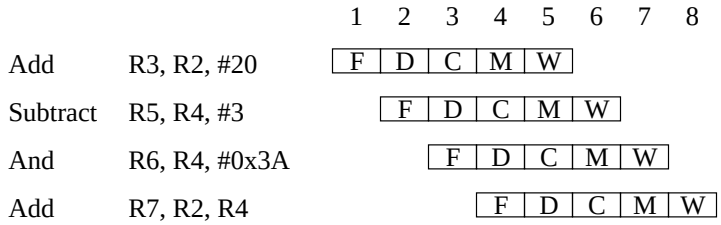
Chapter 6
Pipelining

6.1. (*a*) The pipeline execution diagram for the given code is shown below.



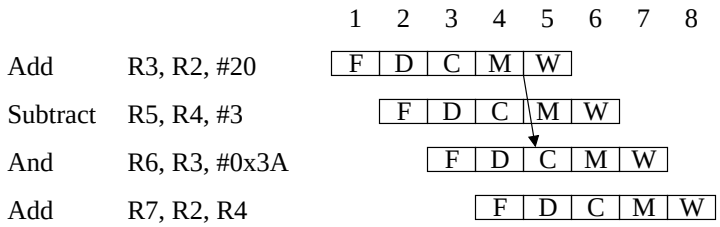
The description of activity in each stage during each cycle is given below. Activity for instructions prior to the first Add instruction and instructions after the second Add instruction is not described, but would nonetheless occur as determined by the type of instruction and the stage of the pipeline.

Cycle	Stage	Activity
1	F	fetching Add instruction
2	F	fetching Subtract instruction
	D	decoding Add instruction, reading register R2 (value 2000)
3	F	fetching And instruction
	D	decoding Subtract instruction, reading register R4 (value 50)
	C	performing arithmetic $2000 + 20 = 2020$ for Add instruction
4	F	fetching Add instruction
	D	decoding And instruction, reading register R4 (value 50)
	C	performing arithmetic $50 - 3 = 47$ for Subtract instruction
	M	no operation for Add instruction
5	D	decoding Add instruction, reading register R2 (value 2000) and register R4 (value 50)
	C	performing logic operation $50 \text{ AND } 3\text{A}_{16} = 50$ for And instruction
	M	no operation for Subtract instruction
	W	write result of 2020 for Add instruction to register R3
6	C	performing arithmetic $2000 + 50 = 2050$ for Add instruction
	M	no operation for And instruction
	W	write result of 47 for Subtract instruction to register R5
7	M	no operation for Add instruction
	W	write result of 50 for And instruction to register R6
8	W	write result of 2050 for Add instruction to register R7

(*b*) The contents of each register during each cycle are described in the table below. The notation '–' is used to indicate either that there is insufficient information to determine the register contents in that cycle, or the value is not relevant for the operation(s) performed by the corresponding instruction. The instruction occupying the IR in each cycle is identified symbolically from the given sequence of instructions.

	1	2	3	4	5	6	7	8
IR	–	Add	Sub	And	Add	–	–	–
PC	1000	1004	1008	1012	1016	1020	1024	1028
RA	–	–	2000	50	50	2000	–	–
RB	–	–	–	–	–	50	–	–
RZ	–	–	–	2020	47	50	2050	–
RY	–	–	–	–	2020	47	50	2050

6.2. (*a*) The pipeline execution diagram for the given code is shown below.



The description of activity in each stage during each cycle is the same as in part (*a*) of Problem 6.1, except that the previous contents of register R3 are read by the And instruction in cycle 4, but the Compute stage uses the new value of 2020 that has been computed by the first Add instruction. The differences with the solution for part (*a*) of Problem 6.1 are highlighted in boldface in the partial table given below.

