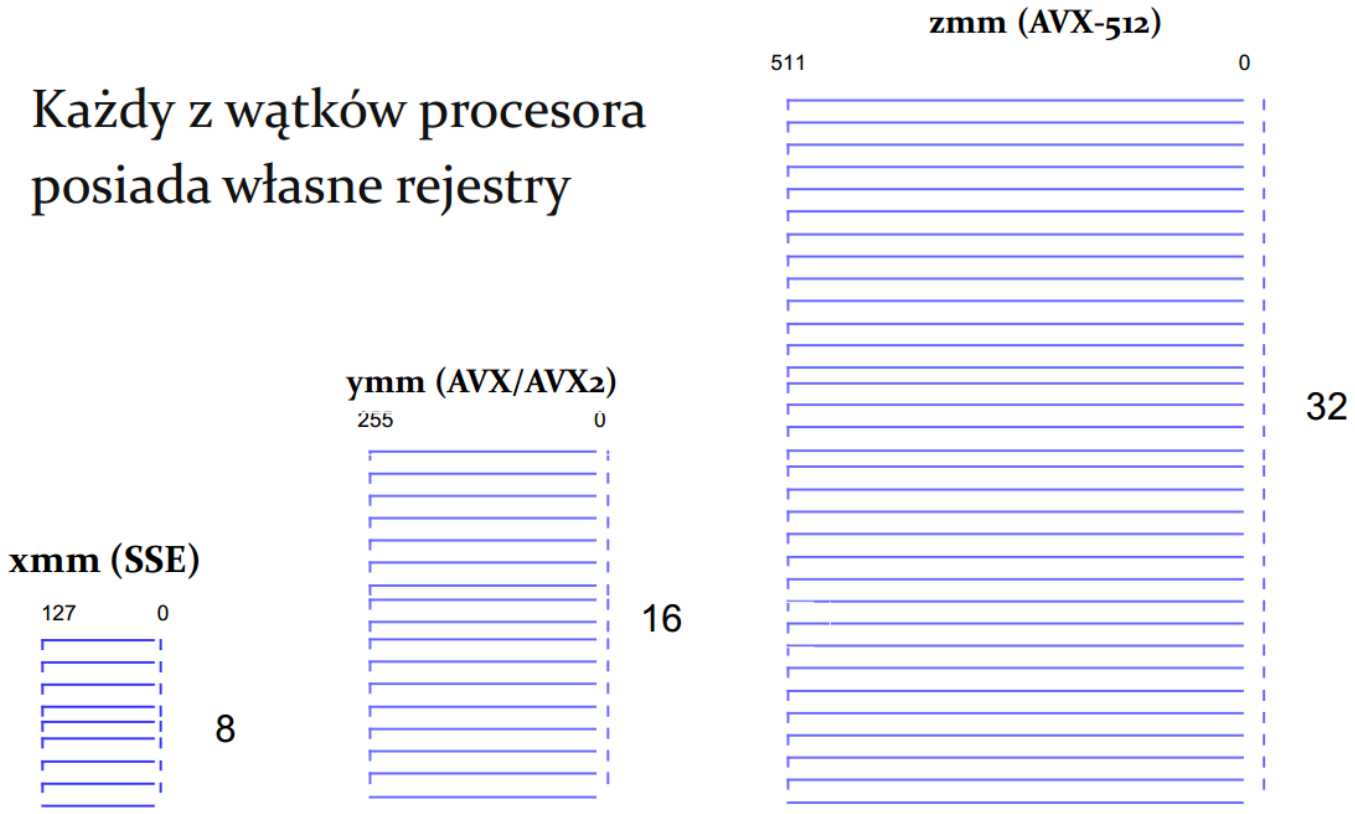
Laboratorium programowania niskopoziomowego

