POLI
ESCOLA SUPERIOR
TECNOLOGIA
GESTÃO
TÉCNICO
GUARDA

COMPUTER ARCITECTURE PROJECT

MOVE DATA IN RAM

Curso:	Lic. Engenharia Informática
Unidade curricular:	Computer Arcihecture
Ano letivo:	2022/2023
Nº e nome de aluno:	1707106 ÖMER BİLAL USTA
Docente:	Prof. Luis Figueiredo
Data:	14 de maio de 2022

INDEXES

1.INTRODUCTION

Our purpose is find faster way to move data in RAM with this report. This report also shows how to move data in ram, copying from an address to put another address. We will see results at the end of this report.

For this , we will use Assembly and C. We will use two array and copy from array1 to array2. In C we have Creation of variables, Cycle counting and result of it, Calling functions and Librarys. In Assembly we have Functions, Registers(to store data temporarily) and Loops.

In C, We created variables(array1, array2, counters). After that, filled array1(to use as source) using loop then called function for move data from array1 into array2. At this point Assembly codes are active, pushed EBP into stack (EBP will point array's pointers). With using EBP, we got array's pointer and put them into registers (ESI AND EDI), also we got counter and put it into ECX (for number of cycles). And then, Loop starts. In Loop, first row does: gets data from array1(using ESI) and put it into register. At second row, we get data from register, put it into array2(using EDI). After moving data, we change ESI, EDI and ECX. Loop works until reach end of arrays.

For this task, we have 5 strategy. Moving data as 1byte, as 2byte, as 4byte, as 8byte, as 16byte.

For 1 byte strategy, We used AL register(8 bit) and MOV instruciton.

For 2 byte strategy, We used AX register(16 bit) and MOV instruction.

For 4 byte strategy, We used EAX register(32 bit) and MOV instruction.

For 8 byte strategy, We used MMX register(64 bit)and MOVQ instruction.

For 16 byte strategy, We used XMM register(128 bit) and MOVDQA instruction.

1Byte is 8 bit, 2 byte is 16 bit, 4 byte is 32 bit, 8 byte is 64 bit, 16 byte is 128 bit.

2.THEORETICAL EXPLANATION

a-) Instructions

a-1)Data Movement Instructions

a-1-1) Mov, Movq, Movdqa

The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not.

Also movq and movdqa instructions are SIMD (Single Instruction Multiple Data) instructions.

MOV uses with 8, 16, 32 bit register.

MOVQ uses with 64 bit registers.

MOVDQA uses with 128 bit registers.

a-1-2) Push

The push instruction places its operand onto the top of the hardware supported stack in memory.

a-1-3) Pop

The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location).

a-2) Arithmetic and Logic Instructions

a-2-1) Add

The add instruction adds together its two operands, storing the result in its first operand.

a-2-2) Sub

The sub instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand.

a-3) Control Flow Instruction

a-3-1) JNZ

JNZ instruction checks Zero Flag. If Zero Flag isn't 0(zero), loop continues. Otherwise Loop stops.

b-) Registers

Registers are in CPU. Some Different register has same size for example EAX and EBX, AL and AH.

Registers are a type of computer memory used to quickly accempt, store and trasnfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often named as Processor registers.

A processor register may hold an instruction, a storage adress or any data(such as bit sequence or individual characters).

The computer needs processor registers for maipulating data and a register for holding a memory adress The register holding the memory location is used to calculate the address of the next instruciton after the execution of the current instruciton is completed.

AH, AL (etc.)registers are 8 bit registers. They can store 1 byte data.

