

Name: Mohith Ankem

Student ID: 017400825

CS259 Project1

Question 1:

I executed the 'gcc program1.c -w' command, which created an exe file. I ran the program interactively and then redirected the output to a txt file, producing the output below.

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program1.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
Line 1 ..
Line 2 mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out>first.txt
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ cat ./first.txt
Line 2 Line 1 ..
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$
```

When a C program sends output to the screen (stdout), it uses a technique called line buffering for printf statements. This means the program holds onto the output until it sees a new line character, then displays everything at once. It's like writing a sentence and hitting enter to show it.

However, when the same program writes its output to a file on the disk, it uses full buffering for printf. In this case, the program collects a larger amount of output before writing it all at once, which is more efficient for disk operations. The write command, on the other hand, doesn't use any buffering. It sends its output immediately, regardless of whether it's going to the screen or a file.

As a result, when you redirect the program's output to a file, you might see something unexpected: the content from the write command appears in the file before the content from the printf command, even if the printf comes first in your code. This happens because the printf output is still being held in the buffer when the write command sends its output directly to the file. This difference in buffering behavior can lead to surprising results when comparing screen output to file output, especially in programs that mix printf and write commands.

Question 2:

With the addition of fflush system call, below is the output.

```
Line 2 mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program2.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
Line 1 ..
Line 2 mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out>second.txt
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ cat ./second.txt
Line 1 ..
Line 2 mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$
```

By adding a fflush command after the printf command, the system is forced to instantly write the printf command's output to the disk before continuing with the program's execution. This step rectifies the discrepancy between the stdout and text file outputs mentioned earlier, ensuring that the outputs are displayed in the intended order.

Question 3:

After running the code twice, I observed that the order in which the ‘nodes’ of the process tree are traversed varies with each execution. This variation is due to the fact that the execution order or rate of parent and child processes is determined by the system’s scheduling algorithm, which schedules the processes differently in each run.

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program3.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
parent 0
child 0
parent 1
child 1
parent 1
child 1
parent 2
child 2
parent 2
child 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ parent 2
child 2
parent 2
child 2
```

Question 4a:

Due to the execution of the wait() system call in the parent process, it is compelled to wait until its child processes have been completed. Consequently, the order in which the nodes of the process tree are traversed in the output follows a right-most, Depth First pattern. The parent process only executes after all its child processes have finished.

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program4a.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
child 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
parent 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
child 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
parent 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$
```

4b:

As expected from the assignment guidelines, we observed multiple printf statements in the disk-saved output. These additional statements resulted from the standard I/O buffer being duplicated when the parent process spawned a child process.

```
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out>fourth.txt && cat fourth.txt
child 0
child 1
child 2
child 0
child 1
parent 2
child 0
parent 1
child 2
child 0
parent 1
parent 2
parent 0
child 1
child 2
parent 0
child 1
parent 2
parent 0
parent 1
parent 2
child 2
parent 0
parent 1
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out>fourth.txt && cat fourth.txt
child 0
child 1
child 2
child 0
child 1
parent 2
child 0
parent 1
child 2
child 0
parent 1
parent 2
parent 0
child 1
child 2
parent 0
child 1
parent 2
parent 0
parent 1
child 2
parent 0
parent 1
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$
```

Question 5:

The program's execution resulted in identical outputs on the screen (stdout) and in the disk file. This consistency is due to intentionally clearing the I/O cache after each printf statement using the `fflush(stdout)` command. As the cache is cleared whenever processes are forked, standard I/O buffer data is not duplicated. Consequently, no redundant printf statements are found in the disk file output.

```

mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program5.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
child 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
parent 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out>fifth.txt && cat fifth.txt
child 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
parent 0
child 1
child 2
parent 2
parent 1
child 2
parent 2
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$

```

Question 6:

Source Code:

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    pid_t pid;
    printf("Immediately before the fork. Only one process at this point.\n");
    fflush(stdout);
    pid = fork();
    printf("Immediately after the fork. This statement should print twice.\n");
    fflush(stdout);
    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        printf("I'm the child. My pid is %d. My parent's pid is %d.\n", getpid(), getppid());
    } else {
        printf("I'm the parent. My pid is %d. My child's pid is %d.\n", getpid(), pid);
        wait(NULL);
    }
    return 0;
}

```

Output:

```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program6.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
I'm the parent. My pid is 211656. My child's pid is 211657.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is 211657. My parent's pid is 211656.
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out > sixth.txt && cat sixth.txt
Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is 211660. My parent's pid is 211659.
I'm the parent. My pid is 211659. My child's pid is 211660.
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$
```

