Must be done Individually (no team-work)

Problem 1. Exploring Instruction Flow and and Instruction level parallelism:

ILP Consider the following pseudo-assembly code.

Top:

A. LOAD R1 = Mem[1234]
B. LOAD R2 = Mem[42]
C. ADD R4 = R1 + R2
D. LOAD R3 = Mem[1976]
E. BEQ R4, #0, Foo
F. SUB R1 = R3 - #1
G. XOR R5 = R1 ^ 0xffffffff
H. ADD R1 = R1 + 1
I. JUMP Bar

Foo:

J. LOAD R3 = Mem[2000]
K. ADD R2 = R4 + #13
L. MUL R3 = R3 × R2

Bar:

M. ADD R2 = R4 + R5
N. SUB R4 = R1 - #8
O. XOR R1 = R3 ^ 0xf0f0f0f0

Calculate the ILP(instruction level parallelism) for each of the following scenarios. Use a table to record the relevant dependencies as part of your answer. A sample table is provided at the end of this problem.

a. Observe all register data (RAW, WAR, WAW) and control dependencies and assume the conditional branch (statement E) is not-taken (falls through to F).

RAW → Following the path of direct dependency

Top to Bar only

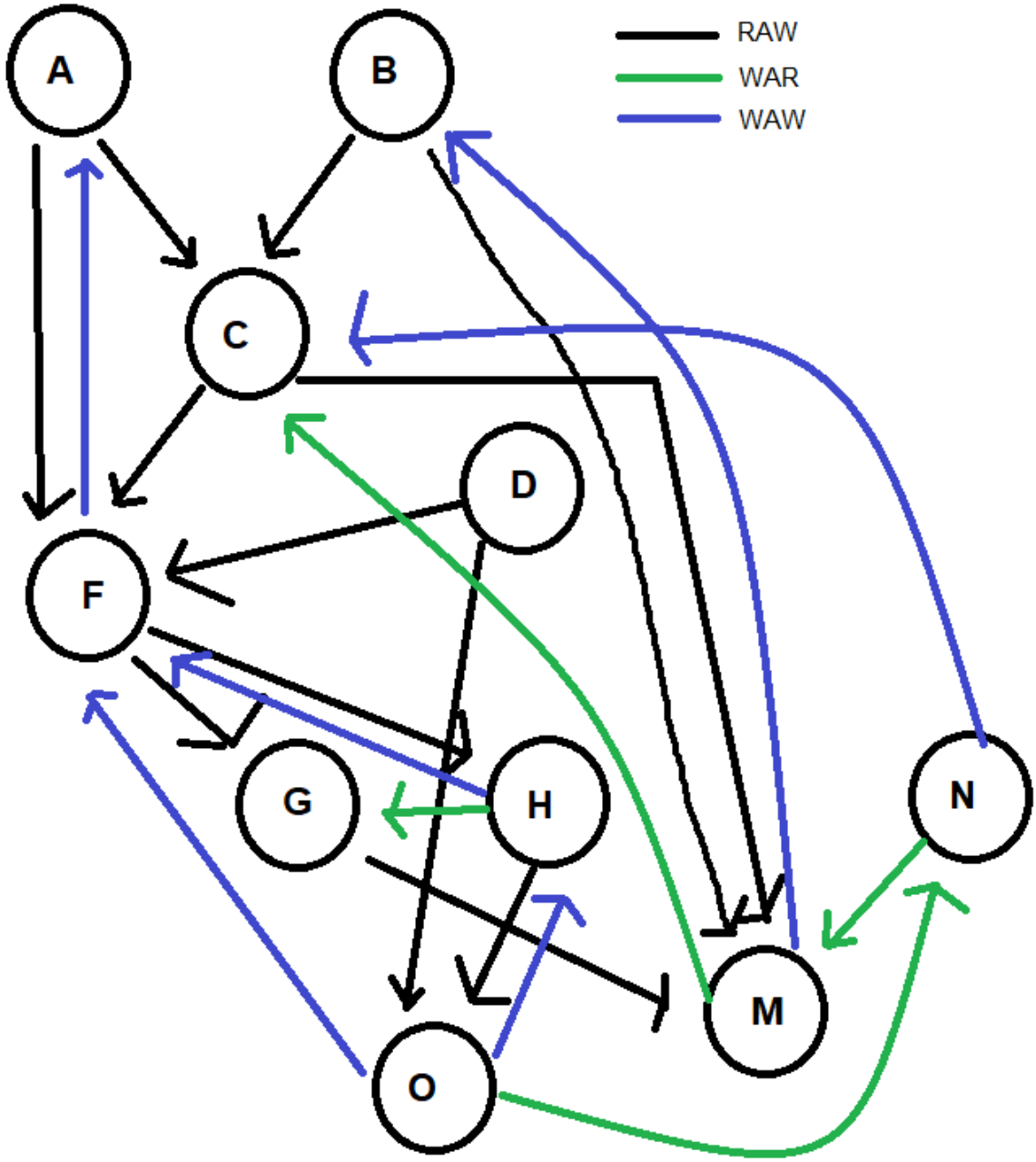
From C → E, this is what is supposed to happen in the pipeline, but since E is not taken, it Falls through to F, Hence C→E,F

