FIT3143 APPLIED #3

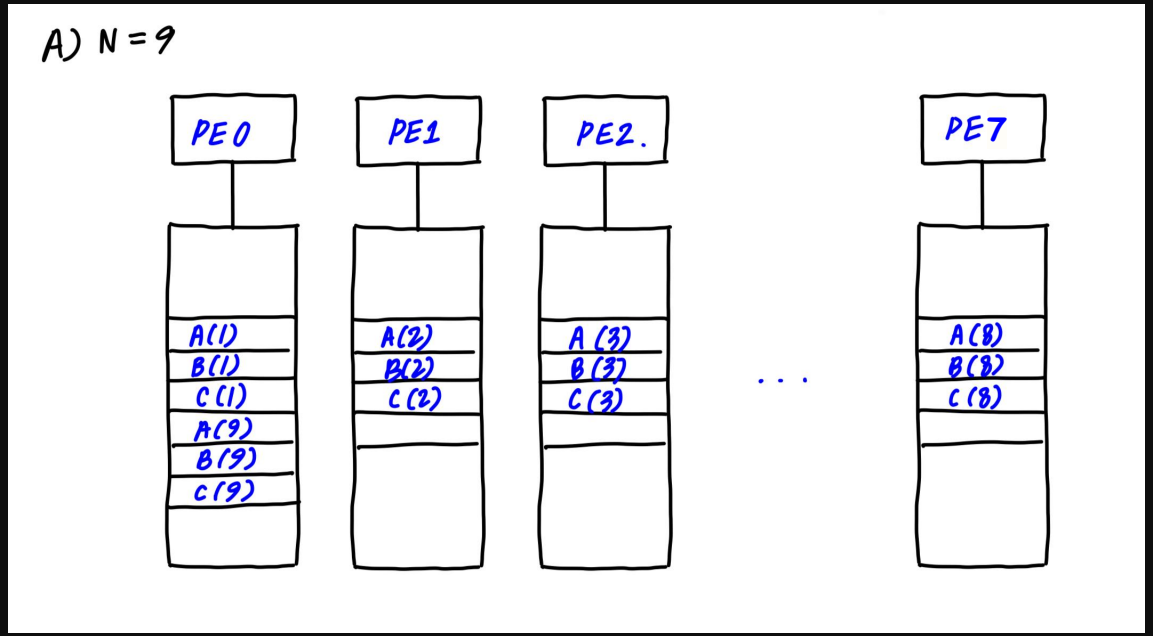
Prepared by Alexis, Ben, Marcus and Kevin
S2 2025

01

ROBERT SLENDERS

SIMD

SIMD: Q1A – Robert




```
DO 10 I = 1,N

10 A(I) = B(I) - C(I)
```

- 2 loops needed to perform array addition for N = 9
- On second loop, PE 1-7 must be deactivated

