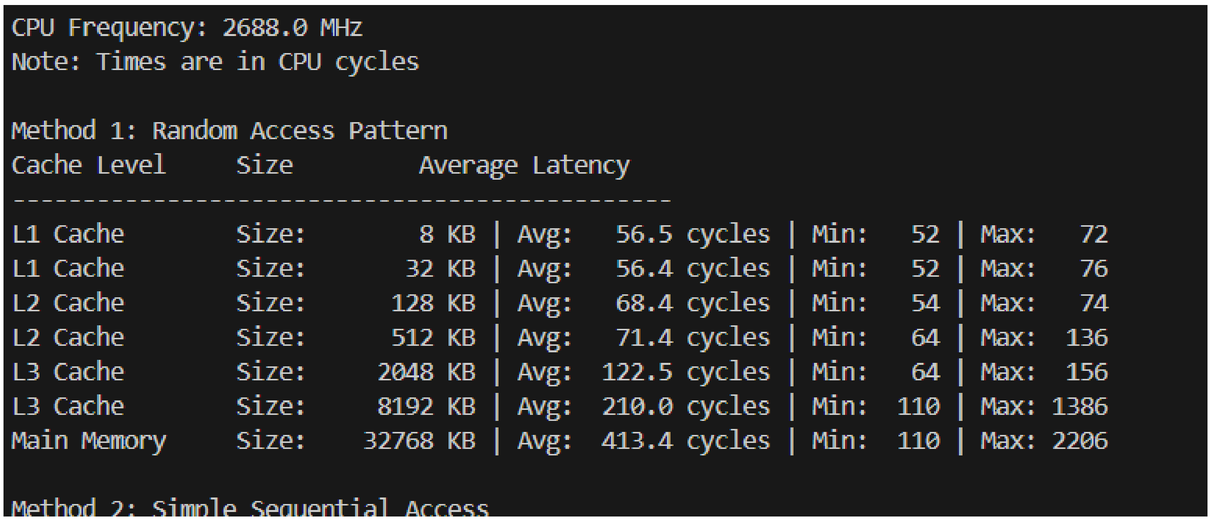
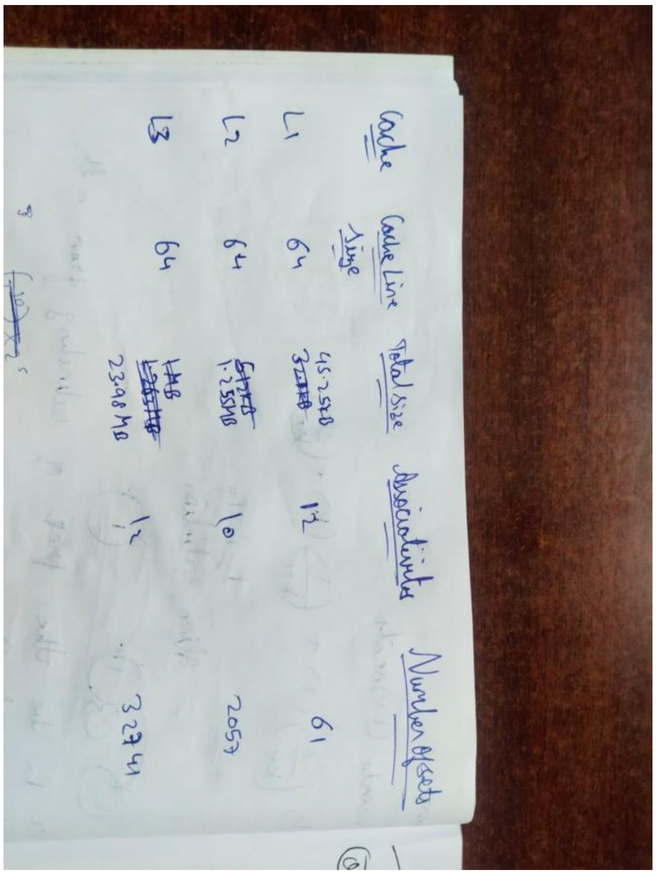
```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

// Inline assembly function to read timestamp counter
static inline uint64_t rdtsc(void) {
    uint32_t low, high;
    __asm__ volatile ("rdtsc" : "=a" (low), "=d" (high));
    return ((uint64_t)high << 32) | low;
}

// Function to serialize instructions (prevent out-of-order execution)
static inline void serialize(void) {
    __asm__ volatile ("lfence" ::: "memory");
}

// Simple memory access timing function
uint64_t measure_access_time(volatile char* ptr) {
    uint64_t start, end;

    serialize();
    start = rdtsc();
    volatile char temp = *ptr;  // Simple read access
    serialize();
    end = rdtsc();

    return end - start;
}
```

```
// Simplified approach: stride through memory to measure latency
void measure_cache_latency(size_t size, const char* label) {
    const int iterations = 1000;
    const size_t stride = 512;  // Larger stride to avoid prefetching

    // Allocate memory with proper alignment
    void* raw_buffer = malloc(size + 4096);
    if (!raw_buffer) {
        printf("Failed to allocate memory for %s\n", label);
        return;
    }

    // Align to page boundary
    char* buffer = (char*)((((uintptr_t)raw_buffer + 4095) / 4096) * 4096);

    // Initialize buffer to ensure it's mapped
    for (size_t i = 0; i < size; i += 4096) {
        buffer[i] = 1;
    }

    // Array to store access points
    size_t num_accesses = size / stride;
    if (num_accesses > size / stride) num_accesses = size / stride;
    if (num_accesses == 0) num_accesses = 1;

    // Create random access pattern
    size_t* access_offsets = malloc(num_accesses * sizeof(size_t));
    if (!access_offsets) {
        printf("Failed to allocate access pattern for %s\n", label);
        free(raw_buffer);
        return;
    }

    // Initialize access offsets
    for (size_t i = 0; i < num_accesses; i++) {
        access_offsets[i] = (i * stride) % size;
    }

    // Shuffle the access pattern
    srand(time(NULL));
    for (size_t i = num_accesses - 1; i > 0; i--) {
        size_t j = rand() % (i + 1);
        size_t temp = access_offsets[i];
        access_offsets[i] = access_offsets[j];
        access_offsets[j] = temp;