Cycle	Stage	Activity
4	F	fetching Add instruction
	D	decoding And instruction, **reading register R3 (value unknown)**
	C	performing arithmetic $50 - 3 = 47$ for Subtract instruction
	M	no operation for Add instruction
5	D	decoding Add instruction, reading register R2 (value 200) and register R4 (value 50)
	C	performing logic operation **2020 AND $3A_{16}$ = 32** for And instruction
	M	no operation for Subtract instruction
	W	write result of 2020 for Add instruction to register R3
6	C	performing arithmetic $2000 + 50 = 2050$ for Add instruction
	M	no operation for And instruction
	W	write result of 47 for Subtract instruction to register R5
7	M	no operation for Add instruction
	W	write result of **32** for And instruction to register R6

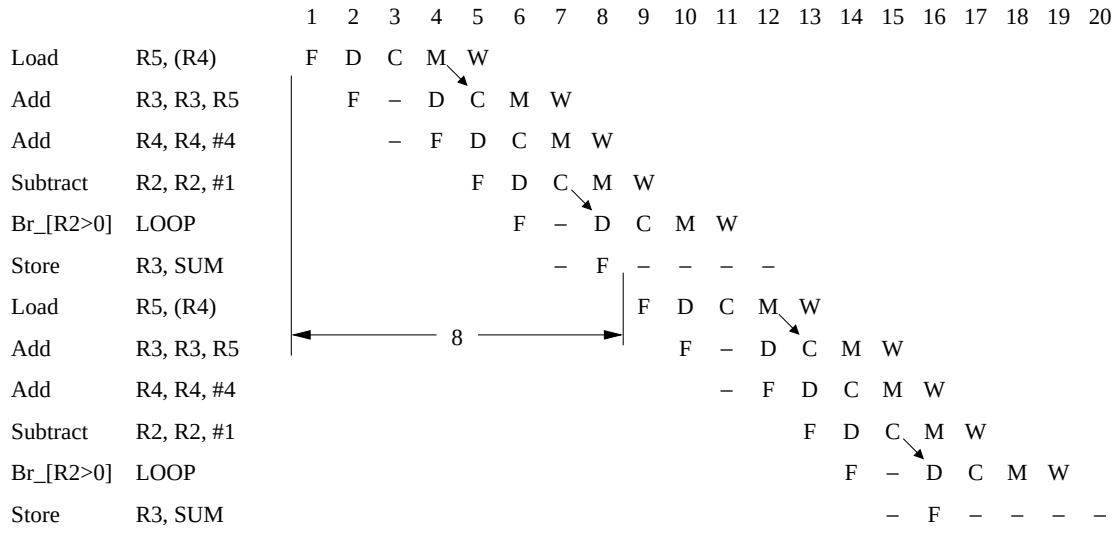
(*b*) The contents of each register during each cycle are described in the table below. The differences with the equivalent table in part (*b*) of Problem 6.1 are highlighted in italics.

	1	2	3	4	5	6	7	8
IR	–	Add	Sub	And	Add	–	–	–
PC	1000	1004	1008	1012	1016	1020	1024	1028
RA	–	–	2000	***R3 value***	50	2000	–	–
RB	–	–	–	–	–	50	–	–
RZ	–	–	–	2020	47	***32***	2050	–
RY	–	–	–	–	2020	47	***32***	2050

The contents of RZ in cycle 6 and RY in cycle 7 are determined as follows:

