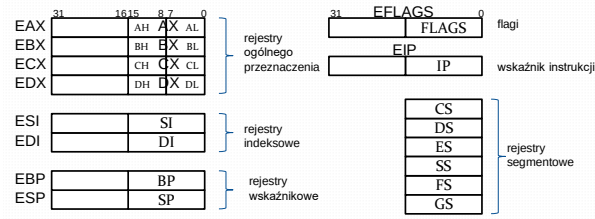
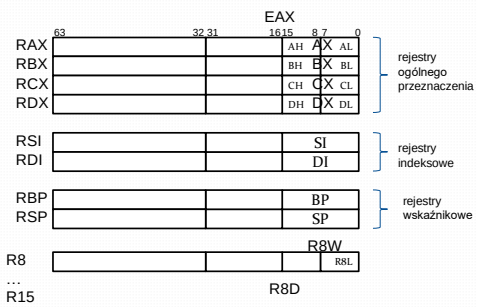
Instrukcje logiczne, przesunięć i rotacji

IA32- rejestry



EM64T- rejestry



Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

Instrukcje logiczne

- AND bitowa funkcja AND
- ANDN bitowa funkcja AND z negacją
- OR bitowa funkcja OR
- XOR bitowa funkcja OR
- NOT bitowa funkcja NOT

Wpływa na flagi: OSZAPC
0SZxP0

Instrukcja AND

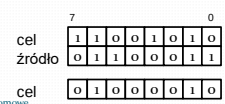
and cel, źródło

Wyznacza iloczyn logiczny zawartości źródła i celu (bit po bicie), wynik umieszcza w miejscu przeznaczenia (cel).

cel := cel and źródło

```
and eax, zmienna
and edx, [ebx+esi*4]
and rax, rdx
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.



Wpływa na flagi: OSZAPC
0SZxx0
Wymagane BMI1

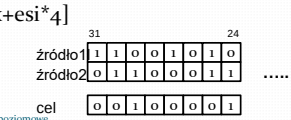
Instrukcja ANDN

andn cel, źródło1, źródło2

Wyznacza iloczyn logiczny zanegowanego źródła1 i źródła2 (bit po bicie), wynik umieszcza w miejscu przeznaczenia (cel). Cel i źródło1 są rejestrami 32|64 bitowymi.

cel := (not źródło1) and źródło2

```
andn  edx, eax, zmienna
andn  eax, edx, [ebx+esi*4]
andn  r8, rax, rdx
```



Wpływa na flagi: OSZAPC
0SZxP0

Instrukcja OR

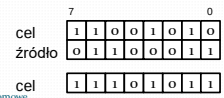
or cel, źródło

Wyznacza sumę logiczną zawartości źródła i celu (bit po bicie), wynik umieszcza w miejscu przeznaczenia (cel).

cel := cel or źródło

```
or    eax, zmienna
or    edx, [ebx+esi*4]
or    rax, r9
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.



Wpływa na flagi: OSZAPC
0SZxP0

Instrukcja XOR

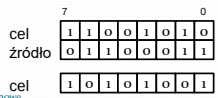
xor cel, źródło

Wyznacza, bit po bicie, sumę modulo 2 zawartości źródła i celu wynik umieszcza w miejscu przeznaczenia (cel).

cel := cel xor źródło

```
xor   eax, zmienna
xor   edx, [ebx+esi*4]
xor   rax, r9
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.



Wpływa na flagi: -

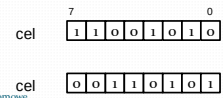
Instrukcja NOT

not cel

Wyznacza negację logiczną zawartości i celu (bit po bicie), wynik umieszcza w miejscu przeznaczenia (cel).

cel := not cel

```
not   eax
not   byte ptr [ebx+esi*4]
not   rdx
```



If ((!a&&b)||(c&&d&&e)) {...}else {...};

```
        mov   al, a          mov   al, a          mov   dl, c
        not   al             andn  al, al, b      mov   al, a
        and   al, b          jnz   then_1         and   dl, d
        mov   dl, c          mov   dl, c          andn  al, al, b
        and   dl, d          and   dl, d          and   dl, e
        and   dl, e          and   dl, e          or    al, dl
        or    al, dl         or    al, dl         jz    else_1
        jz    else_1         jz    else_1   then_1:
then_1:              then_1:
```


Instrukcje przesunięć i rotacji

- SAR przesunięcie arytmetyczne w prawo
- SHR przesunięcie logiczne w prawo
- SAL przesunięcie arytmetyczne w lewo
- SHL przesunięcie logiczne w lewo
- SARX przesunięcie arytmetyczne w prawo
- SHRX przesunięcie logiczne w prawo
- SHLX przesunięcie logiczne w lewo
- SHRD przesunięcie w prawo double
- SHLD przesunięcie w lewo double
- ROR rotacja w prawo
- ROL rotacja w lewo
- RCR rotacja w prawo przez przeniesienie
- RCL rotacja w lewo przez przeniesienie
- RORX rotacja w prawo

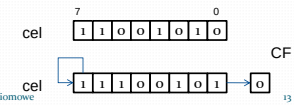
Wpływa na flagi: OSZAPC
(0x)SZxPC

Instrukcja SAR

sar cel, ile

Przesunięcie arytmetyczne celu w prawo o ile bitów. Ile=1, cl lub wartość 0-31|63 .

```
sar   eax, 1
sar   [ebx+esi*4], cl
sar   rdx, cl
```



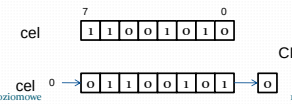
Wpływa na flagi: OSZAPC
(0x)SZxPC

Instrukcja SHR

shr cel, ile

Przesunięcie logiczne w prawo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
shr   eax, 1
shr   [ebx+esi*4], cl
shr   rdx, cl
```



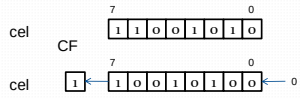
Wpływa na flagi: OSZAPC
(0x)SZxPC

Instrukcja SAL

sal cel, ile

Przesunięcie arytmetyczne celu w lewo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
sal   eax, 1
sal   [ebx+esi*4], cl
sal   rdx, cl
```



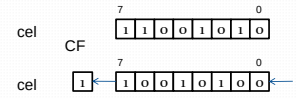
Wpływa na flagi: OSZAPC
(0x)SZxPC

Instrukcja SHL

shl cel, ile

Przesunięcie logiczne celu w lewo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
shl   eax, 1
shl   [ebx+esi*4], cl
shl   rdx, cl
```



Przykład – oblicz sumę krawędzi prostopadłościanu

```
mov   eax, a          ;wczytaj a
add   eax, b          ;dodaj b
add   eax, c          ;dodaj c
sal   eax, 2          ; ×4
```


Wpływa na flagi: -----
Wymaga BMI2

Instrukcja SARX

sarx cel, źródło, ile

Przesunięcie arytmetyczne źródła w prawo o ile bitów i zapisanie w celu. Cel i ile są rejestrami 32|64 bitowymi .

```
sarx   eax, zmienna, edx
sarx   eax, [ebx+esi*4], ecx
sarx   rdx, rax, rcx
```

cel

Wpływa na flagi: ------
Wymaga BMI2

Instrukcja SHRX

shrx cel, źródło, ile

Przesunięcie logiczne źródła w prawo o ile bitów i zapisanie w celu. Cel i ile są rejestrami 32|64 bitowymi .

```
shrx   eax, zmienna, edx
shrx   eax, [ebx+esi*4], ecx
shrx   rdx, rax, rcx
```


Wpływa na flagi: ------
Wymaga BMI2

Instrukcja SHLX

shlx cel, źródło, ile

Przesunięcie logiczne źródła w lewo ile bitów i zapisanie w celu. Cel i ile są rejestrami 32|64 bitowymi .

```
shlx   eax, zmienna, edx
shlx   eax, [ebx+esi*4], ecx
shlx   rdx, rax, rcx
```

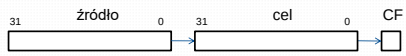

Wpływa na flagi: OSZAPC
(Ox)SZxPC

Instrukcja SHRD

shrd cel, źródło, ile

Przesunięcie źródła:celu w prawo o ile bitów. Ile=cl lub wartość 0-31|64. Rejestr źródła (16,32,64) pozostaje bez zmian.

```
shrd   eax, ecx, 15
shrd   [ebx+esi*4], edx, cl
shrd   zmienna, rax, cl
```



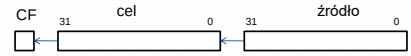
Wpływa na flagi: OSZAPC
(Ox)SZxPC

Instrukcja SHLD

shld cel, źródło, ile

Przesunięcie źródła:celu w lewo o ile bitów. Ile=cl lub wartość 0-31|63. Rejestr źródła (16,32,64) pozostaje bez zmian.

```
shld   eax, ecx, 15
shld   [ebx+esi*4], edx, cl
shld   zmienna, rax, cl
```



Przykład – podziel 256-bitową liczbę a przez 64

```
mov    rcx, offset a       ;wczytaj adres a
mov    rax, [rcx+8]        ;wczytaj drugie 64 bity
shrd   [rcx], rax, 6       ;przesuń pierwsze
mov    rax, [rcx+16]       ;wczytaj trzecie 64 bity
shrd   [rcx+8], rax, 6     ;przesuń drugie
mov    rax, [rcx+24]       ;wczytaj czwarte 64 bity
shrd   [rcx+16], rax, 6    ;przesuń trzecie
shr    rax, 6              ;przesuń czwarte
mov    [rcx+24], rax       ;zapisz
```

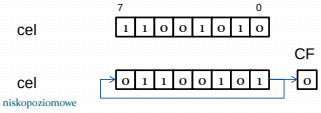

Wpływa na flagi: OSZAPC
(0x)----C

Instrukcja ROR

ror cel, ile

Rotacja (obrót) celu w prawo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
ror    eax, 1
ror    [ebx+esi*4], cl
ror    rdx, cl
```



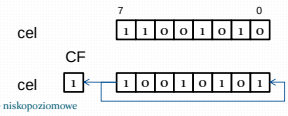
Wpływa na flagi: OSZAPC
(0x)----C

Instrukcja ROL

rol cel, ile

Rotacja (obrót) celu w lewo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
rol   eax, 1
rol   [ebx+esi*4], cl
rol   rdx, cl
```



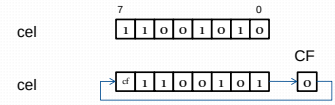
Wpływa na flagi: OSZAPC
(0x)----C

Instrukcja RCR

rcr cel, ile

Rotacja (obrót) przez przeniesienie celu w prawo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
rcr   eax, 1
rcr   [ebx+esi*4], cl
rcr   rdx, cl
```



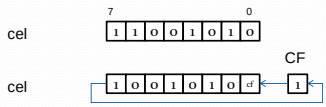
Wpływa na flagi: OSZAPC
(0x)----C

Instrukcja RCL

ror cel, ile

Rotacja (obrót) przez przeniesienie celu w lewo o ile bitów. Ile=1, cl lub wartość 0-31|63.