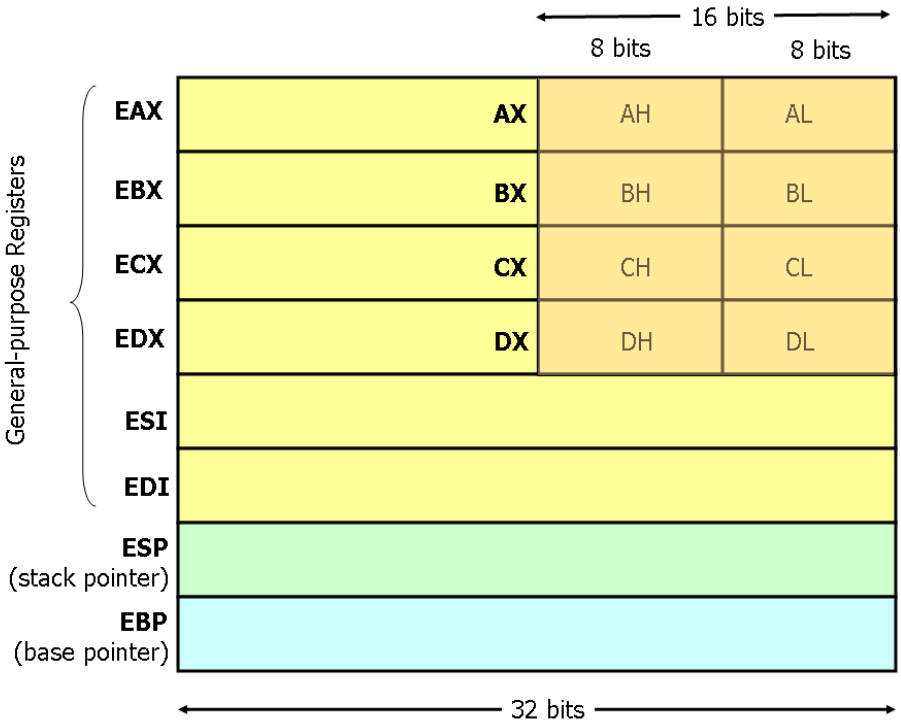
AX, BX (etc.) registers are 16 bit registers. They can store 2 byte data.

EAX, EBX, ESI, EDI (etc.) registers are 32 bit registers. They can store 4 byte data.

ESI and EDI are index registers. They contains memory adress. ESI is for source address, EDI is for destination address.

The EBP register is placed at the top of the stack and is used to point to what's in the stack.

ECX is called the Counter register. It is used as a loop counter and for shifts.



c-) SIMD (Single Instruction Multiple Data)

SIMD is a computational technique for processing a number of data values (generally a power of two) using a single instruction, with the data for the operands packed into special wide registers. One instruction can therefore do the work of many separate instructions. This type of parallel processing instruction is commonly called SIMD (single instruction, multiple data).

A technique for processing multiple data values using a single instruction, with the data for the operands packed into wide registers. One instruction can therefore do the work of many. SIMD instructions are very powerful for computations on media data.

SIMD works with 64, 128, 256, 528 bit registers (MMX, XMM, YMM, ZMM). In this project, we worked with MMX and XMM registers. MMX is 64 bit, XMM is 128 bit register.

Also We need special instructions for this. We worked with movdqa and movq. Movq is for 64 bit, movdqa is for 128 bit.

Not: Movdqa used with memory, that memory must be aligned at 16 bytes.

MMX REGISTER

MMX register are 64 bit registers. They can store 8 byte data.

<table>
<tr><td colspan="8">quadword</td></tr>
<tr><td colspan="4">doubleword</td><td colspan="4">doubleword</td></tr>
<tr><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td></tr>
<tr><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td></tr>
</table>

XMM REGISTER

XMM registers are 128 bit registers. They can store 16 byte data

<table>
<tr><td colspan="8">double float</td><td colspan="8">double float</td></tr>
<tr><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td><td colspan="4">float</td></tr>
<tr><td colspan="8">quadword</td><td colspan="8">quadword</td></tr>
<tr><td colspan="4">doubleword</td><td colspan="4">doubleword</td><td colspan="4">doubleword</td><td colspan="4">doubleword</td></tr>
<tr><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td><td colspan="2">word</td></tr>
<tr><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td><td>byte</td></tr>
</table>

3. TECHNICAL CHAPTER (codes)

At C side, variables created, arrays filled with data, called functions and calculated CPU cycles.