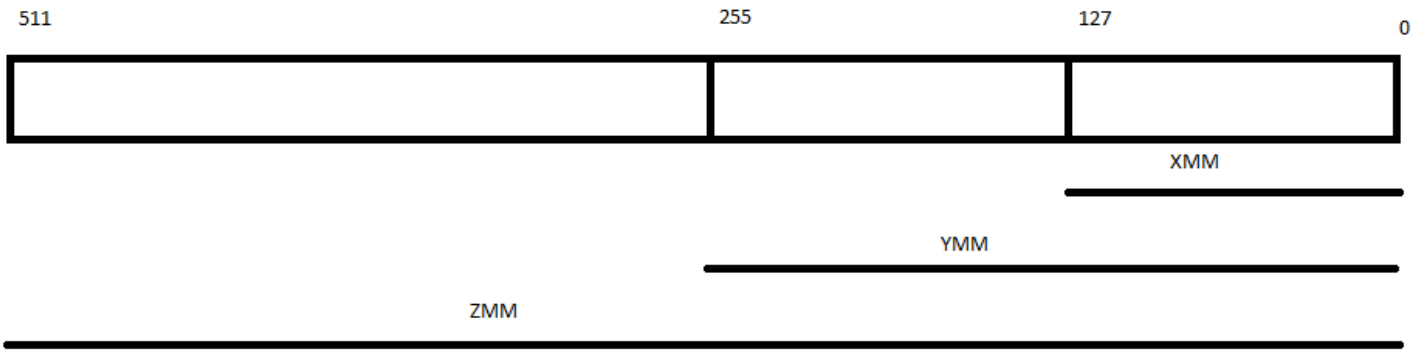
LAB 13 – Podstawowe operacje przy użyciu instrukcji wektorowych AVX.

AVX O P I S

Operacje na wektorach to operacje na kilku różnych danych (tego samego typu) wykonywane przez jedną instrukcję, dzięki którym można znacząco przyśpieszyć wykonywanie obliczeń. Aktualnie większość produkowanych procesorów posiada wbudowane instrukcje wektorowe. Na laboratoriach będziemy używać instrukcji wektorowych AVX/AVX2 pracujących na rejestrach YMM lub rzadziej XMM. Poniżej przypomnienie informacji z wykładu:



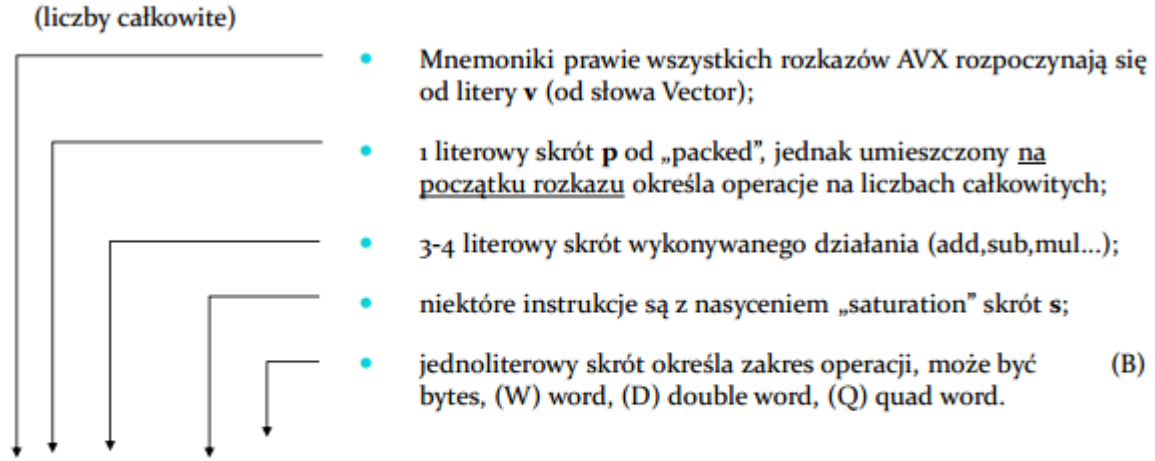
Należy tutaj wyraźnie zaznaczyć, że rejestry XMM są częścią rejestrów YMM, a te znowu są częścią rejestrów ZMM, dlatego **nie** należy mylić, że są to osobne rejestry, które można używać w jednym czasie. Poniżej przedstawiono schemat obrazujący te zależności (jest on analogiczny do rejestrów procesora AX/EAX/RAX):