SIMD: Q1B – Robert

```
DO 10 I = 1,N

10 A(I) = B(I) - C(I)
```

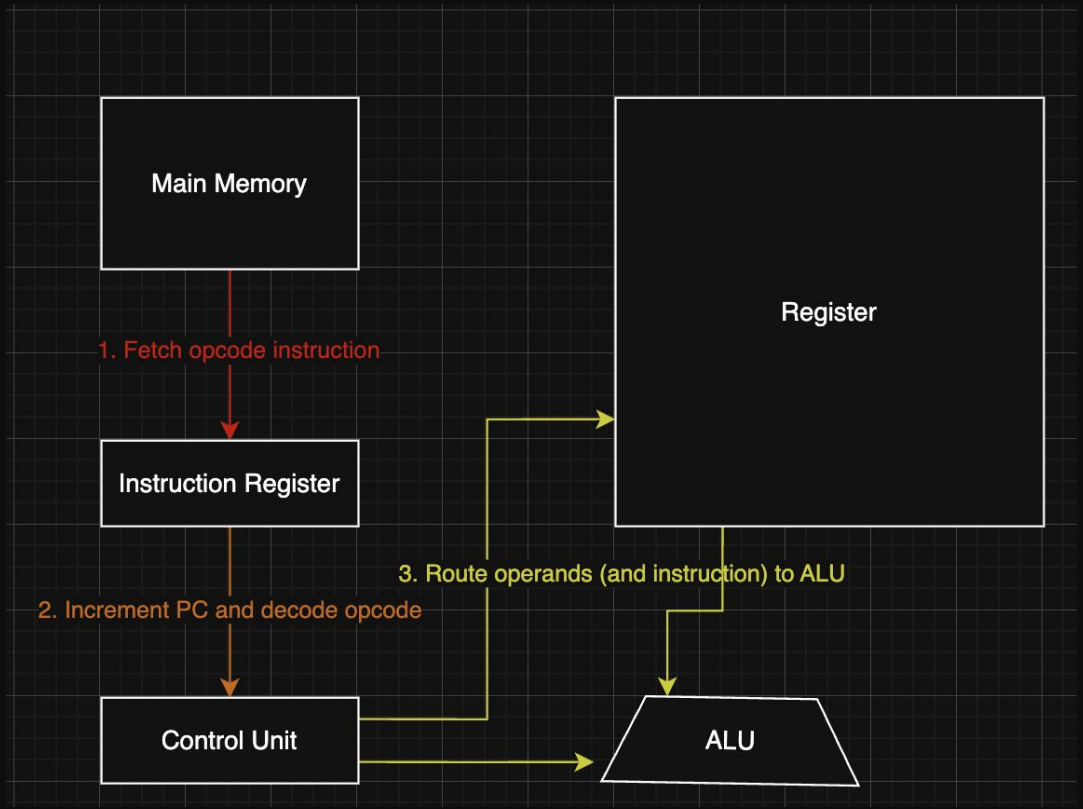
The statement is not true. In general a speedup of M is not achieved, as on the final iteration, some processing elements will be idle. The exception to this is when N is an integer multiple of M, where the theoretical speedup can be achieved. As N grows larger, the speedup approaches the theoretical as the processing elements idle for a shorter part of the overall time. In practice, some overhead is also incurred when processing in parallel, which limits the speedup.

