ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

INSTITUTO POLITÉCNICO DA GUARDA

PROJECT 1

USING SIMD INSTRUCTIONS AND PURE ASSEMBLY, MAKE A FUNCTION TO ADD EFFICIENTLY ANY SIZE OF INTEGER VECTORS

Curso	**Engenharia Informática**
Unidade Curricular	Arquitetura de Computadores
Ano Letivo	2020/2021
Docente	Luis Figueiredo
Coordenador da área disciplinar	Fernando Melo Rodrigues
Data	23-05-2021
Alunos	Tiago Silva 1009268

1. PROJECT – effective sum

1.1 Simplified Algorithm

Given 2 groups of aligned integers.

The addition should be made using any available registers from ymm to eax accordingly.

num_of_elements = total amount of grouped integers (31).

```
for each group of 8
  sum using ymm registers
  decrease 8 from the num_of_elements every time an addition is made
end
on the remaining amount, for each group of 4
  sum using xmm registers
  decrease 4 from the num_of_elements every time an addition is made
end
on the remaining amount, for each group of 2
  sum using mmx registers
  decrease 2 from the num_of_elements every time an addition is made
end
on the remaining amount (if any)
  sum using eax registers
end
```

1.2.1 C CODE

```c
#include "stdio.h";
#include "conio.h";

/*
Project 1:
Using SIMD instructions and pure assembly, make a function to add efficiently
any size of integer vectors
*/

//indicate that we have an external function
extern "C" {
  void effective_sum(int*, int*, int*, int);
}

const int NUMBER_OF_ELEMENTS = 31;
const int alignment = 32; // 32 alignment for YMM 256 bit usage

int main() {

  // 31 characters array
  //  8 executed on YMM (3 times) | 24 added
  //  4 executed on XMM (1 time)  |  4 added
  //  2 executed on MMX (1 time)  |  2 added
  //  1 executed on EAX (1 time)  |  1 added

  __declspec(align(alignment))
    int source1[NUMBER_OF_ELEMENTS] = {
       1,  2,  3,  4,  5,  6,  7,  8,
       9, 10, 11, 12, 13, 14, 15, 16,
      17, 18, 19, 20, 21, 22, 23, 24,
      25, 26, 27, 28, 29, 30, 31
    };
  __declspec(align(alignment))
    int source2[NUMBER_OF_ELEMENTS] = {
       1,  2,  3,  4,  5,  6,  7,  8,
       9, 10, 11, 12, 13, 14, 15, 16,
      17, 18, 19, 20, 21, 22, 23, 24,
      25, 26, 27, 28, 29, 30, 31
    };
  __declspec(align(alignment))
    int destination[NUMBER_OF_ELEMENTS] = {
      0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0
    };
```

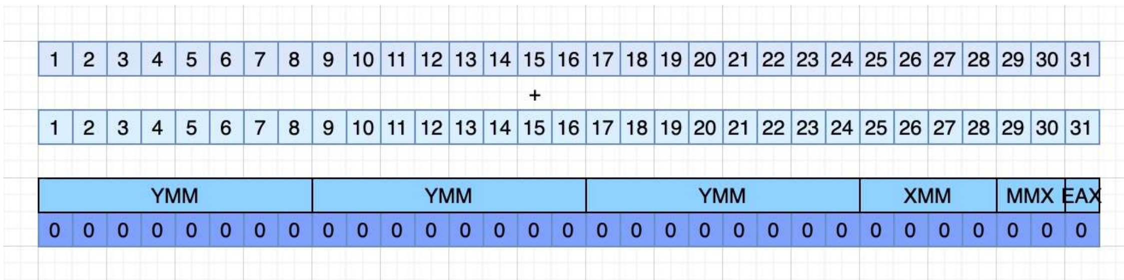
```
int i;

  _asm {
    mov eax, NUMBER_OF_ELEMENTS
    push eax                    ; push the value of NUMBER_OF_ELEMENTS to the stack
    lea eax, destination        ; load effective address of destination and put on eax
    push eax                    ; push the address of 'destination' to the stack
    lea eax, source2            ; load effective address of source2 put on eax
    push eax                    ; push the address of 'source2' to the stack
    lea eax, source1            ; load effective address of source1 put on eax
    push eax                    ; push the address of 'source1' to the stack
    call effective_sum
    add esp, 16                 ; set esp back to where it was before we use the stack
  }

  //Validation
  for (i = 0; i < NUMBER_OF_ELEMENTS; i++) {
    printf("source1[%d] + source2[%d] -> (%d + %d) = %d\n",
        i, i, source1[i], source2[i], destination[i]
    );
  };

  _getch();
  return 0;
}
```