Question 7:

Source Code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int pid, pid2, pid3, pid4;
    int Apid, Bpid;

    printf("Immediately before the fork. Only one process at this point.\n");
    fflush(stdout);

    pid = fork();

    if (pid > 0) {
        Apid = getpid();
        printf("node A. Pid is %d\n", Apid);
        fflush(stdout);
        wait(NULL); // Wait for node C (child)
    } else if (pid == 0) {
        Bpid = getpid();
        printf("\tnode B. Pid is %d, parent's pid is %d\n", Bpid, getppid());
        fflush(stdout);
    } else {
        printf("Failed to fork\n");
    }

    pid2 = fork();

    if (pid2 > 0) {
        wait(NULL);
    } else if (pid2 == 0 && getppid() == Bpid) {
```

```

printf("\t\t\node D. Pid is %d, parent's pid is %d\n", getpid(), getppid());
fflush(stdout);
} else if (pid2 == 0 && getppid() == Apid) {
printf("\t\t\node C. Pid is %d, parent's pid is %d\n", getpid(), getppid());
fflush(stdout);
pid3 = fork();
if (pid3 == 0) {
printf("\t\t\node E. Pid is %d, parent's pid is %d\n", getpid(), getppid());
fflush(stdout);
} else if (pid3 > 0) {
pid4 = fork();
if (pid4 == 0) {
printf("\t\t\node F. Pid is %d, parent's pid is %d\n", getpid(), getppid());
fflush(stdout);
}
if (pid4 > 0) {
wait(NULL);
}
}
} exit(0);
}

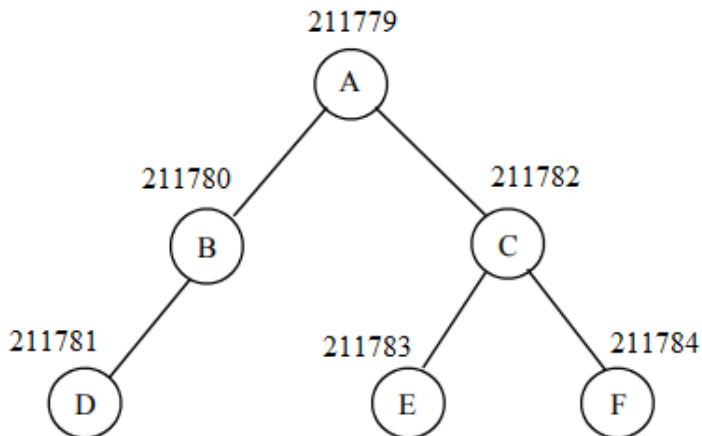
```

Output:

```

mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
Immediately before the fork. Only one process at this point.
node A. Pid is 211773
    node B. Pid is 211774, parent's pid is 211773
        node D. Pid is 211775, parent's pid is 211774
    node C. Pid is 211776, parent's pid is 211773
        node E. Pid is 211777, parent's pid is 211776
        node F. Pid is 211778, parent's pid is 211776
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out > seventh.txt && cat seventh.txt
Immediately before the fork. Only one process at this point.
node A. Pid is 211779
    node B. Pid is 211780, parent's pid is 211779
        node D. Pid is 211781, parent's pid is 211780
    node C. Pid is 211782, parent's pid is 211779
        node E. Pid is 211783, parent's pid is 211782
        node F. Pid is 211784, parent's pid is 211782
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$

```



Question 8:

Source Code:

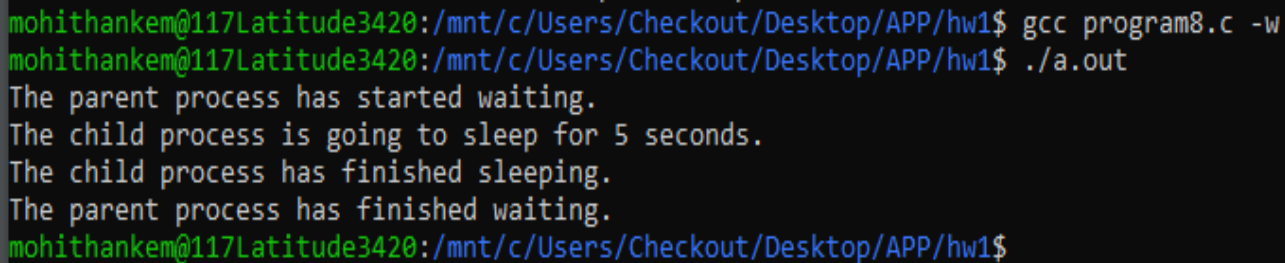
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main() {
    int status = 0;
    int pid = fork();

    if (pid) {
        printf("The parent process has started waiting.\n");
        fflush(stdout);
        wait(&status);
    } else {
        printf("The child process is going to sleep for 5 seconds.\n");
        sleep(5);
        printf("The child process has finished sleeping.\n");
        status = 1;
        exit(0);
    }

    printf("The parent process has finished waiting.\n");
    exit(0);
}
```