From the pipeline path, we skip instruction E and go down to BAR

WAR → Later write must not overwrite a pending read

WAW → Earlier write register must not overwrite a later write register

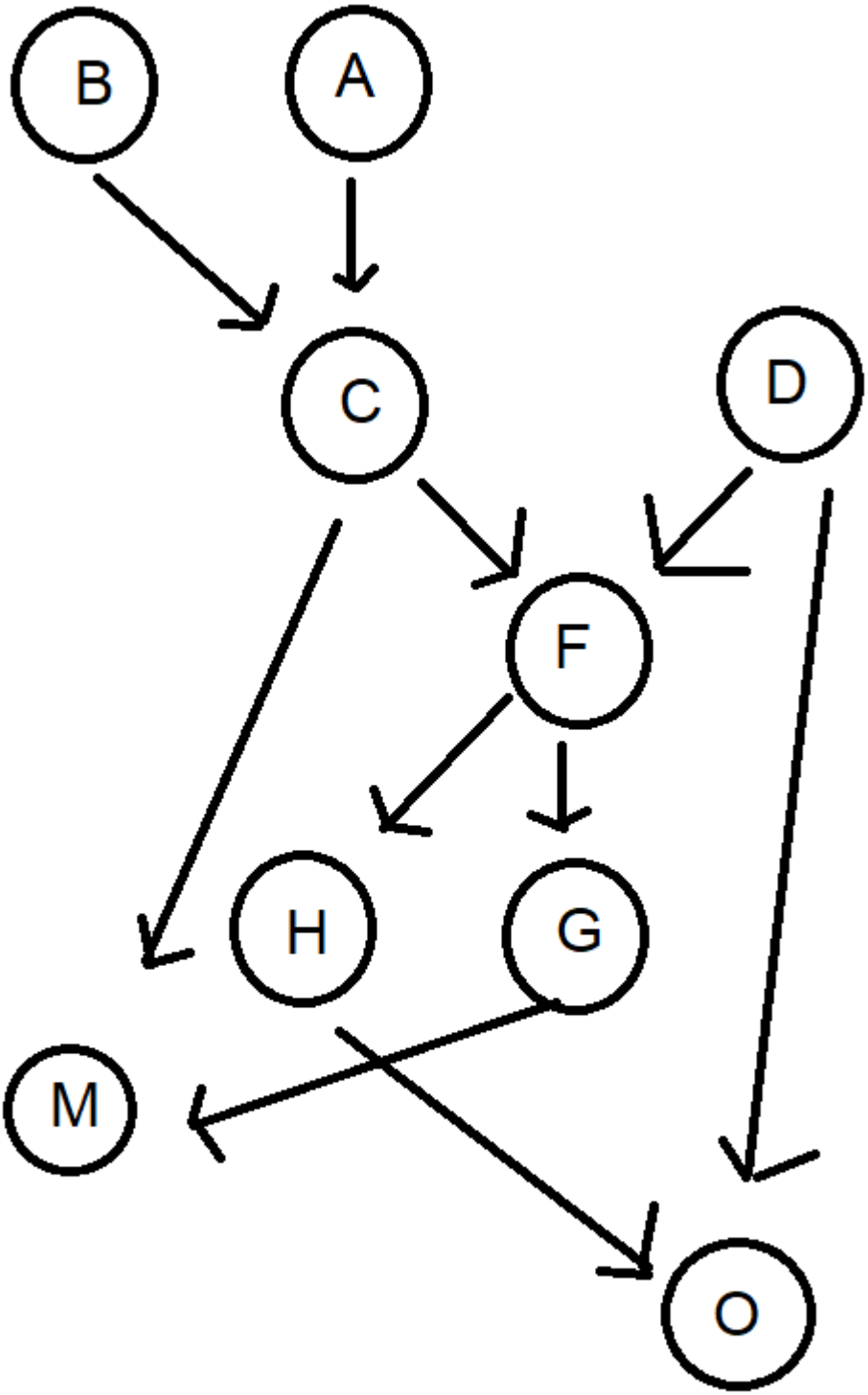
RAW	WAR	WAW
A→ C	H → G	H → F
B–>C	M → C	O → F
C→E, F	N → M	F→ A
A→ F	0 –> N	M → B
D→F		N→C
F→G		O → H
F→ H		
B→M		
C→ M		
G→ M		
H→ O		
D→ O		



