** A set of single-bit values used for alignment traps, performance monitors, and to monitor floating-point register usage.
- **Performance monitor data registers:** Used to support performance monitor hardware.
- **Processor identifiers:** Describe processor implementation-dependent features.
- **Application registers:** A collection of special-purpose registers. Table 21.5 provides a brief definition of each.

Table 21.5 IA-64 Application Registers

Kernel registers (KR0-7)	Convey information from the operating system to the application.
Register stack configuration (RSC)	Controls the operation of the register stack engine (RSE).
RSE Backing store pointer (BSP)	Holds the address in memory that is the save location for r32 in the current stack frame.
RSE Backing store pointer to memory stores (BSPSTORE)	Holds the address in memory to which the RSE will spill the next value.
RSE NaT collection register (RNAT)	Used by the RSE to temporarily hold NaT bits when it is spilling general registers.
Compare and exchange value (CCV)	Contains the compare value used as the third source operand in the cmpxchg instruction.
User NaT collection register (UNAT)	Used to temporarily hold NaT bits when saving and restoring general registers with the ld8.fill and st8.spill instructions.
Floating-point status register (FPSR)	Controls traps, rounding mode, precision control, flags, and other control bits for floating-point instructions.
Interval time counter (ITC)	Counts up at a fixed relationship to the processor clock frequency.
Previous function state (PFS)	Saves value in CFM register and related information.
Loop count (LC)	Used in counted loops and is decremented by counted-loop-type branches.
Epilog count (EC)	Used for counting the final (epilog) state in modulo-scheduled loops.