02

BEN LEAHY

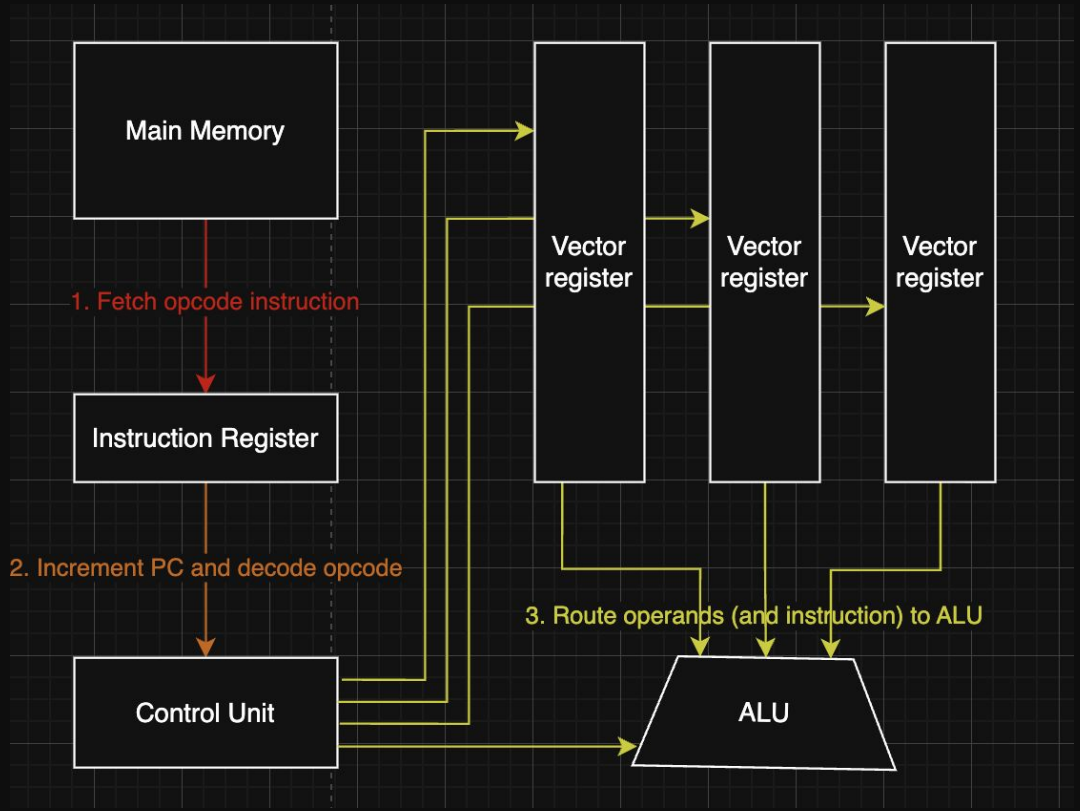
VECTOR PROCESSING

Question 2A Ben Leahy



Complete steps 1 and 2 for each individual operation calculated in ALU

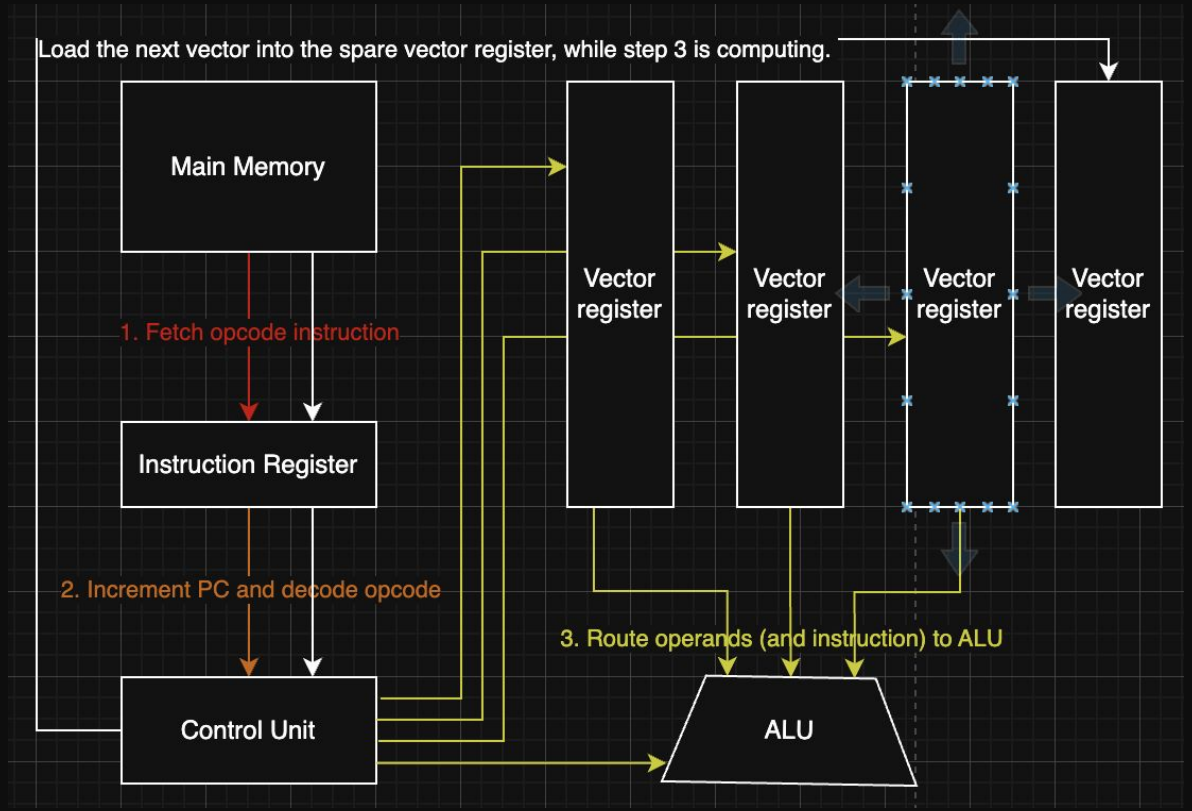
Question 2A Ben Leahy



Complete steps 1 and 2 once for each operation that applies to an entire vector

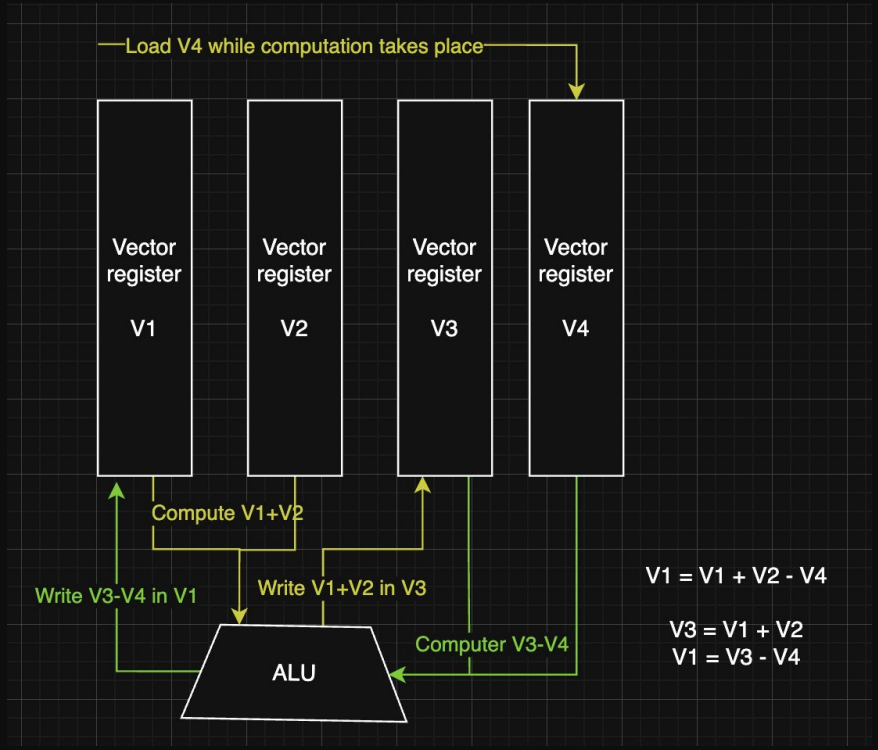
We require specific hardware, but we can save on steps 1 and 2.

Question 2A Ben Leahy



Chaining

Question 2A Ben Leahy



If we wanted to compute V1 = V1 + V2 - V4 with 3 operand vector architecture, then we break it into two steps.

V3 = V1 + V2
V1 = V3 - V4

Then we can use chaining to start loading what is needed into V4 while the first operation takes place. This means we save that loading time.

Question 2B Ben Leahy

Question: Present and explain how _mm256_mul_ps is used for vector processing in C/C++.

What is _mm256_mul_ps ?
It's a function in c. It takes two vector registers as parameters, and multiples them element by element. (vector size 8, for elements sized float 32 → 32 bits). It takes advantage of any vector processing hardware inside your computer.

Syntax

```
extern __m256 _mm256_mul_ps(__m256 m1, __m256 m2);
```

Arguments

m1	float32 vector used for the operation
m2	float32 vector also used for the operation

