CSCE 5610 – Computer System Architecture
Assignment 2

Uday Bhaskar Valapadasu
11696364

Question 1 Solution:

a)

Calculating the CPI for 5-stage and 12-stage with pipelines data hazards: -

5-Stage:

- Clock cycle = 1ns
- Stall = 1 stall every 5 instructions = 1/5
- The average CPI (Cycles Per Instruction) due to data hazards is:

$$CPI_{5\text{-stage}} = 1 + 1/5 = 1.2$$

12-stage pipeline:

- Clock cycle = 0.6 ns.
- Stall = 3 stalls every 8 instructions = 3/8.
- The average CPI due to data hazards is:

$$CPI_{12\text{-stage}} = 1 + 3/8 = 1.375$$

Calculating each pipeline execution time per instruction: -

- The execution time per instruction is given by the formula:

Execution time = I x Clock cycle Time × CPI

- For the 5-stage pipeline:
  - Execution time 5-stage = 1 ns x 1.2 = 1.2 ns
- For the 12-stage pipeline:
  - Execution time 12-stage = 0.6 ns x 1.375 = 0.825 ns

Calculating the speedup of the 12-stage pipeline over the 5-stage pipeline: -

- Speedup is the ratio of the execution time of the 5-stage pipeline to the execution time of the 12-stage pipeline:

Speedup = Execution time of 5-stage / Execution time of 12-stage = 1.22/0.825 ~ 1.455

Thus, the speedup of the 12-stage pipeline over the 5-stage pipeline, considering only data hazards, is approximately **1.455**.

b)

Calculating the stalls caused by branch misprediction:-

Branch misprediction rate for both the machines = 5%.
Constituted Branch instructions = 20% of total instructions.
Thus, the frequency of branch mispredictions for both machines is:

Mispredict frequency = 0.2 × 0.05 = 0.01

This means 1 out of 100 instructions is a mis predicted branch.

Determine the penalties due to branch mispredictions: -
5-stage pipeline:

- Branch misprediction penalty = 2 cycles.
- The average number of cycles lost per instruction due to branch mispredictions is:

Branch penalty-stage = 0.01 x 2 = 0.02 cycles per instruction

12-stage pipeline:

- Branch misprediction penalty = 5 cycles.
- The average number of cycles lost per instruction due to branch mispredictions is:

Branch penalty 12-stage = 0.01 x 5 = 0.05 cycles per instruction

Adding the branch misprediction penalty to the original CPI.
Now we add the branch misprediction penalty to the CPIs calculated from data hazards:

- 5-stage pipeline CPI:
  - $CPI_{5\text{-stage}} = 1.2 + 0.02 = 1.22$
- 12-stage pipeline CPI:
  - $CPI_{12\text{-stage}} = 1.375 + 0.05 = 1.425$

Thus, after accounting for both data hazards and branch misprediction, the CPI of the 5-stage pipeline is **1.22**, and the CPI of the 12-stage pipeline is **1.425.**

Speedup **=** Execution time of 5-stage / Execution time of 12-stage
= (1*1.22*1)/ (1*1.425*0.6)
= 1.22/0.855
= 1.42

Question 2 Solution:

1)

To count the number of static instructions, we simply count the unique instructions in the code, regardless of how many times they are executed. Let's go through the code:

1. li $t1, 0
2. li $t2, 10
3. beq $t1, $t2, Exit
4. addi $t1, $t1, 1
5. j L

Each of these lines represents a unique instruction.

"Exit:" is a label, not an instruction, so it's not counted.

Therefore, the total number of static instructions is **5**.

To calculate the number of dynamic instructions for this MIPS assembly code snippet, we need to analyze how many times each instruction is executed. Let's break it down step by step:

1. li $t1, 0 - Executed once
2. li $t2, 10 - Executed once
3. L: beq $t1, $t2, Exit - Executed 11 times (10 times it's false, 1 time it's true)
4. addi $t1, $t1, 1 - Executed 10 times
5. j L - Executed 10 times

The loop runs 10 times because $t1 starts at 0 and increments until it reaches 10, at which point the beq condition becomes true and the loop exits.