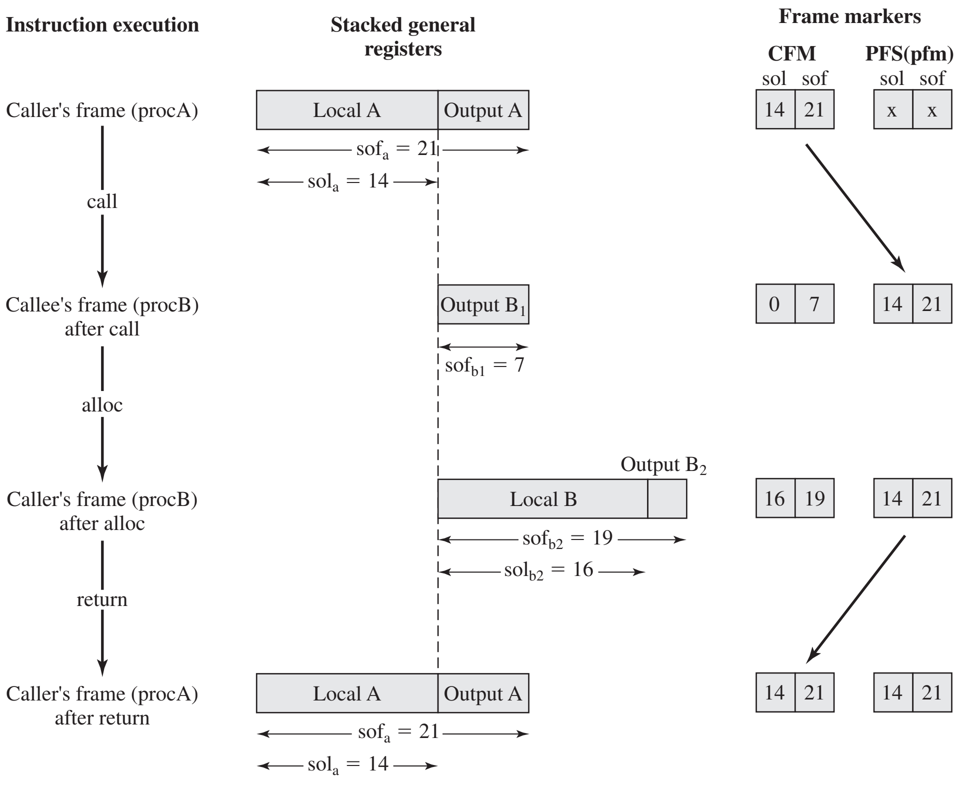


Figure 21.8 Register Stack Behavior on Procedure Call and Return

Register Stack

The register stack mechanism in IA-64 avoids unnecessary movement of data into and out of registers at procedure call and return. The mechanism automatically provides a called procedure with a new **frame** of up to 96 registers (r32 through r127) upon procedure entry. The compiler specifies the number of registers required by a procedure with the alloc instruction, which specifies how many of these are local (used only within the procedure) and how many are output (used to pass parameters to a procedure called by this procedure). When a procedure call occurs, the IA-64 hardware renames registers so that the local registers from the previous frame are hidden and what were the output registers of the calling procedure now have register numbers starting at r32 in the called procedure. Physical registers in the range r32 through r127 are allocated in a circular-buffer fashion to virtual registers associated with procedures. That is, the next register allocated after r127 is r32. When necessary, the hardware moves register contents between registers and memory to free up additional registers when procedure calls occur, and restores contents from memory to registers as procedure returns occur.

Figure 21.8 illustrates register stack behavior. The alloc instruction includes sof (size of frame) and sol (size of locals) operands to specify the required number of registers. These values are stored in the CFM register. When a call occurs, the sol and sof values from the CFM are stored in the sol and sof fields of the previous function state (PFS) application register (Figure 21.9). Upon return these sol and sof values

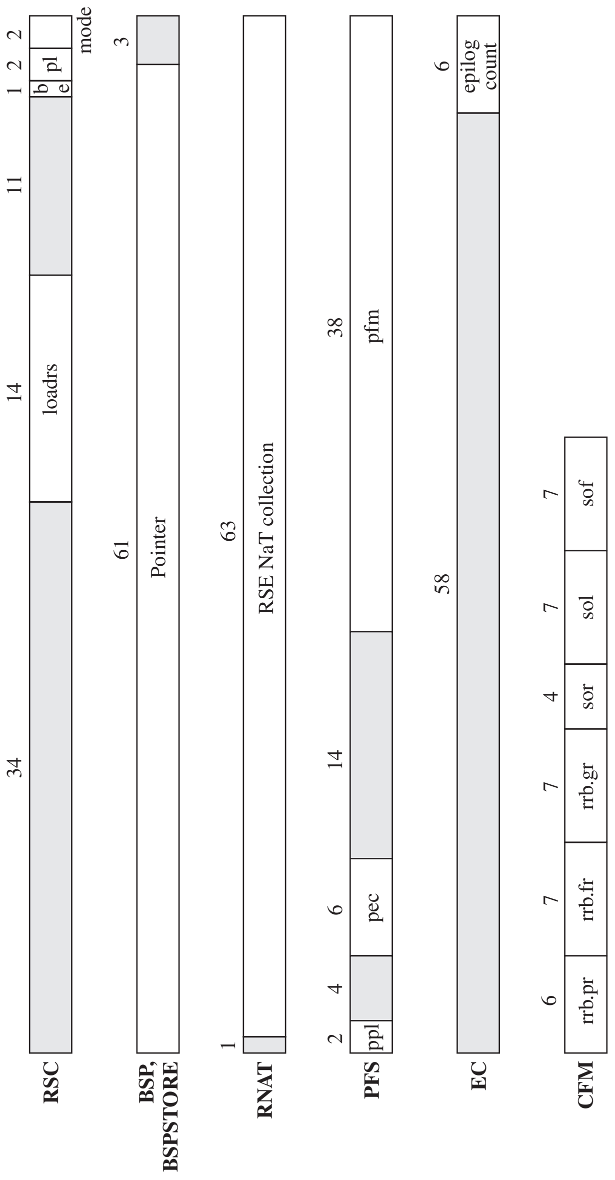


Figure 21.9 Formats of Some IA-64 Registers

must be restored from the PFS to the CFM. To allow nested calls and returns, previous values of the PFS fields must be saved through successive calls so that they can be restored through successive returns. This is a function of the alloc instruction, which designates a general register to save the current value of the PFS fields before they are overwritten from the CFM fields.

