

SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I
BRODOGRADNJE

NAPREDNE ARHITEKTURE RAČUNALA (250)

IZVJEŠTAJ 2

Toni Biuk
Tino Melvan

Split, svibanj 2018.

SADRŽAJ

Specifikacije računala.....	1
1. Performance i pristup memoriji	2
1.1. Kod	2
1.2. Rezultati.....	2
2. Impakti linija brze memorije.....	4
2.1. Kod	4
2.2. Rezultati.....	4
3. L1, L2 (, L3) brza memorija	6
3.1. Kod	6
3.2. Rezultati.....	6
4. Razine paralelizma.....	8
4.1. Kod	8
4.2. Rezultati.....	8
5. Popunjavanje brze memorije.....	10
5.1. Kod	10
5.2. Rezultati.....	11
6. Konzistencija brze memorije	12
6.1. Kod	12
6.2. Rezultati.....	13
7. Problem hardvera	15
7.1. Kod	15
7.2. Rezultati.....	16
8. Prilog.....	17
8.1. inc/Utility/Traces.hpp	17
8.2. inc/Utility/Clock.hpp	17
8.3. src/Utility/Clock.cpp.....	17
8.4. CMakeLists.txt	18

Specifikacije računala

Tablica 1: Specifikacije računala

Operacijski sustav	Microsoft Windows 10 Pro Build 17134
Procesor	Intel® Core™ i5-8600k @ 3.60 GHz – 4.30 GHz
Broj jezgri	6
Broj niti	6
Brza memorija	9MB SmartCache

1. Performance i pristup memoriji

1.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <cstdint>
#include <cstdlib>

#define BIT (1U)
#define BYTE (8U * BIT)
#define KILOBYTE (1024U * BYTE)
#define MEGABYTE (1024U * KILOBYTE)

#define ARR_MEMORY_SIZE (256U * MEGABYTE)
#define ARR_LENGTH (ARR_MEMORY_SIZE / (sizeof(int32_t) * BYTE))

int32_t main() {
    Clock clock;
    auto *arr = new int32_t[ARR_LENGTH]();

    clock.Start();
    for (auto i = 0U; i < ARR_LENGTH; i++) {
        arr[i] *= 3;
    }
    INFO("Loop_1: %lld ms\n", clock.ElapsedMiliSeconds());

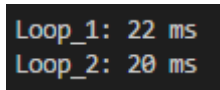
    clock.Start();
    for (auto i = 0U; i < ARR_LENGTH; i += 16) {
        arr[i] *= 3;
    }
    INFO("Loop_2: %lld ms\n", clock.ElapsedMiliSeconds());

    delete[] arr;

    return EXIT_SUCCESS;
}
```

1.2. Rezultati

Dani kod izvršava dvije petlje te mjeri vrijeme potrebno da se one izvrše. Prva prolazi kroz svaki element danog niza, dok druga kroz svaki šesnaesti. Druga petlja, dakle, izvršava šesnaest puta manje prolazaka kroz petlju, pa bi za očekivati bilo da će ona i mnogo kraće trajati. Međutim, kada vrijeme zapravo izmjerimo (Slika 1.1) vidimo da to nije slučaj. U danom primjeru, druga petlja je svega 10% brža. Razlog je sporost pristupanja memoriji, kako bi se dohvatio element niza, dok je sama aritmetička operacija zanemariva (vremenski govoreći).