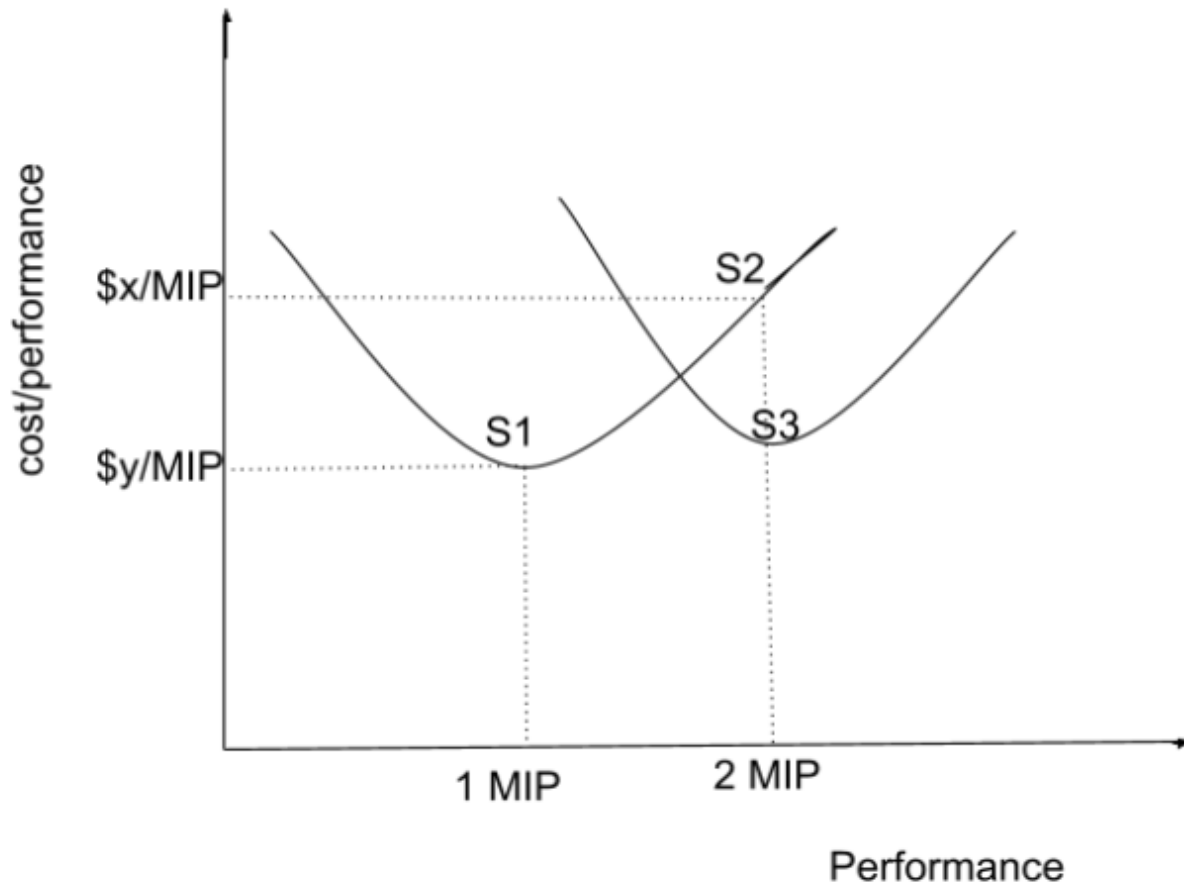
Output:



```
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ gcc program8.c -w
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$ ./a.out
The parent process has started waiting.
The child process is going to sleep for 5 seconds.
The child process has finished sleeping.
The parent process has finished waiting.
mohithankem@117Latitude3420:/mnt/c/Users/Checkout/Desktop/APP/hw1$
```

Question 9:

Typically, using two CPUs in parallel offers a more advantageous cost-to-performance ratio than doubling the speed of a single existing CPU. This is mainly due to the cost efficiency of using two CPUs in parallel rather than trying to double the performance of a single CPU. The decision between redesigning a single processor to achieve 2 MIPs or using two existing processors in parallel depends on various factors, including cost, complexity, and performance scaling. The provided curve supports this argument:



The diagram clearly demonstrates this phenomenon. System S1, with 1 MIP of processing power, falls within the optimal utilization range of the Sequential processing curve. However, System S2, with 2 MIPs of processing power, enters the over-utilization segment of the curve. In this phase, technology is pushed to its limits, resulting in significantly higher costs for marginal performance improvements.