At Assembly side, data moved between arrays.

a-) C

Firstly I created arrays and initial_counter,final_counter. Then filled array1 and called function. When function is done, we come back to C side and (final_counter – initial_counter) shows CPU cyles (It is calculated while function works.). And this is repeat for other functions.

Not:Between functions execution, we reseted destination array(array2).

You can see C codes below.

```
#include "conio.h"
#include <stdio.h>
#include <intrin.h>
extern "C" { //indicate that we have an external function
        int move_array_1byte(int* var, int* varr, int arraysize);
        int move_array_2byte(int* var, int* varr, int arraysize);
        int move_array_4byte(int* var, int* varr, int arraysize);
        int move_array_8byte(int* var, int* varr, int arraysize);
        int move_array_16byte(int* var, int* varr, int arraysize);

}
int main(int argc, char* argv[])
{
        #define arraysize 8
        __declspec(align(16))
        int array1[arraysize];
        int array2[arraysize];
        unsigned __int64 initial_counter, final_counter;

        for (int i = 0; i < arraysize; i++)
        {
                array1[i] = i;
        }

        initial_counter = __rdtsc();
        move_array_1byte(array1, array2, (arraysize*4));
        final_counter = __rdtsc();
        printf("\nCode executed in %I64d CPU cycles\n", final_counter - initial
_counter);


        for (int i = 0; i < arraysize; i++)
        {
                array2[i] = 0;
        }

        initial_counter = __rdtsc();
        move_array_2byte(array1, array2, (arraysize * 4));
```

```
        final_counter = __rdtsc();
        printf("\nCode executed in %I64d CPU cycles\n", final_counter - initial
_counter);


        for (int i = 0; i < arraysize; i++)
        {
                array2[i] = 0;
        }

        initial_counter = __rdtsc();
        move_array_4byte(array1, array2, (arraysize * 4));
        final_counter = __rdtsc();
        printf("\nCode executed in %I64d CPU cycles\n", final_counter - initial
_counter);


        for (int i = 0; i < arraysize; i++)
        {
                array2[i] = 0;
        }

        initial_counter = __rdtsc();
        move_array_8byte(array1, array2, (arraysize * 4));
        final_counter = __rdtsc();
        printf("\nCode executed in %I64d CPU cycles\n", final_counter - initial
_counter);



        int array4[arraysize];
        initial_counter = __rdtsc();
        move_array_16byte(array1, array4, (arraysize * 4));
        final_counter = __rdtsc();
        printf("\nCode executed in %I64d CPU cycles\n", final_counter - initial
_counter);



}
```