```
Loop_1: 22 ms
Loop_2: 20 ms
```

Slika 1.1. Ispis primjera 1

S obzirom na to da program ne mora direktno pristupati glavnoj memoriji, već umjesto toga pristupa brzoj, gdje je prvim pristupom učitani cijeli niz, svo vrijeme utrošeno na to postaje mnogo kraće.

2. Impakti linija brze memorije

2.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <stdint>
#include <cstdlib>

#define BIT (1U)
#define BYTE (8U * BIT)
#define KILOBYTE (1024U * BYTE)
#define MEGABYTE (1024U * KILOBYTE)

#define ARR_MEMORY_SIZE (256U * MEGABYTE)
#define ARR_LENGTH (ARR_MEMORY_SIZE / (sizeof(int32_t) * BYTE))

#define INCREMENT (1024U)

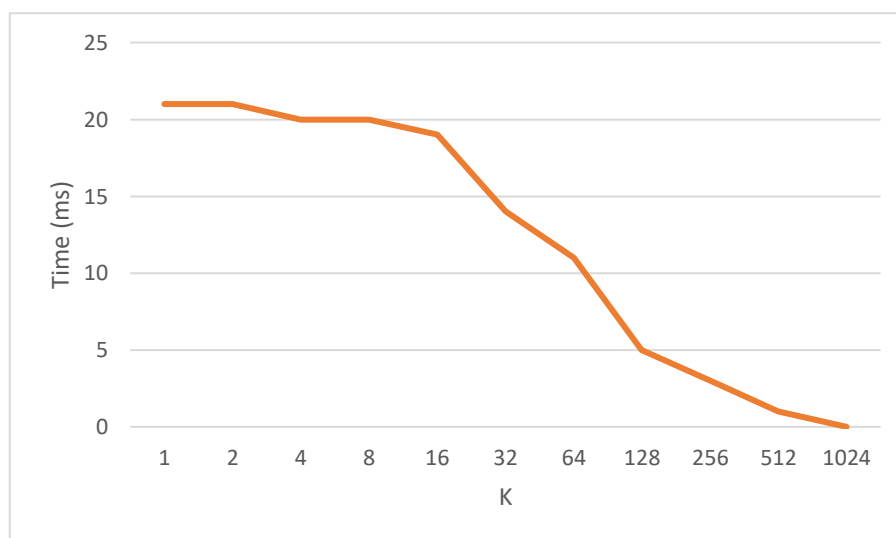
int32_t main() {
    Clock clock;
    auto *arr = new int32_t[ARR_LENGTH]();
    const auto K = INCREMENT;

    clock.Start();
    for (auto i = 0U; i < ARR_LENGTH; i += K) {
        arr[i] *= 3;
    }
    INFO("Loop: %lld ms\n", clock.ElapsedMiliSeconds());

    delete[] arr;

    return EXIT_SUCCESS;
}
```

2.2. Rezultati



Slika 2.1. Ovisnost veličine koraka o vremenu izvršavanja koda u "primjeru 2"

U ovom primjeru ispituje se brzina izvršavanja petlje ovisno o veličini koraka. Za broj koraka manjih od 16, vrijeme izvršavanja je relativno slično, iako se uočava mali pad zbog manjeg broja operacija koje je potrebno izvršiti. Vrijeme pristupa brzoj memoriji za takve korake je konstantno jer brza memorija posjeduje linije od 64 bajta. Također, upravo zato što ima linije od 64 bajta, kod koraka većih od 16 učit će se pad. To se događa zbog činjenice da brza memorija može spremiti točno 16 elemenata u svoju jednu liniju (*int32_t* velik je 4 bajta, pa vrijedi $16 * 4\text{bajta} = 64\text{bajta}$). S daljnjim porastom koraka, program više neće morati dirati sve linije brze memorije, već će određene moći preskakati, što će uzrokovati velika ubrzanja.

3. L1, L2 (, L3) brza memorija

3.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <cstdint>
#include <cstdlib>

#define BIT (1U)
#define BYTE (8U * BIT)
#define KILOBYTE (1024U * BYTE)
#define MEGABYTE (1024U * KILOBYTE)

#define ARR_MEMORY_SIZE (8U * MEGABYTE)
#define ARR_LENGTH (ARR_MEMORY_SIZE / (sizeof(int32_t) * BYTE))

int32_t main() {
    Clock clock;

    const auto steps = 64U * 1024U * 1024U;
    auto *arr = new int32_t[ARR_LENGTH]();

    INFO("Array size: %llu\n", ARR_LENGTH);
    clock.Start();
    for (auto i = 0U; i < steps; i++) {
        arr[(i * 16U) % ARR_LENGTH]++;
    }

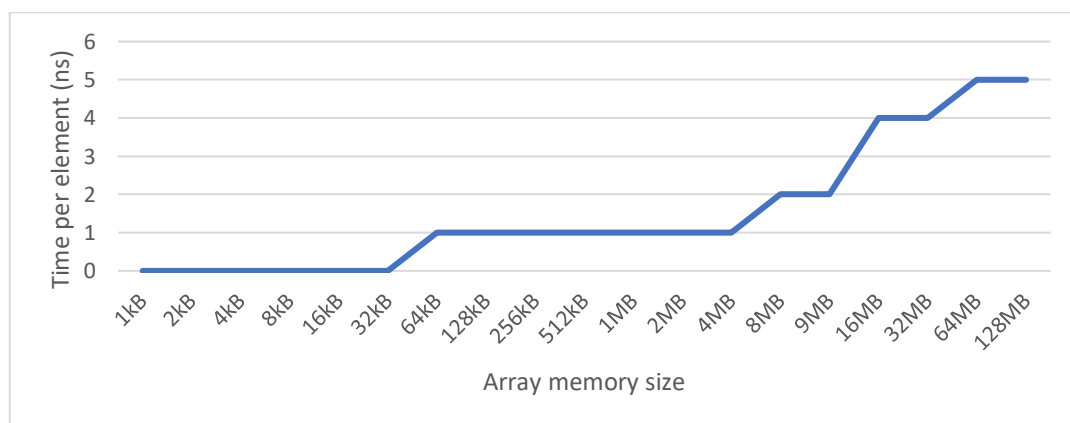
    const auto elapsedNS = clock.ElapsedNanoSeconds();
    INFO("Time per element: %lld ns\n", elapsedNS / steps);