```
rcl   eax, 1
rcl   [ebx+esi*4], cl
rcl   rdx, cl
```



Wpływa na flagi: ------
Wymaga BMI2

Instrukcja RORX

rorx cel, źródło, ile

Rotacja źródła w prawo o ile bitów i zapisanie w celu. Cel jest rejestrem 32|64 bitowym. Ile przyjmuje wartość 0-31|63.

```
rorx  eax, zmienna, 12
rorx  eax, [ebx+esi*4], 7
rorx  rdx, rax, 44
```


Przykłady

```
and   eax, offffefffh     ;zeruje 12 bit eax

or    ax, 01000h          ;ustawia 12 bit ax

xor   rax, 01000h         ;neguje 12 bit rax

and   eax, eax            ;m. in. zeruje CF

xor   eax, eax            ;zeruje eax
```


Przykłady

```
and   eax, 070h     ;maska
sar   eax, 4        ;eax – 3bitowa liczba


mov   ah, al        ;zamienia al na jego
and   ax, 0f00fh    ; szesnastkową reprezentację
shr   ah, 4         ;ASCII w ah, al.
add   ax, 3030h
```

Przykłady - *4,25

```
movsxd    rax, zmienna      ;typu int – 32 bity
mov       rdx, rax          ;powiel
sal       rax, 2            ;x4
sar       rdx, 2            ;/4
adc       rax, rdx          ;x4¼
mov       zmienna, eax      ;zapisz

movsxd    rdx, eax          ; test
cmp       rax, rdx          ;porównaj
jnz       blad              ;przekroczenie zakresu
```


Przykłady

```
;odwracanie bitów z rcx; rdx liczba odwracanych bitów 1-64
invb    proc
        xor     rax, rax        ;rax:=0
        dec     rdx             ;eliminacja zera
        js      @e
        and     rdx, 3fh        ;ograniczenie zakresu
@p:     shr     rcx, 1          ;lsb->CF
        rcl     rax, 1          ;CF->msb
        dec     rdx             ;licznik--
        jns     @p              ;następny jeśli nieujemne
@e:     ret
invb    endp
```

Operacje na znacznikach, bitach i bajtach

Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

Operacje na flagach

- STC Ustawienie CF
- CLC Zerowanie CF
- CMC Zanegowanie CF
- CLD Zerowanie DF – flagi kierunku
- STD Ustawienie DF
- LAHF Przesłanie flag do rejestru AH
- SAHF Przesłanie rejestru AH do flag
- PUSHF/PUSHFD/ PUSHFQ Wysłanie flag na stos
- POPF/POPFD/POPFQ Pobranie flag ze stosu
- STI Ustawienie IF – flagi przerwań
- CLI Zerowanie IF

Wpływa na flagi: C

Instrukcja STC

```
stc
```

Ustawienie flagi CF.

```
CF := 1
```

```
stc
```


Wpływa na flagi: C

Instrukcja CLC

```
clc
```

Zerowanie flagi CF.

```
CF := 0
```

```
clc
```


Wpływa na flagi: C

Instrukcja CMC

```
cmc
```

Zanegowanie flagi CF.

```
CF := not CF
```

```
cmc
```

Wpływa na flagi: D

Instrukcja STD

std

Ustawienie flagi kierunku DF. Jeżeli DF=1 instrukcje łańcuchowe zmniejszają rejestr ESI lub EDI.

DF := 1

std

Wpływa na flagi: D

Instrukcja CLD

cld

Zerowanie flagi kierunku DF. Jeżeli DF=0 instrukcje łańcuchowe zwiększają rejestr ESI lub EDI.

DF := 0

cld

Wpływa na flagi: -

Instrukcja LAHF

lahf

Przesłanie flag do rejestru AH

AH := lo(FLAGS)

lahf

Wpływa na flagi: SZAPC

Instrukcja SAHF

sahf

Przesłanie rejestru AH do flag. Bity 1,3,5 są ignorowane.

lo(FLAGS) := AH

sahf

Wpływa na flagi: -

Instrukcja PUSHF/PUSHFD/PUSHFQ

pushf/pushfd/pushfq

Przesyła zawartość Flag/Eflag/Rflag na stos.

pushf

pushfd

Wpływa na flagi: OSZAPC

Instrukcja POPF/POPFD/POPFQ

popf/popfd/popfq

Pobiera zawartość Flag/EFlag ze stosu.

popf

popfd

Wpływa na flagi: I

Instrukcja STI

sti

Ustawienie flagi przerwań IF lub VIF. Włącza po następnej instrukcji system przerwań maskowalnych.

IF := 1

sti

Wpływa na flagi: I

Instrukcja CLI

cli

Zerowanie flagi przerwań IF lub VIF. Wyłącza system przerwań maskowalnych.

IF := 0

cli

Operacje na bitach

- BT Testowanie bitu
- BTS Testowanie bitu z ustawianiem
- BTR Testowanie bitu z zerowaniem
- BTC Testowanie bitu z negacją
- TEST Porównanie logiczne
- BSF Przeszukiwanie bitów w przód
- BSR Przeszukiwanie bitów wstecz
- LZCNT Zlicza zerowe bity od najstarszego
- TZCNT Zlicza zerowe bity od najmłodszego
- BEXTR Wycina ciąg bitów
- BLSI Kopiuje najmłodszy ustawiony bit
- BLSR Zeruje najmłodszy ustawiony bit
- BLSMSK Tworzy maskę do bitu=0
- BZHI Zeruje starsze bity

Wpływa na flagi: OSZAPC
xxxxxC

Instrukcja BT

bt baza, nr

Wyznacza wartość bitu nr (rejestr lub wartość) w bazie (rejestr lub zmienna) i umieszcza ją w CF.

CF := bit bazy numer nr

bt zmienna, eax
bt edx, 12
bt rcx, 37

Wpływa na flagi: OSZAPC
xxxxxC

Instrukcja BTS

bts baza, nr

Wyznacza wartość bitu nr (rejestr lub wartość) w bazie (rejestr lub zmienna) i umieszcza ją w CF. Następnie ustawia badany bit.

CF := bit bazy numer nr
bit bazy numer nr:=1

bts zmienna, eax
bts edx, 12
bts rcx, 37

Wpływa na flagi: OSZAPC
xxxxxC

Instrukcja BTR

btr baza, nr

Wyznacza wartość bitu nr (rejestr lub wartość) w bazie (rejestr lub zmienna) i umieszcza ją w CF. Następnie zeruje badany bit.

CF := bit bazy numer nr
bit bazy numer nr := 0

btr zmienna, eax
btr edx, 12
btr rcx, 37

Wpływa na flagi: OSZAPC
xxxxxC

Instrukcja BTC

btc baza, nr

Wyznacza wartość bitu nr (rejestr lub wartość) w bazie (rejestr lub zmienna) i umieszcza ją w CF. Następnie neguje badany bit.

CF := bit bazy numer nr

bit bazy numer nr := not bit bazy numer nr

```
btc   zmienna, eax
btc   edx,12
btc   rcx, 37
```


Wpływa na flagi: OSZAPC
0SZxP0

Instrukcja TEST

test cel, źródło

Wyznacza iloczyn logiczny(bit po bicie) zawartości celu i źródła (rejestr lub wartość), wynik jest pominięty, ustawia flagi.

cel and źródło

```
test   eax, zmienna
test   edx, [ebx+esi*4]
```


Wpływa na flagi: OSZAPC
xxZxxx

Instrukcja BSF

bsf cel, źródło

Przeszukiwanie bitów w przód. Szuka w rejestrze lub zmiennej źródła najmłodszego bitu=1, jego indeks umieszcza w rejestrze celu (ZF = 0). Jeśli źródło = 0, wówczas ZF = 1, a cel jest niezdefiniowany

cel := indeks najmłodszego bitu = 1 źródła

```
bsf   eax, zmienna
bsf   edx,esi
bsf   rcx, rdx
```


Wpływa na flagi: OSZAPC
xxZxxx

Instrukcja BSR

bsr cel, źródło

Przeszukiwanie bitów wstecz. Szuka w rejestrze lub zmiennej źródła najstarszego bitu=1, jego indeks umieszcza w rejestrze celu (ZF = 0). Jeśli źródło = 0, wówczas ZF = 1, a cel jest niezdefiniowany

cel :=indeks najstarszego bitu=1 źródła

```
bsr   eax, zmienna
bsr   edx, esi
bsr   rcx, rdx
```


Wpływa na flagi: OSZAPC
xxZxxC
Wymaga LZCNT

Instrukcja LZCNT

lzcnt cel, źródło

Zlicza starsze (wiodące) zerowe bity źródła (16|32|64) i ilość zapisuje do rejestru celu. Dla celu = 0 ZF = 1. Dla celu = rozmiarowi źródła CF = 1.

cel := liczba wiodących zer w źródle

```
lzcnt  eax, zmienna
lzcnt  edx, esi
lzcnt  rcx, rdx
```


Wpływa na flagi: OSZAPC
xxZxxC
Wymaga BMI1

Instrukcja TZCNT

tzcnt cel, źródło

Zlicza od najmłodszego zerowe bity źródła (16|32|64) i ilość zapisuje do rejestru celu. Dla celu = 0 ZF = 1. Dla celu=rozmiarowi źródła CF = 1.

cel := liczba końcowych zer w źródle