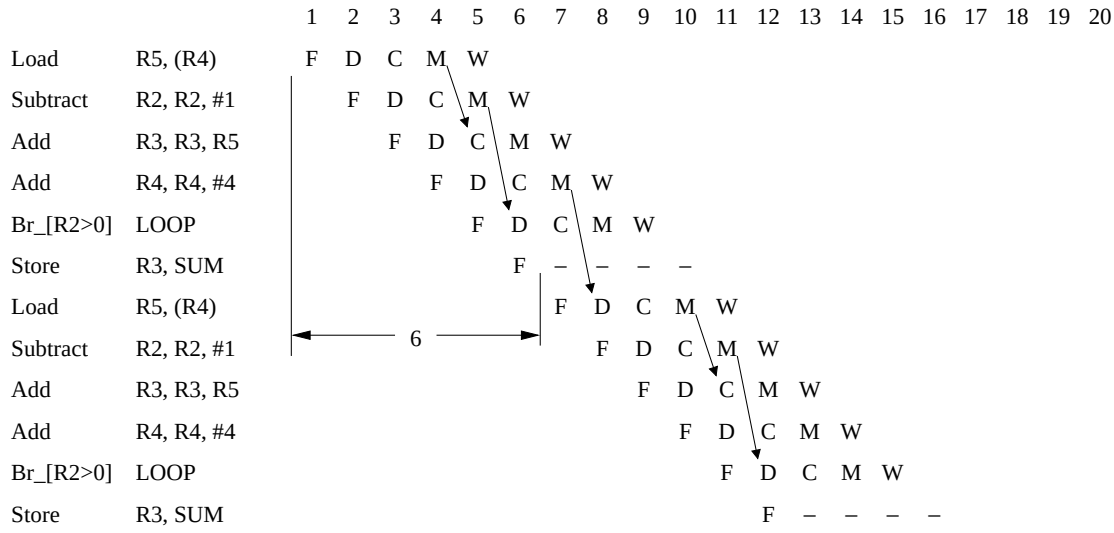
$$(2020 \text{ AND } 3\text{A}_{16}) = (7\text{E}4_{16} \text{ AND } 3\text{A}_{16}) = 20_{16} = 32$$

6.3. The problem statement indicated forwarding paths to the ALU from RY and RZ. For branch instructions to incur only a one-cycle penalty as discussed in Sections 6.6.1 and 6.6.2, additional forwarding paths to the dedicated comparator in the Decode stage are also required. The resulting pipeline execution diagram for the given code is shown below.



There are 8 cycles between the starting points of successive passes through the loop. For correct forwarding of the results of the Load and Subtract instructions, a stall cycle is necessary for each case to ensure that the immediately following instruction uses the correct operand value. Without the additional forwarding path to the Decode stage as described above, each Branch instruction would have to stall for a total of three cycles to allow the Subtract result to be placed in register R2.

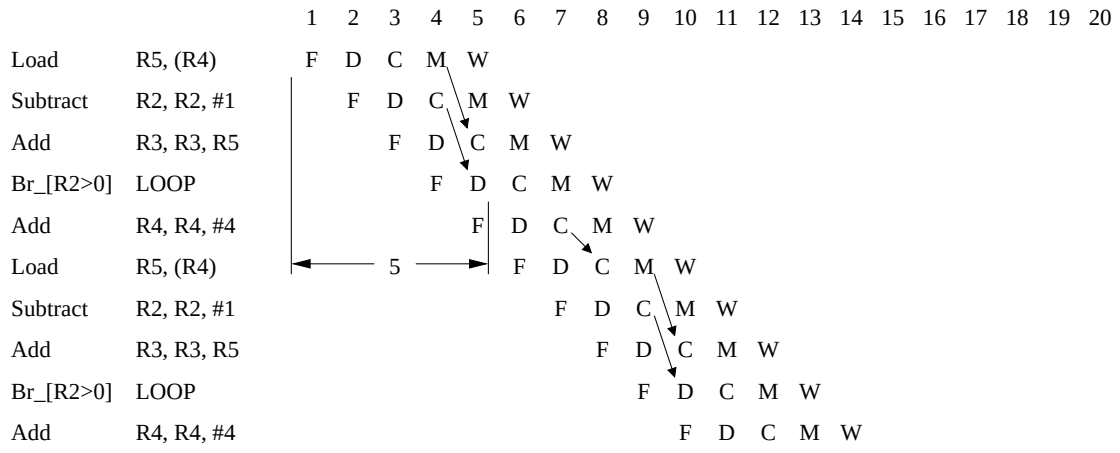
6.4. To enhance performance, instructions the loop body can be reordered to eliminate the stall cycles in each pass through the loop. The reordering enables an instruction to effectively fill a stall cycle with useful activity. As shown below, there are now 6 cycles between the starting points of successive passes through the loop, but still two instances of forwarding between instructions *within* one loop pass.