    delete[] arr;

    return EXIT_SUCCESS;
}
```

3.2. Rezultati

Mijenjajući veličinu ulaznog niza cijelih brojeva (`ARR_MEMORY_SIZE`), iz prethodno danog programa dobije se graf prikazan u nastavku.



Slika 3.1. Ovisnost veličine niza i vremena izvršavanja po elementu niza u primjeru 3.

Promatranjem ovisnosti niza i vremena izvršavanja možemo odrediti veličinu brze memorije i njenih dijelova. Iz grafa, očite promijene nastupaju poslije 32kB i 9MB. Iz toga se može iščitati da je L1 velik 32kB, dok je L3 velik 9MB. Ako pak ne znamo da procesor ima L3 brzu memoriju, dalo bi se zaključiti da je L2 velik 9MB, te da L3 ne postoji. Međutim, to nije slučaj, već samo L1 i L2 imaju slične performanse u ovome slučaju, a i vrijednosti su zaokruživane na donji cijeli broj. L2 bi trebao biti velik 256kB. Ako podatke nije moguće smjestiti u L3 brzu memoriju, vrijeme izvršavanja počet će brzo rasti zbog brojnih pristupanja glavnoj memoriji, što je vidljivo iz grafa.

4. Razine paralelizma

4.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <cstdint>
#include <cstdlib>

#define BIT (1U)
#define BYTE (8U * BIT)
#define KILOBYTE (1024U * BYTE)
#define MEGABYTE (1024U * KILOBYTE)

#define ARR_LENGTH (2U)

int32_t main() {
    Clock clock;

    const auto steps = 32U * MEGABYTE;
    auto *arr = new int32_t[ARR_LENGTH]();

    clock.Start();
    for (auto i = 0U; i < steps; i++) {
        arr[0U]++;
        arr[0U]++;
    }
    INFO("Elapsed time: %lld ms\n", clock.ElapsedMiliSeconds());

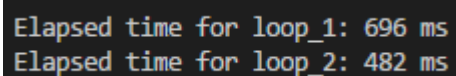
    clock.Start();
    for (auto i = 0U; i < steps; i++) {
        arr[0U]++;
        arr[1U]++;
    }
    INFO("Elapsed time: %lld ms\n", clock.ElapsedMiliSeconds());

    delete[] arr;

    return EXIT_SUCCESS;
}
```

4.2. Rezultati

Rezultati izvršavanja prethodno danog koda, uz isključenu optimizaciju kompajlera, dani su na slici ispod.



```
Elapsed time for loop_1: 696 ms
Elapsed time for loop_2: 482 ms
```

Slika 4.1. Rezultati izvršavanja primjera 4.

Druga petlja brža je od prve zbog činjenice da moderni procesori mogu pristupati višestrukim lokacijama unutar L1 brze memorije istovremeno. Prva petlja ovakvo nešto ne može iskoristavati zbog toga što pristupa istom elementu, pa paralelizacija nije moguća zbog ovisnosti između više operacija. S druge strane, ovakav tip paralelizacije urednu prolazi kod petlje broj dva, zbog toga što se radi o dva različita elementa.