```
tzcnt  eax, zmienna
tzcnt  edx, esi
tzcnt  rcx, rdx
```

Wpływa na flagi: OSZAPC
0xZxx0
Wymaga BMI1

Instrukcja BEXTR

bextr cel, źródło, st_ile

Wycina z rejestru|zmiennej źródła (32|64) ciąg bitów i umieszcza w rejestrze celu. Początkowy bit określa rejestr st_ile[7:0], a ilość bitów st_ile[15:8]. Jeśli cel = 0, wówczas ZF = 1.

cel := źródło[start + ile - 1: start]

```
bextr  eax, zmienna, edx
bextr  edx, esi, eax
bextr  rcx, rdx, rax
```


Wpływa na flagi: OSZAPC
0SZxxC
Wymaga BMI1

Instrukcja BLSI

blsi cel, źródło

Izoluje z rejestru lub zmiennej źródła najmłodszy bit=1 i umieszcza w rejestrze celu (CF = 1). Zeruje pozostałe bity. Jeśli źródło = 0, wówczas CF = 0, a cel = 0.

cel := (-źródło) and źródło

```
blsi   eax, zmienna
blsi   edx, esi
blsi   rcx, rdx
```


Wpływa na flagi: OSZAPC
0SZxxC
Wymaga BMI1

Instrukcja BLSR

blsr cel, źródło

Kopiuje bity z rejestru lub zmiennej źródła (32|64) i umieszcza w rejestrze celu, zeruje najmłodszy bit = 1 (CF = 0). Jeśli źródło = 0, wówczas CF = 1, a cel = 0.

cel := (źródło - 1) and źródło

```
blsr   eax, zmienna
blsr   edx, esi
blsr   rcx, rdx
```


Wpływa na flagi: OSZAPC
0S0xxC
Wymaga BMI1

Instrukcja BLSMSK

blsmsk cel, źródło

Ustawia młodsze bity rejestru celu (32|64) na 1 aż do numeru najmłodszego bitu=1 z rejestru lub zmiennej źródła włącznie (CF = 0). Zeruje pozostałe bity. Jeśli źródło = 0, wówczas CF = 1, a cel = not 0.

cel := (źródło - 1) xor źródło

```
blsmsk   eax, zmienna
blsmsk   edx, esi
blsmsk   rcx, rdx
```


Wpływa na flagi: OSZAPC
0SZxxC
Wymaga BMI2

Instrukcja BZHI

bzhi cel, źródło, idx

Kopiuje bity z rejestru lub zmiennej źródła (32|64) do rejestru celu (32|64) i kasuje starsze bity od numeru z rejestru idx (CF = 0). Jeśli idx > 31|63, wówczas CF = 1.

cel := źródło; cel[rozmiar - 1: idx] = 0

```
bzhi   eax, zmienna, edx
bzhi   edx, esi, eax
bzhi   rcx, rdx, rax
```


Wpływa na flagi: -

Instrukcja SETcc

SETcc cel

Jeśli jest spełniony warunek cc, ustawia bajt na 1, w przeciwnym wypadku na 0.

if cc then cel := 1
else cel := 0

```
sets   al
setge  [esi+8]
```

Instrukcje SETcc

- SETE/SETZ — Ustaw bajt jeśli equal/ zero
- SETNE/SETNZ — Ustaw bajt jeśli not equal/ not zero
- SETS — Ustaw bajt jeśli sign (negative)
- SETNS — Ustaw bajt jeśli not sign (non-negative)
- SETO — Ustaw bajt jeśli overflow
- SETNO — Ustaw bajt jeśli not overflow
- SETPE/SETP — Ustaw bajt jeśli parity even/ parity
- SETPO/SETNP — Ustaw bajt jeśli parity odd/ not parity
- SETA/SETNBE — Ustaw bajt jeśli above/ not below or equal
- SETAE/SETNB/SETNC — Ustaw bajt jeśli above or equal/ not below/ not carry
- SETB/SETNAE/SETC — Ustaw bajt jeśli below/ not above or equal/ carry
- SETBE/SETNA — Ustaw bajt jeśli below or equal/ not above
- SETG/SETNLE — Ustaw bajt jeśli greater/ not less or equal
- SETGE/SETNL — Ustaw bajt jeśli greater or equal/ not less
- SETL/SETNGE — Ustaw bajt jeśli less/ not greater or equal
- SETLE/SETNG — Ustaw bajt jeśli less or equal/ not greater

Przykład – int na bin(string)

```
procedure s2b(var s:string;i:integer);
asm
    push ebx
    bsr edx,ecx     //w edx nr najstarszej 1
    jnz @1
    mov word ptr [eax],$3001
    jmp @e
@1: inc edx
    mov [eax],dl    //długość
    dec edx
    inc eax
@p: bt ecx,edx      //testuj
    setc bl
    add bl,$30
    mov [eax],bl    //zapisz znak
    inc eax
    dec edx
    jns @p
@e: pop ebx
end;
```

```
procedure s2b2(var s:string;i:integer);
asm
    bsr edx,ecx          //w edx nr najstarszej 1
    jnz @1
    mov word ptr [eax],$3001
    jmp @e
@1: mov [eax],dl         //długość
    inc [eax]
@p: inc eax
    bt ecx,edx           //testuj
    setc [eax]
    add byte ptr [eax],$30 //zapisz znak
    dec edx
    jns @p
@e:
end;
```


Wpływa na flagi: -

Instrukcja RDPID

rdpid cel

Czyta 32-bitowy identyfikator procesora do rejestru celu.

cel=PID

rdpid eax

rdpid rdx

Wpływa na flagi: -

Instrukcja RDTSC

rdtsc

Read Time Stamp Counter. Czyta 64-bitowy licznik do rejestrów EDX: EAX.

EDX: EAX := licznik

rdtsc

Przykład – funkcja pomiaru czasu

```
function Pomiar(a:integer):integer;
var Cykle_H,Cykle_L:integer;
asm
    rdtsc
    mov Cykle_H,edx
    mov Cykle_L,eax
    ...
    ...
    rdtsc
    sub eax,Cykle_L
    sbb edx,CykleH
    sub EAX,9       ; odliczenie 9?
end;
```


Przykład – funkcja random

```
function MyRandom0(a:integer):integer;overload;
asm
    push ebx
    xor ebx,ebx
    imul edx,[ebx+MySeed],$08088405
    inc edx
    mov [ebx+MySeed],edx
    mul edx
    mov eax,edx
    pop ebx
end;

function MyRandom1(a:integer):integer;overload;
asm
    push eax
    rdtsc
    bswap eax
    pop edx
    mul edx
    mov eax,edx
end;
```

```
function MyRandom2(a:integer):integer;overload;
asm
    push eax
    rdtsc
    ror al,3
    imul edx,MySeed,$08088405
    ror ah,1
    inc edx
    ror eax,1
    mov MySeed,edx
    bswap eax
    xor eax,edx
    pop edx
    mul edx
    mov eax,edx
end;
```

Wpływa na flagi: 00000C

Instrukcja RDRAND

rdrand cel

Czyta 16|32|64-bitową liczbę losową (deterministic random bit generator) do rejestru celu wg normy NIST SP 800-90A. Jeśli CF = 1 wartość jest prawidłowa.

rdrand rax

Wpływa na flagi: 00000C

Instrukcja RDSEED

rdseed cel

Czyta 32|64-bitową liczbę losową (non-deterministic random bit generator) do rejestru celu wg norm NIST SP 800-90B i NIST SP800-90C. Jeśli CF = 1 wartość jest prawidłowa.

rdseed rax

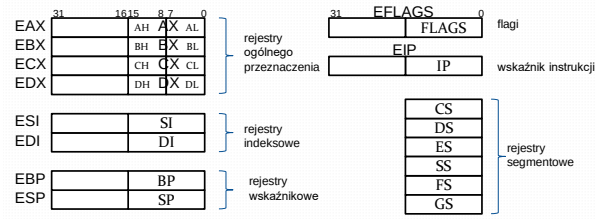
Przykład

Oblicz wartość Losową (64)

```
;rcx – wskaźnik do liczby int64
random proc;
        xor     rax, rax  ;nieudane
        mov     rdx, 10   ;liczba powtórzeń
@p:     rdseed  r8        ;czytaj liczbę
        jc      @ok
        dec     rdx       ;licznik--
        jnz     @pz
        ret
@ok:    mov     [rcx], r8 ;zapisz wartość
        inc     rax       ;udane
        ret
random endp;
```

Instrukcje arytmetyczne

IA32- rejestry



Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

Instrukcje arytmetyczne

- ADD dodawanie całkowitoliczbowe
- ADC dodawanie z przeniesieniem
- ADCX dodawanie z przeniesieniem bez znaku
- ADOX dodawanie z przeniesieniem bez znaku
- SUB odejmowanie
- SBB odejmowanie z pożyczką
- MUL mnożenie bez znaku
- MULX mnożenie bez znaku
- IMUL mnożenie ze znakiem
- DIV dzielenie bez znaku
- IDIV dzielenie ze znakiem
- INC inkrementacja (zwiększenie)
- DEC dekrementacja (zmniejszenie)
- NEG zmiana znaku
- CMP porównanie

Wpływa na flagi: OSZAPC

Instrukcja ADD

```
add    cel, źródło
```

Dodaje zawartość źródła i celu, sumę umieszcza w miejscu przeznaczenia (cel).

```
cel := cel + źródło
```

```
add    eax, zmienna
add    edx, [ebx+esi*4]
add    rcx, rbx
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.