Moreover, the reordering also introduces another forwarding situation *across* successive loop passes for the new value of register R4. The new value in register RY is forwarded to the Decode stage, as described in the solution to Problem 6.3 for branch comparisons. The forwarded value would be captured into register RA or register RB for later use in the Compute stage, in addition to being used within the Decode stage for the comparison.

6.5. With a branch delay slot, a useful instruction should be placed in that position to execute to completion in every pass through the loop. Unlike the solutions for Problem 6.3 and Problem 6.4, the position after the branch instruction will not be occupied by the the Store instruction that would be fetched and discarded.

As shown in the diagram below, reordering the instructions to utilize the branch delay slot results in an execution where there are 5 cycles between the starting points of successive passes through the loop and two instances of forwarding between instructions *within* each loop pass.



Similar to the solution of Problem 6.3, there is also a forwarding situation between instructions *across* loop passes. In this case, the value of register RZ is forwarded to the Compute stage.

6.6. The statement of the problem indicated that the Compute stage generates the value 17 when the Add instruction is being fetched, and that Compute generates the value 198 when LShiftL instruction is being fetched. With that information, the following pipeline execution diagram can be generated, along with an indication that the value 17 is placed in register RZ at the beginning of cycle 2, and the value 198 at the beginning of cycle 3.

				1	2	3	4	5	
previous instr.		F	D	"17"	M	W			
previous instr.			F	D	"198"	M	W		
Add	R3, R2, R1			F	D	C	M	W	
LShiftL	R3, R3, #1				F	D	C	M	W
values in RZ:					17	198	130	260	

The statement of the problem also indicated that the values of registers R1 and R2 are 30 and 100. Hence, the computed value of 130 for the Add instruction is placed in register RZ at the beginning of cycle 4. Finally, the value of 130 is forwarded from RZ to the Compute stage during cycle 4 for the shift operation of the LShiftL instruction. The shifted result of 260 is placed in RZ at the beginning of cycle 5.

6.7. Let N represent the total number of instructions that are executed for the program. Branches constitute 20 percent of the total, or $0.2N$. Each branch has one delay slot. If all of the delay slots are occupied with NOP instructions, the total number of cycles (neglecting the last four cycles to complete the execution of the final instruction) is $N + 0.2N = 1.2N$.

If it is possible to usefully fill 70 percent of the delay slots with another instruction (leaving 30 percent of the slots with NOP instructions), then the total number of cycles is $N + (0.2 \times 0.3)N = 1.06N$.

The speedup of the second case relative to the initial case is

$$\left(\frac{1.2}{1.06} - 1\right) \times 100 = 13 \text{ percent}$$

6.8. Let N represent the total number of instructions that are executed. Branches constitute 20 percent of the total, or $0.2N$. Each branch has *two* delay slots. If all of the delay slots are occupied with NOP instructions, the total number of cycles (neglecting the last four cycles to complete the execution of the final instruction) is $N + 2(0.2N) = 1.4N$.

If it is possible to usefully fill the first delay slot 70 percent of the time, and the second delay slot 10 percent of the time, then the total number of cycles is $N + 0.2((2 \times 0.3)N + (0.7 \times 0.9)N) = 1.25N$. The term 2×0.3 accounts for the two-cycle penalty in the 30 percent of cases when neither delay slot is filled with a useful instruction. For the other 70 percent of the cases when the first delay slot is filled, 90 percent of that subset has the second delay slot filled with a NOP. The term 0.7×0.9 accounts for just a one-cycle penalty in those cases. All remaining cases have no penalty.

The clock rates for the processors in Problems 6.7 and 6.8 are the same. Hence, in terms of the total number of cycles, the result from Problem 6.7 ($1.06N$) is better than the result for this problem ($1.25N$). The speedup is

$$\left(\frac{1.25}{1.06} - 1\right) \times 100 = 18 \text{ percent}$$

6.9. Let N represent the total number of instructions that are executed. Branches constitute 20 percent of the total, or $0.2N$. The branches are taken 75 percent of the time.