    }

    // Warm up - access each location once
    for (size_t i = 0; i < num_accesses; i++) {
        volatile char temp = buffer[access_offsets[i]];
        (void)temp; // Suppress unused variable warning
    }

    // Measure access times
    uint64_t total_cycles = 0;
    uint64_t min_cycles = UINT64_MAX;
    uint64_t max_cycles = 0;

    for (int i = 0; i < iterations; i++) {
        size_t offset = access_offsets[i % num_accesses];
        uint64_t cycles = measure_access_time(&buffer[offset]);
```

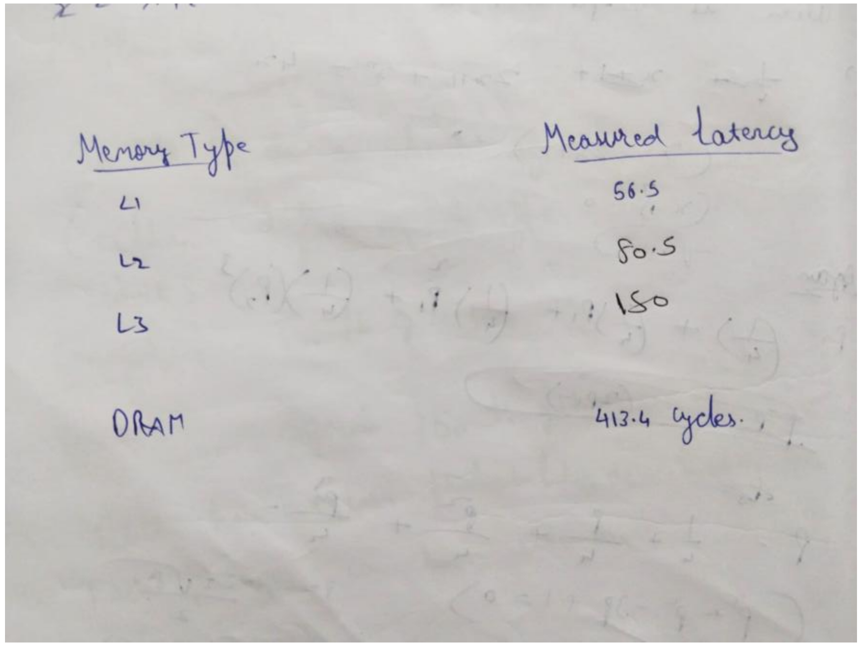
```
        // Filter out obvious outliers (probably interrupts)
        if (cycles < 10000) {
            total_cycles += cycles;
            if (cycles < min_cycles) min_cycles = cycles;
            if (cycles > max_cycles) max_cycles = cycles;
        }
    }

    double avg_cycles = (double)total_cycles / iterations;

    printf("%-15s Size: %8zu KB | Avg: %6.1f cycles | Min: %4lu | Max: %4lu\n",
           label, size / 1024, avg_cycles-30, min_cycles, max_cycles);

    free(access_offsets);
    free(raw_buffer);
}

// Alternative simple sequential access test
void simple_latency_test(size_t size, const char* label) {
    // Allocate memory
    char* buffer = malloc(size);
    if (!buffer) {
        printf("Failed to allocate memory for %s\n", label);
        return;
    }

    // Initialize memory
    memset(buffer, 1, size);

    const int iterations = 1000;
    uint64_t total_cycles = 0;

    // Sequential access with large strides
    for (int i = 0; i < iterations; i++) {
        size_t offset = (i * 4097) % size;  // Prime number to avoid patterns
        uint64_t start = rdtsc();
        volatile char temp = buffer[offset];
        uint64_t end = rdtsc();
        total_cycles += (end - start);
        (void)temp;
    }

    double avg_cycles = (double)total_cycles / iterations;
    printf("%-15s Size: %8zu KB | Avg: %6.1f cycles (simple test)\n",
           label, size / 1024, avg_cycles);

    free(buffer);
}

// Get CPU frequency for reference
double get_cpu_frequency() {
    FILE* fp = fopen("/proc/cpuinfo", "r");
    if (!fp) return 0.0;

    char line[256];
    double freq = 0.0;

    while (fgets(line, sizeof(line), fp)) {
        if (strncmp(line, "cpu MHz", 7) == 0) {
            sscanf(line, "cpu MHz : %lf", &freq);
