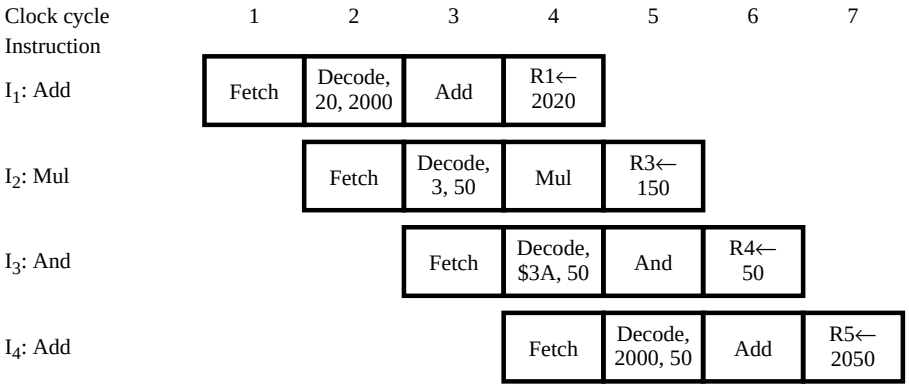
Chapter 8 – Pipelining

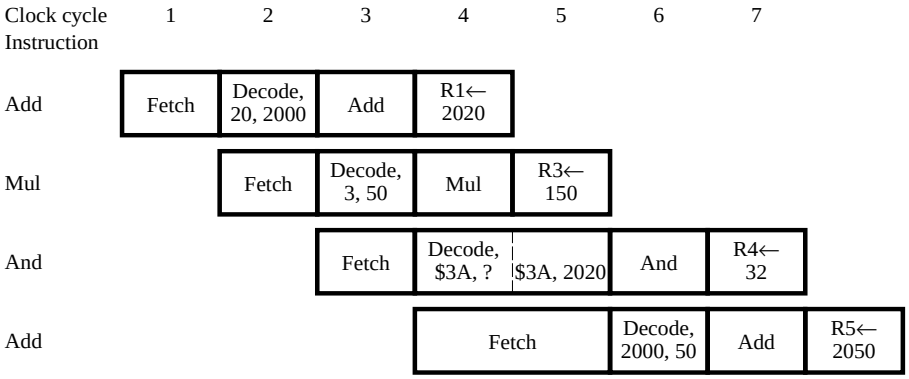
8.1. (a) The operation performed in each step and the operands involved are as given in the figure below.



(b)

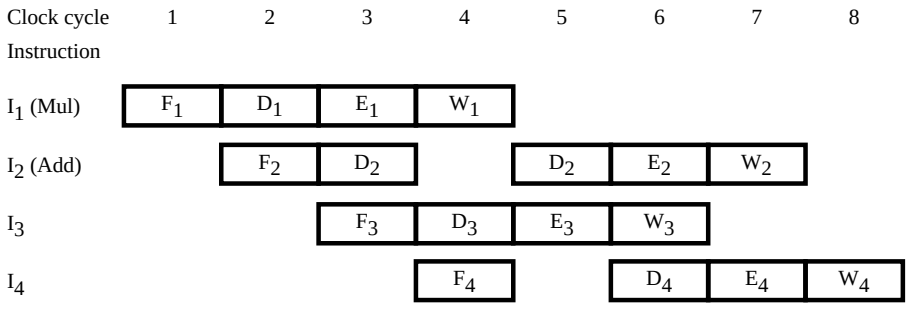
Clock cycle	2	3	4	5
Buffer B1	Add instruction (I_1)	Mul instruction (I_2)	And instruction (I_3)	Add instruction (I_4)
Buffer B2	Information from a previous instruction	Decoded I_1 Source operands: 20, 2000	Decoded I_2 Source operands: 3, 50	Decoded I_3 Source operands: $3A, 50
Buffer B3	Information from a previous instruction	Information from a previous instruction	Result of I_1: 2020 Destination = R1	Result of I_2: 150 Destination = R3

8.2. (a)

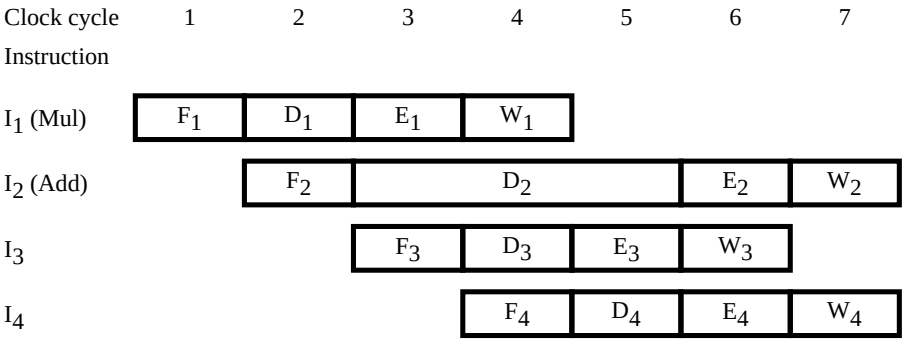


(b) Cycles 2 to 4 are the same as in P8.1, but contents of R1 are not available until cycle 5. In cycle 5, B1 and B2 have the same contents as in cycle 4. B3 contains the result of the multiply instruction.