ILP is defined as = (Number of instructions in dataflow graph)/(depth of dataflow graph).
where depth = number of edges
ILP_{RAW} = 9/3 = 3
ILP_{WAR} = 4/1 = 4
ILP_{WAW} = 6/2 = 3
ILP_{ALL} = 6/2 = 3

b. Repeat (a), but only for true dependencies (and no control dependencies; I.e., (F) will be executed in the instruction stream)
Assuming that E is skipped, we just continue the same as 1A, only keeping the RAW section

RAW	WAR	WAW
A→ C	H → G	H → F
B–>C	M → C	O → F
C→E, F	N → M	F→ A
D→F	0 –> N	M → B
F→G		N→C
F→ H		O → H
C→ M		
G→ M		
H→ O		
D→ O		



ILP is defined as = (Number of instructions in dataflow graph)/(depth of dataflow graph).
where depth = number of edges

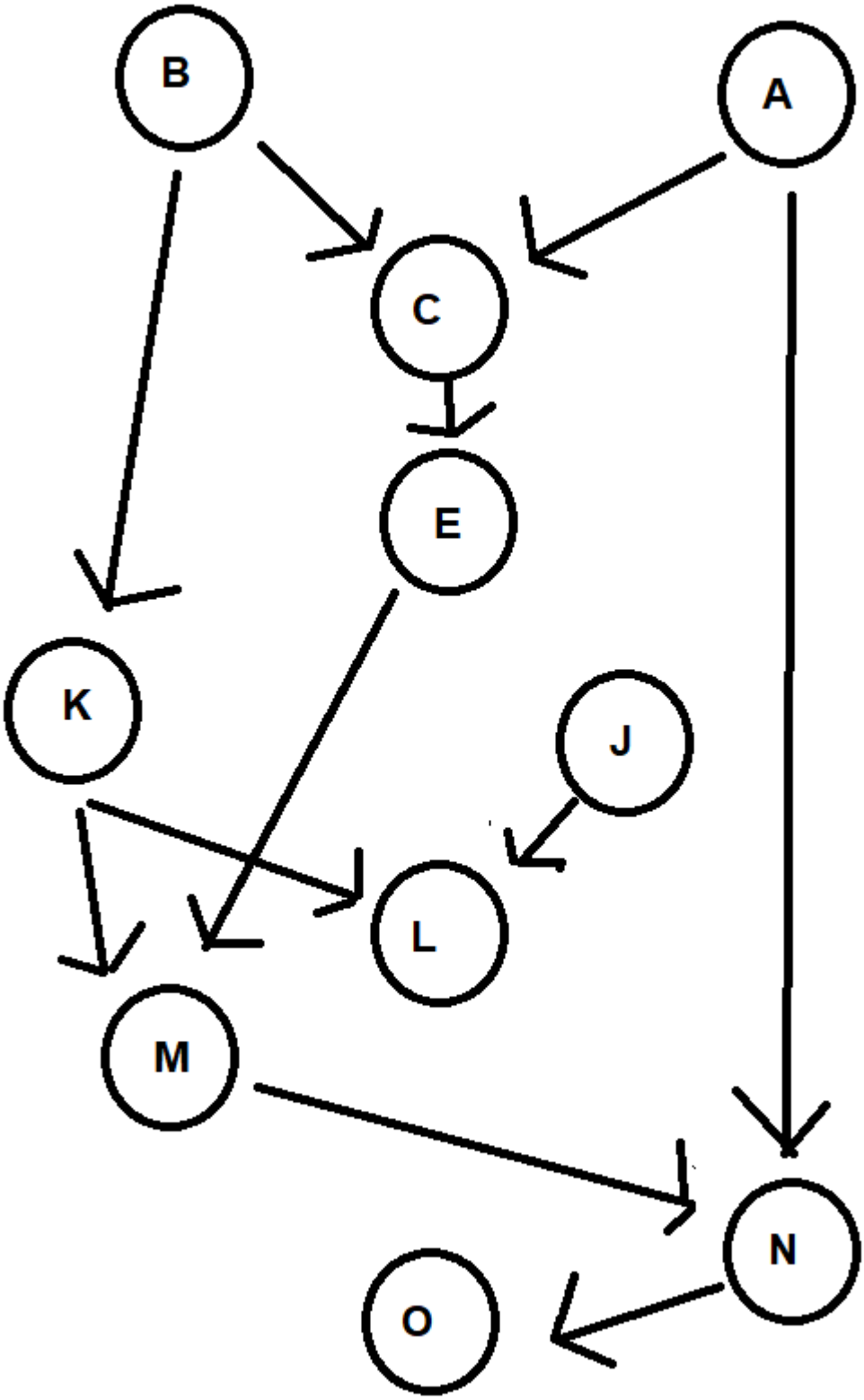
= 9/3 = 3

c. Repeat (a), but only for true dependencies (no control dependencies) and assuming (E) is taken (branch to J).

WAR → Later write must not overwrite a pending read
green is strictly the register that the data that it is being written to and is dependent on an earlier read

WAW → Earlier write register must not overwrite a later write register

RAW	WAR	WAW
A → C	K → C	E → C
B → C	L → D	K → B
C – > E	M → L	L → D
B → K	N → K	M → K
E → K	O→ N	N → E
J → L		O → A
K → L		
K→ M		
E → M		
M → N		
A → N		
N → O		
L → O		


B
A
C
E
K
J
L
M
N
O

ILP is defined as = (Number of instructions in dataflow graph)/(depth of dataflow graph).
where depth = number of edges
= 10/2 = 5

HINT: The intention of the problem is not to use a specific pipeline. It is however, to identify different kinds of data dependences and the degree of ILP (Instruction Level Parallelism) achievable

Normally you want do this graphically using a data flow graph. Where a node of the graph is one instruction (A, B, C, ...) and you have a directed edge from one node to the other if there is a dependence. You may want to use a different color for different dependence types.

Once you have the graph, you can find out what would be the maximum parallelism you would obtain for that graph (computation) if you had all the hardware you needed. So then the depth of the data flow graph gives us the minimum execution time possible for the graph. The total number of instructions in the graph will give up the execution time without any extra hardware. This means everything must be executed sequentially and one at a time.
Then ILP is defined as = (Number of instructions in dataflow graph)/(depth of dataflow graph).

Sample table to produce to show your work for each senarion:
If there is a RAW dependency between A and C and WAR dependency between F and C then

RAW	WAR	WAW
A-C	F-C	

Problem 2. Instruction Flow and Branch Prediction

Write down all assumptions. If you need more space, attach additional pages to the back, and label accordingly. Please save your work for this problem after you submit your answers. We will build on this assignment as we go through the semester and see alternative processor designs.