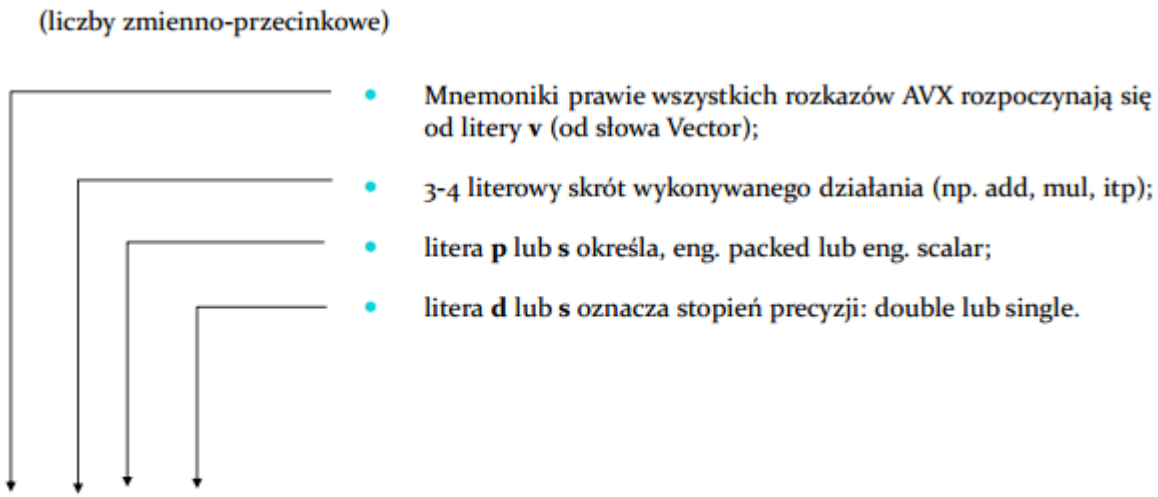
Wśród rozkazów typu AVX wyróżniamy grupy AVX, AVX2, FMA:

- AVX operuje na rejestrach ymm oraz xmm
- AVX2 wyłącznie na rejestrach ymm
- FMA na rejestrach ymm
- Instrukcje szyfrujące algorytmem AES na rejestrach xmm

Poniżej przedstawiono sposób budowania rozkazów AVX (przypomnienie z wykładu):



Rozkaz **vpaddb** wykonuje dodawanie (add) wektorowo/równolegle (p) liczb całkowitych w zakresie 8 bitów (b) i ewentualnie z nasyceniem



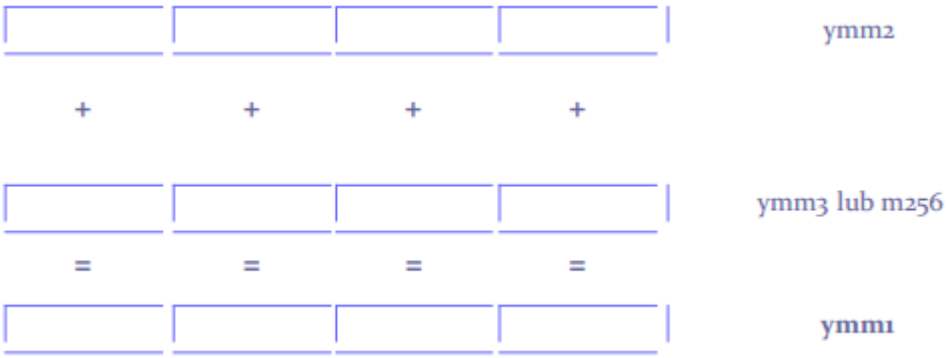
Rozkaz **vaddpd** wykonuje dodawanie (add) równoległe/spakowane (p) liczb zmiennoprzecinkowych podwójnej precyzji (d)