Current Frame Marker and Previous Function State

The CFM register describes the state of the current general register stack frame, associated with the currently active procedure. It includes the following fields:

- **sof:** size of stack frame
- **sol:** size of locals portion of stack frame
- **sor:** size of rotating portion of stack frame; this is a subset of the local portion that is dedicated to software pipelining
- **register rename base values:** Values used in performing register rotation general, floating-point and predicate registers

The PFS application register contains the following fields:

- **pfm:** Previous frame marker; contains all of the fields of the CFM
- **pec:** Previous epilog count
- **ppl:** Previous privilege level

21.5 ITANIUM ORGANIZATION

Intel's Itanium processor is the first implementation of the IA-64 instruction set architecture. The first version of this implementation, known as Itanium, was released in 2001, followed in 2002 by the Itanium 2. The Itanium organization blends superscalar features with support for the unique EPIC-related IA-64 features. Among the superscalar features are a six-wide, ten-stage-deep hardware pipeline, dynamic prefetch, branch prediction, and a register scoreboard to optimize for compile time nondeterminism. EPIC related hardware includes support for predicated execution, control and data speculation, and software pipelining.

Figure 21.10 is a general block diagram of the Itanium organization. The Itanium includes nine execution units: two integer, two floating-point, four memory, and three branch execution units. Instructions are fetched through an L1 instruction cache and fed into a buffer that holds up to eight bundles of instructions. When deciding on functional units for instruction dispersal, the processor views at most two instruction bundles at a time. The processor can issue a maximum of six instructions per clock cycle.

The organization is in some ways simpler than a conventional contemporary superscalar organization. The Itanium does not use reservation stations, reorder buffers, and memory ordering buffers, all replaced by simpler hardware for speculation. The register remapping hardware is simpler than the register aliasing typical of superscalar machines. Register dependency-detection logic is absent, replaced by explicit parallelism directives precomputed by the software.