So, the total number of dynamic instructions is: 1 + 1 + 11 + 10 + 10 = **33 dynamic instructions.**

2)

We are comparing the performance of two processors:

1. P1:

- **Clock rate = 1 GHz (1 cycle = 1 ns).**
- **CPI = 1.2 for this program.**

2. P2:

- **Clock rate = 2 GHz (1 cycle = 0.5 ns).**
- **CP| = 1 for this program.**

CPU Execution Time Formula

The execution time of a processor is given by:

CPU Time = (CPI × Instruction Count) / Clock Rate

CPU Time for Each Processor:

For P1 (1 GHz, CPI= 1.2):

$$\text{CPU Time}_{p1} = (1.2 \times 55) / 1 \times 10^9 = 66 \text{ ns.}$$

- **For P2 (2 GHZ, CPI = 1):**

$$\text{CPU Time}_{p2} = (1 \times 55) / 2 \times 10^9 = 27.5 \text{ ns.}$$

Processor P2 is fastest.

Finding The Fastest Processor and Calculating the Speedup :-

Speedup = CPU Time_{p1} **/** CPU Time_{p2} **= 66/27.5 ~ 2.4**

So, P2 is 2.4 times faster than P1 in executing the given program.

Question 3 Solution:

Functional Units:

Instruction Memory	2ns
Register Read	1ns
ALU (for address calculation or arthimetic operations)	2ns
Data Memory (for load and store operations)	2ns
Register Write	1ns

Given the program below:

```
lw $a2, 0($a2)
subi $t0, $t1, 1
addi $v0, $0, 0
add $t0, $0, $0
add $t4, $a0, $a1
```

1)

The cycle time of a single-cycle implementation is the total sum of the functional units latencies involved in the instruction path. The memory—related instructions consume more time choosing the longest the path:

$$\text{Cycle time}_{\text{single-time}} = 2\text{ns} + 1\text{ns} + 2\text{ns} + 2\text{ns} + 1\text{ns} = 8\text{ns}$$

So, the cycle time for the single-cycle implementation is **8ns**.

In pipelined implementation, the cycle time is determined by the stage having the longest pipeline latency such that each stage handles part of the instruction during each clock cycle.

- We can observe the ALU, Data Memory & Instruction Memory have the highest latency of 2ns

So, the cycle time of the pipelined implementation is **2ns**.

2) **Pipeline diagram to show the execution of these five instructions:**

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
lw $a2, 0($a2)	**IF**	**ID**	**EX**	**MEM**	**WB**				
subi $t0, $t1, 1		**IF**	**ID**	**EX**	**MEM**	**WB**			
addi $v0, $0, 0			**IF**	**ID**	**EX**	**MEM**	**WB**		
add $t0, $0, $0				**IF**	**ID**	**EX**	**MEM**	**WB**	
add $t4, $a0, $a1					**IF**	**ID**	**EX**	**MEM**	**WB**

3) **Single-Cycle Execution:**

Total Cycles: Each instruction takes one cycle.

Total Cycles = 5

Execution Time:

Total Time = Total Cycles x Cycle Time

= 5 x 8 ns

= 40 ns

4) **Pipelined Execution:**

Total Cycles: n + (k - 1) = 5 + (5 - 1) = 9

Execution Time:

Total Time = Total Cycles x Cycle Time

= 9 × 2 ns

= 18 ns

These computations demonstrate how pipelining is substantially faster than single-cycle execution by overlapping instruction steps to reduce total execution time. Because it can process numerous instructions at once, the pipelined implementation is more efficient and takes less time to execute overall.

Question 4 Solution:

a)

Instruction	IF	ID	EX	MEM	WB	Stalls	Reason
sub $19, $19, $20	1	2	3	4	5	0	-
addi $9, $20, 5	2	3	4	5	6	0	-
sw $9, 0($21)	3	4	5	6	7	2	RAW on $9
addi $10, $19, 2	6	7	8	9	10	2	RAW on $19
sll $10, $10, $3	7	8	9	10	11	2	RAW on $10
add $10, $10, $21	8	9	10	11	12	2	RAW on $10
addi $11, $20, 1	9	10	11	12	13	0	-
sll $11, $11, 3	10	11	12	13	14	2	RAW on $11
add $11, $11, $21	11	12	13	14	15	2	RAW on $11
lw $11, 0($11)	12	13	14	15	16	2	RAW on $11
sub $9, $9, $11	13	14	15	16	17	2	RAW on $11
sw $9, 0($10)	14	15	16	17	18	2	RAW on $9
beq $9, $0, Loop	15	16	17	18	19	1	Branch stall (ID)
addi $9, $12, 2	16	17	18	19	20	0	-

Detailed Breakdown of Stalls:

