20

Pipelining and Scheduling

sched-ule: a procedural plan that indicates the time and sequence of each operation

Webster's Dictionary

A simple computer can process one instruction at a time. First it fetches the instruction, then decodes it into opcode and operand specifiers, then reads the operands from the register bank (or memory), then performs the arithmetic denoted by the opcode, then writes the result back to the register back (or memory); and then fetches the next instruction.

Modern computers can execute parts of many different instructions at the same time. At the same time the processor is writing results of two instructions back to registers, it may be doing arithmetic for three other instructions, reading operands for two more instructions, decoding four others, and fetching yet another four. Meanwhile, there may be five instructions delayed, awaiting the results of memory-fetches.

Such a processor usually fetches instructions from a single flow of control; it's not that several programs are running in parallel, but the adjacent instructions of a single program are decoded and executed simultaneously. This is called *instruction-level parallelism* (ILP), and is the basis for much of the astounding advance in processor speed in the last decade of the twentieth century.

A *pipelined* machine performs the write-back of one instruction in the same cycle as the arithmetic "execute" of the next instruction and the operand-read of the previous one, and so on. A *very-long-instruction-word* (VLIW) issues several instructions in the same processor cycle; the compiler must ensure that they are not data-dependent on each other. A *superscalar* machine

issues two or more instructions in parallel *if they are not related by data dependence* (which it can check quickly in the instruction-decode hardware); otherwise it issues the instructions sequentially – thus, the program will still operate correctly if data-dependent instructions are adjacent, but it will run faster if the compiler has not scheduled non-data-dependent instructions adjacent to each other. A *dynamic-scheduling* machine reorders the instructions as they are being executed, so that it can issue several non-data-dependent instructions simultaneously, and may need less help from the compiler. Any of these techniques produce instruction-level parallism.

The more instructions can be executed simultaneously, the faster the program will run. But why can't all the instructions of the program be executed in parallel? After all, that would be the fastest possible execution.

There are several kinds of *constraints* on instruction execution; we can optimize the program for instruction-level parallelism by finding the best *schedule* that obeys these constraints:

Data dependence: If instruction A calculates a result that's used as an operand of instruction B, then B cannot execute before A is finished.

Functional unit: If there are k_{fu} multipliers (adders, etc.) on the chip, then at most k_{fu} multiplication (addition, etc.) instructions can execute at once.

Instruction issue: The instruction-issue unit can issue at most k_{ii} instructions at a time.

Register: At most k_{r} registers can be in use at a time; more specifically, any schedule must have some valid register allocation.

The functional-unit, instruction-issue, and register constraints are often lumped together as *resource constraints* or *resource hazards*.

On a pipelined machine, even if "B cannot *execute* before A," there may be some parts of B's execution (such as instruction-fetch) that can proceed concurrently with A; Figures 20.2 and 20.3 give details.

There are also pseudo-constraints that can often be made to disappear by renaming variables:

Write-after-write: If instruction A writes to a register or memory location, and B writes to the same location, then the order of A and B must not be changed. But often it is possible to modify the program so that A and B write to different locations.

Write-after-read: If A must read from a location before B writes to it, then A and B's order of execution must not be swapped, unless renaming can be done so that they use different locations.

	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
ADD	I-Fetch	Read	Unpack	Shift Add	Round Add	Round Shift	Write			
MULT	I-Fetch	Read	Unpack	MultA	MultA	MultA	MultB	MultB Add	Round	Write
CONV	I-Fetch	Read	Unpack	Add	Round	Shift	Shift	Add	Round	Write

FIGURE 20.1. Functional unit requirements of instructions (on the MIPS R4000 processor). This machine's floating-point ADD instruction uses the instruction-fetch unit for one cycle; reads registers for one cycle; unpacks exponent and mantissa; then for the next cycle uses a shifter and an adder; then uses both the adder and a rounding unit; then the rounding unit and a shifter; then writes a result back to the register file. The MULT and CONV instructions use functional units in a different order.

Resource usage of an instruction. We might describe an instruction in terms of the number of cycles it takes to execute, and the resources it uses at different stages of execution. Figure 20.1 shows such a description for three instructions of a hypothetical machine.

If the ith cycle of instruction A uses a particular resource, and the jth cycle of instruction B uses the same resource, then B cannot be scheduled exactly $i - j$ cycles after A, as illustrated in Figure 20.2.

However, some machines have several functional units of each kind (e.g., more than one adder); on such a machine it does not suffice to consider instructions pairwise, but we must consider all the instructions scheduled for a given time.

Data-dependence of an instruction. The same considerations apply to data-dependence constraints. The result of some instruction A is written back to the register file during the **Write** stage of its execution (see Figure 20.1); if instruction B uses this register, then the **Read** stage of B must be after the **Write** stage of A. Some machines have bypass circuitry that may allow the arithmetic stage of B to follow immediately after the arithmetic stage of A; for example, the **Shift/Add** stage of an ADD instruction might be able to immediately follow the **Round** stage of a MULT. These situations are shown in Figure 20.3.

