Assume a machine that uses the delay model of Example 10.6 (loads take two clocks, all other instructions take one clock). Also assume that the machine can execute any two instructions at once. Find a shortest possible execution of this fragment. Do not forget to consider which register is best used for each of the copy steps. Also, remember to exploit the information given by register descriptors as was described in Section 8.6, to avoid unnecessary loads and stores.

10.5 Software Pipelining

As discussed in the introduction of this chapter, numerical applications tend to have much parallelism. In particular, they often have loops whose iterations are completely independent of one another. These loops, known as *do-all* loops, are particularly attractive from a parallelization perspective because their iterations can be executed in parallel to achieve a speed-up linear in the number of iterations in the loop. Do-all loops with many iterations have enough parallelism to saturate all the resources on a processor. It is up to the scheduler to take full advantage of the available parallelism. This section describes an algorithm, known as *software pipelining*, that schedules an entire loop at a time, taking full advantage of the parallelism across iterations.

10.5.1 Introduction

We shall use the do-all loop in Example 10.12 throughout this section to explain software pipelining. We first show that scheduling across iterations is of great importance, because there is relatively little parallelism among operations in a single iteration. Next, we show that loop unrolling improves performance by overlapping the computation of unrolled iterations. However, the boundary of the unrolled loop still poses as a barrier to code motion, and unrolling still leaves a lot of performance "on the table." The technique of software pipelining, on the other hand, overlaps a number of consecutive iterations continually until it runs out of iterations. This technique allows software pipelining to produce highly efficient and compact code.

Example 10.12: Here is a typical do-all loop:

```
for (i = 0; i < n; i++)
    D[i] = A[i]*B[i] + c;
```

Iterations in the above loop write to different memory locations, which are themselves distinct from any of the locations read. Therefore, there are no memory dependences between the iterations, and all iterations can proceed in parallel.

We adopt the following model as our target machine throughout this section. In this model

- The machine can issue in a single clock: one load, one store, one arithmetic operation, and one branch operation.
- The machine has a loop-back operation of the form

  ```
  BL R, L
  ```

  which decrements register R and, unless the result is 0, branches to location L.
- Memory operations have an auto-increment addressing mode, denoted by `++` after the register. The register is automatically incremented to point to the next consecutive address after each access.
- The arithmetic operations are fully pipelined; they can be initiated every clock but their results are not available until 2 clocks later. All other instructions have a single-clock latency.

If iterations are scheduled one at a time, the best schedule we can get on our machine model is shown in Fig. 10.17. Some assumptions about the layout of the data also also indicated in that figure: registers `R1`, `R2`, and `R3` hold the addresses of the beginnings of arrays A, B, and D, register `R4` holds the constant c, and register `R10` holds the value $n-1$, which has been computed outside the loop. The computation is mostly serial, taking a total of 7 clocks; only the loop-back instruction is overlapped with the last operation in the iteration. □

```
     //   R1, R2, R3 = &A, &B, &D
     //   R4         = c
     //   R10        = n-1

L:   LD   R5, 0(R1++)
     LD   R6, 0(R2++)
     MUL  R7, R5, R6
     nop
     ADD  R8, R7, R4
     nop
     ST   0(R3++), R8        BL R10, L
```

Figure 10.17: Locally scheduled code for Example 10.12

In general, we get better hardware utilization by unrolling several iterations of a loop. However, doing so also increases the code size, which in turn can have a negative impact on overall performance. Thus, we have to compromise, picking a number of times to unroll a loop that gets most of the performance improvement, yet doesn't expand the code too much. The next example illustrates the tradeoff.

Example 10.13: While hardly any parallelism can be found in each iteration of the loop in Example 10.12, there is plenty of parallelism across the iterations. Loop unrolling places several iterations of the loop in one large basic block, and a simple list-scheduling algorithm can be used to schedule the operations to execute in parallel. If we unroll the loop in our example four times and apply Algorithm 10.7 to the code, we can get the schedule shown in Fig. 10.18. (For simplicity, we ignore the details of register allocation for now). The loop executes in 13 clocks, or one iteration every 3.25 clocks.