1. Instruction 3 (sw) has 2 stalls because $9 is produced by instruction 2 (addi) and needs to be written back before sw.
2. Instruction 4 (addi) stalls for 2 cycles because $19 is produced by instruction 1 (sub) and needs to be written back.
3. Instruction 5 (sll) stalls for 2 cycles due to RAW hazard on $10 produced by instruction 4 (addi).
4. Instruction 6 (add) stalls for 2 cycles because $10 is still needed from the previous sll.
5. Instruction 8 (sll) stalls for 2 cycles because $11 is produced by instruction 7 (addi).
6. Instruction 9 (add) stalls for 2 cycles for RAW on $11.
7. Instruction 10 (lw) stalls for 2 cycles for RAW on $11 (which is used in instruction 9).
8. Instruction 11 (sub) stalls for 2 cycles due to RAW on $11 from lw.
9. Instruction 12 (sw) stalls for 2 cycles due to RAW on $9.
10. Instruction 13 (beq) stalls for 1 cycle due to branch resolution in the ID stage (branch decision).

Total Stalls: 17 stalls

b)

Optimally Reordered Code:

1. sub $19, $19, $20
2. addi $11, $20, 1
3. addi $9, $20, 5
4. addi $10, $19, 2
5. sll $11, $11, 3
6. sw $9, 0($21)

7. sll $10, $10, $3
8. add $11, $11, $21
9. add $10, $10, $21
10. lw $11, 0($11)
11. sub $9, $9, $11
12. sw $9, 0($10)
13. beq $9, $0, Loop
14. addi $9, $12, 2

Pipeline execution with stalls:

Instruction	IF	ID	EX	MEM	WB	Stalls	Reason
sub $19, $19, $20	1	2	3	4	5	0	-
addi $11, $20, 1	2	3	4	5	6	0	-
addi $9, $20, 5	3	4	5	6	7	0	-
addi $10, $19, 2	4	5	6	7	8	1	RAW on $19
sll $11, $11, 3	5	6	7	8	9	1	RAW on $11
sw $9, 0($21)	6	7	8	9	10	0	-
sll $10, $10, $3	7	8	9	10	11	0	-
add $11, $11, $21	8	9	10	11	12	0	-
add $10, $10, $21	9	10	11	12	13	0	-
lw $11, 0($11)	10	11	12	13	14	1	RAW on $11
sub $9, $9, $11	11	12	13	14	15	1	RAW on $11
sw $9, 0($10)	12	13	14	15	16	1	RAW on $9
beq $9, $0, Loop	13	14	15	16	17	1	Branch stall (ID)
addi $9, $12, 2	14	15	16	17	18	0	-

Total stalls: 6

Explanation of optimizations:

1. Independent instructions (addi $11, $20, 1 and addi $9, $20, 5) are moved earlier to avoid waiting for $19.
2. addi $10, $19, 2 is placed immediately after $19 is available.
3. sw $9, 0($21) is positioned to reduce stalls between dependencies on $10 and $11.
4. The order of sll and add for $10 and $11 is optimized to minimize stalls.
5. lw $11, 0($11) is placed late to reduce its effect on following instructions.
6. The beq branch remains at the end to preserve the loop's functionality.

c)

Optimally Reordered Code (with Forwarding):

1. sub $19, $19, $20
2. addi $9, $20, 5
3. addi $10, $19, 2
4. sw $9, 0($21)

5. sll $10, $10, $3
6. add $10, $10, $21
7. addi $11, $20, 1
8. sll $11, $11, 3
9. add $11, $11, $21
10. lw $11, 0($11)
11. sub $9, $9, $11
12. sw $9, 0($10)
13. beq $9, $0, Loop
14. addi $9, $12, 2

Pipeline execution with forwarding:

Instruction	IF	ID	EX	MEM	WB	Stalls	Reason
sub $19, $19, $20	1	2	3	4	5	0	-
addi $9, $20, 5	2	3	4	5	6	0	-
addi $10, $19, 2	3	4	5	6	7	0	Forwarding from EX
sw $9, 0($21)	4	5	6	7	8	0	Forwarding from EX
sll $10, $10, $3	5	6	7	8	9	0	Forwarding from EX
add $10, $10, $21	6	7	8	9	10	0	Forwarding from EX
addi $11, $20, 1	7	8	9	10	11	0	-
sll $11, $11, 3	8	9	10	11	12	0	Forwarding from EX
add $11, $11, $21	9	10	11	12	13	0	Forwarding from EX
lw $11, 0($11)	10	11	12	13	14	0	Forwarding from MEM
sub $9, $9, $11	11	12	13	14	15	0	Forwarding from MEM
sw $9, 0($10)	12	13	14	15	16	0	Forwarding from EX
beq $9, $0, Loop	13	14	15	16	17	1	Branch stall (ID)
addi $9, $12, 2	14	15	16	17	18	0	-