5. Popunjavanje brze memorije

5.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <cstdint>
#include <cstdlib>

#define BIT (1U)
#define BYTE (8U * BIT)
#define KILOBYTE (1024U * BYTE)
#define MEGABYTE (1024U * KILOBYTE)

#define ARR_MEMORY_SIZE (24U * MEGABYTE)
#define ARR_LENGTH (ARR_MEMORY_SIZE / (sizeof(int32_t) * BYTE))

#define MAX_STEP (576U)

typedef uint8_t byte;

int64_t UpdateEveryKthByte(byte *arr, const uint32_t K) noexcept {
    Clock clock;
    clock.Start();

    const auto rep = MEGABYTE / BYTE;
    auto p = 0U;

    if (!arr) {
        return INT64_MAX;
    }

    for (auto i = 0U; i < rep; i++) {
        arr[p]++;
        p += K;
        if (p >= ARR_LENGTH) {
            p = 0U;
        }
    }

    return clock.ElapsedMiliSeconds();
}

int32_t main() {
    auto *arr = new byte[ARR_LENGTH]();

    for (auto step = 0U; step < MAX_STEP; step++) {
        const auto elapsedMiliSeconds = UpdateEveryKthByte(arr, step);
        INFO("Step:%u\tTime: %lld ms\n", step, elapsedMiliSeconds);
    }

    delete[] arr;

    return EXIT_SUCCESS;
}
```

5.2. Rezultati

Rezultat izvršavanja programa iz ovog primjera, uz mijenjanje veličine niza jest graf u nastavku. On je pokazuje ovisnost vremena izvršavanja u odnosu na veličinu niza (1-24MB) te veličinu koraka (1-600). Apscisa predstavlja korak, dok ordinata predstavlja veličinu niza.



Slika 5.1. Relativno izvršavanje različitih veličina (vremenski) nizova ovisno o veličini koraka

Na grafu, tamnije mrlje predstavljaju relativno sporije izvršavanje u odnosu na ostatak. Tako možemo uočiti da su posebno loše veličine koraka 128, 256, 384 i 512.

U konkretnom procesoru radi se o 12-way asocijativnoj brzoj memoriji, što znači da je ona podijeljena u 12 setova. Različiti koraci pristupaju različitim setovima (jedan od tih 12), a posebni koraci mogu natjerati mapiranje u jako malo setova (najgori slučaj, sve mapirano u jedan). Time se smanji iskoristivost brze memorije, pa samim time treba više učitavati i spremati nove vrijednosti. Sve navedeno uveliko smanji izvršavanje.

Također, na grafu, možemo primijetiti da je krajnje lijeva strana najsvjetlija. To je zbog toga što tu najviše susjednih koraka koristi isti set brze memorije.

6. Konzistencija brze memorije

6.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <cstdint>
#include <cstdlib>
#include <thread>
#include <vector>

#define BIT (1U)
#define BYTE (8U * BIT)
#define KILOBYTE (1024U * BYTE)
#define MEGABYTE (1024U * KILOBYTE)

#define ARR_MEMORY_SIZE (256U * MEGABYTE)
#define ARR_LENGTH (ARR_MEMORY_SIZE / (sizeof(int32_t) * BYTE))

#define NUM_OF_THREADS (4U)

auto *s_counter = new int32_t[ARR_LENGTH]();

static void UpdateCounter(const uint32_t position) noexcept {
    for (auto j = 0U; j < 100000000U; j++) {
        s_counter[position] = s_counter[position] + 3;
    }
}

int32_t main() {
    Clock clock;

    const uint32_t positions1[] = { 0, 1, 2, 3 };
    const uint32_t positions2[] = { 16, 32, 48, 64 };

    std::vector<std::thread> threads;
    threads.reserve(NUM_OF_THREADS);

    clock.Start();
    for (auto i = 0U; i < NUM_OF_THREADS; i++) {
        threads.emplace_back(std::thread(UpdateCounter, positions1[i]));
    }

    for (auto& thread : threads) {
        thread.join();
    }
    INFO("Positions_1 took %.2f seconds\n", clock.ElapsedMiliSeconds() / 1000.0);

    threads.clear();
    clock.Start();
    for (auto i = 0U; i < NUM_OF_THREADS; i++) {
        threads.emplace_back(std::thread(UpdateCounter, positions2[i]));
    }

    for (auto& thread : threads) {
        thread.join();
    }
    INFO("Positions_2 took %.2f seconds\n", clock.ElapsedMiliSeconds() / 1000.0);

    return EXIT_SUCCESS;
}
```

6.2. Rezultati

S problemom konzistencije brza memorija susreće se gotovo na svim više-jezgrenim računalima. To je uzrokovano činjenicom da svaka od jezgara ima svoju zasebnu brzu memoriju (L1 i L2 dio), što se može vidjeti na slici dolje. Na njoj je prikazan ispis programa *CPU-Z* te prikazuje detaljne podatke o brzjoj memoriji za procesor koji se nalazi unutar računala .