Przykład – oblicz sumę krawędzi prostopadłościanu

```
mov    eax, a          ;wczytaj a
add    eax, b          ;dodaj b
add    eax, c          ;dodaj c
add    eax, eax        ; ×2
add    eax, eax        ; ×2
```

Wpływa na flagi: OSZAPC

Instrukcja ADC

adc cel, źródło

Dodaje zawartość źródła, celu i przeniesienia, sumę umieszcza w miejscu przeznaczenia (cel).

cel := cel + źródło + CF

```
adc   eax, zmienna
adc   edx, [ebx+esi*4]
add   rcx, rbx
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.

Przykład – oblicz sumę liczb całkowitych 128-bitowych

```
mov   esi, offset b     ;adres zmiennej b
mov   edi, offset a     ;adres zmiennej a
mov   eax, [esi]        ;najmłodsze  podwójne słowo b
add   [edi], eax        ;dodajemy do najmłodszego a
mov   eax, [esi+4]      ;drugie podwójne słowo b
adc   [edi+4], eax      ;do drugiego a plus przeniesienie
mov   eax, [esi+8]      ;trzecie podwójne słowo b
adc   [edi+8], eax      ;do trzeciego a plus przeniesienie
mov   eax, [esi+12]     ;czwarte podwójne słowo b
adc   [edi+12], eax     ;do czwartego a plus przeniesienie
```


Przykład – oblicz sumę liczb całkowitych 128-bitowych

```
mov   rsi, offset b     ;adres zmiennej b
mov   rdi, offset a     ;adres zmiennej a
mov   rax, [rsi]        ;najmłodsze  poczwórne słowo b
add   [rdi], rax        ;dodajemy do najmłodszego a
mov   rax, [rsi+8]      ;drugie poczwórne słowo b
adc   [rdi+8], rax      ;do drugiego a plus przeniesienie
```


Wpływa na flagi: -----C
Wymagane: ADX

Instrukcja ADCX

adcx cel, źródło

Dodaje bez znaku zawartość źródła, celu i przeniesienia, sumę umieszcza w miejscu przeznaczenia (cel – rejestr 32|64 bitowy).

cel := cel + źródło + CF

```
adcx   eax, zmienna
adcx   edx, [ebx+esi*4]
adcx   rcx, rbx
```


Wpływa na flagi: O-----
Wymagane: ADX

Instrukcja ADOX

adox cel, źródło

Dodaje bez znaku zawartość źródła, celu i flagi przepełnienia, sumę umieszcza w miejscu przeznaczenia (cel – rejestr 32|64 bitowy).

cel := cel + źródło + OF

```
adox   eax, zmienna
adox   edx, [ebx+esi*4]
adox   rcx, rbx
```


Wpływa na flagi: OSZAPC

Instrukcja SUB

sub cel, źródło

Odejmuje zawartość źródła od celu, różnicę umieszcza w miejscu przeznaczenia (cel).

cel := cel - źródło

```
sub   ecx, zmienna
sub   ebx, [ebx+esi*4]
sub   rax, rdx
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.

Wpływa na flagi: OSZAPC

Instrukcja SBB

sbb cel, źródło

Odejmuje zawartość źródła od celu z uwzględnieniem pożyczki, różnicę umieszcza w miejscu przeznaczenia (cel).

cel := cel - (źródło + CF)

```
sbb   edx, zmienna
sbb   eax, [ebx+esi*4]
sbb   rax, rdx
```

Uwaga:
Jeśli źródło jest adresowane w trybie prostym może mieć do 32 bitów.

Wpływa na flagi: OxxxxC

Instrukcja MUL

mul źródło

Mnoży bez znaku zawartość akumulatora (al, ax, eax, rax) i źródła, wynik umieszcza w miejscu przeznaczenia (ax, dx:ax, edx:eax, rdx:rax). Flagi CF i OF są zerem, jeśli starsza połowa bitów wyniku jest równa zero.

wynik := acc * źródło

```
mul   zmienna
mul   word ptr[ebx + esi*4]
```


Przykład – oblicz pole prostokąta o bokach a, b

```
        mov   eax, a              ;a
        mul   b                   ; a * b
        jc    poza_int            ;jeśli przekroczony zakres
        mov   pole, eax           ;zapisz
        ...
        ...
poza_int: ...
```


Wpływa na flagi: ------
Wymaga BMI2

Instrukcja MULX

mulx cel1, cel2, źródło

Mnoży bez znaku zawartość rejestru edx | rdx i źródła, wynik umieszcza w rejestrach celu(cel1:cel2).

cel1:cel2 := e(r)dx * źródło

```
mulx  ecx, ebx, [tab + esi*4]
mulx  rcx, rbx, rax
```


Wpływa na flagi: OxxxxC

Instrukcja IMUL -1

imul źródło

Mnoży ze znakiem zawartość akumulatora (al, ax, eax, rax) i źródła, wynik umieszcza w miejscu przeznaczenia (ax, dx:ax, edx:eax, rdx:rax). Flagi CF i OF są zerem, jeśli iloczyn mieści się w młodszej połowie bitów wyniku.

wynik := acc * źródło

```
imul  zmienna
imul  ecx
```


Wpływa na flagi: OxxxxC

Instrukcja IMUL -2

imul cel, źródło

Mnoży ze znakiem zawartość rejestru celu (16|32|64 bity) przez źródło, wynik umieszcza w miejscu przeznaczenia (cel). Flagi CF i OF są zerem, jeśli iloczyn mieści się w rejestrze celu.

cel := cel * źródło

```
imul  eax, zmienna
imul  rdx, rcx
```

Wpływa na flagi: OxxxxC

Instrukcja IMUL -3

imul cel, źródło1, źródło2

Mnoży ze znakiem zawartość źródła1 (16|32|64 bity) przez źródło2 (stała), wynik umieszcza w miejscu przeznaczenia (cel). Flagi CF i OF są zerem, jeśli iloczyn mieści się w rejestrze celu.

cel := źródło1 * źródło2

```
imul   eax, zmienna, 5
imul   cx, dx, 77
```

Uwaga:
Jeśli źródło2 jest adresowane w trybie prostym może mieć do 32 bitów.

Przykład – oblicz sumę krawędzi prostopadłościanu

```
mov    eax, a          ;wczytaj a
add    eax, b          ;dodaj b
add    eax, c          ;dodaj c
imul   eax, eax, 4     ; ×4
```


Wpływa na flagi: xxxxxx

Instrukcja DIV

div źródło

Dzieli bez znaku zawartość AX, DX:AX, EDX:EAX, RDX:RAX przez źródło, iloraz umieszcza w AL, AX, EAX, RAX a resztę w AH, DX, EDX, RDX.

```
div    byte ptr zmienna
div    ebx
```


Wpływa na flagi: xxxxxx

Instrukcja IDIV

idiv źródło

Dzieli ze znakiem zawartość AX, DX:AX, EDX:EAX, RDX:RAX przez źródło, iloraz umieszcza w AL, AX, EAX, RAX a resztę w AH, DX, EDX, RDX.

```
idiv   byte ptr zmienna
idiv   ebx
```


Wpływa na flagi: OSZAP

Instrukcja INC

inc cel

Zwiększa zawartość celu o 1.

```
inc    zmienna
inc    edx
inc    rcx
```


Wpływa na flagi: OSZAP

Instrukcja DEC

dec cel

Zmniejsza zawartość celu o 1.

```
dec    zmienna
dec    edx
dec    r8
```

Wpływa na flagi: OSZAPC

Instrukcja NEG

```
neg   cel
```

Zmienia znak celu w kodzie U2.

```
cel := -cel
```

```
neg   ax
neg   byte ptr[ebx+esi*4]
neg   r11
```

Flaga CF=0, tylko dla argumentu=0.

Wpływa na flagi: OSZAPC

Instrukcja CMP

```
cmp   źródło1, źródło2
```

Porównuje zawartość źródła1 i źródła2, wynik nie jest zapamiętywany, tylko są ustawiane flagi.

```
źródło1-źródło2
```

```
cmp   ax, zmienna
cmp   edx, [ebx+esi*4]
cmp   rcx, 123
```

Uwaga:

Jeśli źródło2 jest adresowane w trybie prostym może mieć do 32 bitów.

Przykład – oblicz sumę kwadratów liczb w tablicy

```
      mov   eax, 0              ;wartość początkowa sumy
      mov   esi, eax            ;indeks
      mov   ebx, offset wektor  ;tablica liczb całkowitych
      mov   ecx, 123            ;licznik
petla: mov  edx, [ebx+esi*4]
      imul  edx, edx            ;kwadrat liczby
      add   eax, edx            ;suma
      inc   esi
      dec   ecx
      jnz   petla               ;wynik w eax
```


Instrukcje arytmetyczne BCD

- DAA — korekta upakowanego kodu BCD po dodawaniu
  Decimal adjust after addition
- DAS — korekta upakowanego kodu BCD po odejmowaniu
  Decimal adjust after subtraction
- AAA — ASCII korekta po dodawaniu
  ASCII adjust after addition
- AAS — ASCII korekta po odejmowaniu
  ASCII adjust after subtraction
- AAM — ASCII korekta po mnożeniu
  ASCII adjust after multiplication
- AAD — ASCII korekta przed dzieleniem
  ASCII adjust before division

!!! Nie działają w trybie 64 bitowym !!!

Wpływa na flagi: OSZAPC

Instrukcja DAA

```
daa
```

Korekta upakowanego kodu BCD (w AL) po dodawaniu. Polega na dodaniu 6 najpierw do młodszego półbajta, a potem do starszego, jeśli ich zawartości były większe od 9 lub wystąpiło przeniesienie AF (CF)

```
daa
```


Wpływa na flagi: OSZAPC

Instrukcja DAS

```
das
```

Korekta upakowanego kodu BCD (w AL) po odejmowaniu. Polega na odjęciu 6 najpierw od młodszego półbajta, a potem od starszego, jeśli ich zawartości były większe od 9 lub wystąpiło przeniesienie AF (CF).

```
das
```

Wpływa na flagi: OSZAPC

Instrukcja AAA

aaa

Korekta nieupakowanego kodu BCD (w AL) po dodawaniu. Polega na dodaniu 6 do młodszego półbajta i 1 do AH, jeśli zawartość AL była większa od 9 lub wystąpiło przeniesienie AF.

aaa

Wpływa na flagi: OSZAPC

Instrukcja AAS

aas

Korekta nieupakowanego kodu BCD (w AL) po odejmowaniu. Polega na odjęciu 6 od młodszego półbajta i 1 od AH, jeśli zawartość AL. była większa od 9 lub wystąpiło przeniesienie AF.

aas

Wpływa na flagi: OSZAPC

Instrukcja AAM

aam

Korekta nieupakowanego kodu BCD (w AX) po mnożeniu. Polega na jednoczesnym wykonaniu:

AH=AL div 10

AL:=AL Mod 10.

aam

Wpływa na flagi: OSZAPC

Instrukcja AAD

aad

Korekta nieupakowanego kodu BCD (w AX) przed dzieleniem. Polega na jednoczesnym wykonaniu:

AL:=AH*10+AL.