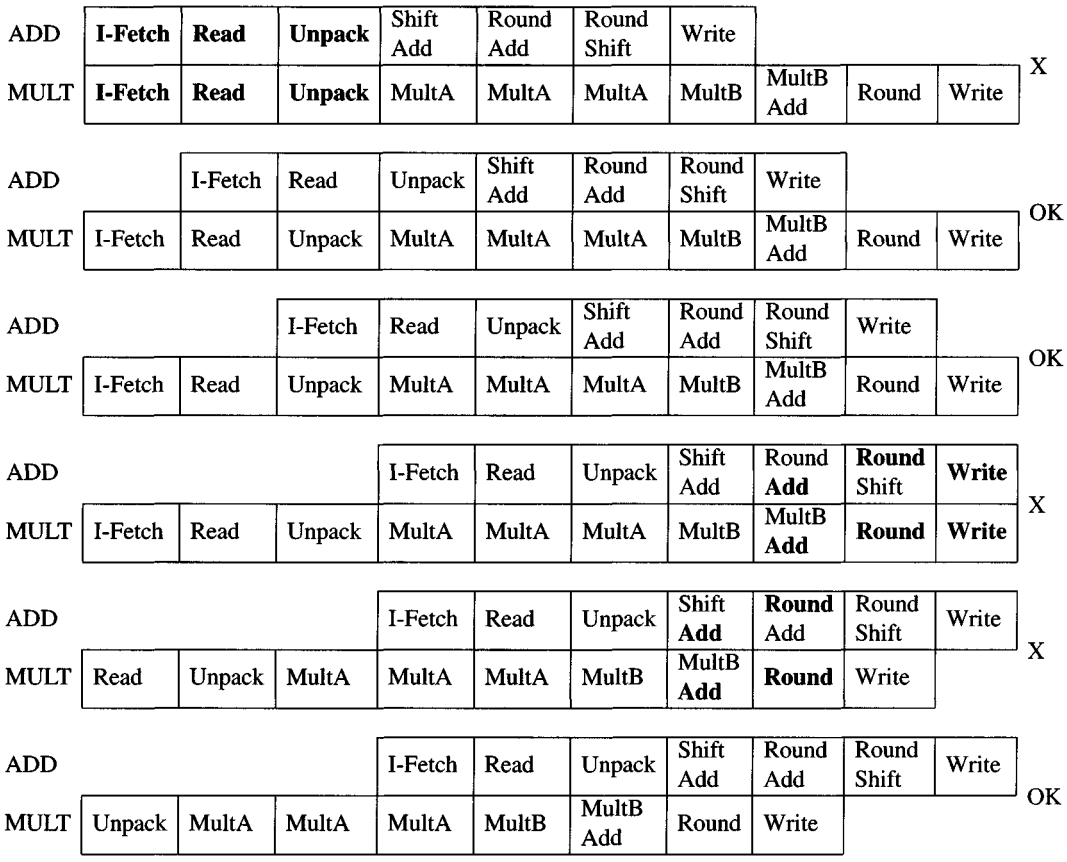


FIGURE 20.2. If there is only one functional unit of each kind, then an ADD cannot be started at the same time as a MULT (because of numerous resource hazards shown in boldface); nor three cycles after the MULT (because of **Add**, **Round**, and **Write** hazards); nor four cycles later (because of **Add** and **Round** hazards). But if there were two adders and two rounding units, then an ADD *could* be started four cycles after a MULT. Or with dual fetch units, multiple-access register file, and dual unpackers, the MULT and ADD could be started simultaneously.

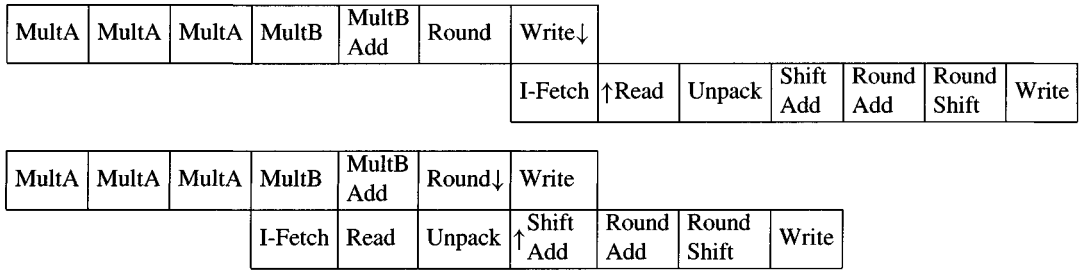


FIGURE 20.3. Data dependence. (Above) If the MULT produces a result that is an operand to ADD, the MULT must write its result to the register file before the ADD can read it. (Below) Special bypassing circuitry can route the result of MULT directly to the Shift and Add units, skipping the Write, Read, and Unpack stages.

20.1 LOOP SCHEDULING WITHOUT RESOURCE BOUNDS

Choosing an optimal schedule subject to data-dependence constraints and resource hazards is difficult – it is NP-complete, for example. Although NP-completeness should never scare the compiler writer (graph coloring is NP complete, but the approximation algorithm for graph coloring described in Chapter 11 is very successful), it remains the case that resource-bounded loop scheduling is hard to do in practice.

I will first describe an algorithm that ignores the resource constraints and finds an optimal schedule subject only to the data-dependence constraints. This algorithm is not useful in practice, but it illustrates the kind of opportunities there are in instruction-level parallelism.

The *Aiken-Nicolau loop pipelining* algorithm has several steps:

1. Unroll the loop;
2. Schedule each instruction from each iteration at the earliest possible time;
3. Plot the instructions in a tableau of iteration-number versus time;
4. Find separated groups of instructions at given slopes;
5. Coalesce the slopes;
6. Reroll the loop.

To explain the notions of *tableau*, *slope*, and *coalesce*, I use Program 20.4a as an example; let us assume that every instruction can be completed in one cycle, and that arbitrarily many instructions can be issued in the same cycle, subject only to data-dependence constraints.

```
for i ← 1 to N
    a    ← j ⊕ V[i − 1]
    b    ← a ⊕ f
    c    ← e ⊕ j
    d    ← f ⊕ c
    e    ← b ⊕ d
    f    ← U[i]
g : V[i] ← b
h : W[i] ← d
    j    ← X[i]
```

(a)

```
for i ← 1 to N

    a_i    ← j_{i−1} ⊕ b_{i−1}
    b_i    ← a_i ⊕ f_{i−1}
    c_i    ← e_{i−1} ⊕ j_{i−1}
    d_i    ← f_{i−1} ⊕ c_i
    e_i    ← b_i ⊕ d_i
    f_i    ← U[i]
g : V[i] ← b_i
h : W[i] ← d_i
    j_i    ← X[i]
```

(b)

PROGRAM 20.4. (a) A **for**-loop to be software-pipelined. (b) After a *scalar-replacement* optimization (in the definition of a); and scalar variables labeled with their iteration-number.

Data dependence through memory. For optimal scheduling of stores and fetches, we need to trace data dependence as a value is stored into memory and then fetched back. As discussed on page 452, dependence analysis of memory references is not trivial! In order to illustrate loop scheduling for Program 20.4a without full-fledged dependence analysis, we can use *scalar replacement* to replace the reference to $V[i-1]$ with the (equivalent) b; now we can see that in the resulting Program 20.4b all memory references are independent of each other, assuming that the arrays U, V, W, X do not overlap.

Next we mark each variable in the loop body to indicate whether *this* iteration's value is used, or the *previous* iteration's value, as shown in Program 20.4b. We can construct a *data-dependence* graph as a visual aid in scheduling; solid edges are data dependences within an iteration, and dotted edges are *loop-carried* dependences, as shown in Graph 20.5a.

Now suppose we unroll the loop; the data-dependence graph is a DAG, as shown in Graph 20.5b. Scheduling DAGs is easy if there are no resource constraints; starting from operations with no predecessors, each operation goes

as soon as its predecessors have all completed:

Cycle	Instructions
1	$a_1 c_1 f_1 j_1 f_2 j_2 f_3 j_3 \ldots$
2	$b_1 d_1$
3	$e_1 g_1 h_1 a_2$
4	$b_2 c_2$
5	$d_2 g_2 a_3$
⋮	⋮

It is convenient to write this schedule in a *tableau* where the rows are successive cycles and the columns are successive iterations of the original loop, as shown in Table 20.6a.