b-) Assembly

You can see Assembly codes below and explanation of functions at second function (Functions are basicly same. Because ot that there is only one explanation).

```
.MODEL FLAT, C
.CODE

PUBLIC move_array_1byte
move_array_1byte PROC
        push ebp
        mov ebp,esp
        mov ESI,[ebp + 8]
        mov EDI,[ebp + 12]
        mov ecx,[ebp + 16]
rp1:
        mov AL, [ESI]
        mov [EDI], AL
        add ESI, 1
        add EDI, 1
```

```
        sub ecx, 1
        jnz rp1

        pop ebp
        ret
move_array_1byte ENDP


PUBLIC move_array_2byte
move_array_2byte PROC
        push ebp              ;Pushed EBP into Stack
        mov ebp,esp           ;ESP's value(address of array1 pointer)
        mov ESI,[ebp + 8]     ;Getting array1's pointer and put into ESI
        mov EDI,[ebp + 12]    ;Getting array2's pointer and put into EDI
        mov ecx,[ebp + 16]    ;Getting (arraysize*4) and put into ECX
rp1:
        mov AX, [ESI]
        mov [EDI], AX         ;Data moved from array1 to array2
        add ESI, 2            ;ESI increased by 2 because this function
        add EDI, 2            ;is 2byte-2byte read and write
        sub ecx, 2            ;Counter decreased by 2
        jnz rp1               ;Checked counter and decided stop or
                              ;continue
        pop ebp               ;Popped EBP from stack
        ret

move_array_2byte ENDP         ;End of Function


PUBLIC move_array_4byte
move_array_4byte PROC
        push ebp
        mov ebp,esp
        mov ESI,[ebp + 8]
        mov EDI,[ebp + 12]
        mov ecx,[ebp + 16]
rp1:
        mov EAX, [ESI]
        mov [EDI], EAX
        add ESI, 4
        add EDI, 4
        sub ecx, 4
        jnz rp1

        pop ebp
        ret
move_array_4byte ENDP


PUBLIC move_array_8byte
move_array_8byte PROC
        push ebp
        mov ebp, esp
        mov ESI, [ebp + 8]
        mov EDI, [ebp + 12]
        mov ecx, [ebp + 16]
rp1:
        movq mm0, [ESI]
        movq[EDI], mm0
        add ESI, 8
```

```
        add EDI, 8
        sub ecx, 8
        jnz rp1

        pop ebp
        ret
move_array_8byte ENDP


PUBLIC move_array_16byte
move_array_16byte PROC
        push ebp
        mov ebp,esp
        mov ESI,[ebp + 8]
        mov EDI,[ebp + 12]
        mov ecx,[ebp + 16]
rp1:
        movdqa xmm0, [ESI]
        movdqa [EDI], xmm0
        add ESI, 16
        add EDI, 16
        sub ecx, 16
        jnz rp1

        pop ebp
        ret

move_array_16byte ENDP
END
```

4.RESULTS

I executed code 3 times and I got 3 diffrent result. Array size is 32 byte(4*8) for 1st, 2nd and 3rd attempt. Array size is 800 byte (4*200) for 4th and 5th attempt.

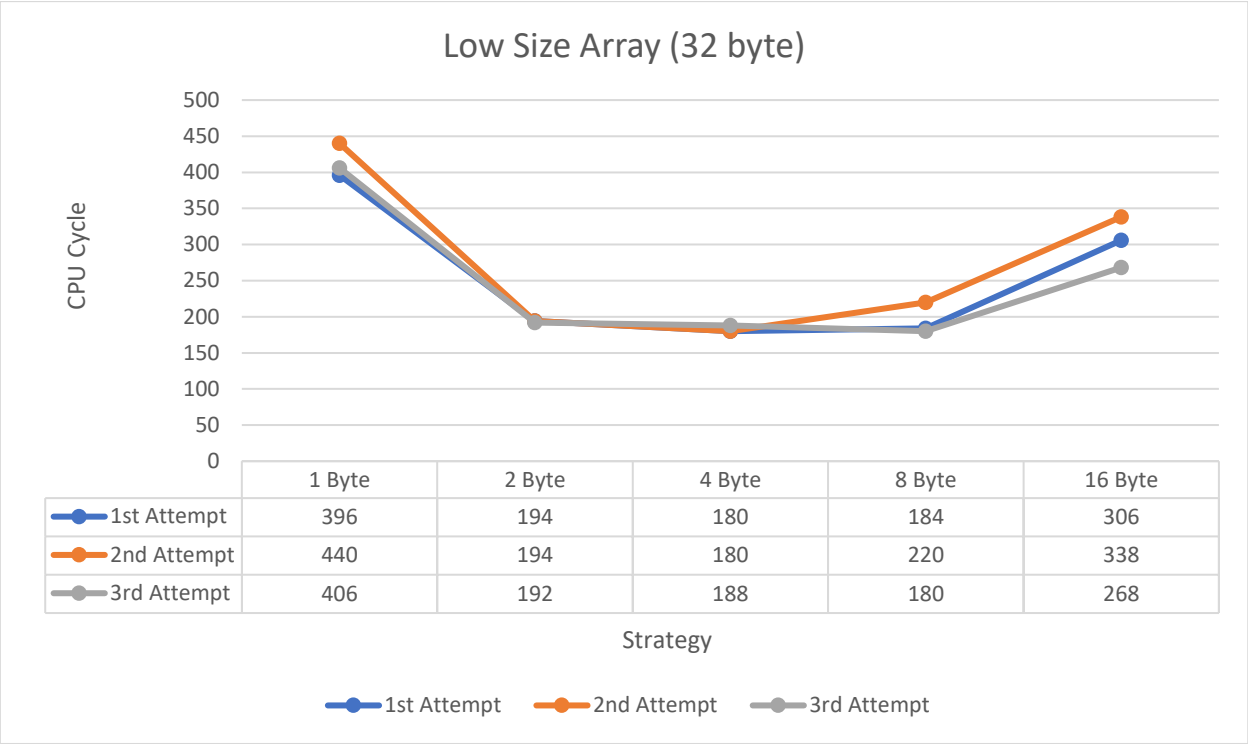
At 1st attempt, 4 byte strategy is fastest, 8 byte strategy is 2nd fastest, 2 byte strategy is 3rd fastest, 16 byte is strategy 4th fastest, 1byte strategy is slowest.