AH=0

aad

Przykład – dodawanie liczb BCD

```
        mov   al, 0
        add   al, al          ;CF=0
petla:  mov   al, [esi]       ;pobierz cyfrę źródła
        adc   al, ds:[edi]    ;dodaj cyfrę celu z przeniesieniem
        aaa                   ;korekta
        mov   ds:[edi], al    ;zapamiętaj cyfrę
        inc   esi             ;następna cyfra
        inc   edi
        dec   ecx
        jnz   petla           ;CF nie zmieniło się od AAA!!!
```

Operacje na łańcuchach

Operacje na łańcuchach

- MOVS/MOVSB/MOVSW/MOVSD/MOVSQ Prześlij łańcuch/bajtów/słów/podwójnych słów/poczwórnych słów
- CMPS/CMPSB/CMPSW/CMPSD/CMPSQ Porównaj łańcuchy/bajtów/słów/podwójnych słów/poczwórnych słów
- SCAS/SCASB/SCASW/SCASD/SCASQ Skanuj łańcuch/bajtów/słów/podwójnych słów/poczwórnych słów
- LODS/LODSB/LODSW/LODSD/LODSQ Ładuj łańcuch/bajtów/słów/podwójnych słów/poczwórnych słów
- STOS/STOSB/STOSW/STOSD/STOSQ Zapamiętaj łańcuch/bajtów/słów/podwójnych słów/poczwórnych słów
- REP Powtarzaj dopóki ECX nie jest zerem
- REPE/REPZ Powtarzaj dopóki equal/zero
- REPNE/REPNZ Powtarzaj dopóki not equal/not zero

Wpływa na flagi: -

Instrukcja MOVS/MOVSB

movs byte ptr [(r|e)di], [(r|e)si]
movsb

Przesyła bajt z pamięci ds:(r|e)si do pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 1 w zależności od flagi DF (0/1).

[(r|e)s:edi] := [ds:(r|e)si]
(r|e)di := (r|e)di ±1
(r|e)si := (r|e)si ±1

movsb

Wpływa na flagi: -

Instrukcja MOVS/MOVSW

movs word ptr [(r|e)di], [(r|e)si]
movsw

Przesyła słowo z pamięci ds:(r|e)si do pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 2 w zależności od flagi DF (0/1).

[es:(r|e)di] := [ds:(r|e)si]
(r|e)di := (r|e)di ±2
(r|e)si := (r|e)si ±2

movsw

Wpływa na flagi: -

Instrukcja MOVS/MOVSD

movs dword ptr [(r|e)di], [(r|e)si]
movsd

Przesyła podwójne słowo z pamięci ds:esi do pamięci es:edi. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 4 w zależności od flagi DF (0/1).

[es:(r|e)di] := [ds:(r|e)si]
(r|e)di := (r|e)di ±4
(r|e)si := (r|e)si ±4

movsd

Wpływa na flagi: -

Instrukcja MOVS/MOVSQ

movs qword ptr [(r|e)di], [(r|e)si]
movsq

Przesyła poczwórne słowo z pamięci ds:(r|e)si do pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 8 w zależności od flagi DF (0/1).

[es:(r|e)di] := [ds:(r|e)si]
(r|e)di := (r|e)di ±8
(r|e)si := (r|e)si ±8

movsq

Wpływa na flagi: OSZAPC

Instrukcja CMPS/CMPSB

cmps byte ptr [(r|e)si], [(r|e)di]

cmpsb

Porównuje bajt z pamięci ds:(r|e)si i z pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 1 w zależności od flagi DF (0/1).

[ds:(r|e)si] - [es:(r|e)di]

(r|e)di := (r|e)di ±1

(r|e)si := (r|e)si ±1

cmpsb

Wpływa na flagi: OSZAPC

Instrukcja CMPS/CMPSW

cmps word ptr [(r|e)si], [(r|e)di]

cmpsw

Porównuje słowo z pamięci ds:(r|e)si i z pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 2 w zależności od flagi DF (0/1).

[ds:(r|e)si] - [es:(r|e)di]

(r|e)di := (r|e)di ± 2

(r|e)si := (r|e)si ± 2

cmpsw

Wpływa na flagi: OSZAPC

Instrukcja CMPS/CMPSD

cmps dword ptr [(r|e)si], [(r|e)di]

cmpsd

Porównuje podwójne słowo z pamięci ds:(r|e)si i z pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 4 w zależności od flagi DF (0/1).

[ds:(r|e)si] - [es:(r|e)di]

(r|e)di := (r|e)di ± 4

(r|e)si := (r|e)si ± 4

cmpsd

Wpływa na flagi: OSZAPC

Instrukcja CMPS/CMPSQ

cmps qword ptr [(r|e)si], [(r|e)di]

cmpsq

Porównuje poczwórne słowo z pamięci ds:(r|e)si i z pamięci es:(r|e)di. Rejestry (r|e)di/(r|e)si są zwiększane/zmniejszane o 8 w zależności od flagi DF (0/1).

[ds:(r|e)si] - [es:(r|e)di]

(r|e)di := (r|e)di ± 8

(r|e)si := (r|e)si ± 8

cmpsq

Wpływa na flagi: OSZAPC

Instrukcja SCAS/SCASB

scas byte ptr [(r|e)di]

scasb

Porównuje bajt akumulatora AL i pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 1 w zależności od flagi DF (0/1).

AL. - [es:(r|e)di]

(r|e)di := (r|e)di ± 1

scasb

Wpływa na flagi: OSZAPC

Instrukcja SCAS/SCASW

scas word ptr [(r|e)di]

scasw

Porównuje słowo akumulatora AX i pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 2 w zależności od flagi DF (0/1).

AX-[es:(r|e)di]

(r|e)di:=(r|e)di ± 2

scasw

Wpływa na flagi: OSZAPC

Instrukcja SCAS/SCASD

scas dword ptr [(r|e)di]

scasd

Porównuje podwójne słowo akumulatora EAX i pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 4 w zależności od flagi DF (0/1).

EAX - [es: (r|e)di]

(r|e)di := (r|e)di ± 4

scasd

Wpływa na flagi: OSZAPC

Instrukcja SCAS/SCASQ

scas qword ptr [(r|e)di]

scasq

Porównuje poczwórne słowo akumulatora RAX i pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 8 w zależności od flagi DF (0/1).

RAX-[es:(r|e)di]

(r|e)di := (r|e)di ±8

scasq

Wpływa na flagi: -

Instrukcja LODS/LODSB

lods byte ptr [(r|e)si]

lodsb

Czyta bajt do akumulatora AL z pamięci ds:(r|e)si. Rejestr (r|e)si jest zwiększany/zmniejszany o 1 w zależności od flagi DF (0/1).

AL := [ds:(r|e)si]

(r|e)si := (r|e)si ±1

lodsb

Wpływa na flagi: -

Instrukcja LODS/LODSW

lods word ptr [(r|e)si]

lodsw

Czyta słowo do akumulatora AX z pamięci ds:(r|e)si. Rejestr (r|e)si jest zwiększany/zmniejszany o 2 w zależności od flagi DF (0/1).

AX = [ds:(r|e)si]

(r|e)si := (r|e)si ±2

lodsw

Wpływa na flagi: -

Instrukcja LODS/LODSD

lods dword ptr [(r|e)si]

lodsd

Czyta podwójne słowo do akumulatora EAX z pamięci ds:(r|e)si. Rejestr (r|e)si jest zwiększany/zmniejszany o 4 w zależności od flagi DF (0/1).

EAX = [ds:(r|e)si]

(r|e)si := (r|e)si ± 4

lodsd

Wpływa na flagi: -

Instrukcja LODS/LODSQ

lods qword ptr [(r|e)si]

lodsq

Czyta poczwórne słowo do akumulatora RAX z pamięci ds:(r|e)si. Rejestr (r|e)si jest zwiększany/zmniejszany o 8 w zależności od flagi DF (0/1).

RAX = [ds:(r|e)si]

(r|e)si := (r|e)si ± 8

lodsq

Wpływa na flagi: -

Instrukcja STOS/STOSB

stos byte ptr [(r|e)di]
stosb

Zapisuje bajt z akumulatora AL do pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 1 w zależności od flagi DF (0/1).

[es:(r|e)di] := AL
(r|e)di:=(r|e)di ± 1

stosb

Wpływa na flagi: -

Instrukcja STOS/STOSW

stos word ptr [(r|e)di]
stosw

Zapisuje słowo z akumulatora AX do pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 2 w zależności od flagi DF (0/1).

[es:(r|e)di] := AX
(r|e)di:=(r|e)di ± 2

stosw

Wpływa na flagi: -

Instrukcja STOS/STOSD

stos dword ptr [(r|e)di]
stosd

Zapisuje podwójne słowo z akumulatora EAX do pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 4 w zależności od flagi DF (0/1).

[es:(r|e)di] := EAX
(r|e)di := (r|e)di ± 4

stosd

Wpływa na flagi: -

Instrukcja STOS/STOSQ

stos qword ptr [(r|e)di]
stosq

Zapisuje poczwórne słowo z akumulatora RAX do pamięci es:(r|e)di. Rejestr (r|e)di jest zwiększany/zmniejszany o 8 w zależności od flagi DF (0/1).

[es:(r|e)di]=RAX
(r|e)di:=(r|e)di ±8

stosq

Wpływa na flagi: -

Prefiks REP
REPNZ/REPNE
REPZ/REPE

Powoduje powtórzenie (R|E)CX razy następującej po nim instrukcji łańcuchowej, jeśli spełniony jest warunek (repnz powtarza dopóty ZF = 0,jeśli ZF = 1 powtarzanie jest przerywane itd.). Jeżeli (R|E)CX = 0, to instrukcja nie zostanie wykonana.

Wpływa na flagi: -

Prefiks REP
REPNZ/REPNE
REPZ/REPE

rep movsb

rep lodsd

rep stosq

rep*ww* cmpsw

rep*ww* scasb

Przykład

```
mov ecx, 100
mov esi, bufor1
mov edi, bufor2
rep movsb
```

Kopiuje zawartość bufora1 do bufora2.

```
mov rax, 0
mov rcx, 100
mov rdi, bufor
rep ds:stosq
```

Zeruje zawartość bufora (800B).

```
mov al, 77
mov ecx, 100
mov edi, bufor
repnz ds:scasb
```

Szuka wartości 77 w buforze. ZF = 1 oznacza znalezienie żądanej wartości.

```
mov al, 0
mov rcx, 100
mov rdi, bufor
repz ds:scasb
```

Szuka wartości <>0 w buforze. ZF = 0 oznacza znalezienie żądanej wartości.