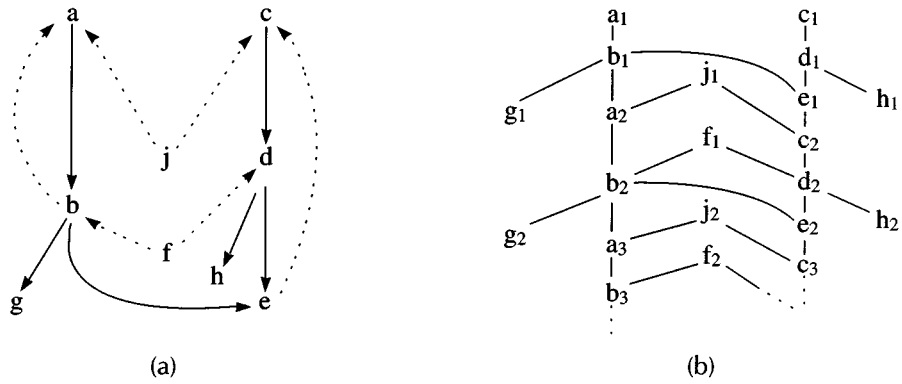
After a few iterations are scheduled, we notice a pattern in the tableau: there is a group of instructions *cdeh* racing down to the lower-right corner with a slope of three cycles per iteration, another group *abg* with a more moderate slope of two cycles per iteration, and a third group fj with zero slope. The key observation is that there are *gaps* in the schedule, separating identical groups, that grow larger at a constant rate. In this case the groups of instructions at iteration $i \geq 4$ are identical to the groups at iteration $i + 1$. In general the groups at iteration i will be identical to the groups at $i + c$, where sometimes $c > 1$; see Exercise 20.1.

Theorems:

- If there are K instructions in the loop, the pattern of identical groups separated by gaps will always appear within K^2 iterations (and usually much sooner).
- We can increase the slopes of the less steeply sloped groups, either closing the gaps or at least making them small and nonincreasing, without violating data-dependence constraints.
- The resulting tableau has a repeating set of m identical cycles, which can constitute the body of a pipelined loop.
- The resulting loop is optimally scheduled (it runs in the least possible time).

See the Further Reading section for reference to proofs. But to see why the loop is optimal, consider that the data-dependence DAG of the unrolled loop has some path of length P to the last instruction to execute, and the scheduled loop executes that instruction at time P.

The result, for our example, is shown in Table 20.6b. Now we can find a repeating pattern of three cycles (since three is the slope of the steepest



GRAPH 20.5. Data-dependence graph for Program 20.4b: (a) original graph, in which solid edges are same-iteration dependences and dotted edges are loop-carried dependences; (b) acyclic dependences of the unrolled loop.

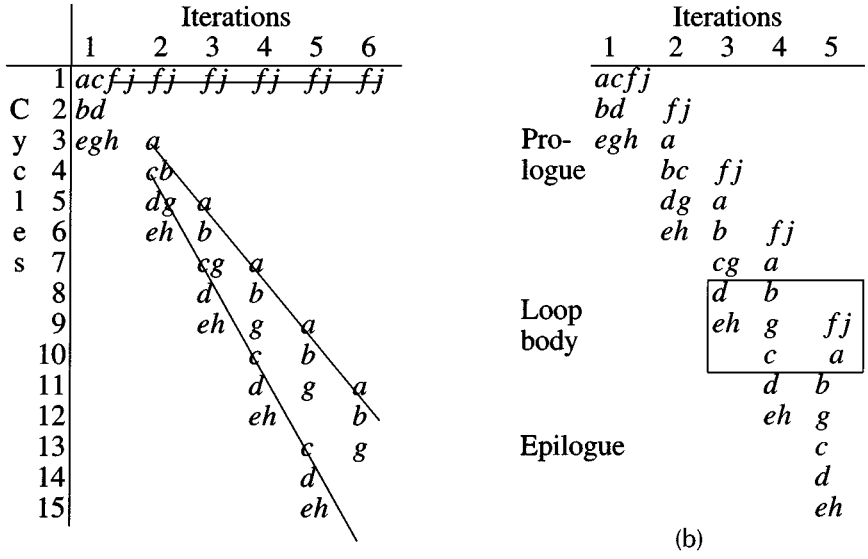


TABLE 20.6. (a) Tableau of software-pipelined loop schedule; there is a group of instructions fj with slope 0, another group abg with slope 2, and a third group $cdeh$ with slope 3. (b) The smaller-slope groups are pushed down to slope 3, and a pattern is found (boxed) that constitutes the pipelined loop.

group). In this case, the pattern does not begin until cycle 8; it is shown in a box. This will constitute the *body* of the scheduled loop. Irregularly scheduled instructions before the loop body constitute a *prologue*, and instructions after it constitute the *epilogue*.

Now we can generate the multiple-instruction-issue program for this loop, as shown in Figure 20.7. However, the variables still have subscripts in this "program": the variable j_{i+1} is live at the same time as j_i. To encode this program in instructions, we need to put in MOVE instructions between the different variables, as shown in Figure 20.8.

This loop is optimally scheduled – assuming the machine can execute 8 instructions at a time, including four simultaneous loads and stores.

Multicycle instructions. Although I have illustrated an example where each instruction takes exactly one cycle, the algorithm is easily extensible to the situation where some instructions take multiple cycles.

20.2 RESOURCE-BOUNDED LOOP PIPELINING

A real machine can issue only a limited number of instructions at a time, and has only a limited number of load/store units, adders, and multipliers. To be practically useful, a scheduling algorithm must take account of resource constraints.

The input to the scheduling algorithm must be in three parts:

1. A program to be scheduled;
2. A description of what resources each instruction uses in each of its pipeline stages (similar to Figure 20.1);
3. A description of the resources available on the machine (how many of each kind of functional unit, how many instructions may be issued at once, restrictions on what kinds of instructions may be issued simultaneously, and so on).

Resource-bounded scheduling is NP-complete, meaning that there is unlikely to be an efficient optimal algorithm. As usual in this situation, we use an approximation algorithm that does reasonably well in "typical" cases.