Introduction

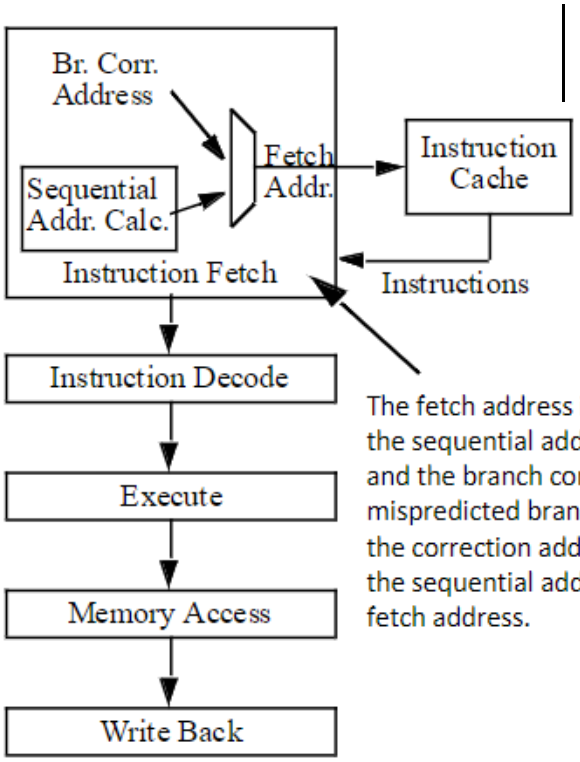
This problem investigates the effects of branches and control flow changes on program performance for a scalar pipeline (to keep the focus on branch prediction). Branch penalties increase as the number of pipeline stages increases between instruction fetch and branch resolution (or condition and target resolution). This effect of pipelined execution drives the need for branch prediction. This problem explores both static branch prediction in **Part C** (we will see dynamic branch prediction based on this design in a future assignment). For this problem the base machine is a 5-Stage pipeline.

***formatting error when converting PDF to DOC**

The 5-Stage Pipeline without Dynamic Branch Prediction

Execution Assumptions:

- unconditional branches execute in the decode stage
- conditional branches execute in the execute stage
- Effective address calculation is performed in the execute stage
- All memory access is performed in the memory access stage
- All necessary forwarding paths exist
- The register file is read after write



The fetch address is a choice between the sequential address generation logic and the branch correction logic. If a mispredicted branch is being corrected the correction address is chosen over the sequential address for the next fetch address.

Part A: Branch Penalties.

What are the branch penalties for unconditional and conditional branches?

Assuming that there are no interrupts and that everything flows smoothly, in order and without error.

Unconditional Branch executes in decode stage, so if there's a stall, there will be a minimum penalty of 3 branch cycles to finish it. Decode is in step 2 so needs at least 3 cycles to finish the instruction. If the stall happens in the beginning, then penalty of 1

Conditional Branch executes in execute stage , so if there's a stall, there will be a minimum penalty of 2 branch cycles. Execute is in step 3, so needs at least 2 cycles to go back to the end of the instruction. If the stall happens in the beginning, then penalty of 2

Unconditional _____Min 1 or 3 _____ Conditional _____2_____

Part B: No Branch Prediction.

This problem will use the bubble sort program. An execution trace, or a sequence of executed basic blocks, is provided for this problem. A basic block is a group of consecutive instructions that are always executed together in sequence.

Example Code: Bubble Sort

BB	Line#	Label	Assembly_Instruction				Comment
1	1	main:	addi	r2,	r0,	ListArray r2 <-	ListArray
	2		addi	r3,	r0,	ListLength	r3 <- ListLength
	3		add	r4,	r0,	r0	i = 0;
2	4	loop1:	bge	r4,	r3,	end	while (i < Length)
							{
3	5		addi	r5,	r4,	1	j = i + 1;
4	6	loop2:	bge	r5,	r3,	cont	while (j < Length)
							{
5	7		lw	r6,	r4(r2)		temp = ListArray[i];
	8		lw	r7,	r5(r2)		temp2 = ListArray[j];
	9		ble	r6,	r7,	skip	if (temp > temp2)
							{
6	10		sw	r7,	r4(r2)		ListArray[i] <- temp;
	11		sw r6,		r5(r2)		ListArray[j] <- temp2;
							}
7	12	skip:	addi r5,		r5,	1	j++;
	13		ba	loop2		}	
8	14	cont:	addi r4,		r4,	1	i++;
	15		ba	loop1		}	
9	16	end:	lw	r1,	(sp)		r1 <- Return Pointer
	17		ba	r1			