Operacje na rejestrach segmentowych

- LDS Załadowanie pełnego wskaźnika z użyciem DS
- LES Załadowanie pełnego wskaźnika z użyciem ES
- LFS Załadowanie pełnego wskaźnika z użyciem FS
- LGS Załadowanie pełnego wskaźnika z użyciem GS
- LSS Załadowanie pełnego wskaźnika z użyciem SS

Wpływa na flagi: -

Instrukcja LDS

lds cel, źródło

Wczytanie pełnego adresu źródła do pary rejestrów ds:cel(32).

ds:cel := wskaźnik do źródła

```
lds esi, tablica
```


Wpływa na flagi: -

Instrukcja LES

les cel, źródło

Wczytanie pełnego adresu źródła do pary rejestrów es:cel(32).

es:cel := wskaźnik do źródła

```
les edi, tablica2
```


Wpływa na flagi: -

Instrukcja LFS

lfs cel, źródło

Wczytanie pełnego adresu źródła do pary rejestrów fs:cel.

fs:cel := wskaźnik do źródła

```
lfs eax, tablica
```


Wpływa na flagi: -

Instrukcja LGS

lgs cel, źródło

Wczytanie pełnego adresu źródła do pary rejestrów gs:cel.

gs:cel := wskaźnik do źródła

```
lgs eax, tablica
```

Wpływa na flagi: -

Instrukcja LSS

lss cel, źródło

Wczytanie pełnego adresu źródła do pary rejestrów ss:cel.

ss:cel := wskaźnik do źródła

lss esp, nowy_stos

Inne operacje

- LOCK — Powoduje niepodzielne wykonanie następnej instrukcji
- LEA — Ładowanie adresu efektywnego
- NOP — Nie wykonuje żadnego działania
- UD2 — Instrukcja niezdefiniowana
- XLAT/XLATB — Tłumaczenie w oparciu o tablicę translacji
- MOVBE — Przesłanie po zamianie kolejności bajtów
- CPUID — Identyfikacja procesora

Wpływa na flagi: -

Prefiks LOCK

lock

Powoduje wystawienie sygnału LOCK procesora i wykonanie w sposób niepodzielny instrukcji:

add, adc, and, brc, btr, bts, cmpxchg, cmpxch8b, cmpxch16b, dec, inc, neg, not, or, sbb, sub, xor, xadd i xchg,

<u>jeśli argument celu jest w pamięci</u>.

lock btr

Przykład

```
        mov     zu, 1   ;inicjalizacja
        ...
        ...
        xor     ax, ax  ;ax=0
@p:
lock    xchg    zu, ax  ;al<->zu
        test    ax, ax  ;czy 0
        jz      @p      ;akt. czekanie
        ...
        ...
        ...
        mov     zu, 1   ;oddaj 1 do zu
        ...
```

```
        mov     zu, 1
        ...
        ...
        ...
        ...
@p:
lock    btr     zu, 0   ;zajmij
        jnc     @p
        ...
        ...
        ...
        mov     zu, 1   ;uwolnij
        ...
```


Wpływa na flagi: -

Instrukcja LEA

lea cel, źródło

Wczytanie wyznaczonego adresu źródła do rejestru celu.

cel := adres źródła

```
lea eax, [edx+esi*4+12]     ; eax = edx + esi * 4 + 12
lea rax, [rdx+rsi*4+12]     ; rax = rdx + rsi * 4 + 12
```


Przykład

```
lea     rax, [rdx]          ;kopiuje rejestr
lea     rbx, [rcx + rdx]    ;suma rejestrów
lea     rdx, [rax +10]      ;suma rejestru i stałej
lea     rax, [rax +1]       ;inkrementacja
lea     rax, [rbx+8*rsi + 3]    ;suma trzech wartości
lea     rax, [rax + 4*rax]  ;mnożenie przez 5
```

Wpływa na flagi: -

Instrukcja NOP

nop

Nic nie robi.

nop

Wpływa na flagi: -

Instrukcja UD2

ud2

Generuje wyjątek *instrukcja niezdefiniowana*, nic nie robi, wprowadzona do testów.

ud2

Wpływa na flagi: -

Instrukcja XLAT/XLATB

xlat arg

xlatb

Tłumaczenie w oparciu o tablicę translacji.

AL := DS:[(R|E)BX+AL]

xlatb

Wpływa na flagi: -

Instrukcja MOVBE

movbe cel, źródło

Przesłanie po zamianie kolejności bajtów. Jeden z argumentów musi być rejestrem (16, 32, 64).

cel := zamień(źródło)

movbe eax, zmienna

przed	12	c4	7f	de
po	de	7f	c4	12

Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
1	1	zarezerwowany	
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
8	TP	flaga umożliwiająca krokowe wykonanie (trap)	X
9	IF	flaga zezwolenia na przerwania (interrupt enable)	X
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S
12, 13	IOPL	poziom uprawnień we/wy (I/O privilege level, od 286)	X
14	NT	nested task flag (od 286)	X
16	RF	flaga wznowienia (resume, od 386)	X
17	VM	flaga trybu Virtual 8086 (od 386)	X
18	AC	alignment check (od 486SX)	X
19	VIF	Virtual interrupt flag (od Pentium)	X
20	VIP	Virtual interrupt pending (od Pentium)	X
21	ID	Identification (od Pentium)	X
3, 5, 15, 22-31	0	zarezerwowany	

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

Wpływa na flagi: -

Instrukcja CPUID

cpuid

Identyfikacja procesora jest możliwa, jeśli bit 21 flaga ID w rejestrze flag może być zmieniana. Na podstawie EAX (czasem też ECX) podaje w EAX,EBX,ECX i EDX różne informacje o procesorze.

cpuid

Test ID

```
        xor     eax, eax  ;0
        pusfd
        pop     edx
        bts     edx, 21   ;ustaw
        push    edx
        popfd
        pushfd
        pop     edx
        btr     edx, 21   ;skasuj
        rcl     eax, 1    ;1b
        push    edx
        popfd
        pushfd
        pop     edx
```

```
        bts     edx, 21   ;ustaw
        rcl     eax, 1    ;10b
        push    edx
        popfd
        pushfd
        pop     edx
        bt      edx, 21   ;sprawdź
        rcl     eax, 1    ;101b
        cmp     eax, 5
        je      cpuidOK
```

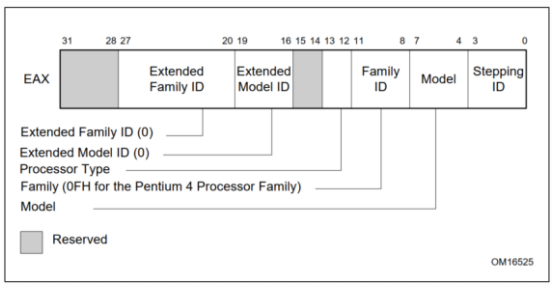

Przykład

```
mov eax, 0
cpuid
```

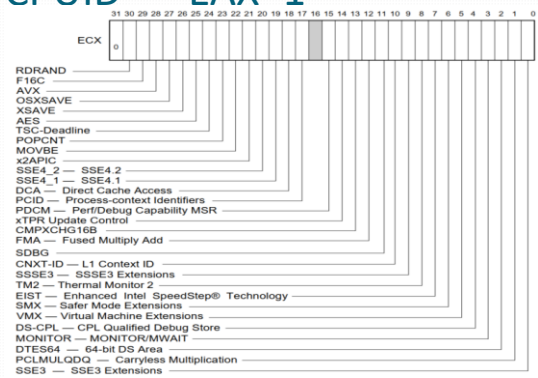
Zwraca wartość maksymalną dla cpuid
oraz identyfikator producenta:

eax = max
ebx = 'Genu'
ecx = 'ntel'
edx = 'ineI'

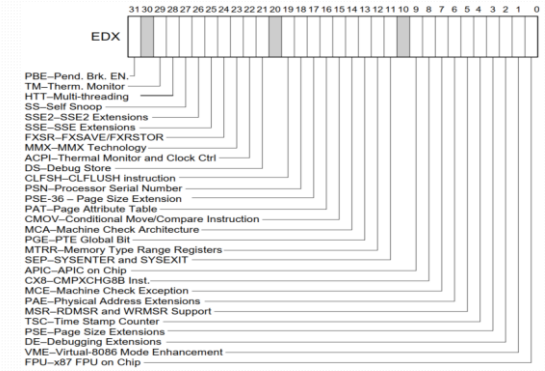
CPUID EAX=1



CPUID EAX=1



CPUID EAX=1



Przykład

Sprawdzenie, czy procesor posiada technologię MMX.

```
mov EAX, 1          ; żądanie informacji o właściwościach
CPUID               ; oFH, oA2H instrukcja CPUID
test EDX, oo8ooooooH ; sprawdzenie bitu technologii MMX (bit 23 w EDX)
jnz                 ; znaleziono technologię MMX
```

Wpływa na flagi: -

Instrukcja XGETBV

xgetbv

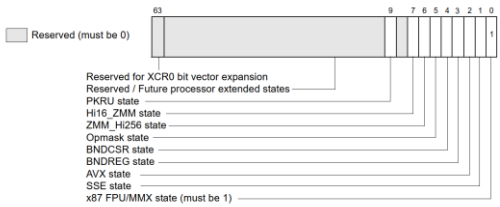
Czyta do edx:eax zawartość rozszerzonego rejestru kontrolnego o indeksie ecx.

xgetbv

Przykład - Sprawdzenie, czy procesor posiada technologię AVX.

```
supports_AVX proc
        mov     eax, 1
        cpuid
        and     ecx, 018000000H
        cmp     ecx, 018000000H  ; sprawdź flagi OSXSAVE i AVX
        jne     not_sup
        ; procesor wspiera inst. AVX i XGETBV jest włączone przez OS
        mov     ecx, 0           ; wybierz rejestr XCR0
        XGETBV                   ; wynik w EDX:EAX
        and     eax, 06H
        cmp     eax, 06H         ; czy OS włączył obsługę XMM i YMM
        jne     not_sup
        mov     eax, 1           ; wspiera AVX
        jmp     done
not_sup: mov    eax, 0           ; nie wspiera AVX
done:   ret
endp
```