At 2nd attempt, 4 byte strategy is fastest, 2 byte strategy is 2nd fastest, 8 byte strategy is 3rd fastest, 16 byte is strategy 4th fastest, 1byte strategy is slowest.

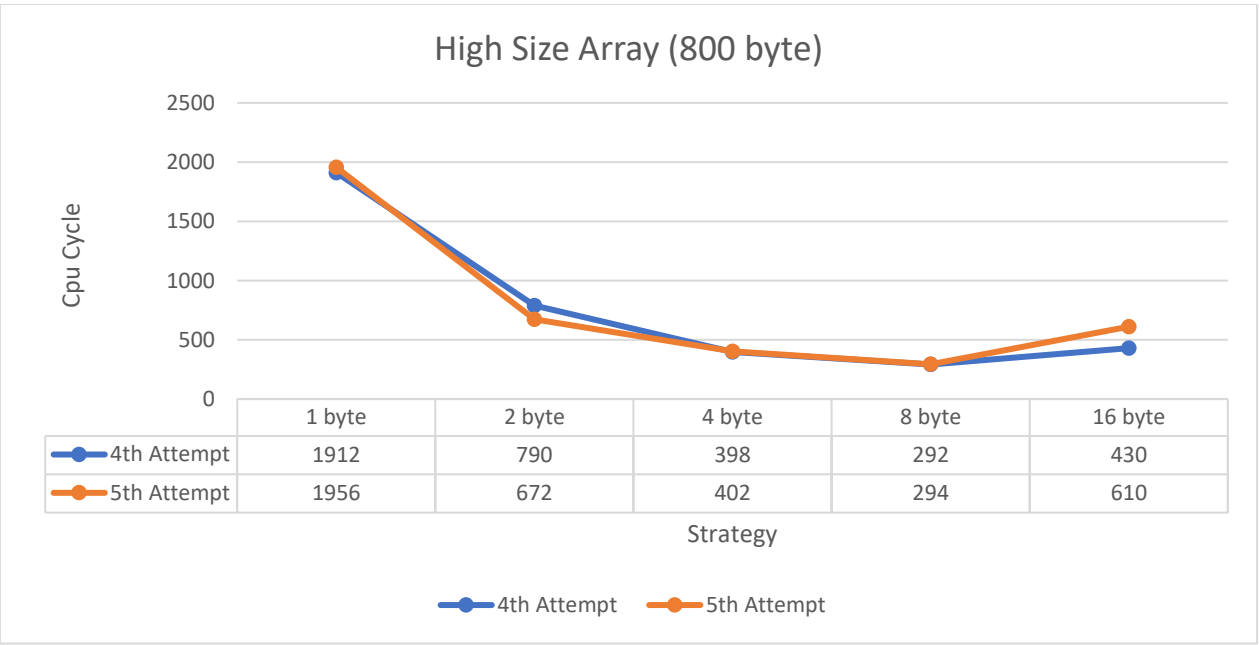
At 3nd attempt, 8 byte strategy is fastest, 4 byte strategy is 2nd fastest, 2 byte strategy is 3rd fastest, 16 byte is strategy 4th fastest, 1byte strategy is slowest.