Execution Trace: Sequence of Basic Blocks Executed:

1 2 3 4 5 6 7 4 5 7 4 5 7 4 8 2 3 4 5 6 7 4 5 7 4 8 2 3 4 5 6 7 4 8 2 3 4 8 2 9

[Hint: An alternate way to represent the same execution trace above is to use the sequence of branch instructions, both conditional and unconditional (i.e. ba), executed.]
Main, Loop1, Loop 2, Loop1, Loop1, etc

1. Fill in the branch execution table with an NT for not taken and a T for taken. This table is recording the execution pattern for each (static) branch instruction. Use the execution trace on Page 2.

Branch Execution - Assume No Branch Prediction:
Also Assuming one bit predictions

Execution Trace: Sequence of Basic Blocks Executed:
1 2 3 4 5 6 7 4 5 7 4 5 7 4 8 2 3 4 5 6 7 4 5 7 4 8 2 3 4 8 2 9

Line 4 - Block 2 - 5 times - following through the execution, it only takes at the last iteration where it finally ends and skips to cont - taken once
Line 6 - Block 4 - 10 times - taken 4 times
Line 9 - Block 5 - 6 times - taken 3 times
Line 13 - Block 7 - 6 times - block 7 to block 4 = taken all 6 times
Line 15 - Block 8 - 4 times - block 8 to block 2 = taken all 4 times
Line 17 - Block 9 - 1 time (end)

Branch Instruction No. (i.e. Line#)	Branch Instruction Execution (dynamic executions of each branch)									
	1	2	3	4	5	6	7	8	9	10
4	NT	NT	NT	NT	T					
6	NT	NT	NT	T	NT	NT	T	NT	T	T
9	NT	T	T	NT	T	NT				
13	T	T	T	T	T	T				
15	T	T	T	T						
17	T									

Using the branch execution table above to calculate the statistics requested in the following table.

Branch Execution Statistics:

Branch Instr. No.	Times Executed	Times Taken	Times Not Taken	% Taken	%Not Taken
4	5	1	4	⅕ = 20%	80%
6	10	4	6	4/10 = 40%	60%
9	6	3	3	3/6 = 50%	50%
13	6	6	0	6/6 = 100%	0%
15	4	4	0	4/4 = 100%	0%
17	1	1	0	1/1 = 100%	0%

2. How many cycles does the trace take to execute (include all pipeline fill and drain cycles)?
[Hint: you don't need to physically simulate the execution trace, just compute the cycle count.]
Execution Trace: Sequence of Basic Blocks Executed:
1 2 3 4 5 6 7 4 5 7 4 5 7 4 8 2 3 4 5 6 7 4 5 7 4 8 2 3 4 8 2 9

Block 1 - 3 instructions x 1 execution = 3 clock periods per instruction
Block 2 - 1 instructions x 5 execution = 5
Block 3 - 1 instructions x 4 execution = 4
Block 4 -1 instructions x 10 execution = 10
Block 5 - 3 instructions x 6 execution = 18
Block 6 - 2 instructions x 3 execution = 6
Block 7 - 2 instructions x 6 execution = 12
Block 8 - 2 instructions x 4 execution = 8
Block 9 - 2 instructions x 1 execution = 2
—----------

CPI = 68 total instructions to finish the execution trace(cycle)
1 clock per instruction
IPC = 1/68 = 0.0147

CPI = Nominal CPI + Penalty cycles for TAKEN BRANCH + Penalty Cycles for NOT TAKEN Branch

3. How many cycles are lost to control dependency stalls?
Based on GeeksForGeeks, Control dependency stalls occur when there's a transfer of control instructions such as branch, jump etc. The number of times that this has jumped or branched is on line 9 and line 4 with instructions skip and end. The number of times that those lines were taken is 6 times and 1 time, so a total of 7 cycles are lost to control dependency stalls

Part C: Static Branch Prediction.