XCR0



Instrukcje warunkowe i skoku

Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

Warunki dotyczące flag

- E/Z equal/ zero ZF=1
- NE/NZ not equal/ not zero ZF=0
- C carry CF=1
- NC not carry CF=0
- O overflow OF=1
- NO not overflow OF=0
- S sign (negative) SF=1
- NS not sign (non-negative) SF=0
- P/PE parity/ parity even PF=1
- NP/PO not parity/ parity odd PF=0

Warunki porównania liczb

- E/Z equal/ zero ZF=1
- NE/NZ not equal/ not zero ZF=0

Dla liczb bez znaku:

- A/NBE above/ not below or equal CF=0 i ZF=0
- AE/NB above or equal/ not below CF=0
- B/NAE below/ not above or equal CF=1
- BE/NA below or equal/ not above CF=1 lub ZF=1

Dla liczb ze znakiem

- G/NLE greater/ not less or equal ZF=0 i SF=OF
- GE/NL greater or equal/ not less SF=OF
- L/NGE less/ not greater or equal SF<>OF
- LE/NG less or equal/ not greater ZF=1 lub SF<>OF

Wpływa na flagi: -

Instrukcja CMOVcc

CMOVcc cel, źródło

Jeśli jest spełniony warunek cc, przesyła źródło do miejsca przeznaczenia (rejestr 16, 32 lub 64 bitowy). Instrukcja wprowadzona w procesorach rodziny P6!

if cc then cel:=źródło

```
cmovz    eax, zmienna
cmovge   edx, [ebx+esi*4]
cmovna   rax, rdx
```


Instrukcje CMOVcc

- CMOVE/CMOVZ Prześlij jeżeli equal/ zero
- CMOVNE/CMOVNZ Prześlij jeżeli not equal/ not zero
- CMOVA/CMOVNBE Prześlij jeżeli above/ not below or equal
- CMOVAE/CMOVNB Prześlij jeżeli above or equal/ not below
- CMOVB/CMOVNAE Prześlij jeżeli below/ not above or equal
- CMOVBE/CMOVNA Prześlij jeżeli below or equal/ not above
- CMOVG/CMOVNLE Prześlij jeżeli greater/ not less or equal
- CMOVGE/CMOVNL Prześlij jeżeli greater or equal/ not less
- CMOVL/CMOVNGE Prześlij jeżeli less/ not greater or equal
- CMOVLE/CMOVNG Prześlij jeżeli less or equal/ not greater
- CMOVC Prześlij jeżeli carry
- CMOVNC Prześlij jeżeli not carry
- CMOVO Prześlij jeżeli overflow
- CMOVNO Prześlij jeżeli not overflow
- CMOVS Prześlij jeżeli sign (negative)
- CMOVNS Prześlij jeżeli not sign (non-negative)
- CMOVP/CMOVPE Prześlij jeżeli parity/ parity even
- CMOVNP/CMOVPO Prześlij jeżeli not parity/ parity odd

Przykład

```
MyMax64 proc
movsxd   rax, ecx
movsxd   rdx, edx
cmp      rax, rdx
cmovl    rax, rdx
ret
MyMax64 endp
```

```
function
TForm1.MyMax(x,y:integer):integer;
asm
  mov eax, x
  cmp eax, y
  jnc  @@exit
  mov eax, y
@@exit:
end;

function
TForm1.MyMax2(x,y:integer):integer;
asm
  mov eax, x
  cmp eax, y
  cmovc eax, y       ;cmovb   eax,y
end;
```


Skoki warunkowe Jcc

- JE/JZ — Skocz jeśli equal/zero
- JNE/JNZ — Skocz jeśli not equal/not zero
- JA/JNBE — Skocz jeśli above/not below or equal
- JAE/JNB — Skocz jeśli above or equal/not below
- JB/JNAE — Skocz jeśli below/not above or equal
- JBE/JNA — Skocz jeśli below or equal/not above
- JG/JNLE — Skocz jeśli greater/not less or equal
- JGE/JNL — Skocz jeśli greater or equal/not less
- JL/JNGE — Skocz jeśli less/not greater or equal
- JLE/JNG — Skocz jeśli less or equal/not greater
- JC — Skocz jeśli carry
- JNC — Skocz jeśli not carry
- JO — Skocz jeśli overflow
- JNO — Skocz jeśli not overflow
- JS — Skocz jeśli sign (negative)
- JNS — Skocz jeśli not sign (non-negative)
- JPO/JNP — Skocz jeśli parity odd/not parity
- JPE/JP — Skocz jeśli parity even/parity

Wpływa na flagi: -

Instrukcja JZ

jz przesunięcie

Przeskakuje do podanej etykiety (adres jest względny 16/32bitowy).

EIP := EIP + przesunięcie

```
        jz      dalej
        ...
dalej:  ...
```


Przykład

Sortowanie

```
procedure MySort1(t:array of integer;n:integer);
asm
        push    edi     ;zabezpiecz rejestry
        push    esi
        mov     esi, n  ;liczba
        dec     esi     ;esi ostatni element
        mov     edx, t  ;adres tablicy
@p:     mov     edi, esi
        dec     edi     ;poprzedzający element
        mov     eax, [edx+esi*4]    ;ostatni do eax
@w:     cmp     eax, [edx+edi*4]    ;czy >=
        jae     @@a
        xchg    eax, [edx+edi*4]    ;zamień elementy
        mov     [edx+esi*4], eax
@@a:    dec     edi     ;pętla wewnętrzna
        jns     @w
        dec     esi     ;pętla główna
        jnz     @p
        pop     esi     ;przywróć rejestry
        pop     edi
end;
```


Instrukcje sterujące przebiegiem programu +

- JMP — Skok bezwarunkowy
- JCXZ/JECXZ/JRCX — Skok jeśli zero w rejestrze CX/ECX/RCX
- LOOP — Pętla z licznikiem CX/ECX/RCX
- LOOPZ/LOOPE — Pętla z licznikiem CX/ECX/RCX i zero/equal
- LOOPNZ/LOOPNE — Pętla z licznikiem CX/ECX/RCX i not zero/not equal
- CALL — Wywołanie podprogramu
- RET — Powrót z podprogramu
- IRET — Powrót z podprogramu obsługi przerwania
- INT — Przerwanie programowe
- INTO — Przerwanie przy przekroczeniu zakresu
- BOUND — sprawdzenie ograniczeń indeksu tablicy
- ENTER — Wysokopoziomowe wejście do podprogramu – utworzenie ramy stosu
- LEAVE — Wysokopoziomowe wyjście z podprogramu – usunięcie ramy stosu

Wpływa na flagi: -

Instrukcja JMP

jmp adres

Przeskakuje do podanej etykiety (adres jest względny 8/16/32bitowy lub bezwzględny).

EIP := EIP + przesunięcie(adres)

CS := segment(adres); EIP := EIP + przesunięcie(adres)

```
jmp     dalej
jmp     eax
jmp     [esi]
jmp     lib1:dalej1
```

Wpływa na flagi: -

Instrukcja JCXZ/JECXZ/JRCXZ

JCXZ/JECXZ/JRCXZ przesunięcie

Skok jeśli zero w rejestrze CX/ECX/RCX do podanej etykiety (adres jest względny 16/32/64 bitowy).

EIP := EIP + przesunięcie

```
petla: ...
       jecxz  dalej
       ...
       jmp    petla
dalej: ...
```


Wpływa na flagi: -

Instrukcja LOOP

loop przesunięcie

Pętla z licznikiem CX/ECX/RCX. Zmniejsza CX/ECX/RCX o 1 i jeśli nie uzyskano zera przeskakuje do podanej etykiety (adres jest względny 8 bitowy).

EIP := EIP + przesunięcie(adres)

```
petla: ...
       ...
       loop   petla
```


Przykład

Silnia

```
function silnia(n:integer):integer;
asm
   mov   ecx, eax
   dec   ecx
@p: imul eax, ecx
   dec   ecx
   jnz   @p
end;
```

```
function silnia1(n:integer):integer;
asm
   mov   ecx, eax
@p: dec  ecx
   jecxz  @e
   imul   eax, ecx
   jmp    @p
@e:
end;
```

```
function silnia2(n:integer):integer;
asm
   mov   ecx, eax
   dec   ecx
@p: imul eax, ecx
   loop   @p
end;
```


Przykład

```
for(int i=0;i<n;i++)
{
…
}
```

```
            xor     rcx,rcx
for_petla:  cmp     rcx,n
            jnb     for_end
            ...
            ...
for_cont:   inc     rcx
            jmp     for_petla
for_end:
```

```
            xor     rcx,rcx
for_petla:  ...
            ...
for_cont:   inc     rcx
            cmp     rcx,n
            jb      for_petla
for_end:
```

```
            mov     rcx,n
for_petla:  ...
            ...
for_cont:   dec     rcx
            jnz     for_petla
for_end:
```


Wpływa na flagi: -

Instrukcja LOOPZ/LOOPE

loopz/loope przesunięcie

Pętla z licznikiem CX/ECX/RCX i zero/equal. Zmniejsza CX/ECX/RCX o 1 i jeśli nie uzyskano zera w CX/ECX/RCX i flaga ZF=1 przeskakuje do podanej etykiety (adres jest względny 8 bitowy).

EIP := EIP + przesunięcie(adres)

```
petla: ...
       ...
       loopz  petla
```


Wpływa na flagi: -

Instrukcja LOOPNZ/LOOPNE

loopnz/loopne przesunięcie

Pętla z licznikiem CX/ECX/RCX i nie zero/equal. Zmniejsza CX/ECX/RCX o 1 i jeśli nie uzyskano zera i flaga ZF=0 przeskakuje do podanej etykiety (adres jest względny 8 bitowy).