MODULO SCHEDULING

Iterative modulo scheduling is a practical, though not optimal, algorithm for resource-bounded loop scheduling. The idea is to use iterative backtracking

	$a_1 \leftarrow j_0 \oplus b_0$	$c_1 \leftarrow e_0 \oplus j_0$	$f_1 \leftarrow U[1]$	$j_1 \leftarrow X[1]$		
	$b_1 \leftarrow a_1 \oplus f_0$	$d_1 \leftarrow f_0 \oplus c_1$	$f_2 \leftarrow U[2]$	$j_2 \leftarrow X[2]$		
	$e_1 \leftarrow b_1 \oplus d_1$	$V[1] \leftarrow b_1$	$W[1] \leftarrow d_1$	$a_2 \leftarrow j_1 \oplus b_1$		
	$b_2 \leftarrow a_2 \oplus f_1$	$c_2 \leftarrow e_1 \oplus j_1$	$f_3 \leftarrow U[3]$	$j_3 \leftarrow X[3]$		
	$d_2 \leftarrow f_1 \oplus c_2$	$V[2] \leftarrow b_2$	$a_3 \leftarrow j_2 \oplus b_2$			
	$e_2 \leftarrow b_2 \oplus d_2$	$W[2] \leftarrow d_2$	$b_3 \leftarrow a_3 \oplus f_2$	$f_4 \leftarrow U[4]$	$j_4 \leftarrow X[4]$	
	$c_3 \leftarrow e_2 \oplus j_2$	$V[3] \leftarrow b_3$	$a_4 \leftarrow j_3 \oplus b_3$		$i \leftarrow 3$	
$L:$	$d_i \leftarrow f_{i-1} \oplus c_i$	$b_{i+1} \leftarrow a_i \oplus f_i$				
	$e_i \leftarrow b_i \oplus d_i$	$W[i] \leftarrow d_i$	$V[i+1] \leftarrow b_{i+1}$	$f_{i+2} \leftarrow U[i+2]$	$j_{i+2} \leftarrow X[i+2]$	
	$c_{i+1} \leftarrow e_i \oplus j_i$	$a_{i+2} \leftarrow j_{i+1} \oplus b_{i+1}$	$i \leftarrow i+1$	if $i < N-2$ goto L		
	$d_{N-1} \leftarrow f_{N-2} \oplus c_{N-1}$	$b_N \leftarrow a_N \oplus f_{N-1}$				
	$e_{N-1} \leftarrow b_{N-1} \oplus d_{N-1}$	$W[N-1] \leftarrow d_{N-1}$	$V[N] \leftarrow b_N$			
	$c_N \leftarrow e_{N-1} \oplus j_{N-1}$					
	$d_N \leftarrow f_{N-1} \oplus c_N$					
	$e_N \leftarrow b_N \oplus d_N$	$W[N] \leftarrow d_N$				

FIGURE 20.7. Pipelined schedule. Assignments in each row happen simultaneously; each right-hand side refers to the value *before* the assignment. The loop exit test $i < N+1$ has been "moved past" three increments of i, so appears as $i < N-2$.

	$a_1 \leftarrow j_0 \oplus b_0$	$c_1 \leftarrow e_0 \oplus j_0$	$f_1 \leftarrow U[1]$	$j_1 \leftarrow X[1]$	
	$b_1 \leftarrow a_1 \oplus f_0$	$d_1 \leftarrow f_0 \oplus c_1$	$f'' \leftarrow U[2]$	$j_2 \leftarrow X[2]$	
	$e_1 \leftarrow b_1 \oplus d_1$	$V[1] \leftarrow b_1$	$W[1] \leftarrow d_1$	$a_2 \leftarrow j_1 \oplus b_1$	
	$b_2 \leftarrow a_2 \oplus f_1$	$c_2 \leftarrow e_1 \oplus j_1$	$f' \leftarrow U[3]$	$j' \leftarrow X[3]$	
	$d \leftarrow f_1 \oplus c_2$	$V[2] \leftarrow b_2$	$a \leftarrow j_2 \oplus b_2$		
	$e_2 \leftarrow b_2 \oplus d_2$	$W[2] \leftarrow d_2$	$b \leftarrow a \oplus f''$	$f \leftarrow U[4]$	$j \leftarrow X[4]$
	$c \leftarrow e_2 \oplus j_2$	$V[3] \leftarrow b$	$a \leftarrow j' \oplus b$		$i \leftarrow 3$
$L:$	$d \leftarrow f'' \oplus c$	$b \leftarrow a' \oplus f'$	$b' \leftarrow b;\ a' \leftarrow a;\ f'' \leftarrow f';\ f' \leftarrow f;\ j'' \leftarrow j';\ j' \leftarrow j$		
	$e \leftarrow b' \oplus d$	$W[i] \leftarrow d$	$V[i+1] \leftarrow b$	$f \leftarrow U[i+2]$	$j \leftarrow X[i+2]$
	$c \leftarrow e \oplus j''$	$a \leftarrow j' \oplus b$	$i \leftarrow i+1$	if $i < N-2$ goto L	
	$d \leftarrow f'' \oplus c$	$b \leftarrow a \oplus f'$	$b' \leftarrow b$		
	$e \leftarrow b' \oplus d$	$W[N-1] \leftarrow d$	$V[N] \leftarrow b$		
	$c \leftarrow e \oplus j'$				
	$d \leftarrow f' \oplus c$				
	$e \leftarrow b \oplus d$	$W[N] \leftarrow d$			

FIGURE 20.8. Pipelined schedule, with move instructions.

to find a good schedule that obeys the functional-unit and data-dependence constraints, and then perform register allocation.

The algorithm tries to place all the instructions of the loop body in a schedule of Δ cycles, assuming that there will also be a prologue and epilogue of the kind used by the Aiken-Nicolau algorithm. The algorithm tries increasing values of Δ until it reaches a value for which it can make a schedule.

A key idea of *modulo scheduling* is that if an instruction violates functional-unit constraints at time t, then it will not fit at time $t + \Delta$, or at any time t' where $t \equiv t'$ modulo Δ.