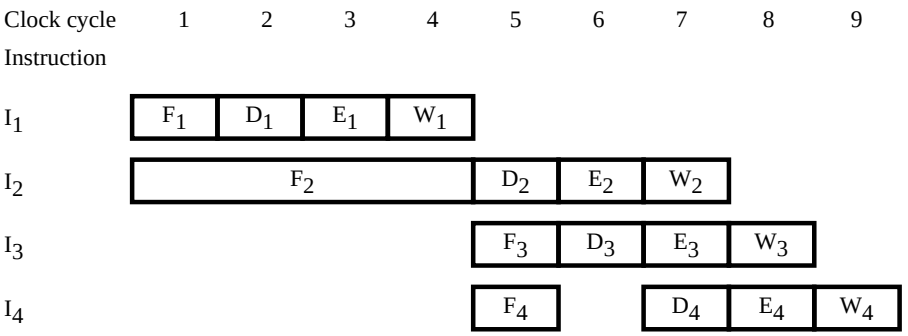
8.3. Step D_2 may be abandoned, to be repeated in cycle 5, as shown below. But, instruction I_1 must remain in buffer B1. For I_3 to proceed, buffer B1 must be capable of holding two instructions. The decode step for I_4 has to be delayed as shown, assuming that only one instruction can be decoded at a time.



8.4. If all decode and execute stages can handle two instructions at a time, only instruction I_2 is delayed, as shown below. In this case, all buffers must be capable of holding information for two instructions. Note that completing instruction I_3 before I_2 could cause problems. See Section 8.6.1.



8.5. Execution proceeds as follows.



8.6. The instruction immediately preceding the branch should be placed after the branch.

LOOP	Instruction 1		LOOP	Instruction 1
	...			...
	Instruction $n-1$	$\Rightarrow$		Instruction $n-1$
	Instruction n			Conditional Branch LOOP
	Conditional Branch LOOP			Instruction n

This reorganization is possible only if the branch instruction does not depend on instruction n.

8.7. The UltraSPARC arrangement is advantageous when the branch instruction is at the end of the loop and it is possible to move one instruction from the body of the loop into the delay slot. The alternative arrangement is advantageous when the branch instruction is at the beginning of the loop.

8.8. The instruction executed on a speculative basis should be one that is likely to be the correct choice most often. Thus, the conditional branch should be placed at the end of the loop, with an instruction from the body of the loop moved to the delay slot if possible. Alternatively, a copy of the first instruction in the loop body can be placed in the delay slot and the branch address changed to that of the second instruction in the loop.

8.9. The first branch (BLE) has to be followed by a NOP instruction in the delay slot, because none of the instructions around it can be moved. The inner and outer loop controls can be adjusted as shown below. The first instruction in the outer loop is duplicated in the delay slot following BLE. It will be executed one more time than in the original program, changing the value left in R3. However, this should cause no difficulty provided the contents of R3 are not needed once the sort is completed. The modified program is as follows:

```
        ADD     R0,LIST,R3
        ADD     R0,N,R1
        SUB     R1,1,R1
        SUB     R1,1,R2
OUTER   LDUB    [R3+R1],R5      Get LIST(j)
        LDUB    [R3+R2],R6      Get LIST(k)
INNER   SUB     R6,R5,R0
        BLE,pt  NEXT
        SUB     R2,1,R2         k ← k−1
        STUB    R5,[R3+R2]
        STUB    R6,[R3+R1]
        OR      R0,R6,R5
NEXT    BGE,pt,a INNER
        LDUB    [R3+R2],R6      Get LIST(k)
        SUB     R1,1,R1
        BGT,pt  OUTER
        SUB     R1,1,R2
```

8.10. Without conditional instructions:

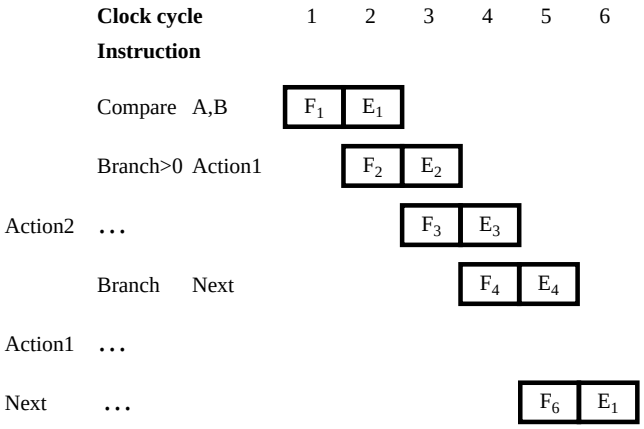
```
            Compare     A,B       Check A − B
            Branch>0    Action1
Action2     …           …         One or more instructions
            Branch      Next
Action1     …           …         One or more instructions
Next        …
```