1.2.2 ASSEMBLY (ASM FILE) CODE

```
.MODEL FLAT, C; USE THE FLAT MEMORY MODEL. USE C CALLING CONVENTIONS
.CODE; INDICATES THE START OF THE CODE SEGMENT.
PUBLIC effective_sum
  effective_sum PROC; FUNCTION NAME
  push ebp                  ; first instruction in the function
  mov ebp, esp
  push esi
  push edi
  push eax
  push ebx
  push ecx
  push edx

  ; MY CODE BLOCK

  ; STACK
  ; ESP -> | EBP                          |
  ;        | EDI                          |
  ;        | POINTER TO SOURCE1           | EBP + 8
  ;        | POINTER TO SOURCE2           | EBP + 12
  ;        | POINTER TO DESTINATION       | EBP + 16
  ;        | NUMBER OF ELEMENTS           | EBP + 20

  mov esi, [ebp + 8]        ; source 1
  mov edi, [ebp + 12]       ; source 2
  mov ecx, [ebp + 16]       ; destination
  mov edx, [ebp + 20]       ; number of elements

  ; -------- YMM --------
  goe_init:                 ; groups_of_eight_initialization
    mov ebx, edx            ; current loop limit
    shr ebx, 3              ; total divided by 8 -> number of executions of ymm registers
    je gof_init             ; if zero, goes to groups_of_four_initialization
  goe:                      ; groups_of_eight
    vmovapd ymm0, [esi]     ; move source1 into ymm0            | Move Aligned Packed Integer Values
    vmovapd ymm1, [edi]     ; move source2 into ymm1            | Move Aligned Packed Integer Values
    vaddpd ymm2, ymm0, ymm1 ; sum ymm0 + ymm1 and put it in ymm2 | Add Packed Integers
    vmovapd [ecx], ymm2     ; move ymm2(result) into destination | Move Aligned Packed Integer Values
    add esi, 32             ; advance source1 pointer 8 numbers ahead     | (8 x 4 = 32 byte)
    add edi, 32             ; advance source2 pointer 8 numbers ahead     | (8 x 4 = 32 byte)
    add ecx, 32             ; advance destination pointer 8 numbers ahead | (8 x 4 = 32 byte)
    sub edx, 8              ; remaining number of elements is reduced by 8
    dec ebx
    jnz goe                 ; if not zero, goes back to groups_of_eight
```

```
; -------- XMM --------
gof_init:                   ; groups_of_four_initialization
  mov ebx, edx              ; current loop limit
  shr ebx, 2                ; total divided by 4 -> number of executions of xmm registers
  jz got_init               ; if zero, goes to groups_of_two_initialization
gof:                        ; groups_of_four
  movdqa xmm0, [esi]        ; move source1 into xmm0            | Move Aligned Packed Integer Values
  paddd xmm0, [edi]         ; add source2 into xmm0             | Add Packed Integers
  movdqa [ecx], xmm0        ; move xmm0(result) to destination | Move Aligned Packed Integer Values
  add esi, 16               ; advance source1 pointer 4 numbers ahead     | (4 x 4 = 16 byte)
  add edi, 16               ; advance source2 pointer 4 numbers ahead     | (4 x 4 = 16 byte)
  add ecx, 16               ; advance destination pointer 4 numbers ahead | (4 x 4 = 16 byte)
  sub edx, 4                ; remaining number of elements is reduced by 4
  dec ebx
  jnz gof                   ; if not zero, goes back to groups_of_four

; -------- MMX--------
got_init:                   ; groups_of_two_initialization
  mov ebx, edx              ; current loop limit
  shr ebx, 1                ; total divided by 2 -> number of executions of mmx registers
  jz goo_init               ; if zero, goes to groups_of_one_initialization
got:                        ; groups_of_two
  movq mm0, [esi]           ; move source1 into mm0            | Move quadword (64bit)
  paddd mm0, [edi]          ; add source2 into mm0             | Add Packed Integers
  movq [ecx], mm0           ; move mm0(result) to destination | Move quadword (64bit)
  add esi, 8                ; advance source1 pointer 2 numbers ahead | (2 x 4 = 8 byte)
  add edi, 8                ; advance source2 pointer 2 numbers ahead | (2 x 4 = 8 byte)
  add ecx, 8                ; advance destination pointer 2 numbers ahead
  sub edx, 2                ; remaining number of elements is reduced by 2
  dec ebx
  jnz got                   ; if not zero, goes back to groups_of_two

; -------- EAX --------
goo_init:                   ; groups_of_one_initialization
  mov ebx, edx              ; current loop limit
  cmp ebx, 0
  jz the_end                ; if zero, goes to the end
goo:                        ; groups_of_one
  mov eax, [esi]            ; move source1 into eax            | Move int (32bit)
  add eax, [edi]            ; add source2 into eax             | Add
  mov [ecx], eax            ; move eax into destination        | Move int (32bit)
the_end:
  pop edx
  pop ecx
  pop ebx
  pop eax
  pop edi
  pop esi
```

```
  pop ebp
  ret                          ; last instruction in the function
  effective_sum ENDP           ; end of the function. Any other function can be written after this line
END
```



2. CACHE

2.1 – **When, Why, How?**

Cache was introduced in April 1965 by Maurice Wilkes and was called “slave memory”.
The principle is to store in a fast access memory (automatically) the content of a slow access memory so the first one can “serve” the user on future requests to the data.
(https://www.cs.auckland.ac.nz/courses/compsci703s1c/resources/Wilkes.pdf)

The time spent on requesting recurrent data repeatedly was heavy and slow and the need for cache memory was a must.