Poniżej znajduje się schematyczny przykład dodawania wektorów o poszczególnych typach:
INT64 (64bitowe liczby stałoprzecinkowe):

cel = źródło1 + źródło2

xmm1 = xmm2 + xmm3/m128

ymm1 = ymm2 + ymm3/m256



Oraz kod w obrazujący dodawanie tych wektorów:

```
vpaddq ymm1, ymm2, ymm3;
```

Przykładowy kod obrazujący sumowanie wektorów liczb typu int (liczby 32-bitowe)

```
void vec_avx_add_int(int* t1, int* t2, int* t3, int n)
{
    __asm
    {
        push esi;
        push edi;
        mov ecx, n;
        shl ecx, 2;
        mov esi, t1;
        mov edx, t2;
        mov edi, t3;

    petla:
        sub ecx, 32;
        vmovdqu ymm0, ymmword ptr[esi + ecx];
        vmovdqu ymm1, ymmword ptr[edx + ecx];
        vpaddd ymm2, ymm1, ymm0;
        vmovdqu ymmword ptr[edi + ecx], ymm2;
        jnz petla;
        pop edi;
        pop esi;
    }
}
```

Zad1: Proszę napisać program wykorzystujący powyższą funkcję i sprawdzić jego działanie dla różnych długości tablicy. Co się stanie jak podamy długość tablicy: 8, 16, 25, 32, 64, 100, 256?

Zad2: Proszę napisać program działający na wektorach liczb całkowitych odejmujący dwa wektory z zapisem do trzeciego dla liczb 32-bitowych.

Zad3: Proszę napisać program działający na wektorach liczb całkowitych mnożący dwa wektory z zapisem do trzeciego dla liczb 32-bitowych.

Zad4: Proszę napisać programy działające na wektorach liczb całkowitych dodające/odejmujące/mnożące dwa wektory z zapisem do trzeciego dla liczb typu INT64.

Obsługa **macierzy** liczb całkowitych przez instrukcje wektorowe AVX

Z punktu widzenia samych instrukcji nie zmienia się nic! Cała obsługa macierzy następuje przy użyciu procesora i rejestrów całkowitych procesora.

Poniżej przykład obsługi dodawania dwóch macierzy liczb całkowitych z zapisem do trzeciej na danych 32-bitowych w kodzie inline dla architektury x86:

```
void mtx2_avx_add_int(int** t1, int** t2, int** t3, int n, int m)
{
    __asm
    {
        push esi;
        push edi;
        mov eax, n;

    petlaN:
        mov esi, t1;
        mov esi, dword ptr[esi + eax * 4 - 4];
        mov edx, t2;
        mov edx, dword ptr[edx + eax * 4 - 4];
        mov edi, t3;
        mov edi, dword ptr[edi + eax * 4 - 4];

        mov ecx, m;
        shl ecx, 2;
    petlaM:
        sub ecx, 32;
        vmovdqu ymm0, ymmword ptr[esi + ecx];
        vmovdqu ymm1, ymmword ptr[edx + ecx];
        vpaddd ymm2, ymm1, ymm0;
        vmovdqu ymmword ptr[edi + ecx], ymm2;
        jnz petlaM;

        dec eax;
        jnz petlaN;
        pop edi;
        pop esi;
    }
}
```

Zad5: Proszę napisać program wykorzystujący powyższą funkcję i sprawdzić jego działanie dla różnych długości tablic. Co się stanie jak podamy długość tablicy jako pierwszy i/lub drugi wymiar 8, 16, 25, 32, 64, 100, 256?

Zad6: Proszę napisać program działający na macierzach stałoprzecinkowych odejmujący dwie macierze z zapisem do trzeciej dla liczb 32bitowych.

Zad7: Proszę napisać program działający na macierzach stałoprzecinkowych mnożący dwie macierze z zapisem do trzeciej dla liczb 32bitowych.

Zad8: Proszę napisać programy działające na macierzach stałoprzecinkowych dodające/odejmujące/mnożące dwie macierze z zapisem do trzeciej dla liczb typu INT64.