If conditional instructions are available, we can use:

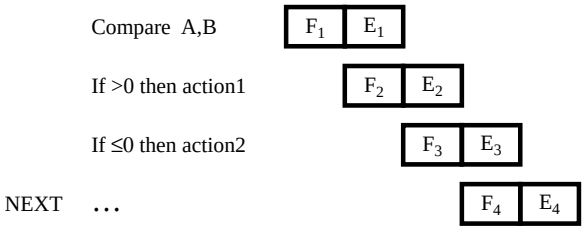
```
            Compare   A,B   Check A − B
            …         …     Action1 instruction(s), conditional
            …         …     Action2 instruction(s), conditional
Next        …
```

In the second case, all Action 1 and Action 2 instructions must be fetched and decoded to determine whether they are to be executed. Hence, this approach is beneficial only if each action consists of one or two instructions.

Without conditional instructions



With conditional instructions



8.11. Buffer contents will be as shown below.

Cycle No.	3	4	5
ALU Operation	+	Shift	O_3
R3	45	130	260
RSLT	198	130	260

8.12. Using Load and Store instructions, the program may be revised as follows:

```
INSERTION   Test        RHEAD
            Branch>0    HEAD
            Move        RNEWREC,RHEAD
            Return
HEAD        Load        RTEMP1,(RHEAD)
            Load        RTEMP2,(RNEWREC)
            Compare     RTEMP1,RTEMP2
            Branch>0    SEARCH
            Store       RHEAD,4(RNEWREC)
            Move        RNEWREC,RHEAD
            Return
SEARCH      Move        RHEAD,RCURRENT
LOOP        Load        RNEXT,4(RCURRENT)
            Test        RNEXT
            Branch=0    TAIL
            Load        RTEMP1,(RNEXT)
            Load        RTEMP2,(RNEWREC)
            Compare     RTEMP1,RTEMP2
            Branch<0    INSERT
            Move        RNEXT,RCURRENT
            Branch      LOOP
INSERT      Store       RNEXT,4(RNEWREC)
TAIL        Store       RNEWREC,4(RCURRENT)
            Return
```

This program contains many dependencies and branch instructions. There very few possibilities for instruction reordering. The critical part where optimization should be attempted is the loop. Given that no information is available on branch behavior or delay slots, the only optimization possible is to separate instructions that depend on each. This would reduce the probability of stalling the pipeline.

The loop may be reorganized as follows.

```
LOOP    Load        RNEXT,4(RCURRENT)
        Load        RTEMP2,(RNEWREC)
        Test        RNEXT
        Load        RTEMP1,(RNEXT)
        Branch=0    TAIL
        Compare     RTEMP1,RTEMP2
        Branch<0    INSERT
        Move        RNEXT,RCURRENT
        Branch      LOOP
INSERT  Store       RNEXT,4(RNEWREC)
TAIL    Store       RNEWREC,4(RCURRENT)
        Return
```

Note that we have assumed that the Load instruction does not affect the condition code flags.

8.13. Because of branch instructions, 120 clock cycles are needed to execute 100 program instructions when delay slots are not used. Using the delay slots will eliminate 0.85 of the idle cycles. Thus, the improvement is given by:

$$\frac{120}{120 - 20 \times 0.85} = 1.081$$

That is, instruction throughput will increase by 8.1%.

8.14. Number of cycles needed to execute 100 instructions:

Without optimization	140
With optimization ($140 - 20 \times 0.85 - 20 \times 0.2$)	127

Thus, throughput improvement is $140/127 = 1.102$, or 10.2%

8.15. Throughput improvement due to pipelining is n, where n is the number of pipeline stages.

	Number of cycles needed to execute one instruction:	Throughput
4-stage:	$1.20 - 0.8 \times 0.20 = 1.04$	4/1.04 = 3.85
6-stage:	$1.4 - 0.8 \times 0.2 - 0.25 \times 0.2 = 1.19$	6/1.19 = 5.04

Thus, the 6-stage pipeline leads to higher performance.

8.16. For a “do while” loop, the termination condition is tested at the beginning of the loop. A conditional branch at that location will be taken when exiting the loop. Hence, it should be predicted not taken. That is, the state machine should be started in the state LNT, unless the loop is not likely to be executed at all.

A “do until” loop is executed at least once, and the branch condition is tested at the end of the loop. Assuming that the loop is likely to be executed several times, the branch should be predicted taken. That is, the state machine should be started in state LT.

8.17. An instruction fetched in cycle j reaches the head of the queue and enters the decode stage in cycle $j + 6$. Assume that the instruction preceding I_1 is decoded and instruction I_6 is fetched in cycle 1. This leads to instructions I_1 to I_6 being in the queue at the beginning of cycle 2. Execution would then proceed as shown below.

Note that the queue is always full, because at most one instruction is dispatched and up to two instructions are fetched in any given cycle. Under these conditions, the queue length would drop below 6 only in the case of a cache miss.