At 4th attempt, 8 byte strategy is fastest, 4 byte strategy is 2nd fastest, 16 byte strategy is 3rd fastest, 2 byte strategy is 4th fastest, 1 byte strategy is slowest.

At 5th attempt, 8byte strategy is fastest, 4 byte strategy is 2nd fastest, 16 byte strategy is 3rd fastest, 2 byte strategy is 4th fastest, 1byte strategy is slowest.



	1 Byte	2 Byte	4 Byte	8 Byte	16 Byte
1st Attempt	396	194	180	184	306
2nd Attempt	440	194	180	220	338
3rd Attempt	406	192	188	180	268



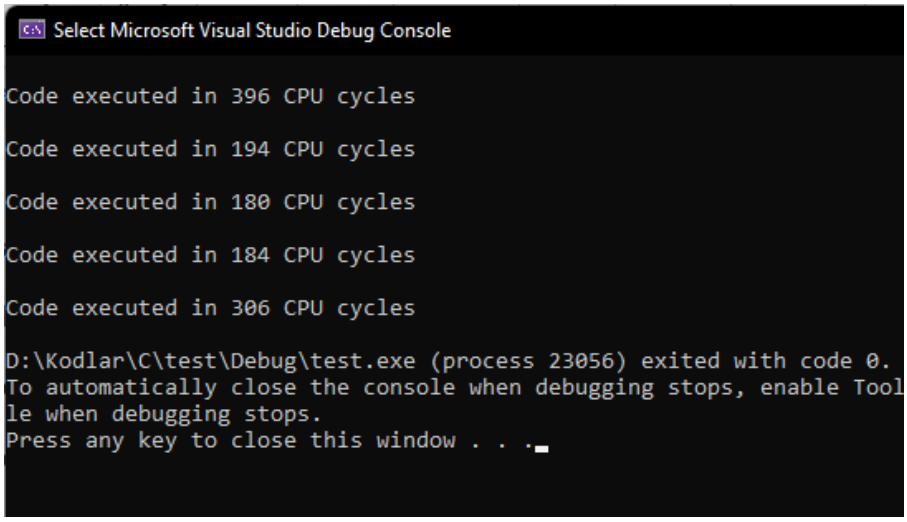
	1 byte	2 byte	4 byte	8 byte	16 byte
4th Attempt	1912	790	398	292	430
5th Attempt	1956	672	402	294	610

At the end, we can see 4 byte strategy looks more stable and faster for low size array, 8 byte strategy is faster for high size array.

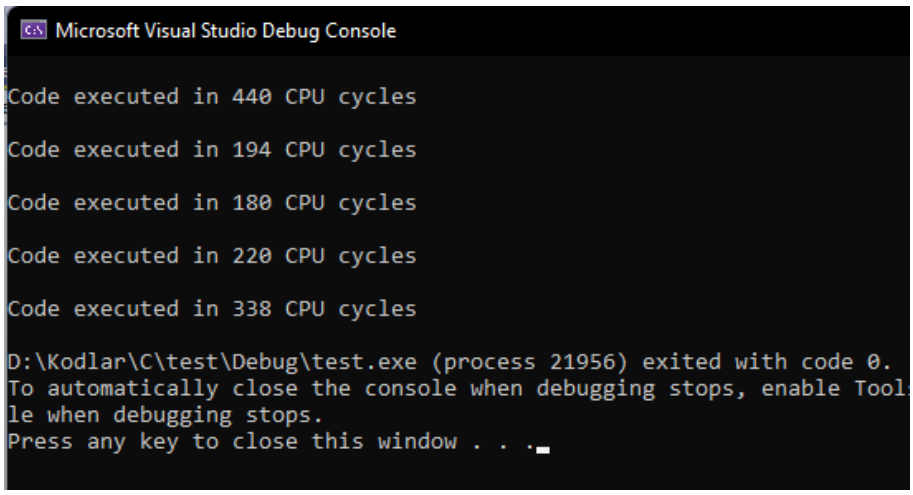
For low size array, 4 byte strategy should be used.