Suppose, for example, we are trying to schedule Program 20.4b with $\Delta = 3$ on a machine that can perform only one load instruction at a time. The following loop-body schedule is illegal, with two different loads at cycle 1:

0		
1	$f_i \leftarrow U[i]$	$j_i \leftarrow X[i]$
2		

We can move f_i from cycle 1 of the loop to cycle 0, or cycle 2:

0	$f_i \leftarrow U[i]$	
1		$j_i \leftarrow X[i]$
2		

0		
1		$j_i \leftarrow X[i]$
3	$f_i \leftarrow U[i]$	

Either one avoids the resource conflict. We could move f_i even earlier, to cycle -1, where (in effect) we are computing f_{i+1}; or even later, to cycle 3, where we are computing f_{i-1}:

0		
1		$j_i \leftarrow X[i]$
2	$f_{i+1} \leftarrow U[i+1]$	

0	$f_{i-1} \leftarrow U[i-1]$	
1		$j_i \leftarrow X[i]$
3		

But with $\Delta = 3$ we can never solve the resource conflict by moving f_i from cycle 1 to cycle 4 (or to cycle -2), because $1 \equiv 4$ modulo 3; the calculation of f would still conflict with the calculation of j:

0		
1	$f_{i-1} \leftarrow U[i-1]$	$j_i \leftarrow X[i]$
2		

Effects on register allocation. Consider the calculation of $d \leftarrow f \oplus c$, which occurs at cycle 0 of the schedule in Figure 20.7. If we place the calculation of d in a later cycle, then the data-dependence edges from the definitions of f and c to this instruction would lengthen, and the data-dependence edges

from this instruction to the use of d in $W[i] \leftarrow d$ would shrink. If a data-dependence edge shrinks to less than zero cycles, then a data-dependence constraint has been violated; this can be solved by also moving the calculations that use d to a later cycle.

Conversely, if a data-dependence edge grows many cycles long, then we must carry several "versions" of a value around the loop (as we carry f, f', f'' around the loop of Figure 20.8), and this means that we are using more temporaries, so that register allocation may fail. In fact, an *optimal* loop-scheduling algorithm should consider register allocation simultaneously with scheduling; but it is not clear whether optimal algorithms are practical, and the *iterated modulo scheduling* algorithm described in this section first schedules, then does register allocation and hopes for the best.

FINDING THE MINIMUM INITIATION INTERVAL

Modulo scheduling begins by finding a lower bound for the number of cycles in the pipelined loop body:

Resource estimator: For any kind of functional unit, such as a multiplier or a memory-fetch unit, we can see how many cycles such units will be used by the corresponding instructions (e.g. multiply or load, respectively) in the loop body. This, divided by the number of that kind of functional unit provided by the hardware, gives a lower bound for Δ. For example, if there are 6 multiply instructions that each use a multiplier for 3 cycles, and there are two multipliers, then $\Delta \geq 6 \cdot 3/2$.

Data-dependence estimator: For any data-dependence cycle in the data-dependence graph, where some value x_i depends on a chain of other calculations that depends on x_{i-1}, the total latency of the chain gives a lower bound for Δ.

Let $\Delta_{\min}$ be the maximum of these estimators.

Let us calculate $\Delta_{\min}$ for Program 20.4b. For simplicity, we assume that one $\oplus$-arithmetic instruction and one load/store can be issued at a time, and every instruction finishes in one cycle; and we will not consider the scheduling of $i \leftarrow i + 1$ or the conditional branch.

Then the *arithmetic resource estimator* is 5 $\oplus$-instructions in the loop body divided by 1 issuable arithmetic instructions per cycle, or $\Delta \geq 5$. The *load/store resource estimator* is 4 load/store instructions in the loop body divided by 1 issuable memory operations per cycle, or $\Delta \geq 4$. The data-dependence estimator comes from the cycle $c_i \rightarrow d_i \rightarrow e_i \rightarrow c_{i+1}$ in Graph 20.5a, whose length gives $\Delta \geq 3$.

Next, we prioritize the instructions of the loop body by some heuristic that decides which instructions to consider first. For example, instructions that are in critical data-dependence cycles, or instructions that use a lot of scarce resources, should be placed in the schedule first, and then other instructions can be filled in around them. Let $H_1, \ldots, H_n$ be the instructions of the loop body, in (heuristic) priority order.

In our example, we could use $H = [c, d, e, a, b, f, j, g, h]$, putting early the instructions that are in the critical recurrence cycle or that use the arithmetic functional unit (since the resource estimators for this loop tell us that arithmetic is in more demand than load/stores).

The scheduling algorithm maintains a set S of scheduled instructions, each scheduled for a particular time t. The value of $SchedTime[h] = none$ if $h \notin S$, otherwise $SchedTime[h]$ is the currently scheduled time for h. The members of S obey all resource and data-dependence constraints.

Each iteration of Algorithm 20.9 places the highest-priority unscheduled instruction h into S, as follows:

1. In the earliest time slot (if there is one) that obeys all dependence constraints with respect to *already-placed predecessors* of h, and respects all resource constraints.
2. But if there is no slot in Δ consecutive cycles that obeys resource constraints, then there can never be such a slot, because the functional units available at time t are the same as those at $t + c \cdot \Delta$. In this case, h is placed without regard to resource constraints, in the earliest time slot that obeys dependence constraints (with respect to already-placed predecessors), and is later than any previous attempt to place h.

Once h is placed, other instructions are removed to make the subset schedule S legal again: any successors of h that now don't obey data-dependence constraints, or any instructions that have resource conflicts with h.

This placement-and-removal could iterate forever, but most of the time either it finds a solution quickly or there is no solution, for a given Δ. To cut the algorithm off if it does not find a quick solution, a *Budget* of $c \cdot n$ schedule placements is allowed (for $c = 3$ or some similar number), after which this value of Δ is abandoned and the next one is tried.

When a def-use edge associated with variable j becomes longer than Δ cycles, it becomes necessary to have more than one copy of j, with MOVE instructions copying the different-iteration versions in bucket-brigade style. This is illustrated in Figure 20.8 for variables a, b, f, j, but I will not show an explicit algorithm for inserting the moves.

```
for Δ ← Δmin to ∞
    Budget ← n · 3
    for i ← 1 to n
        LastTime[i] ← 0
        SchedTime[i] ← none
    while Budget > 0  and  there are any unscheduled instructions
            Budget ← Budget − 1
            let h be the highest-priority unscheduled instruction
            tmin ← 0
            for each predecessor p of h
                if SchedTime[p] ≠ none
                    tmin ← max(tmin, SchedTime[p] + Delay(p, h))
            for t ← tmin to tmin + Δ − 1
                if SchedTime[h] = none
                    if h can be scheduled without resource conflicts
                        SchedTime[h] ← t
            if SchedTime[h] = none
                    SchedTime[h] ← max(tmin, 1 + LastTime[h])
            LastTime[h] ← SchedTime[h]
            for each successor s of h
                if SchedTime[s] ≠ none
                    if SchedTime[h] + Delay(h, s) > SchedTime[s]
                        SchedTime[s] ← none
            while the current schedule has resource conflicts
                    let s be some instruction (other than h) involved in a resource conflict
                    SchedTime[s] ← none
    if all instructions are scheduled
            RegisterAllocate()
            if register allocation succeeded without spilling
                    return and report a successfully scheduled loop.

Delay(h, s) =
    Given a dependence edge h_i → s_{i+k}, so that h uses the value of s from the kth previous iteration
      (where k = 0 means that h uses the current iteration's value of s);
    Given that the latency of the instruction that computes s is l cycles;
    return  l − kΔ
```

ALGORITHM 20.9. Iterative modulo scheduling.