With static prediction (where branches are assumed to be not-taken), the execution time is $N + (0.2 \times 0.75)N = 1.15N$. In other words, the not-taken assumption is incorrect 75 percent of the time, incurring a penalty of one cycle each time.

With dynamic prediction, let f_{wrong} represent the fraction of executed branches that are mispredicted. The execution time is therefore $N + (0.2 \times f_{\text{wrong}})N = (1 + 0.2 \times f_{\text{wrong}})N$.

(*a*) For the execution time with dynamic prediction to be no worse than the execution time with static prediction, it must be the case that $(1 + 0.2 \times f_{\text{wrong}}) \leq 1.15$. After rearrangment of the equation, the solution is $f_{\text{wrong}} \leq 0.75$. This result is equivalent to the requirement that the dynamic prediction must be correct at least 25 percent of the time.

(*b*) If branches are correctly predicted 90 percent of the time, then $f_{\text{wrong}} = 0.1$ and the execution time with dynamic prediction is $1.02N$. The speedup of dynamic prediction in this case over static prediction is

$$\left(\frac{1.15}{1.02} - 1\right) \times 100 = 13 \text{ percent}$$

6.10. The additional control logic required to forward the value of RZ to the ALU must select the correct input to MuxA and MuxB. For the logic to make the correct selection, appropriate information is necessary.

In the interstage buffers, it is necessary to have the register identifiers for the source operands, as well as the register identifier for the destination. These identifiers are introduced into the pipeline when the instruction is decoded, and they move to later stages of the pipeline.

The additional logic compares register identifiers held in different interstage buffers. The destination identifier in interstage buffer B3 feeding the Memory stage is compared against each of the source identifiers in interstage buffer B2 feeding the Compute stage. If there is a match between the destination and one of the sources, then there is a potential dependency. The source register identifier that matches dictates which multiplexer requires an appropriate control signal input setting to select the value of RZ for the corresponding ALU input.

Note, however, that the comparisons described above are meaningful only if the control signal setting for writing a destination register is 1 in interstage buffer B3 feeding the Memory stage. That control signal setting causes the destination register to be modified later in the Write stage. In such a case, the match between the destination register identifier in interstage buffer B3 and one of the source register identifiers in interstage buffer B2 represents a *true* dependency, and forwarding is required. When the register-write control signal setting is 0 in interstage buffer B3, then there is no dependency regardless of any match from the register-identifier comparisons.

There is also a more subtle consideration for any instruction set architecture which specifies that the contents of register R0 are required to always be the constant zero. If R0 is named as the destination register of an instruction for which the register-write control signal setting is 1, any apparent dependency involving R0 with another instruction should be ignored. *No* forwarding of any new result destined for R0 should be done because that result will *not* be written to R0. Thus, the control logic discussed above must perform an additional comparison of the destination register identifier against the R0 register identifier in order to detect this situation.

6.11. Additional logic is required to forward the value of RY to the ALU.

First, MuxA and MuxB in Figure 6.5 must be expanded with an extra input for the value of RY, in addition to the value of RZ.

Second, new register-identifier comparisons must be introduced into the control logic already described in the solution to Problem 6.10. To determine whether the value of register RY should be forwarded, the destination register identifier in interstage buffer B4 feeding the Write stage must be compared with the source register identifiers in interstage buffer B2 feeding the Compute stage. The register-write control signal setting in interstage buffer B4 must also be used to determine whether or not a true dependency exists. Furthermore, consideration must be given to the possibility of constant register R0 as a destination.

6.12. The statement of the problem describes a situation in which there are two apparent dependencies arising from register-identifier matches between the Memory and Compute stages (Add $\rightarrow$ Or), and also between the Write and Compute stages (Subtract $\rightarrow$ Or), *at the same time*.

In fact, the dependency between the Memory and the Compute stages is the only the true dependency because it reflects the most recently generated value for the register involved. The value from the Memory stage is one that should be forwarded. The result in the Write stage is the older result and will be written first to the destination register. The result in the Memory stage is the newer result that will be written later to the same register. The instruction in the Compute stage must use the newer result to correctly reflect the semantics of the instruction sequence being executed.