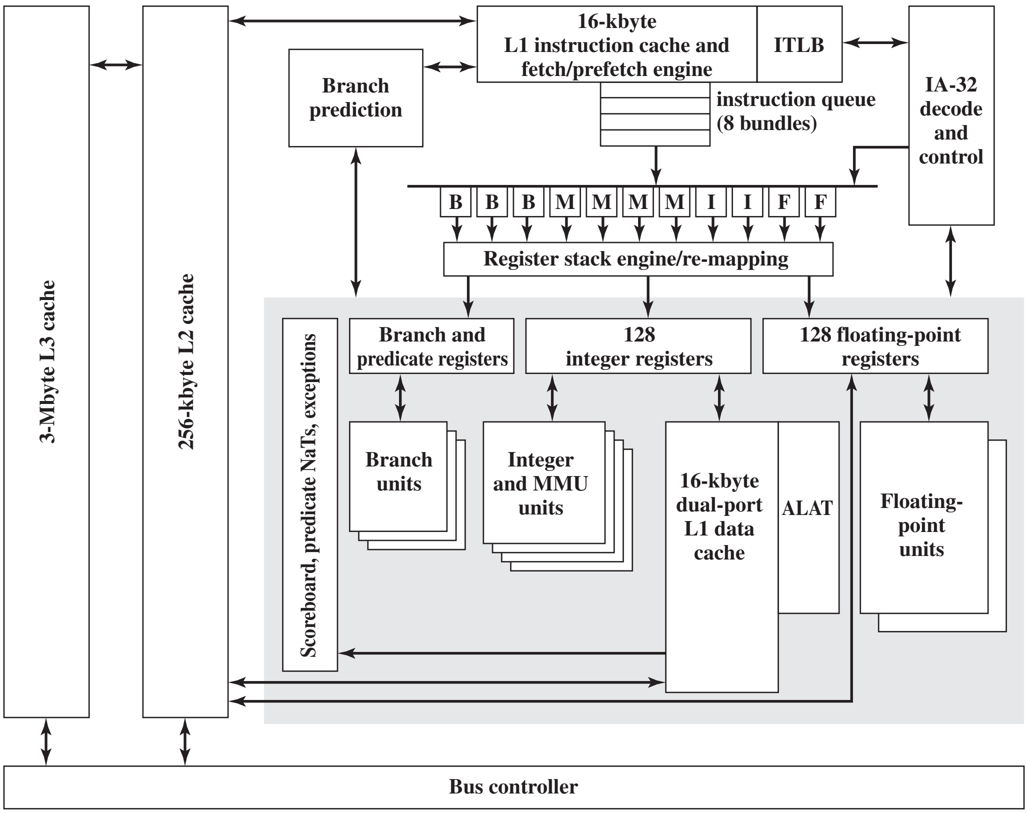

16-kbyte
L1 instruction cache and fetch/prefetch engine
ITLB
Branch prediction
instruction queue (8 bundles)
IA-32 decode and control
B B B M M M M I I F F
Register stack engine/re-mapping
3-Mbyte L3 cache
256-kbyte L2 cache
Scoreboard, predicate NaTs, exceptions
Branch and predicate registers
128 integer registers
128 floating-point registers
Branch units
Integer and MMU units
16-kbyte dual-port L1 data cache
ALAT
Floating-point units
Bus controller

Figure 21.10 Itanium 2 Processor Organization

Using branch prediction, the fetch/prefetch engine can speculatively load an L1 instruction cache to minimize cache misses on instruction fetches. The fetched code is fed into a decoupling buffer that can hold up to eight bundles of code.

Three levels of cache are used. The L1 cache is split into a 16-kbyte instruction cache and a 16-kbyte data cache, each 4-way set associative with a 32-byte line size. The 256-kbyte L2 cache is 6-way set associative with a 64-byte line size. The 3-Mbyte L3 cache is 4-way set associative with a 64-byte line size. All three levels of cache are on the same chip as the processor for the Itanium 2. For the original Itanium, the L3 cache is off-chip but on the same package as the processor.

The Itanium 2 uses an 8-stage pipeline for all but floating-point instructions. Figure 21.11 illustrates the relationship between the pipeline stages and the Itanium 2 organization. The pipeline stages are:

- **Instruction pointer generation (IPG):** Delivers an instruction pointer to the L1I cache.
- **Instruction rotation (ROT):** Fetch instructions and rotate instructions into position so that bundle 0 contains the first instruction that should be executed.