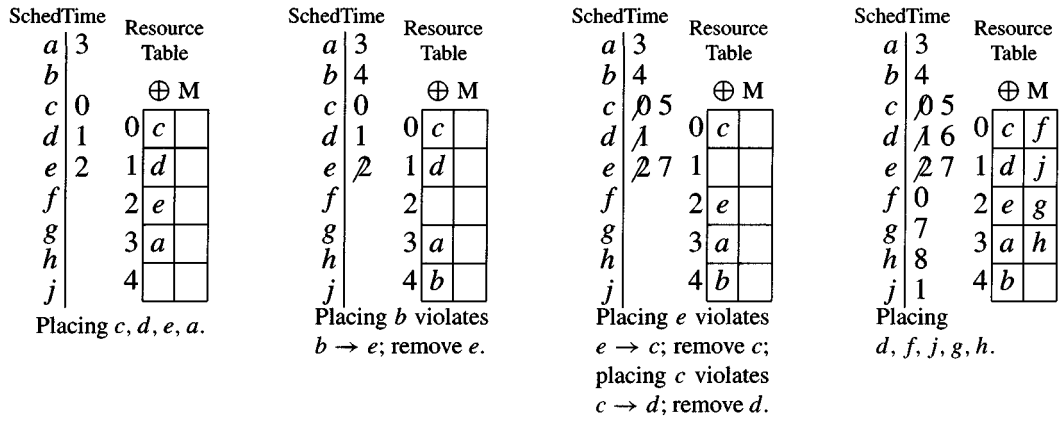


FIGURE 20.10. Iterative modulo scheduling applied to Program 20.4b. Graph 20.5a is the data-dependence graph; $\Delta_{min} = 5$ (see page 479); $H = [c, d, e, a, b, f, j, g, h]$.

Checking for resource conflicts is done with a *resource reservation table*, an array of length Δ. The resources used by an instruction at time t can be entered in the array at position t mod Δ; adding and removing resource-usage from the table, and checking for conflicts, can be done in constant time.

This algorithm is not guaranteed to find an optimal schedule in any sense. There may be an optimal, register-allocable schedule with initiation-interval Δ, and the algorithm may fail to find any schedule with time Δ, or it may find a schedule for which register-allocation fails. The only consolation is that it is reported to work very well in practice.

The operation of the algorithm on our example is shown in Figure 20.10.

OTHER CONTROL FLOW

I have shown scheduling algorithms for simple straight-line loop bodies. What if the loop contains internal control flow, such as a tree of if-then-else statements? One approach is to compute both branches of the loop, and then use a *conditional move* instruction (provided on many high-performance machines) to produce the right result.

For example, the loop at left can be rewritten into the loop at right, using a conditional move:

```
for i ← 1 to N                    for i ← 1 to N
    x ← M[i]                          x ← M[i]
    if x > 0                          u' ← z * x
      u ← z * x                       u ← A[i]
    else u ← A[i]                     if x > 0 move u ← u'
    s ← s + u                         s ← s + u
```

The resulting loop body is now straight-line code that can be scheduled easily.

But if the two sides of the **if** differ greatly in size, and the frequently executed branch is the small one, then executing both sides in every iteration will be slower than optimal. Or if one branch of the **if** has a side effect, it must not be executed unless the condition is true.

To solve this problem we use *trace scheduling*: we pick some frequently executed straight-line path through the branches of control flow, schedule this path efficiently, and suffer some amount of ineffiency at those times where we must jump into or out of the trace. See Section 8.2 and also the Further Reading section of this chapter.

SHOULD THE COMPILER SCHEDULE INSTRUCTIONS?

Many machines have hardware that does dynamic instruction rescheduling at run time. These machines do *out-of-order execution*, meaning that there may be several decoded instructions in a buffer, and whichever instruction's operands are available can execute next, even if other instructions that appeared earlier in the program code are still awaiting operands or resources.

Such machines first appeared in 1967 (the IBM 360/91), but did not become common until the mid-1990s. Now it appears that most high-performance processors are being designed with dynamic (run-time) scheduling. These machines have several advantages and disadvantages, and it is not yet clear whether static (compile-time) scheduling or out-of-order execution will become standard.

Advantages of static scheduling. Out-of-order execution uses expensive hardware resources and tends to increase the chip's cycle time and wattage. The static scheduler can schedule earlier the instructions whose future data-dependence path is longest; a real-time scheduler cannot know the length of the data-dependence path leading from an instruction (see Exercise 20.3). The scheduling problem is NP-complete, so compilers – which have no real-time constraint on their scheduling algorithms – should in principle be able to find better schedules.

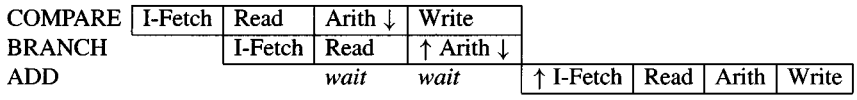


FIGURE 20.11. Dependence of ADD's instruction-fetch on result of BRANCH.

Advantages of dynamic scheduling. Some aspects of the schedule are unpredictable at compile time, such as cache misses, and can be better scheduled when their actual latency is known (see Figure 21.5). Highly pipelined schedules tend to use many registers; typical machines have only 32 register names in a five-bit instruction-field, but out-of-order execution with run-time *register renaming* can use hundreds of actual registers with a few static names (see the Further Reading section). Optimal static scheduling depends on knowing the precise pipeline state that will be reached by the hardware, which is sometimes difficult to determine in practice. Finally, dynamic scheduling does not require that the program be recompiled (i.e. rescheduled) for each different implementation of the same instruction set.

20.3 BRANCH PREDICTION

In many floating-point programs, such as Program 20.4a, the basic blocks are long, the instructions are long-latency floating-point operations, and the branches are very predictable **for**-loop exit conditions. In such programs the problem, as described in the previous sections, is to schedule the long-latency instructions.

But in many programs – such as compilers, operating systems, window systems, word processors – the basic blocks are short, the instructions are quick integer operations, and the branches are harder to predict. Here the main problem is fetching the instructions fast enough to be able to decode and execute them.