The logic discussed in Problems 6.10 and 6.11 must be implemented in a manner that gives priority to a dependence between the Memory and the Compute stages when the situation described above arises. In other words, forwarding from the Write stage to the Compute stage happens only if there is no dependence from the Memory stage involving the same source register in the Compute stage.

Similar to the solutions of Problems 6.10 and 6.11, the control logic for the situation described here still requires that the register-write control signal settings in the Memory and Write stages be considered to verify that a true dependency exists, and that the possible use of constant register R0 as the destination also be considered.

6.13. The instruction sequence consists of four memory-access instructions and four arithmetic instructions.

Using the *original* order of instructions, there are two cases to consider for dispatching pairs of instructions in the superscalar processor.

Case 1: The processor imposes the restriction that two instructions being dispatched in the same cycle must be of distinct types to reduce the complexity at each execution unit. In the pipeline execution diagram below, memory-access instructions are denoted as M_i and arithmetic instructions are denoted as A_j.

	1	2	3	4	5	6	7	8	9
M_1	F	D	C	M	W				
M_2	F		D	C	M	W			
M_3		F		D	C	M	W		
M_4		F			D	C	M	W	
A_1			F	D	C	W			
A_2			F		D	C	W		
A_3				F		D	C	W	
A_4				F			D	C	W

Case 2: The processor allows two instructions of the same type to be dispatched in the same cycle with more complex buffering at each execution unit. Simultaneously dispatched instructions of the same type are still handled with pipelining in their execution unit, so the completion of all activity requires the same number of cycles as the first case above.

	1	2	3	4	5	6	7	8	9
M_1	F	D	C	M	W				
M_2	F	D		C	M	W			
M_3		F	D		C	M	W		
M_4		F	D			C	M	W	
A_1			F	D	C	W			
A_2			F	D		C	W		
A_3				F	D		C	W	
A_4				F	D			C	W

Now, rather than preserving the original order of instructions, it is possible to *interleave* the memory-access and arithmetic instructions in a revised instruction sequence. If such an interleaved sequence is used, the pipeline execution diagram below shows that one less cycle is required to complete all activity.

	1	2	3	4	5	6	7	8	9
M_1	F	D	C	M	W				
A_1	F	D	C	W					
M_2		F	D	C	M	W			
A_2		F	D	C	W				
M_3			F	D	C	M	W		
A_3			F	D	C	W			
M_4				F	D	C	M	W	
A_4				F	D	C	W		

6.14. The sequence of N instructions consists of 75 percent arithmetic instructions and 25 percent memory-access (Load/Store) instructions. There are no branch instructions.

On a basic five-stage pipelined processor, the execution time is N cycles (neglecting the final four cycles for the last instructions).

On a superscalar processor, the arithmetic instructions (the largest fraction) pass one at a time through the corresponding execution unit and therefore dictate the execution time as $0.75N$ cycles.

The speedup with superscalar execution is

$$\left(\frac{1}{0.75} - 1\right) \times 100 = 33 \text{ percent}$$

If the distribution of instructions had been 50 percent each for the arithmetic and memory-access categories, the ideal speedup would have been a factor of 2, or 100 percent. The uneven distribution between the two categories of instructions given in this question limits the achievable speedup.

6.15. The sequence of N instructions consists of 65 percent arithmetic instructions, 25 percent memory-access (Load/Store) instructions, and 15 percent branch instructions that are never taken.

On a basic five-stage pipelined processor, the ideal execution time for a sequence of N instructions is N cycles. (Even static prediction for never-taken branches would result in no branch penalties.)

On a superscalar processor, the branches are handled in the fetch unit; they are not dispatched to the arithmetic unit or the memory unit. The contribution of 65 percent for arithmetic instructions dictates the execution time, assuming perfect branch prediction and no penalties. Hence, the execution time is $0.65N$ cycles.