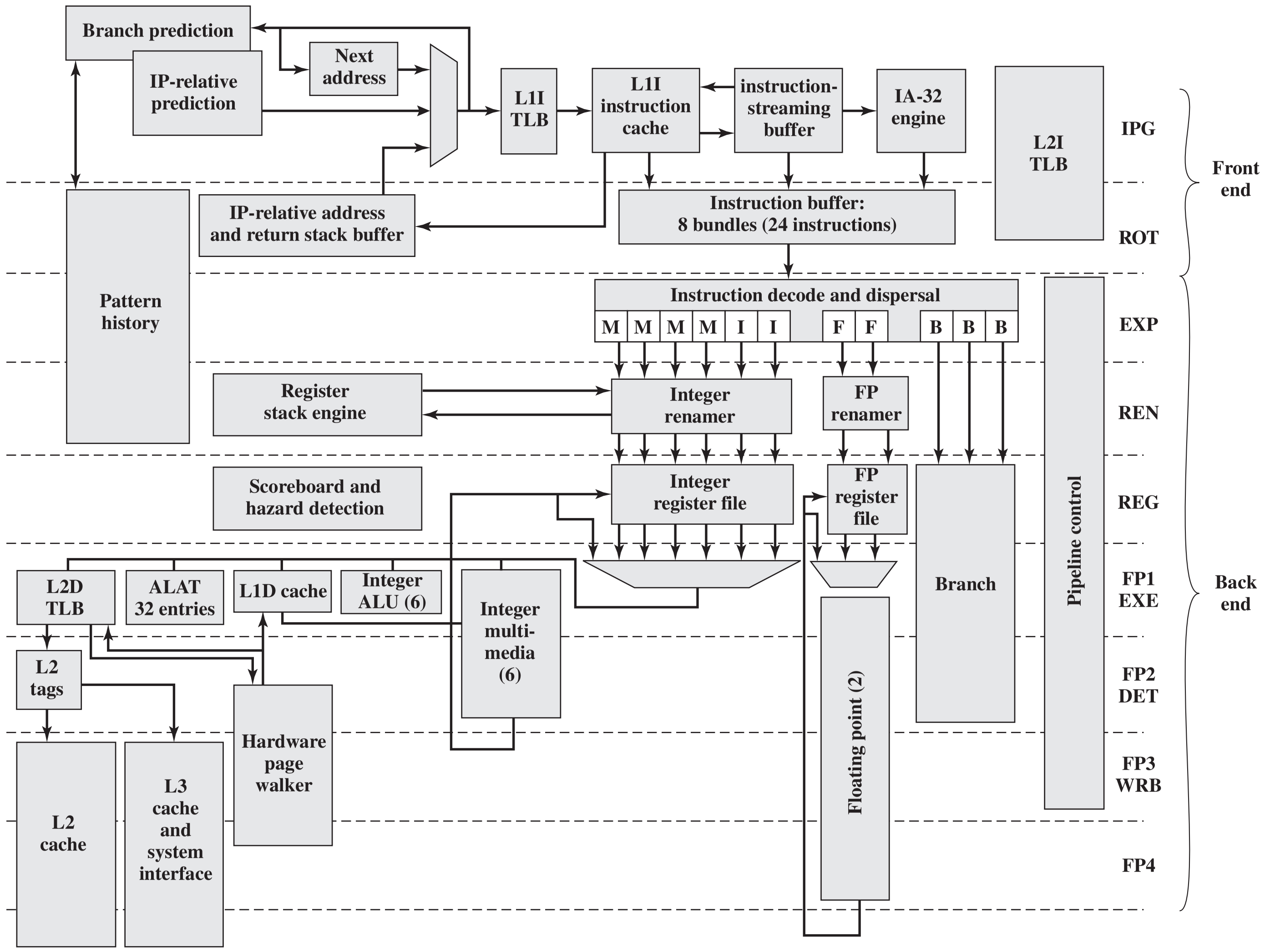


Figure 21.11 Itanium 2 Processor Pipeline [MCNA03]

- **Instruction template decode, expand and disperse (EXP):** Decode instruction templates, and disperse up to 6 instructions through 11 ports in conjunction with opcode information for the execution units.
- **Rename and decode (REN):** Rename (remap) registers for the register stack engine; decode instructions.
- **Register file read (REG):** Delivers operands to execution units.
- **ALU execution (EXE):** Execute operations.
- **Last stage for exception detection (DET):** Detect exceptions; abandon result of execution if instruction predicate was not true; resteer mispredicted branches.
- **Write back (WRB):** Write results back to register file.

For floating-point instructions, the first five pipeline stages are the same as just listed, followed by four floating-point pipeline stages, followed by a write-back stage.

21.6 RECOMMENDED READING AND WEB SITES

[HUCK00] provides an overview of IA-64; another overview is [DULO98]. [SCHL00a] provides a general discussion of EPIC; a more thorough treatment is provided in [SCHL00b]. Two other good treatments are [HWU01] and [KATH01]. [CHAS00] and [HWU98] provide introductions to predicated execution. Volume 1 of [INTE00a] contains a detailed treatment of software pipelining; two articles that provide a good explanation of the topic, with examples, are [JARP01] and [BHAR00].