Figure 20.11 illustrates the pipeline stages of a COMPARE, BRANCH, and ADD instruction. Until the BRANCH has executed, the instruction-fetch of the successor instruction cannot be performed because the address to fetch is unknown.

Suppose a superscalar machine can issue four instructions at once. Then, in waiting three cycles after the BRANCH is fetched before the ADD can be fetched, 11 instruction-issue slots are wasted (3×4 minus the slot that the BRANCH occupies).

Some machines solve this problem by fetching the instructions immediately following the branch; then if the branch is not taken, these fetched-and-decoded instructions can be used immediately. Only if the branch is taken are there stalled instruction slots. Other machines assume the branch will be taken, and begin fetching the instructions at the target address; then if the branch falls through, there is a stall. Some machines even fetch from both addresses simultaneously, though this requires a very complex interface between processor and instruction-cache.

Modern machines rely on *branch prediction* to make the right guess about which instructions to fetch. The branch prediction can be *static* – the compiler predicts which way the branch is likely to go and places its prediction in the branch instruction itself; or *dynamic* – the hardware remembers, for each recently executed branch, which way it went last time, and predicts that it will go the same way.

STATIC BRANCH PREDICTION

The compiler can communicate predictions to the hardware by a 1-bit field of the branch instruction that encodes the predicted direction.

To save this bit, or for compatibility with old instruction sets, some machines use a rule such as "backward branches are assumed to be taken, forward branches are assumed to be not-taken." The rationale for the first part of this rule is that backward branches are (often) loop branches, and a loop is more likely to continue than to exit. The rationale for the second part of the rule is that it's often useful to have predicted-not-taken branches for exceptional conditions; if *all* branches are predicted taken, we could reverse the sense of the condition to make the exceptional case "fall through" and the normal case take the branch, but this leads to worse instruction-cache performance, as discussed in Section 21.2. When generating code for machines that use forward/backward branch direction as the prediction mechanism, the compiler can order the basic blocks of the program in so that the predicted-taken branches go to lower addresses.

Several simple heuristics help predict the direction of a branch. Some of these heuristics make intuitive sense, but all have been validated empirically:

Pointer: If a loop performs an equality comparison on pointers (`p=null` or `p=q`) then predict the condition as *false*.

Call: A branch is *less* likely to the successor that dominates a procedure call (many conditional calls are to handle exceptional situations).

Return: A branch is *less* likely to a successor that dominates a return-from-procedure.

Loop: A branch is *more* likely to the successor (if any) that is the header of the loop containing the branch.

Loop: A branch is *more* likely to the successor (if any) that is a loop preheader, if it does not postdominate the branch. This catches the results of the optimization described in Figure 18.7, where the iteration count is more likely to be > 0 than $= 0$. (B postdominates A if any path from A to program-exit must go through B; see Section 19.5.)

Guard: If some value r is used as an operand of the branch (as part of the conditional test), then a branch is *more* likely to a successor in which r is live and which does not postdominate the branch.

There are some branches to which more than one of the heuristics apply. A simple approach in such cases is to give the heuristics a priority order and use the first heuristic in order that applies (the order in which they are listed above is a reasonable prioritization, based on empirical measurements).

Another approach is to index a table by every possible subset of conditions that might apply, and decide (based on empirical measurements) what to do for each subset.

SHOULD THE COMPILER PREDICT BRANCHES?

Perfect static prediction results in a dynamic mispredict rate of about 9% (for C programs) or 6% (for Fortran programs). The "perfect" mispredict rate is not zero because any given branch does not go in the same direction more than 91% of the time, on average. If a branch did go the same direction 100% of the time, there would be little need for it! Fortran programs tend to have more predictable branches because more of the branches are loop branches, and the loops have longer iteration counts.

Profile-based prediction, in which a program is compiled with extra instructions to count the number of times each branch is taken, executed on sample data, and recompiled with prediction based on the counts, approaches the accuracy of perfect static prediction.

Prediction based the heuristics described above results in a dynamic mispredict rate of about 20% (for C programs), or about half as good as perfect (or profile-based) static prediction.

A typical hardware-based branch-prediction scheme uses two bits for every branch in the instruction cache, recording how the branch went the last two times it executed. This leads to misprediction rates of about 11% (for C

programs), which is about as good as profile-based prediction.

A mispredict rate of 10% can result in very many stalled instructions – if each mispredict stalls 11 instruction slots, as described in the example on page 484, and there is one mispredict every 10 branches, and one-sixth of all instructions are branches, then 18% of the processor's time is spent waiting for mispredicted instruction fetches. Therefore it will be necessary to do better, using some combination of hardware and software techniques. Relying on heuristics that mispredict 20% of the branches is better than no predictions at all, but will not suffice in the long run.

FURTHER READING

Hennessy and Patterson [1996] explain the design and implementation of high-performance machines, instruction-level parallelism, pipeline structure, functional units, caches, out-of-order execution, register renaming, branch prediction, and many other computer-architecture issues, with comparisons of compiler versus run-time-hardware techniques for optimization. Kane and Heinrich [1992] describe the pipeline constraints of the MIPS R4000 computer, from which Figures 20.1 and 20.2 are adapted.

CISC computers of the 1970s implemented complex instructions sequentially using an internal microcode that could do several operations simultaneously; it was not possible for the compiler to interleave parts of several macroinstructions for increased parallelism. Fisher [1981] developed an automatic scheduling algorithm for microcode, using the idea of trace scheduling to optimize frequently executed paths, and then proposed a very-long-instruction-word (VLIW) architecture [Fisher 1983] that could expose the microoperations directly to user programs, using the compiler to schedule.

Aiken and Nicolau [1988] were among the first to point out that a single loop iteration need not be scheduled in isolation, and presented the algorithm for optimal (ignoring resource constraints) parallelization of loops.

Many variations of the multiprocessor scheduling problem are NP-complete [Garey and Johnson 1979; Ullman 1975]. The *iterative modulo scheduling* algorithm [Rau 1994] gets good results in practice. In the absence of resource constraints, it is equivalent to the Bellman-Ford shortest-path algorithm [Ford and Fulkerson 1962]. Optimal schedules can be obtained (in principle) by expressing the constraints as an integer linear program [Govindarajan et al.

1996], but integer-linear-program solvers can take exponential time (the problem is NP-complete), and the register-allocation constraint is still difficult to express in linear inequalities.