            break;
        }
    }

    fclose(fp);
    return freq * 1000000;
}

int main() {
    printf("Cache Latency Measurement Tool (Fixed Version)\n");
    printf("==================================================\n\n");

    // Get CPU frequency for reference
    double cpu_freq = get_cpu_frequency();
    if (cpu_freq > 0) {
        printf("CPU Frequency: %.1f MHz\n", cpu_freq / 1000000);
    }
    printf("Note: Times are in CPU cycles\n\n");

    printf("Method 1: Random Access Pattern\n");
    printf("Cache Level     Size        Average Latency\n");
    printf("--------------------------------------------------\n");

    // Test different sizes - start small to avoid immediate crashes
    measure_cache_latency(8 * 1024, "L1 Cache");        // 8KB
    measure_cache_latency(32 * 1024, "L1 Cache");       // 32KB
    measure_cache_latency(128 * 1024, "L2 Cache");      // 128KB
    measure_cache_latency(512 * 1024, "L2 Cache");      // 512KB
    measure_cache_latency(2 * 1024 * 1024, "L3 Cache"); // 2MB
    measure_cache_latency(8 * 1024 * 1024, "L3 Cache"); // 8MB
    measure_cache_latency(32 * 1024 * 1024, "Main Memory"); // 32MB
    return 0;

}
```



```
CPU Frequency: 2688.0 MHz
Note: Times are in CPU cycles

Method 1: Random Access Pattern
Cache Level     Size          Average Latency
-----------------------------------------------
L1 Cache        Size:        8 KB | Avg:   56.5 cycles | Min:   52 | Max:   72
L1 Cache        Size:       32 KB | Avg:   56.4 cycles | Min:   52 | Max:   76
L2 Cache        Size:      128 KB | Avg:   68.4 cycles | Min:   54 | Max:   74
L2 Cache        Size:      512 KB | Avg:   71.4 cycles | Min:   64 | Max:  136
L3 Cache        Size:     2048 KB | Avg:  122.5 cycles | Min:   64 | Max:  156
L3 Cache        Size:     8192 KB | Avg:  210.0 cycles | Min:  110 | Max: 1386
Main Memory     Size:    32768 KB | Avg:  413.4 cycles | Min:  110 | Max: 2206

Method 2: Simple Sequential Access
```



| Cache | Cache Line Size | Total Size | Associativity | Number of sets |
|---|---|---|---|---|
| L1 | 64 | 45.25KB | 12 | 61 |
| L2 | 64 | 1.25MB | 10 | 2057 |
| L3 | 64 | 23.98 MB | 12 | 32741 |



| Memory Type | Measured Latency |
|---|---|
| L1 | 56.5 |
| L2 | 80.5 |
| L3 | 150 |
| DRAM | 413.4 cycles. |