Zad9: Sprawdzić i przetestować działanie wektorów i macierzy na architekturze X64, dla różnych typów danych: 1/2/4/8 bajtowych danych

Liczby zmiennoprzecinkowe

Operacje na liczbach całkowitych i zmiennoprzecinkowych nie różnią się aż tak bardzo w kontekście podstawowych operacji arytmetycznych takich jak dodawanie, odejmowanie czy mnożenie.

Obsługa tablic, czy macierzy dla wszystkich typów danych odbywa się przez procesor.

Odczytywanie i zapisywanie danych jest takie samo dla liczb całkowitych i zmiennoprzecinkowych

Jedyna różnica jest w samej arytmetyce, a więc inaczej należy traktować dane typu INT, a inaczej typu FLOAT podczas np. dodawania, stąd instrukcje wykonujące dodawanie będą inaczej się nazywały (na poprzednich stronach tego laboratorium jest przedstawiona budowa rozkazów całkowitych i zmiennoprzecinkowych liczb), bo muszą inaczej interpretować zapisane bity w rejestrach wektorowych.

Poniżej przedstawiono odejmowanie wektorów zmiennoprzecinkowych

```
void vec_avx_sub_float(float* t1, float* t2, float* t3, int n)
{
    __asm
    {
        push esi;
        push edi;
        mov ecx, n;
        shl ecx, 2;
        mov esi, t1;
        mov edx, t2;
        mov edi, t3;
```

```
    petla:
        sub ecx, 32;
        vmovups ymm0, ymmword ptr[esi + ecx];
        vmovups ymm1, ymmword ptr[edx + ecx];
        vsubps ymm2, ymm1, ymm0;
        vmovups ymmword ptr[edi + ecx], ymm2;
        jnz petla;
        pop edi;
        pop esi;
    }
}
```

oraz odejmowanie macierzy zmiennoprzecinkowych

```
void mtx2_avx_sub_float(float** t1, float** t2, float** t3, int n, int m)
{
    __asm
    {
        push esi;
        push edi;
        mov eax, n;

    petlaN:
        mov esi, t1;
        mov esi, dword ptr[esi + eax * 4 - 4];
        mov edx, t2;
        mov edx, dword ptr[edx + eax * 4 - 4];
        mov edi, t3;
        mov edi, dword ptr[edi + eax * 4 - 4];

        mov ecx, m;
        shl ecx, 2;
    petlaM:
        sub ecx, 32;
        vmovups ymm0, ymmword ptr[esi + ecx];
        vmovups ymm1, ymmword ptr[edx + ecx];
        vsubps ymm2, ymm1, ymm0;
        vmovups ymmword ptr[edi + ecx], ymm2;
        jnz petlaM;

        dec eax;
        jnz petlaN;
        pop edi;
        pop esi;
    }
}
```

Zadania do samodzielnej realizacji:

Wektory

Zad10: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzielenie

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech wektorów (A, B, C) typu **float** o długości co najmniej 1000 elementów w architekturze **X86**.

Zad11: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzieleni

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech wektorów (A, B, C) typu **double** o długości co najmniej 1000 elementów w architekturze **X86**.

Zad12: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzielenie

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech wektorów (A, B, C) typu **float** o długości co najmniej 1000 elementów w architekturze **X64**.

Zad13: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzieleni

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech wektorów (A, B, C) typu **double** o długości co najmniej 1000 elementów w architekturze **X64**.

Macierze

Zad14: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzielenie

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech macierzy (A, B, C) typu **float** o długości co najmniej 1000x1000 elementów w architekturze **X86**.

Zad15: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzielenie

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech macierzy (A, B, C) typu **double** o długości co najmniej 1000x1000 elementów w architekturze **X86**.

Zad16: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzielenie

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech macierzy (A, B, C) typu **float** o długości co najmniej 1000x1000 elementów w architekturze **X64**.

Zad17: Wykonaj:

- Dodawanie
- Odejmowanie
- Mnożenie
- Dzielenie

Danych przy użyciu instrukcji wektorowych na rejestrach YMM dla trzech macierzy (A, B, C) typu **double** o długości co najmniej 1000x1000 elementów w architekturze **X64**.