For an overview of the Itanium processor architecture, see [SHAR00]; [INTE00b] provides a more detailed treatment. [MCNA03] and [NAFF02] describe the Itanium 2 in some detail.

[EVAN03], [TRIE01], and [MARK00] contain more detailed treatments of the topics of this chapter. Finally, for an exhaustive look at the IA-64 architecture and instruction set, see [INTE00a].

BHAR00 Bharandwaj, J., et al. "The Intel IA-64 Compiler Code Generator." *IEEE Micro,* September/October 2000.

CHAS00 Chasin, A. "Predication, Speculation, and Modern CPUs." *Dr. Dobb's Journal,* May 2000.

DULO98 Dulong, C. "The IA-64 Architecture at Work." *Computer,* July 1998.

EVAN03 Evans, J., and Trimper, G. *Itanium Architecture for Programmers.* Upper Saddle River, NJ: Prentice Hall, 2003.

HUCK00 Huck, J., et al. "Introducing the IA-64 Architecture." *IEEE Micro,* September/October 2000.

HWU98 Hwu, W. "Introduction to Predicated Execution." *Computer,* January 1998.

HWU01 Hwu, W.; August, D.; and Sias, J. "Program Decision Logic Optimization Using Predication and Control Speculation." *Proceedings of the IEEE,* November 2001.

INTE00a Intel Corp. *Intel IA-64 Architecture Software Developer's Manual (4 volumes).* Document 245317 through 245320. Aurora, CO, 2000.

INTE00b Intel Corp. *Itanium Processor Microarchitecture Reference for Software Optimization.* Aurora, CO, Document 245473. August 2000.

JARP01 Jarp, S. "Optimizing IA-64 Performance." *Dr. Dobb's Journal,* July 2001.

KATH01 Kathail. B.; Schlansker, M.; and Rau, B. "Compiling for EPIC Architectures." *Proceedings of the IEEE,* November 2001.

MARK00 Markstein, P. *IA-64 and Elementary Functions.* Upper Saddle River, NJ: Prentice Hall PTR, 2000.

MCNA03 McNairy, C., and Soltis, D. "Itanium 2 Processor Microarchitecture." *IEEE Micro,* March-April 2003.

NAFF02 Naffziger, S., et al. "The Implementation of the Itanium 2 Microprocessor." *IEEE Journal of Solid-State Circuits,* November 2002.

SCHL00a Schlansker, M.; and Rau, B. "EPIC: Explicitly Parallel Instruction Computing." *Computer,* February 2000.

SCHL00b Schlansker, M.; and Rau, B. *EPIC: An Architecture for Instruction-Level Parallel Processors.* HPL Technical Report HPL-1999-111, Hewlett-Packard Laboratories (www.hpl.hp.com), February 2000.

SHAR00 Sharangpani, H., and Arona, K. "Itanium Processor Microarchitecture." *IEEE Micro*, September/October 2000.

TRIE01 Triebel, W. *Itanium Architecture for Software Developers.* Intel Press, 2001.

Recommended Web sites:

- **Itanium:** Intel's site for the latest information on IA-64 and Itanium.
- **HP Itanium Technology site:** Good source of information.
- **IMPACT:** This is a site at the University of Illinois, where much of the research on predicated execution has been done. A number of papers on the subject are available.

21.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

advanced load	IA-64 architecture	software pipeline
branch predication	instruction completer	speculative loading
bundle	instruction group	stack frame
control speculation	Itanium	stop
data speculation	major opcode	syllable
execution unit	NaT bit	template field
explicitly parallel instruction computing (EPIC)	predicate register	very long instruction word (VLIW)
hoist	predication	
	register stack	

Review Questions

21.1 What are the different types of execution units for IA-64?
21.2 Explain the use of the template field in an IA-64 bundle.
21.3 What is the significance of a stop in the instruction stream?
21.4 Define predication and predicated execution.
21.5 How can predicates replace a conditional branch instruction?
21.6 Define control speculation.
21.7 What is the purpose of the NaT bit?
21.8 Define data speculation.
21.9 What is the difference between a hardware pipeline and a software pipeline?
21.10 What is the difference between stacked and rotating registers?