Question 2B Ben Leahy

week5 > C vector_addition_multi_threading.c > main()

```c
int main() {
    // Ie use case here could be finding E[X]. Obv not normal use case for vectors but just for fun
    float a[8] = {1,2,3,4,5,6,7,8};
    float b[8] = {0.1, 0.3, 0.05, 0.2, 0.05, 0.01, 0.01, 0.01};
    float c[8];

    // Fetching and decodingh the same opcode each time.
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < 8; i++) {
        c[i] = a[i] * b[i];
    }

    // Ignore this in terms of our analysis, just for printing.
    for (int i = 0; i < 8; i++) {
        printf("%f\n", c[i]);
    }

    return 0;
}
```

week5 > C vector_addition_vectorisation.c > main()

```c
#include <immintrin.h>
#include <stdio.h>

int main() {
    float a[8] = {1,2,3,4,5,6,7,8};
    float b[8] = {2,2,2,2,2,2,2,2};
    float c[8];

    //Takes advantage of AVX hardware
    __m256 va = _mm256_loadu_ps(a);   // load array a into a 256-bit vector register (8 numbers * 32bits per num)
    __m256 vb = _mm256_loadu_ps(b);
    __m256 vc = _mm256_mul_ps(va, vb); // multiply element-wise using vector optimistion, returns vector
    _mm256_storeu_ps(c, vc);          // store vector result into array c

    for (int i = 0; i < 8; i++) {
        printf("%f\n", c[i]);
    }
    return 0;
}

// Command: gcc -O2 -mavx vector_addition_vectorisation.c -o va_v
```

Question 2B Ben Leahy

week5 > C vector_addition_mt_and_v.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <immintrin.h>   // AVX intrinsics

int main() {
    // At runtime, assign memory in heap for vectors.
    // The first param means the storage adress will be a multiple of this number. It should be a power of 2.
    // The second param is just the size in bytes.
    int n = 32;
    float *a = (float*) aligned_alloc(32, n * sizeof(float));
    float *b = (float*) aligned_alloc(32, n * sizeof(float));
    float *c = (float*) aligned_alloc(32, n * sizeof(float));

    // Ignore this in our analysis, we assume we would be passed this in as a parameter.
    for (int i = 0; i < n; i++) {
        a[i] = i + 1;
        b[i] = i*i - 1;
    }

    // Each thread processes a sub-vector of n=8
    //(if our total vector was not a multiple of 8 then the final thread would be assigned a vector of n <8)
    // Save by computing multiple 8 number vector parts in parallel. Cost is thread creation overhead.
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < n; i += 8) {
        __m256 vec_a = _mm256_loadu_ps(&a[i]);
        __m256 vec_b = _mm256_loadu_ps(&b[i]);

        // Save on instruction and operand fetch and decode.
        __m256 vec_c = _mm256_mul_ps(vec_a, vec_b);

        _mm256_storeu_ps(&c[i], vec_c);
    }

    // Print result
    for (int i = 0; i < n; i++) {
        printf("c[%d] = %f\n", i, c[i]);
    }

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Question 2b Robert

AVX has 3 vectors operands, whereas SSE4 has 2 operands.
This allows us to not overwrite one of the source operands.
This is particularly useful if the next operation requires that source operand again for the next instruction.
The other key difference is that AVX extends 128-bit SSE4 instructions to 256-bit, allowing the storage of 8 32-bit integer or floats in a single vector, as opposed to 4 for SSE4.

03

MARCUS MICCELLI

MIMD ARCHITECTURES & INTERCONNECTION TOPOLOGY

A) MIMD: DISTRIBUTED MEMORY VS. SHARED MEMORY

Feature	*Distributed Memory MIMD*	*Shared Memory MIMD*
Memory Access	*Local memory → less traffic*	*Global memory → more traffic*
Communication	*Message passing sends and synchronises*	*Via shared variables and synchronisation*
Scalability	*Highly scalable*	*Limited scalability → IPC bottleneck*
Programming Complexity	*Explicit communication → hard*	*Implicit communication → easy*
Fault Tolerance	*Better isolation as faults are localised*	*Faults can affect entire system*

Distributed Memory MIMD *is great for large-scale systems like clusters or supercomputers.*

Shared Memory MIMD *is ideal for smaller systems where ease of programming and fast communication are key.*

B) HYPERCUBE INTERCONNECTION TOPOLOGY

A n-dimensional hypercube has 2^n vertices, each labelled as a binary string of length n. Any vertex v is adjacent to another vertex u if v and u's labels only differ by one bit (i.e. 110 and 111 have an edge in the 3-dimensional hypercube which is the regular cube.).