In this scenario, using parallelism shifts the curve downward and to the right, where similar performance improvements can be achieved at the same or even lower cost-to-performance ratios compared to the original system. This is shown by System S3, where we achieve 2 MIPs of computing power with a more cost-effective ratio than System A.

Generally, using two processors in parallel tends to offer a better cost/performance ratio because it leverages existing hardware, minimizing redesign costs. However, the actual performance might be slightly below 2 MIPs due to overhead from parallel processing. Despite this, the cost savings usually outweigh the performance hit, especially for small-scale systems. If your application can handle parallel tasks efficiently and doesn't require perfect linear scaling, the parallel processing approach is likely the better choice.

Question 10:

Instruction-Level Parallelism (ILP), which includes techniques like pipelining (intra-instruction parallelism) and the use of multiple functional units (inter-instruction parallelism), faces inherent limitations due to various hazards that can slow down the pipeline, preventing it from reaching peak performance.

Data hazards occur when two instructions in the pipeline either need to access the same variable or one instruction depends on the result of a previous one. For example, given the instructions:

- MUL R2, R3
- ADD R1, R5 The second instruction relies on the output of the first, creating a data hazard. While hardware pipeline interlocks can address this, they require extra hardware and may limit the effectiveness of ILP, especially as the pipeline depth increases.

Control hazards arise with conditional branch statements, which delay the decision on which instruction to execute next. For example:

- IF $X > 10$, THEN GO TO S3
- S2: $A = B + C$
- S3: $F = H - L$ In this case, the execution of either S2 or S3 depends on the outcome of the condition in S1. If the condition is true, the pipeline will need to flush S2, starting S3 only after S1 is resolved, which is known as a pipeline flush.

The potential speedup of a pipeline is constrained by the number of stages and the instruction stride. As more stages are added, the overhead caused by managing the latches increases and can sometimes surpass the benefits of additional stages, especially in overly aggressive pipelines. Moreover, hazards, particularly control hazards, break the instruction stride, forcing the pipeline to reload.

Resource hazards occur when multiple functional units are used to execute instructions in parallel, and the required resource is unavailable when needed. In multi-function unit ILP, improving parallelism requires sophisticated out-of-order execution mechanisms to handle instruction retirements properly, which further complicates achieving high parallelism.

In conclusion, the effectiveness of pipelining is limited by several factors, including an increase in pipeline stages, instruction stride disruptions, and hazards data, control, and resource-related. Additionally, the need for extra hardware to mitigate these hazards can constrain the aggressiveness of ILP, and in extreme cases, pipelining may even degrade performance.

Question 11:

HyperThreading, a technology developed by Intel, enhances a processor's ability to switch between threads more quickly by expanding the number of registers that store thread states. This gives the appearance of doubling the number of logical processors recognized by the operating system, even though the physical processor count remains unchanged. Despite this, HyperThreading doesn't enable true parallelism, generally providing around a 15% performance improvement.

To take full advantage of HyperThreading, software must be designed to support multi-threading. Developers need to write code that efficiently implements multi-threading to maximize the technology's benefits. For optimal

performance, the number of active threads should be matched to the number of logical processors available in the system.

HyperThreading's impact is particularly noticeable in workloads that involve heavy multitasking or applications that can exploit concurrent thread execution, such as video rendering, gaming, or complex simulations. In these scenarios, the ability to manage multiple threads more effectively can reduce idle CPU time and improve overall throughput. However, it's crucial to understand that HyperThreading is not a substitute for more physical cores. While it enhances the efficiency of existing cores by better managing thread-level parallelism, it can't match the raw performance gains from adding more cores. Nevertheless, in environments where optimizing cost and performance is key, HyperThreading provides a valuable means of improving efficiency without the need for additional hardware.

Additionally, the operating system must be optimized to handle HyperThreading properly. In multicore systems, instead of allocating multiple threads to a single physical core, the operating system should assign threads to different physical cores. This approach enhances resource utilization and improves overall performance.

