```
(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 22 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 15.6 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

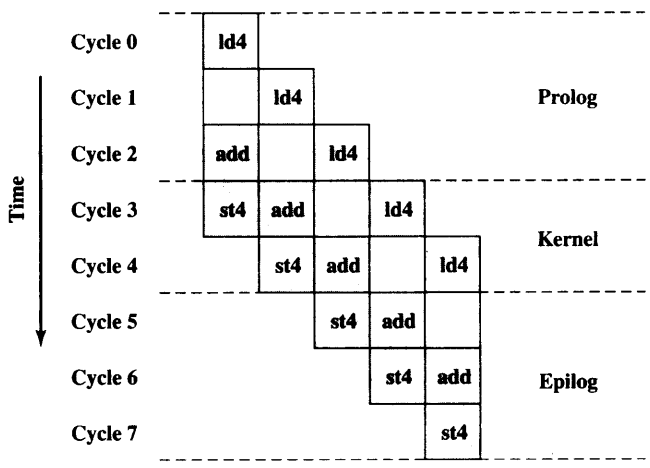


**Figure 15.6** Software Pipelining Example

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 15.6 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 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 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 15.4 shows a trace of the execution of this example.

## 15.4 IA-64 INSTRUCTION SET ARCHITECTURE

Figure 15.7 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 the following:

- **General registers:** 128 general-purpose 64-bit registers. Associated with each register is a NaT bit used to track deferred speculative exceptions, as