Among the different algorithms surrounding cache we can highlight the rewriting of the oldest stored data (1) and data stored by a scheduling date (2).
(1) – in a similar way as today’s recording dash cameras, there’s a loop that rewrites the oldest data with the newest one, so the length of data available is always the newest until X amount of data ago.

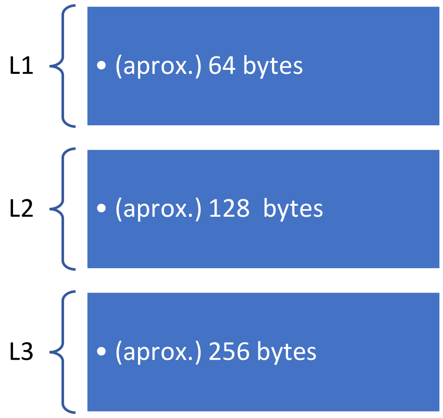
(2) – for the scheduling algorithm, we expect an expiry date for that data and as so new data should be fed in once the expiration date is reached.

Cache blocks

The basic unit for cache storage, it may contain multiple bytes/words of data.
Cache has a static full size; therefore, it will be able to store less bigger blocks or more smaller blocks.

number of blocks = cache size / block size

When the blocks are big, the time spent to load the data is *longer* and the amount of *cache misses* also increases.
When the blocks are too small, we won’t get much benefit from the cache as we’ll have to get a lot more data more frequently from the memory.



How is the cache organized and how we calculate each part?

ADDRESS		
TAG	INDEX	OFFSET

OFFSET: Defines the byte that we want within the blocks

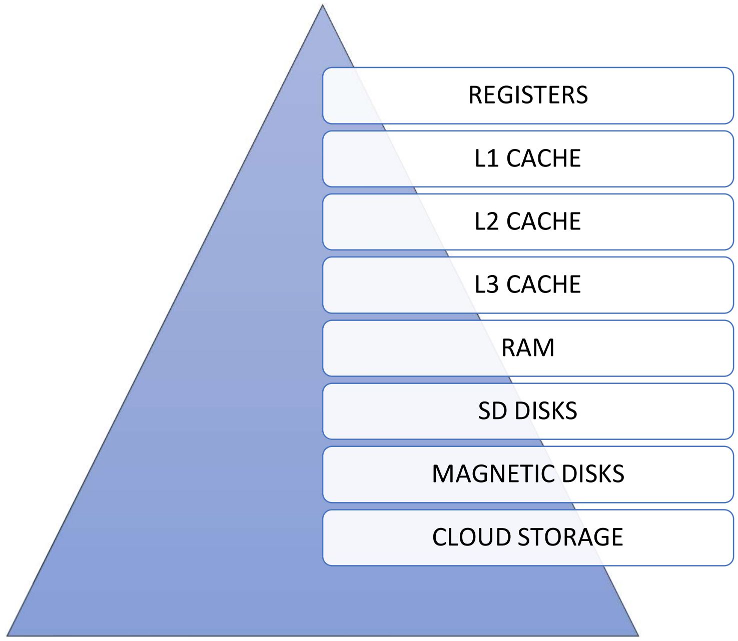
INDEX: Defines the block that we are on the cache

TAG: Defines the required matching bits on the cache

```
Size of blocks     = 2^offset bits
Number of blocks   = 2^index bits
TAG                = index bits – offset bits
```

Cache size: block size x amount of blocks

2.2 Hierarchy of Memory:



2.3 Principle of 10/90

The principal states that 90% of the time is used by 10% of the code.
This includes loops, if statements and all the instructions that on their own end up consuming resources.

2.4 Principle of continuality

When a program accesses a specific byte in an address, it’s highly probable to access nearby addresses in a close future.

2.5 Principle of temporality

When a program accesses a specific byte in an, it is highly probable to access that address again in a close future.

Conclusion:

The project was very exciting, and I have learnt a lot with it.
I look forward to extending it by giving the ability to use ZMM registers and the dynamic memory allocation.
Unfortunately, my investigation didn't give me an answer on the usage of ZMM which is why they are missing.

I want to thank the teacher for all the help and support ever since the start, my effort alone couldn't have made it this far without him.

References:

PDF from classes (main source of information)

https://www.cs.auckland.ac.nz/courses/compsci703s1c/resources/Wilkes.pdf

https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec16.pdf