Total stalls: 1

Explanation of optimizations:

1. Instruction order is adjusted to minimize data hazards by using forwarding effectively.
2. addi $10, $19, 2 follows the instruction that updates $19 to allow immediate forwarding.
3. Instructions that modify $10 and $11 are grouped together to maximize forwarding benefits.
4. The lw instruction is delayed to minimize its impact on later instructions.
5. The beq branch remains at the end to preserve the loop's functionality.

d)

Cycle	Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Forwarding/Bypassing
1	sub $19, $19, $20	IF	ID	EX	MEM	WB															
2	addi $9, $20, 5		IF	ID	EX	MEM	WB														
3	sw $9, 0($21)			IF	ID	EX	MEM	WB													
4	addi $10, $19, 2				IF	ID	EX	MEM	WB												Forward from MEM to EX
5	sll $10, $10, $3					IF	ID	EX	MEM	WB											Forward from EX to ID
6	add $10, $10, $21						IF	ID	EX	MEM	WB										Forward from EX to EX
7	addi $11, $20, 1							IF	ID	EX	MEM	WB									
8	sll $11, $11, 3								IF	ID	EX	MEM	WB								Forward from EX to EX
9	add $11, $11, $21									IF	ID	EX	MEM	WB							Forward from EX to EX
10	lw $11, 0($11)										IF	ID	EX	MEM	WB						Forward from MEM to EX
11	sub $9, $9, $11											IF	ID	EX	MEM	WB					Forward from MEM to EX
12	sw $9, 0($10)												IF	ID	EX	MEM	WB				Forward from EX to EX
13	beq $9, $0, Loop													IF	ID	EX	MEM	WB			1 stall (branch)
14	addi $9, $12, 2														IF	ID	EX	MEM	WB		

Explanation:

- Forwarding is used to avoid RAW stalls by bypassing data between stages.
- The branch decision in beq is made in the ID stage, reducing the stall to 1 cycle for the branch.
- Data dependencies are resolved using forwarding from MEM to EX and EX to EX stages.

Question 5 Solution:

Calculating the average instruction frequencies between the two programs (gobmk and mcf):

Instruction	gobmk Frequency (%)	mcf Frequency (%)	Average Frequency (%)
Loads	21	35	(21+35)/2=28
Stores	12	11	(12+11)/2=11.5
Branches	14	24	(14+24)/2=19
Jumps	2	1	(2+1)/2=1.5
ALU operations	50	29	(50+29)/2=39.5

Split Branches into Taken and Not Taken:

Since 50% of branches are taken and 50% are not taken, we split the average branch frequency (19%):

- Taken Branches: 0.50×19=9.5%0.50 \times 19 = 9.5\%0.50×19=9.5%
- Not-Taken Branches: 0.50×19=9.5%0.50 \times 19 = 9.5\%0.50×19=9.5%

Calculate the Frequency of Other Instructions:

We subtract the total of all known frequencies from 100% to find the frequency of "other" instructions:

$$100\% - (39.5\% + 28\% + 11.5\% + 19\% + 1.5\%) = 0.5\%$$

Clock Cycles for Each Instruction Type:

Instruction	Clock Cycles
ALU operations	1.0
Loads	3.5
Stores	2.8
Branches (Taken)	4.0
Branches (Not Taken)	2.0
Jumps	2.4
Other Instructions	3.0

Calculate the Effective CPI

Instruction	Average Frequency (%)	Clock Cycles	Contribution to CPI = Frequency × Clock Cycles
ALU operations	39.5	1.0	39.5×1.0=0.395
Loads	28	3.5	28×3.5=0.98
Stores	11.5	2.8	11.5×2.8=0.322
Branches (Taken)	9.5	4.0	9.5×4.0=0.38
Branches (Not Taken)	9.5	2.0	9.5×2.0=0.19
Jumps	1.5	2.4	1.5×2.4=0.036
Other Instructions	0.5	3.0	0.5×3.0=0.015

Sum the Contributions:

Effective CPI=0.395+0.98+0.322+0.38+0.19+0.036+0.015=2.

The effective CPI is 2.318.