The speedup with superscalar execution is

$$\left(\frac{1}{0.65} - 1\right) \times 100 = 54 \text{ percent}$$

6.16. For the proposed branch-related enhancement of the instruction set, different prediction hints are given below for the cases considered in this question.

(*a*) For a loop with a backward branch at the end (and no other branch for loop exit in the body of the loop), it can be assumed that the number of passes will be larger than one, hence the *likely-to-be-taken (LT)* hint can be used to initialize the prediction history.

(*b*) For a loop with a forward conditional branch at the beginning to exit the loop and backward unconditional branch to make a another pass through the loop, the LT hint can be used to initialize the prediction history of the backward branch at the end, just as in part (*a*). For the forward branch, the opposite hint of *likely-not-to-be-taken (LNT)* can be used to intialize the branch history.

6.17. The assembly-language program and prediction hints for the IF-THEN statement in this question are given below.

(*a*) A convenient approach for generating assembly-language instruction for an IF-THEN-ELSE statement is to use the opposite of the comparison condition given in the high-level language statement for branching to the sequence of instructions corresponding to the ELSE clause, allowing the not-taken outcome of the branch instruction to continue to the THEN clause. In this manner, the order of the THEN and ELSE clauses in the assembly-language representation matches the high-level representation. The THEN clause does, however, require an unconditional branch to skip over the ELSE clause.

```
IF:       Load                    R2, A
          Load                    R3, B
          Branch_if_[R2]≤[R3]     ELSE
THEN:     Add                     R2, R2, #1
          Store                   R2, A
          Branch                  END_IF
ELSE:     Add                     R3, R3, #1
          Store                   R3, B
END_IF:   next instruction
```

(*b*) With no other information to bias the preference for the prediction hint of the conditional branch, either LT or LNT can be chosen as the initial prediction for that instruction. For the unconditional branch, however, LT should be the initial prediction.

(*c*) If it is known that $A > B$ most of the the time, then the LNT hint should be used to initialize the prediction history of the conditional branch so that the instruction sequence beginning at the label THEN is executed the first time that the branch instruction is fetched. The unconditional branch should still use the LT hint.

6.18. The two cases for execution analysis described in this question are discussed below.

(*a*) The assembly-language instructions for the solution to this problem are the same as the solution given for Problem 6.17. First, execution is considered for static prediction with the branch-not-taken assumption. If $A > B$, then the conditional branch is not taken, hence there is no branch penalty. The unconditional branch does, however, introduce a penalty of one cycle. Finally, the dependence between the Load instruction and the conditional branch requires two stall cycles. Six instructions would be executed in this case, contributing one cycle each, but three additional cycles would also be required due to a branch penalty and load-to-branch stalls. A total of nine cycles are required before the instruction at the label END_IF can be fetched.

If $A \leq B$, then the conditional branch is taken, which incurs a penalty of one cycle. The ELSE clause does not have an unconditional branch, hence there is one less instruction executed than for the first case above. The two stall cycles due the load-to-branch dependence are still incurred. With five executed instructions, a branch penalty, and two stall cycles, the total is eight cycles before the instruction at the label END_IF can be fetched.

(*b*) Now, execution is considered for delayed branching with one delay slot. In this case, the conventional sequence of instructions (as shown in the solution to Problem 6.17) cannot be used — something must be done with the delay slot after each branch instruction. The assembly-language instructions and the dependencies are such that the delay slot after the conditional branch cannot be filled with an earlier instruction, hence a NOP must be used. The two stall cycles due to the dependency involving the conditional branch are still incurred. For the unconditional branch, the preceding Store instruction can be moved to fill that delay slot.

For the case of $A > B$, the total is eight cycles before the instruction at the label END_IF can be fetched.

For the case of $A \leq B$, the NOP in the delay slot of the conditional branch has the same contribution to execution time as the discarded Add instruction in part (*a)*. The unconditional branch is not executed in this case, hence the total is also eight cycles before the instruction at the label END_IF can be fetched.