In conclusion, while HyperThreading can offer a noticeable performance boost, its effectiveness depends on how well both software and the operating system manage and utilize multiple threads. By aligning the number of threads with logical processors and ensuring efficient thread management by the OS, the potential of HyperThreading can be fully realized.

Question 12:

The need for programmers to explicitly implement parallelism in their software stems from several key trends in both micro-architecture and macro-architecture.

Heat and Power Constraints: Boosting clock speeds is no longer a feasible option due to the excessive power consumption and heat generation, which require significant cooling efforts.

Limits of Instruction-Level Parallelism: The performance benefits from instruction-level parallelism (ILP) have plateaued, as the potential gains have already surpassed the critical threshold.

Pervasive Distributed Computing: With the growing prevalence of cluster, grid, and cloud computing, parallel processing has become a dominant method in modern computing systems.

Need for Cost-Effective Performance: Achieving higher performance without escalating costs now relies heavily on the shift to parallel computing.

However, exposing hardware-level parallelism to software development brings inherent challenges:

Complexity of Parallel Programming: Writing parallel programs is considerably more difficult, involving additional hurdles in designing, re-engineering, debugging, testing, profiling, and scaling applications.

Cognitive Challenges: Human cognition struggles with managing the temporal aspects and the numerous possible combinations of instruction interleavings that parallel execution entails.

New Types of Bugs: Parallel programming introduces previously unknown bugs, such as race conditions and deadlocks, which require new strategies to identify and resolve.

Intermittent and Subtle Bugs: Bugs in parallel systems often occur sporadically and are difficult to detect, making them more likely to slip through the cracks during software testing and quality assurance processes.

In conclusion, while modern hardware and computing trends necessitate the adoption of parallelism in software, these shifts present a range of challenges that must be addressed to fully capitalize on the potential performance improvements offered by parallel computing.

Question 13:

Flynn's Taxonomy categorizes computer architectures into four primary types: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD).

Single Instruction Single Data (SISD): In this architecture, the processor processes one instruction on a single data element at a time, executing tasks sequentially. While methods like pipelining can introduce a form of parallelism, it doesn't offer true parallel processing.

Single Instruction Multiple Data (SIMD): SIMD enables the processor to apply the same instruction across multiple data elements simultaneously, making it ideal for repetitive operations on large datasets. For example, it can add two matrices with 100 elements each as quickly as it can add two individual numbers.

Multiple Instruction Single Data (MISD): In this architecture, multiple instructions are applied concurrently to a single data element. A processor using MISD can simultaneously add, subtract, multiply, and divide two numbers, completing all operations in the time it would take to perform just one.

Multiple Instruction Multiple Data (MIMD): MIMD allows for the execution of different instructions on multiple data elements concurrently, offering the highest level of parallelism. For instance, while adding two numbers, the processor could simultaneously be multiplying another pair of numbers.

Although MISD architectures have historically been considered impractical due to limited real-world applications, the growth in computing power has opened potential new uses. One area where MISD could be beneficial is speculative computation, which involves simulating various outcomes simultaneously, making this architecture potentially more relevant in modern computing contexts.

Question 14:

(a)

The speedup achieved through pipelining is largely dependent on the number of stages in the pipeline, denoted by K . However, several factors can limit the efficiency of pipelining, such as the number of stages, imbalances between those stages, and disruptions to the instruction flow. As K increases, the system requires more latches and gates, which introduces latch overhead. In some cases, this overhead can exceed the actual processing work performed at each stage, particularly in aggressively pipelined systems.

Another challenge arises from hazards, especially control hazards, which can disrupt the instruction flow and force the pipeline to reload. Additionally, when the number of pipeline stages far exceeds the number of instructions, the startup cost of the pipeline becomes significant. In these cases, the expected speedup is reduced, and the formula for speedup becomes $NK / (K + N - 1 + \text{latch overhead})$, rather than simply K , when the number of sequential instructions (N) is small.

As a result, the potential benefits of pipelining are constrained by the number of stages, stage imbalances, and instruction flow. In extreme scenarios, the perceived speedup may be misleading, as the total time per instruction in a pipelined machine could end up being equal to or even longer than in a non-pipelined machine.

(b)

In extremely poor conditions, where the pipeline is stalled and flushed after every single instruction, pipelining could result in a net overall performance reduction. This is due to the high latch overhead and the short instruction stride. In such scenarios, a non-pipelined machine would perform better than a pipelined machine. This is because the penalty associated with starting and stopping the pipeline is significant, leading to decreased overall performance.