Static branch prediction is a compile-time technique of influencing branch execution in order to reduce control dependency stalls. Branch opcodes are supplemented with a static prediction bit that indicates a likely direction during execution of the branch. This is done based on profiling information, ala that in Part B. For this part of **Problem 1,** new branch opcodes are introduced:

bget - branch greater than or equal with static predict taken
bgen - branch greater than or equal with static predict not-taken
blet - branch less than or equal with static predict taken
blen - branch less than or equal with static predict not-taken

Static branch prediction information is processed in the decode stage of the 5-stage pipeline. When a branch instruction with static predict taken (i.e. **bget**) is decoded the machine predicts taken. Conversely, when a branch instruction with static predict not-taken (i.e. **bgen**) is decoded the machine predicts not-taken.

1. Pretend you are the compiler, rewrite each conditional branch instruction in the original code sequence using the new conditional branch instructions with static branch prediction encoded.

B = Branch
Conditional Branches
BGE = Branch Greater than Equal - Lines 4 and 6 - Mostly Non taken = BGEN
BLE = Branch Less Than Equal - Line 9- 50/50, can be BLET or BLEN (flipped a coin) - BLEN will be represented in the example below

Branch Instr. No.	Times Executed	Times Taken	Times Not Taken	% Taken	%Not Taken
4	5	1	4	⅕ = 20%	80%
6	10	4	6	4/10 = 40%	60%
9	6	3	3	3/6 = 50%	50%

BB	Line#	Label	Assembly_Instruction			Comment
1	1	main:	addi	r2,	r0, ListArray	r2 <- ListArray
	2		addi	r3,	r0, ListLength	r3 <- ListLength
	3		add	r4,	r0, r0	i = 0;
2	4	loop1:	**BGEN**	r4,	r3, end	while (i < Length)
						{
3	5		addi	r5,	r4, 1	j = i + 1;
4	6	loop2:	**BGEN**	r5,	r3, cont	while (j < Length)
						{
5	7		lw	r6,	r4(r2)	temp = ListArray[i];
	8		lw	r7,	r5(r2)	temp2 = ListArray[j];
	9		**BLEN**	r6,	r7, skip	if (temp > temp2)
						{
6	10		sw	r7,	r4(r2)	ListArray[i] <- temp;

```
        11              sw    r6,   r5(r2)                    ListArray[j] <- temp2;
                                                                }
7       12      skip:   addi  r5,   r5,   1                    j++;
        13              ba    loop2                 }
8       14      cont:   addi  r4,   r4,   1                    i++;
        15              ba    loop1           }
9       16      end:    lw    r1,   (sp)                r1 <- Return Pointer
        17              ba    r1
```

2. Assuming the same execution trace, what is the new total cycle count of the modified code sequence incorporating static branch prediction instructions. Indicate the resultant IPC.

Execution Trace: Sequence of Basic Blocks Executed:

1 2 3 4 5 6 7 4 5 7 4 5 7 4 8 2 3 4 5 6 7 4 5 7 4 8 2 3 4 5 6 7 4 8 2 3 4 8 2 9

When a branch instruction with static predict taken (i.e. **bget**) is decoded the machine predicts taken. Conversely, when a branch instruction with static predict not-taken (i.e. **bgen**) is decoded the machine predicts not-taken.

BGET → Taken

BLEN —> NonTaken

4 trace path with BGET : 1, 2, 9 (end)

6 trace path with BGET: Nah

9 trace path with BLEN : Nah

9 trace path with BLET : Nah

so this forces instruction line 4 to be taken, which takes the entire loop to the end label, which ends the loop. None of the instructions afterwards can proceed since BGET forces it to end in block 2

Branch Instruction No. (i.e. Line#)	**Branch Instruction Execution (dynamic executions of each branch)**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
4	T									
6	NT									
9(BLEN)	NT									
9(BLET)	NT									

1 2 3 4 5 6 7 4 5 7 4 5 7 4 8 2 3 4 5 6 7 4 5 7 4 8 2 3 4 5 6 7 4 8 2 3 4 8 2 9

Turns to:
1,2 9

Block 1 - 3 instructions x 1 execution = 3 clock periods per instruction
Block 2 - 1 instructions x 5 execution = 5
Block 9 - 2 instructions x 1 execution = 2
—----------

CPI = 10 total instructions to finish the execution trace(cycle)
1 clock per instruction
IPC = 1/10 = 0.1