So finding the number of hops between any two vertices (nodes) is simply counting the number of different bits at each position in our two strings. The strings "100" and "011" have different bits in every character in the string, so the number of hops here is 3.

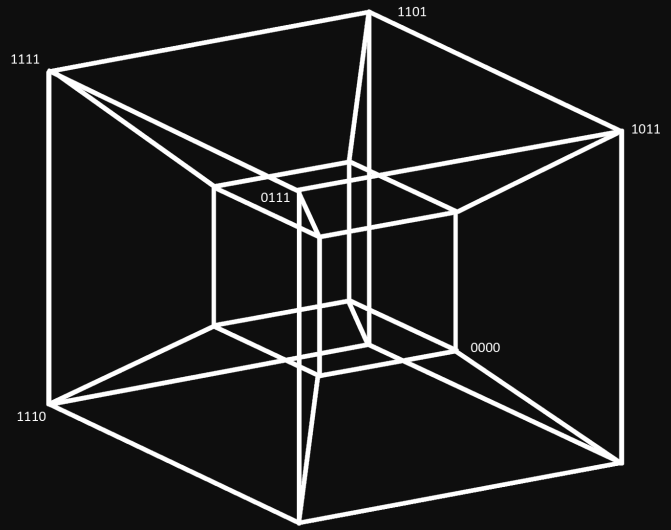
1 0 0
0 1 1

B) HYPERCUBE INTERCONNECTION TOPOLOGY

If new nodes need to be inserted we need to change to a 4-dimensional hypercube, which will support twice as many nodes. Labels for the hypercube will now have 4 bits rather than three.

Hypercubes are optimised for performance, not flexibility — imagine the case when the expected number of nodes is 2(n-1) + 1.

Good architectures for frequent inserting/deleting depends on the use case.



Some labels removed for clarity.

B) HYPERCUBE INTERCONNECTION TOPOLOGY

Ring

- *Adding or removing only requires modifying neighbour nodes.*

Star

- *Good if only inserting or deleting leaf nodes i.e. centralised control.*
- *Easy scaling when most traffic is hub-node.*

Complete Graph

- *If network is small, cost is manageable.*
- *Good when latency is critical.*

Binary Tree

- *Naturally hierarchical — insertion/deletion impact is localised.*
- *Hence, only affected subtree needs adjustment.*

04

ALEXIS RYU

DISTRIBUTED MEMORY WITH OPENMPI

Q4: Issue Identification

```
if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
} else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
```

- We can see from the given code that both rank 0 and rank 1 call MPI_Recv first. This causes a deadlock → processes are blocked in such a way that no further communication or computation can happen.
- Each process is waiting for a message or an event that will never occur.

Q4 Solution 1: Swap MPI_Send and MPI_Recv

```
if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat)
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
} else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat)
```

- Swapping the order of `MPI_Send` and `MPI_Recv` for one (and only one) of the processes eliminates the deadlock
- Simple and effective, but relies on carefully coordinating the order across processes
- If the program gets bigger or involves more ranks, it becomes harder to manage compared to other approaches

Q4 Solution 2: Use MPI_Sendrcv

```
int other = (rank == 0) ? 1 : 0;   // rank 0 talks to 1, 1 talks to 0
MPI_Sendrecv(&outmsg, 1, MPI_CHAR, other, tag,   // send
             &inmsg,  1, MPI_CHAR, other, tag,   // receive
             MPI_COMM_WORLD, &stat);

MPI_Get_count(&stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d (inmsg=%c)\n",
       rank, count, stat.MPI_SOURCE, stat.MPI_TAG, inmsg);
```

- Using `MPI_Sendrcv` allows us to do a simultaneous send and receive → each call posts both directions, eliminating a deadlock scenario
- If rank 0 and rank 1 both call `MPI_Sendrcv` with each other as dest/source, MPI is free to progress both halves (neither side is stuck waiting for the other to send first)

THANK YOU !!!!!!!!

guys for listening!!!!!

AI DECLARATION

No AI was used in the creation of this assessed work.