A loop unrolled k times takes at least $2k + 5$ clocks, achieving a throughput of one iteration every $2 + 5/k$ clocks. Thus, the more iterations we unroll, the faster the loop runs. As $n \rightarrow \infty$, a fully unrolled loop can execute on average an iteration every two clocks. However, the more iterations we unroll, the larger the code gets. We certainly cannot afford to unroll all the iterations in a loop. Unrolling the loop 4 times produces code with 13 instructions, or 163% of the optimum; unrolling the loop 8 times produces code with 21 instructions, or 131% of the optimum. Conversely, if we wish to operate at, say, only 110% of the optimum, we need to unroll the loop 25 times, which would result in code with 55 instructions. □

10.5.2 Software Pipelining of Loops

Software pipelining provides a convenient way of getting optimal resource usage and compact code at the same time. Let us illustrate the idea with our running example.

Example 10.14: In Fig. 10.19 is the code from Example 10.12 unrolled five times. (Again we leave out the consideration of register usage.) Shown in row i are all the operations issued at clock i; shown in column j are all the operations from iteration j. Note that every iteration has the same schedule relative to its beginning, and also note that every iteration is initiated two clocks after the preceding one. It is easy to see that this schedule satisfies all the resource and data-dependence constraints.

We observe that the operations executed at clocks 7 and 8 are the same as those executed at clocks 9 and 10. Clocks 7 and 8 execute operations from the first four iterations in the original program. Clocks 9 and 10 also execute operations from four iterations, this time from iterations 2 to 5. In fact, we can keep executing this same pair of multi-operation instructions to get the effect of retiring the oldest iteration and adding a new one, until we run out of iterations.

Such dynamic behavior can be encoded succinctly with the code shown in Fig. 10.20, if we assume that the loop has at least 4 iterations. Each row in the figure corresponds to one machine instruction. Lines 7 and 8 form a 2-clock loop, which is executed $n - 3$ times, where n is the number of iterations in the original loop. □

```
L:  LD
    LD
        LD
    MUL LD
        MUL  LD
    ADD      LD
        ADD      LD
    ST       MUL  LD
        ST       MUL
                 ADD
                      ADD
                 ST
                      ST    BL (L)
```

Figure 10.18: Unrolled code for Example 10.12

Clock	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

Figure 10.19: Five unrolled iterations of the code in Example 10.12

```
1)        LD
2)        LD
3)        MUL   LD
4)              LD
5)              MUL   LD
6)        ADD         LD
7)  L:                MUL   LD
8)        ST    ADD         LD    BL (L)
9)                          MUL
10)             ST    ADD
11)
12)                   ST    ADD
13)
14)                         ST
```

Figure 10.20: Software-pipelined code for Example 10.12

The technique described above is called *software pipelining*, because it is the software analog of a technique used for scheduling hardware pipelines. We can think of the schedule executed by each iteration in this example as an 8-stage pipeline. A new iteration can be started on the pipeline every 2 clocks. At the beginning, there is only one iteration in the pipeline. As the first iteration proceeds to stage three, the second iteration starts to execute in the first pipeline stage.

By clock 7, the pipeline is fully filled with the first four iterations. In the steady state, four consecutive iterations are executing at the same time. A new iteration is started as the oldest iteration in the pipeline retires. When we run out of iterations, the pipeline drains, and all the iterations in the pipeline run to completion. The sequence of instructions used to fill the pipeline, lines 1 through 6 in our example, is called the *prolog*; lines 7 and 8 are the *steady state*; and the sequence of instructions used to drain the pipeline, lines 9 through 14, is called the *epilog*.

For this example, we know that the loop cannot be run at a rate faster than 2 clocks per iteration, since the machine can only issue one read every clock, and there are two reads in each iteration. The software-pipelined loop above executes in $2n + 6$ clocks, where n is the number of iterations in the original loop. As $n \rightarrow \infty$, the throughput of the loop approaches the rate of one iteration every two clocks. Thus, software scheduling, unlike unrolling, can potentially encode the optimal schedule with a very compact code sequence.

Note that the schedule adopted for each individual iteration is not the shortest possible. Comparison with the locally optimized schedule shown in Fig. 10.17 shows that a delay is introduced before the ADD operation. The delay is placed strategically so that the schedule can be initiated every two clocks without resource conflicts. Had we stuck with the locally compacted schedule,