For high size array, 8 byte strategy should be used.

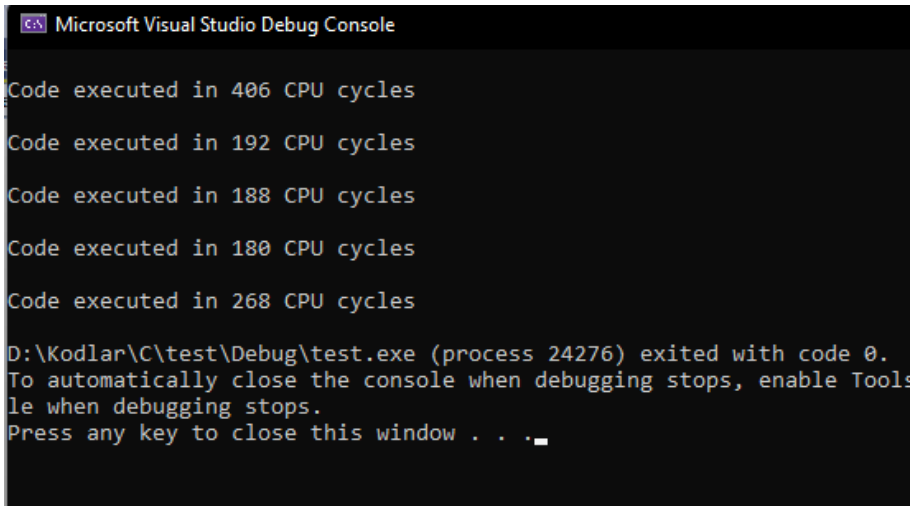
a-) 1st attempt



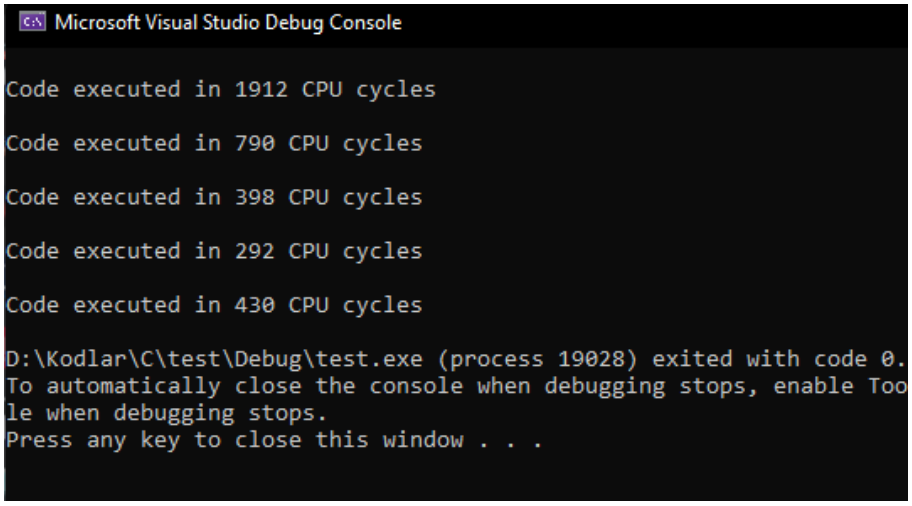
b-) 2nd attempt



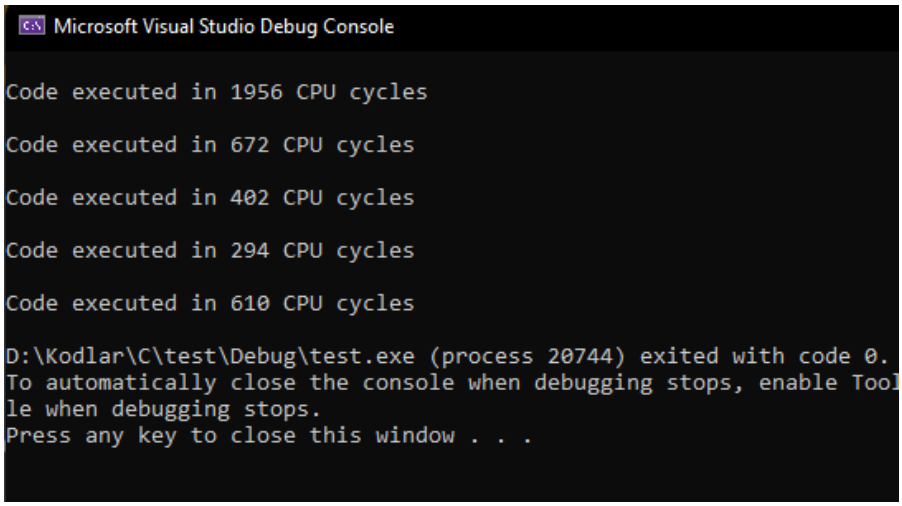
c-) 3rd attempt



d-) 4th attempt



e-) 5th attempt



5.Cache Memory

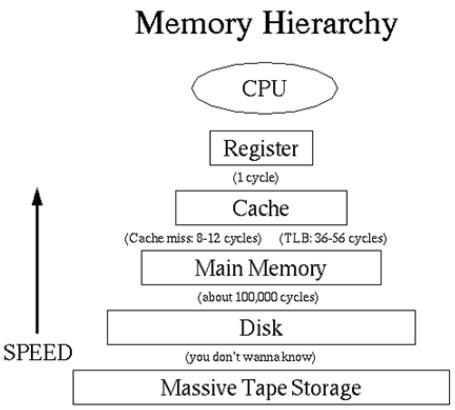
Cache memory is a small-sized type of volatile computer memory that provides high-speed data access to a processor and stores frequently used computer programs, applications and data.

A temporary storage of memory, cache makes data retrieving easier and more efficient. It is the fastest memory in a computer, and is typically integrated onto the motherboard and directly embedded in the processor or main random access memory (RAM).

Cache memory is a chip-based computer component that makes retrieving data from the computer's memory more efficient. It acts as a temporary storage area that the computer's processor can retrieve data from easily. This temporary storage area, known as a cache, is more readily available to the processor than the computer's main memory source.

Cache memory is sometimes called CPU (central processing unit) memory because it is typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU. Therefore, it is more accessible to the processor, and able to increase efficiency, because it's physically close to the processor.