L1 D-Cache	
Size	32 KBytes x 6
Descriptor	8-way set associative, 64-byte line size
L1 I-Cache	
Size	32 KBytes x 6
Descriptor	8-way set associative, 64-byte line size
L2 Cache	
Size	256 KBytes x 6
Descriptor	4-way set associative, 64-byte line size
L3 Cache	
Size	9 MBytes
Descriptor	12-way set associative, 64-byte line size

Slika 6.1. Ispis programa CPU-Z

Problem možemo demonstrirati danim programom. Ako ga pokrenemo (bez ikakvih optimizacija) dobijemo ispis dan u nastavku (Slika 6.2).

```
Positions_1 took 1.28 seconds
Positions_2 took 0.18 seconds
```

Slika 6.2. Ispis primjera 6

Razlog ovakvog ponašanja direktno je povezan s duljinama linije brze memorije i ponašanjem brzih memorija kada se radi o višenitnosti. Prvo, bitno je napomenuti da su elementi niza alocirani dinamički spremljeni redom u memoriji, jedan iza drugoga. S obzirom na to da brza memorija u korištenoj mašini koristi 64-bajtnu liniju, u svaku od njih može se spremiti najviše šesnaest 32-bitnih cijelih brojeva (engl. *Integer*). To znači da, u prvom našem slučaju, gdje pristupamo elementima s pozicijama 0, 1, 2 i 3 u nizu, imamo veliku vjerojatnost da će svi traženi elementi biti na istoj liniji brze memorije. S druge strane, u drugom slučaju, gdje

pristupamo elementima s pozicijama 16, 32, 48 i 64, imamo obrnutu situaciju kada imamo veliku vjerojatnost da su svi elementi na različitoj liniji brze memorije. Razlog zašto je to toliko bitno jest, kada jezgra mijenja vrijednost u svojoj brzoj memoriji, također onemogućuje svim drugim jezgrama korištenje stare vrijednosti za odgovarajuću adresu. Nadalje, kada se onemogućuje nešto u broj memoriji, onemogućuje se cijela linija, a ne samo njezin dio. To znači da, niti jedna jezgra, u svom sljedećem pristupu, neće pronaći traženu vrijednost za adresu u brzoj memoriji, što rezultira u mnogo sporijem izvršavanju.

7. Problem hardvera

7.1. Kod

```
#include <Utility/Clock.hpp>
#include <Utility/Traces.hpp>

#include <cstdint>
#include <cstdlib>

uint32_t A = 0;
uint32_t B = 0;
uint32_t C = 0;
uint32_t D = 0;
uint32_t E = 0;
uint32_t F = 0;
uint32_t G = 0;

static void WeirdnessABCD() noexcept {
    for (auto i = 0U; i < 200000000U; i++) {
        A++;
        B++;
        C++;
        D++;
    }
}

static void WeirdnessACEG() noexcept {
    for (auto i = 0U; i < 200000000U; i++) {
        A++;
        C++;
        E++;
        G++;
    }
}

static void WeirdnessAC() noexcept {
    for (auto i = 0U; i < 200000000U; i++) {
        A++;
        C++;
    }
}

int32_t main() noexcept {
    Clock clock;

    clock.Start();
    WeirdnessABCD();
    auto elapsedTimeMS = clock.ElapsedMiliSeconds();
    INFO("A++; B++; C++; D++;\t\t%lld ms\n", elapsedTimeMS);

    clock.Start();
    WeirdnessACEG();
    elapsedTimeMS = clock.ElapsedMiliSeconds();
    INFO("A++; C++; E++; G++;\t\t%lld ms\n", elapsedTimeMS);

    clock.Start();
    WeirdnessAC();
    elapsedTimeMS = clock.ElapsedMiliSeconds();
    INFO("A++; C++;\t\t\t%lld ms\n", elapsedTimeMS);

    return EXIT_SUCCESS;
}
```

7.2. Rezultati

Ovaj primjer trebao bi demonstrirati čudno ponašanje brze memorije kao posljedica ovisnosti o drugom hardveru. Iako u potpunosti znamo kako bi se brza memorija trebala ponašati, činjenica je da je i dalje teško predvidjeti. Pogledom na primjer dan u poglavlju 7.1, moglo bi se krivo zaključiti da će se funkcija koja inkrementira manje varijabli (**WeirdnessAC**) uvijek (ili barem u prosijeku) izvršavati brže od funkcije koja inkrementira više varijabli (**WeirdnessABCD**, **WeirdnessACEG**), naravno, ako su one po svemu ostalome identične. Puštanje sveukupnog programa otkriti će, pak, da to nije slučaj. Iz priloženog rezultata (Slika 6.2) može se vidjeti da je **WeirdnessABCD** duplo sporija od **WeirdnessACEG**, a treća funkcija, **WeirdnessAC**, ima gotovo jednako vrijeme izvršavanja kao i prethodno spomenuta.