Problems

21.1 Suppose that an IA-64 opcode accepts three registers as operands and produces one register as a result. What is the maximum number of different operations that can be defined in one major opcode family?

21.2 What is the maximum effective number of major opcodes?

21.3 At a certain point in an IA-64 program, there are 10 A-type instructions and six floating-point instructions that can be issued concurrently. How many syllables may appear without any stops between them?

21.4 In Problem 21.3,
 a. How many cycles are required for a small IA-64 implementation having one floating-point unit, two integer units, and two memory units?
 b. How many cycles are required for the Itanium organization of Figure 21.10?

21.5 The initial Itanium implementation had two M-units and two I-units. Which of the templates in Table 21.3 cannot be paired as two bundles of instructions that could be executed completely in parallel?

21.6 An algorithm that can utilize four floating-point instructions per cycle is coded for IA-64. Should instruction groups contain four floating-point operations? What are the consequences if the machine on which the program runs has fewer than four floating-point units?

21.7 In Section 21.3, we introduced the following constructs for predicated execution:

```
        cmp.crel p2, p3 = a, b
(p1)    cmp.crel p2, p3 = a, b
```

where crel is a relation, such as eq, ne, etc.; p1, p2, and p3 are predicate registers; a is either a register or an immediate operand; and b is a register operand.

Fill in the following truth table:

p1	comparison	p2	p3
not present	0		
not present	1		
0	0		
0	1		
1	0		
1	1		

21.8 For the predicated program in Section 21.3, which implements the flowchart of Figure 21.4, indicate

a. Those instructions that can be executed in parallel

b. Those instructions that can be bundled into the same IA-64 instruction bundle

21.9 The IA-64 architecture includes a set of multimedia instructions comparable to those in the IA-32 Pentium architecture (Table 10.11). One such instruction type is the parallel compare instruction of the form pcmp1, pcmp2, or pcmp4, which does a parallel compare 1, 2, or 4 bytes at a time. The instruction pcmp1.gt ri = rj, rk compares the two source operands (rj, rk) byte by byte. For each byte, if the byte in rj is greater than the byte in rk, then the corresponding byte in ri is set to all ones; otherwise the destination byte is set to all zeros. Both operands are interpreted as signed.

Suppose the registers r14 and r15 contain the ASCII strings (see Table F.1) "00000000" and "99999999" respectively and the register r16 contains an arbitrary string of eight characters. Determine whether the comments in the following code fragment are appropriate.

```
      pcmp1.gt    r8 = r14,r16       // if some char < "0" or
      pcmp1.gt    r9 = r16,r15 ;;    // if some char > "9"
      cmp.ne      p6,p0 = r8,r0 ;;   // p6 = true or
      cmp.ne      p7,p0 = r9,r0 ;;   // p7 = true so that
(p6)  br error                       // this branch executes or
(p7)  br error ;;                    // this branch executes
```

21.10 Consider the following source code segment:

```
for ( i = 0; i < 100; i++ )
      if (A[i] < 50 )
            j = j + 1;
      else
            k = k + 1;
```

a. Write a corresponding Pentium assembly code segment.

b. Rewrite as an IA-64 assembly code segment using predicated execution techniques.

21.11 Consider the following C program fragment dealing with floating-point values:

```
a[i] = p * q;
c = a[j];
```

The compiler cannot establish that $i \neq j$, but has reason to believe that it probably is.

a. Write an IA-64 program using an advanced load to implement this C program. *Hint:* the floating-point load and multiply instructions are ldf and fmpy, respectively.

b. Recode the program using predication instead of the advanced load.

c. What are the advantages and disadvantages of the two approaches compared with each other?

21.12 Assume that a stack register frame is created with size equal to SOF = 48. If the size of the local register group is SOL = 16,

a. How many output registers (SOO) are there?

b. Which registers are in the local and output register groups?