In order to be close to the processor, cache memory needs to be much smaller than main memory. Consequently, it has less storage space. It is also more expensive than main memory, as it is a more complex chip that yields higher performance.



What it sacrifices in size and price, it makes up for in speed. Cache memory operates between 10 to 100 times faster than RAM, requiring only a few nanoseconds to respond to a CPU request.

Cache memory has levels; L1 Cache, L2 Cache, L3 Cache.

L1 cache, or primary cache, is extremely fast but relatively small, and is usually embedded in the processor chip as CPU cache.

L2 cache, or secondary cache, is often more capacious than L1. L2 cache may be embedded on the CPU, or it can be on a separate chip or coprocessor and have a high-speed alternative system bus connecting the cache and CPU. That way it doesn't get slowed by traffic on the main system bus.

Level 3 (L3) cache is specialized memory developed to improve the performance of L1 and L2. L1 or L2 can be significantly faster than L3, though L3 is usually double the speed of DRAM. With multicore processors, each core can have dedicated L1 and L2 cache, but they can share an L3 cache. If an L3 cache references an instruction, it is usually elevated to a higher level of cache.

6.SOURCES

1-) docs.oracle.com

2-) cs.virginia.edu

3-) eecg.utoronto.ca

4-) techtarget.com

5-) stackoverflow.com

6-) Lecture Slides(moodle.ipg.pt/course/view.php?id=8539)