For a code like this:

```
if (cond1) GOTO ref1:
    if(cond2) GOTO ref2:
        if(cond3) GOTO ref3:
        .
        .
        .
    n
```

The pipeline gets flushed every statement, which acts like a non-pipelines processor.

Question 15:

S1: $X = 5$

S2: $Y = X + X$

S3: IF $Y > 7$ THEN GOTO S5

S4: $A = B + C$

S5: $Z = X + Y$

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
S1	FI	DA	FO	EX										
S2		FI	DA	-	FO	EX								
S3			FI	-	DA	-	FO	EX						
S4				-	FI	-	-							
S5				-		-	-	-	FI	DA	FO	EX		

Question 16:S1: $A = B + C$ S2: $D = E + F$ S3: $G = A * Y$ S4: $Z = H + G$ S5: $W = I * J$ S6: $F = W * Z$ S7: $H = K * L$

(a)

Time	1	2	3	4	5
Adder 1	$A=B+C$		$Z=H+G$		
Adder 2	$D=E+F$				
Multiplier		$G=A*Y$	$W=I*J$	$F=W*Z$	$H=K*L$
Hazard	S3: A	S4: G	S6: W, Z, Multiplier	S7: Multiplier	

(b)

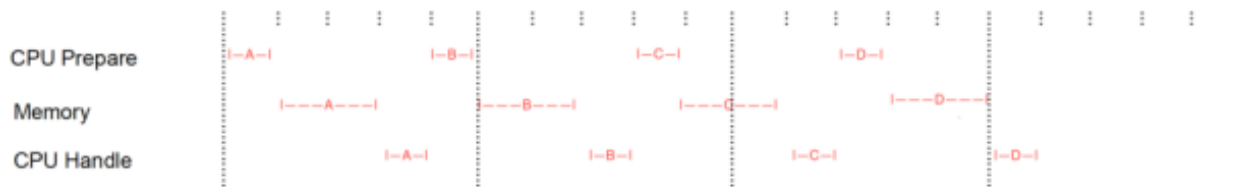
Time	1	2	3	4	5
Adder 1	$A=B+C$		$Z=H+G$		
Adder 2	$D=E+F$				
Multiplier 1		$G=A*Y$	$W=I*J$	$F=W*Z$	
Multiplier 2				$H=K*L$	
Hazard	S3: A	S4: G	S6: W, Z		

(c)

Time	1	2	3	4	5
Adder 1	$A=B+C$		$Z=H+G$		
Adder 2	$D=E+F$				
Adder 3					
Adder 4					
Multiplier 1		$G=A*Y$	$W=I*J$	$F=W*Z$	
Multiplier 2				$H=K*L$	
Multiplier 3					
Multiplier 4					
Hazard	S3: A	S4: G	S6: W, Z		

Question 17:

a) Non-Interleaved



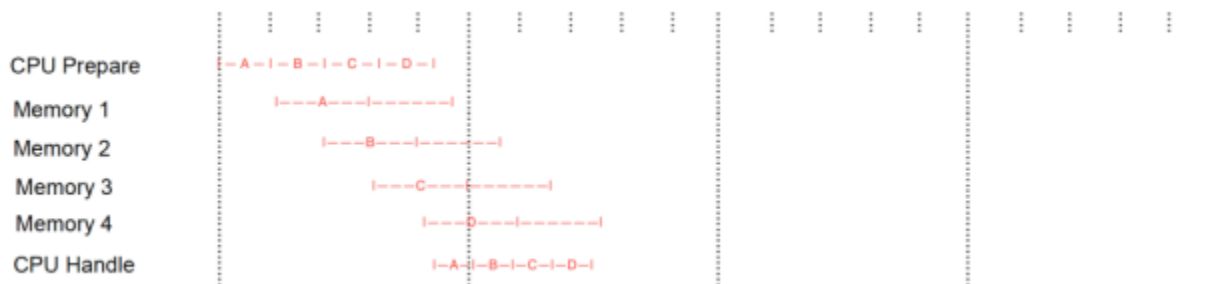
Total Time: 3.2, CPU Utilization: $8/16 = 50\%$

(b) Two-Way Interleaved:



Total time: 2, CPU Utilization: $8/10 = 80\%$, Speedup: $3.2/2 = 1.6$

(c) Four-way Interleaved:



Total time: 1.6, CPU Utilization: $8/8 = 100\%$, Speedup: $3.2/1.6 = 2$