A++; B++; C++; D++;	99 ms
A++; C++; E++; G++;	49 ms
A++; C++;	49 ms

Slika 7.1. Ispis primjera 7

Razlozi zašto bi ovo mogao biti slučaj su razni. Najlakše bi bilo zaključiti kako ovo ima veze s redoslijedom kojim instrukcije završe na procesoru, ili pak s činjenicom koliko se brzo otkrije podudaranje u brzoj memoriji. Problem može biti i dublji, ako kompajler (tj. procesor) koristi SSE (engl. *Streaming SIMD Extensions*) instrukcije. Takva vrsta ne radi s jednom po jednom varijablom, već ih kupi i do četiri odjednom, ukoliko su instrukcije dovoljno slične. Sve u svemu, **WeirdnessABCD** je najsporija vjerojatno zbog činjenice što se tada podaci još ne nalaze u broj memoriji. Što se tiče razlike između **WeirdnessACEG** i **WeirdnessAC**, ona je slična zbog situacije kao i u prvom primjeru prvog zadatka.

Zanimljiva je i činjenica da moderni kompajleri ovakve situacije jako dobro mogu optimizirati, pa tako GCC kompajler uz optimizaciju -O3 ima gotovo zanemarivo izvršavanje ovog koda.

A++; B++; C++; D++;	0 ms
A++; C++; E++; G++;	0 ms
A++; C++;	0 ms

Slika 7.2. Primjer 7 uz O3 optimizaciju GCC kompajlera

8. Prilog

8.1. inc/Utility/Traces.hpp

```
#ifndef UTILITY_TRACES_HPP
#define UTILITY_TRACES_HPP

#include <cstdio>

#define INFO(fmt, ...) do { fprintf(stderr, fmt, __VA_ARGS__); } while (0);

#if _DEBUG
#define DEBUG(fmt, ...) do { INFO(fmt, __VA_ARGS__); } while (0);
#else
#define DEBUG(mt, ...)
#endif

#endif // !UTILITY_TRACES_HPP
```

8.2. inc/Utility/Clock.hpp

```
#ifndef UTILITY_CLOCK_HPP
#define UTILITY_CLOCK_HPP

#include <chrono>

class Clock {
public:
    Clock() noexcept;

    void Start() noexcept;
    int64_t ElapsedMiliSeconds() const noexcept;
    int64_t ElapsedNanoSeconds() const noexcept;

private:
    std::chrono::time_point<std::chrono::system_clock> StartTime;
};

#endif // !UTILITY_CLOCK_HPP
```

8.3. src/Utility/Clock.cpp

```
#include "Utility/Clock.hpp"

Clock::Clock() noexcept {
    this->Start();
}

void Clock::Start() noexcept {
    StartTime = std::chrono::system_clock::now();
}

int64_t Clock::ElapsedMiliSeconds() const noexcept {
    const auto CurrentTime = std::chrono::system_clock::now();
    const std::chrono::duration<double> elapsed_seconds = CurrentTime - StartTime;
    return std::chrono::duration_cast<std::chrono::milliseconds>(elapsed_seconds).count();
}
```

```
int64_t Clock::ElapsedNanoSeconds() const noexcept {
    const auto CurrentTime = std::chrono::system_clock::now();
    const std::chrono::duration<double> elapsed_seconds = CurrentTime - StartTime;
    return
std::chrono::duration_cast<std::chrono::nanoseconds>(elapsed_seconds).count();
}
```

8.4. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1)

get_filename_component(DIR_NAME "${CMAKE_CURRENT_LIST_DIR}" NAME)
project(${DIR_NAME})

set(CMAKE_BUILD_TYPE Release)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "-Wall -Wextra")
set(CMAKE_CXX_FLAGS_DEBUG "-g")
set(CMAKE_CXX_FLAGS_RELEASE "-O3")

include_directories(inc/)

set(HELLO_HEADER_FILES
    inc/Utility/Clock.hpp)
set(HELLO_SOURCE_FILES
    src/Utility/Clock.cpp)
set(MAIN_FILE
    src/main.cpp)

add_executable(${PROJECT_NAME}
    ${HELLO_HEADER_FILES}
    ${HELLO_SOURCE_FILES}
    ${MAIN_FILE})
```