Ball and Larus [1993] describe and measure the static branch-prediction heuristics shown in Section 20.3. Young and Smith [1994] show a profile-based static branch-prediction algorithm that does *better* than optimal static predition; the apparent contradiction in this statement is explained by the fact that their algorithm replicates some basic blocks, so that a branch that's 80% taken (with a 20% misprediction rate) might become two different branches, one almost-always taken and one almost-always not taken.

EXERCISES

20.1 Schedule the following loop using the Aiken-Nicolau algorithm:

```
for i ← 1 to N
    a ← X[i − 2]
    b ← Y[i − 1]
    c ← a × b
    d ← U[i]
    e ← X[i − 1]
    f ← d + e
    g ← d × c
h : X[i] ← g
j : Y[i] ← f
```

a. Label all the scalar variables with subscripts i and $i-1$. **Hint:** In this loop there are no loop-carried scalar-variable dependences, so none of the subscripts will be $i-1$.

b. Perform *scalar replacement* on uses of $X[\,]$ and $Y[\,]$. **Hint:** Now you will have subscripts of $i-1$ and $i-2$.

c. Perform *copy propagation* to eliminate variables a, b, e.

d. Draw a data-dependence graph of statements c, d, f, g, h, j; label intra-iteration edges with 0 and loop-carried edges with 1 or 2, depending on the number of iterations difference there is in the subscript.

e. Show the Aiken-Nicolau *tableau* (as in Table 20.6a).

f. Find the identical groups separated by increasing gaps. **Hint:** The identical groups will be c cycles apart, where in this case c is greater than one!

g. Show the steepest-slope group. **Hint:** The slope is not an integer.

h. Unroll the loop k times, where k is the denominator of the slope.

i. Draw the data-dependence graph of the unrolled loop.

j. Draw the tableau for the schedule of the unrolled loop.

k. Find the slope of the steepest-slope group. **Hint:** Now it should be an integer.

l. Move the shallow-slope group(s) down to close the gap.

m. Identify the loop body, the prologue, and the epilogue.

n. Write a schedule showing placement of the prologue, loop body, and epilogue in specific cycles, like Figure 20.7.

o. Eliminate the subscripts on variables in the loop body, inserting move instructions where necessary, as in Figure 20.8.

20.2 Do parts a–d of Exercise 20.1. Then use iterated modulo scheduling to schedule the loop for a machine that can issue three instructions at a time, of which at most one can be a memory instruction and at most one can be a multiply instruction. Every instruction completes in one cycle.

e. Explicitly represent the increment instruction $i_{i+1} \leftarrow i_i + 1$ and the loop branch k : *if* $i_{i+1} \leq N$ *goto loop* in the data-dependence graph, with an edge from i to itself (labeled by 1), from i to k (labeled by 0), and from k to every node in the loop body (labeled by 1).

f. Calculate $\Delta_{\min}$ based on data-dependence cycles, the 2-instruction per cycle limit, the 1-load/store-per-cycle limit, and the 1-multiply-per-cycle limit. Remark: the Δ required for a data-dependence cycle is the length of the cycle divided by the sum of the edge-labels (where edge labels show iteration distance, as described in Exercise 20.1d).

g. Run Algorithm itermod, showing the SchedTime and Resource tables each time a variable has to be removed from the schedule, as in Figure 20.10. Use the priority order $H = [i, k, c, d, g, f, h, j]$.

h. Eliminate the subscripts on variables in the loop body, inserting move instructions where necessary, as in Figure 20.8. If the move instructions don't fit into the 3-instruction-issue limit, then it's time to increase Δ and try again.

20.3 Consider the following program:

L :	L :
a : $a \leftarrow U[i]$	a : $a \leftarrow U[i]$
b : $b \leftarrow a \times a$	d : $d \leftarrow d \times a$
c : $V[i] \leftarrow b$	b : $b \leftarrow a \times a$
i : $i \leftarrow i + 1$	c : $V[i] \leftarrow b$
d : $d \leftarrow d \times a$	i : $i \leftarrow i + 1$
e : if $d < 1.0$ goto L	e : if $d < 1.0$ goto L
(I) Unscheduled	(II) Scheduled

Suppose these loops are to be run on an out-of-order execution machine with these characteristics: Each instruction takes exactly one cycle, and may be executed as soon as its operands are ready *and* all preceding conditional branches have been executed. Several instructions may be executed at once, except that there is only one multiply unit. If two multiply instructions are ready, the instruction from an earlier iteration, or occurring first in the same iteration, is executed.

The program was originally written as shown in loop (I); the compiler has rescheduled it as loop (II). For each of the two loops:

a. Draw the data-dependence graph, showing loop-carried dependences with a dashed line.

b. Add the control dependence as a loop-carried edge from e to each of the other nodes.

c. To simulate how the machine will execute the loop, show the Aiken-Nicolau *tableau*, with the restriction that b and d must never be put in the same cycle. In a cycle where b and d's predecessors are both ready, prefer the instruction from the earlier iteration, or from earlier in the same iteration.

d. Compute the steepest slope in the tableau; how many cycles per iteration does the loop take?

e. Can compiler scheduling be useful for dynamically rescheduling (out-of-order execution) machines?

20.4 On many machines, instructions after a conditional branch can be executed even before the branch condition is known (the instructions do not *commit* until after the branch condition is verified).

Suppose we have an out-of-order execution machine with these characteristics: An add or branch takes one cycle; a multiply takes 4 cycles; each instruction may be executed as soon as its operands are ready. Several instructions may be executed at once, except that there is only one multiply unit. If two multiply

instructions are ready, the instruction from an earlier iteration, or occurring first in the same iteration, is executed.

For a machine with this behavior, do parts a–e of Exercise 20.3 for the following programs:

$L:$	$L:$
$a: a \leftarrow e \times u$	$b: b \leftarrow e \times v$
$b: b \leftarrow e \times v$	$a: a \leftarrow e \times u$
$c: c \leftarrow a + w$	$c: c \leftarrow a + w$
$d: d \leftarrow c + x$	$d: d \leftarrow c + x$
$e: e \leftarrow d + y$	$e: e \leftarrow d + y$
$f:$ if $e > 0.0$ goto L	$f:$ if $e > 0.0$ goto L
(I) Unscheduled	(II) Scheduled

20.5 Write a short program that contains an instance of each of the branch-prediction heuristics *(pointer, call, return, loop header, loop preheader, guard)* described on pages 485–486. Label each instance.

20.6 Use branch-prediction heuristics to predict the direction of each of the conditional branches in the programs of Exercise 8.6 (page 185) and Figure 18.7b (page 414); explain which heuristic applies to each prediction.