EIP := EIP + przesunięcie(adres)

```
petla: ...
       ...
       loopnz         petla
```

Wpływa na flagi: -

Instrukcja CALL

call adres

Instrukcja call wywołuje podprogram, wysyła na stos adres powrotu EIP|RIP lub CS:EIP|RIP. Parametr adres wpisuje do EIP|RIP lub CS:EIP|RIP.

push e(r)ip; (push cs);

E(R)IP := przesunięcie adres; (CS := selektor segmentu)

call procedura

call eax

call [esi]

Wpływa na flagi: -

Instrukcja RET

ret (ile)

Instrukcja ret wraca z podprogramu, pobiera ze stosu adres powrotu do EIP|RIP lub CS:EIP|RIP. Jeśli posiada parametr ile, to dodatkowo usuwa ze stosu ile bajtów (niepotrzebne już parametry aktualne wywołania).

pop e(r)ip; (pop cs); (e(r)sp:=e(r)sp+ile)

ret

ret 6

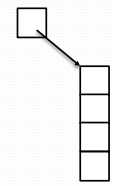
Przykład

Silnia

```
function silnia3(n:integer):integer;
asm
  and   eax,eax
  cmovz eax,one
  jz    @e
  push  eax
  dec   eax
  call  silnia3
  pop   edx
  imul  eax,edx
@e:
end;

one  db  1,0,0,0,0,0,0,0
silnia4 proc;
  cmp   rcx,1
  cmovbe rax, one
  jbe   @e
  push  rcx
  dec   rcx
  call  silnia4
  pop   rcx
  imul  rax, rcx
@e: ret
silnia4 endp;
```


Przykład

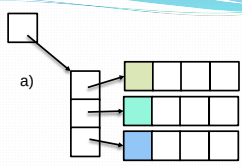
Tablice - wektory

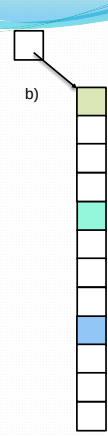


Zmienna tablicowa jest wskaźnikiem na początek obszaru pamięci, który zawiera tablicę.

Przykład

Tablice – Macierze Dynamiczne i statyczne





Zmienna tablicowa jest wskaźnikiem na początek obszaru pamięci, który zawiera:
a) wektor wskaźników na wektory wierszy;
b) całą tablicę (macierz statyczna).

Przykład

Oblicz sumę elementów tablicy typu int (32)

```
;rcx – wskaźnik do macierzy dynamicznej
;rdx – liczba wierszy
;r8 – liczba kolumn
suma_et proc;
        xor   rax, rax  ;suma=0
@pz:    mov   r9, r8    ;liczba kolumn
        mov   r10, [rcx+8*rdx-8];adr. wiersza
@pw:    movsxd r11, [r10+4*r9-4]
        add   rax, r11
        dec   r9        ;liczba kolumn--
        jnz   @pw
        dec   rdx       ;liczba wierszy--
        jnz   @pz
        ret
suma_et endp;
```

Wpływa na flagi: -

Instrukcja INT

int nr

Wywołuje przerwanie programowe o numerze nr (0-255). Numery z zakresu 0-31 są zarezerwowane. Instrukcja int działa podobnie do instrukcji call, jednak dodatkowo wysyła na stos flagi i wchodząc do podprogramu część z nich zeruje.

int 21h

Przerwania i wyjątki zarezerwowane

Wektor Nr	Mnemonik	Opis	Źródło
0	#DE	Błąd dzielenia	Instrukcje DIV i IDIV.
1	#DB	Debuger	Dowolne odwołanie do kodu i danych.
2		Przerwanie NMI	Przerwanie niemaskowalne Non-maskable external.
3	#BP	Breakpoint	Instrukcja INT 3.
4	#OF	Overflow	Instrukcja INTO.
5	#BR	BOUND przekroczony zakres	Instrukcja BOUND.
6	#UD	Błędny kod	Instrukcja UD2 lub zarezerwowany kod.1
7	#NM	Urządzenie nie dostępne (Brak koprocesora)	Instrukcje zmiennoprzecinkowe lub WAIT/FWAIT.
8	#DF	Błąd Double	Dowolna instrukcja generująca wyjątek, NMI lub INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Instrukcje zmiennoprzecinkowe.2
10	#TS	Błędny TSS	Przełączanie zadań lub dostęp do TSS.
11	#NP	Segment nieobecny	Ładowanie rejestrów segmentowych lub dostęp do segmentów systemowych.
12	#SS	Segment stosu uszkodzony	Operacje na stosie I ładowanie rejestru SS.
13	#GP	Ogólna ochrona	Dowolne odwołanie do pamięci i inne zabezpieczenia.
14	#PF	Błąd strony	Dowolne odwołanie do pamięci.
15		Zarezerwowane	
16	#MF	Błąd zmiennoprzecinkowy (błąd matematyczny)	Instrukcje zmiennoprzecinkowe lub WAIT/FWAIT.
17	#AC	Kontrola wyrównania	Dowolne odwołanie do danych w pamięci.3
18	#MC	Kontrola maszyny	Kod błędu (o ile występuje) i źródło zależą od modelu.4
19	#XM	Wyjątek SIMD	Instrukcje zmiennoprzecinkowe SIMDs
20-31		Zarezerwowane	
32-255		Przerwania maskowalne	Przerwania zewnętrzne INTR lub instrujcje INT *n*.

1. Instrukcja UD2 została wprowadzona w procesorze Pentium Pro. 2. Procesory IA-32 po procesorze Intel386 nie generują tego wyjątku. 3. Wyjątek wprowadzony w procesorze Intel486. 4. Wyjątek wprowadzony w procesorze Pentium i poprawiony w procesorach rodziny P6. 5. Wyjątek wprowadzony w procesorze Pentium III.

Wpływa na flagi: -

Instrukcja INTO

into

Wywołuje przerwanie programowe (4) w przypadku ustawienia flagi OF (nadmiaru).

into

Wpływa na flagi: -

Instrukcja IRET/IRETD/IRETQ

iret/iretd/iretq

Instrukcja iret/iretd/iretq wraca z podprogramu obsługi przerwania, pobiera ze stosu adres powrotu do CS:EIP|RIP oraz flagi zachowane przy wywołaniu przerwania.

pop e(r)ip; pop cs; pop (e(r)flags)

iret

Wpływa na flagi: -

Instrukcja BOUND

bound idx, gr

Sprawdza, czy indeks tablicy (wartość 16/32 bitowa ze znakiem) zawarty w rejestrze idx nie przekracza jej granic określonych przez strukturę w pamięci gr złożoną z granicy dolnej i górnej. W przypadku przekroczenia granicy generowany jest wyjątek przekroczenia granicy tablicy.

bound eax, granice1

Wpływa na flagi: -

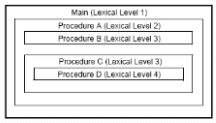
Instrukcja ENTER

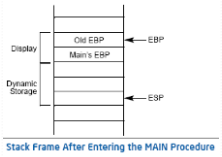
enter storage, level

Tworzy ramę stosu w podprogramie z uwzględnieniem poziomu zagnieżdżenia podprogramów lokalnych (level) i rozmiaru w bajtach zmiennych lokalnych (storage). Na stosie umieszcza wskaźniki ram stosu: podprogramu wywołującego, wszystkich poziomów nadrzędnych i własny.

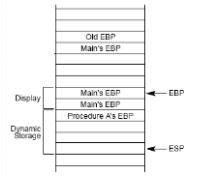
```
PUSH EBP;
FRAME_PTR ← ESP;
IF LEVEL > 0 THEN
        DO (LEVEL – 1) times
                EBP ← EBP – 4;
                PUSH Pointer(EBP); (* doubleword wskazywane przez EBP *)
        OD;
        PUSH FRAME_PTR;
FI;
EBP ← FRAME_PTR;
ESP ← ESP – STORAGE;
```

Instrukcja ENTER



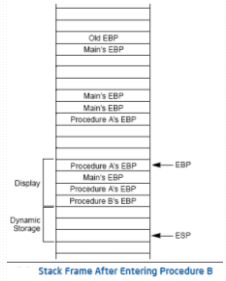


Stack Frame After Entering the MAIN Procedure

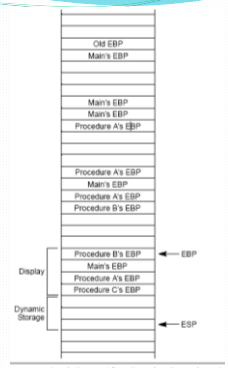


Stack Frame After Entering Procedure A

Instrukcja ENTER



Stack Frame After Entering Procedure B



Stack Frame After Entering Procedure C

Wpływa na flagi: -

Instrukcja LEAVE

leave

Usuwa ramę stosu (utworzoną instrukcją ENTER) przed wyjściem z podprogramu. Przypisuje do wskaźnika stosu rejestr bazowy, następnie zdejmuje rej. basowy ze stosu.

esp:=ebp; pop ebp

leave

Przykład

Sortowanie

```
procedure Sort(var A: array of Integer);

 procedure QuickSort(var A: array of Integer; iLo, iHi: Integer);
 var
  Lo, Hi, Mid, T: Integer;
 begin
  Lo := iLo;
  Hi := iHi;
  Mid := A[(Lo + Hi) div 2];
  repeat
   while A[Lo] < Mid do Inc(Lo);
   while A[Hi] > Mid do Dec(Hi);
   if Lo <= Hi then
   begin
    T := A[Lo];
    A[Lo] := A[Hi];
    A[Hi] := T;
    Inc(Lo);
    Dec(Hi);
   end;
  until Lo > Hi;
  if Hi > iLo then QuickSort(A, iLo, Hi);
  if Lo < iHi then QuickSort(A, Lo, iHi);
 end;

 begin
  QuickSort(A, Low(A), High(A));
 end;
```



```
procedure mysortq(var A:array of integer;iLo,iHi:integer);
asm
    push edi
    push esi
    push ilo
    push ihi
    call @s
    jmp @e

@s: enter 0,0          //sortowanie eax=A; [ebp+8]=ihi; [ebp+12]=ilo
    mov esi,[ebp+8]    //esi=hi:=ihi
    mov edi,[ebp+12]   //edi=lo:=ilo
    mov ecx,esi        //ecx=Mid := A[(Lo + Hi) div 2]
    add ecx,edi
    sar ecx,1
    mov ecx,[eax+ecx*4]   //mid
@r:                    //repeat
    cmp [eax+edi*4],ecx   //while A[Lo] < Mid do Inc(Lo)
    jge @1
    inc edi
    jmp @r
@1: cmp [eax+esi*4],ecx   //while A[Hi] > Mid do Dec(Hi)
    jle @2
    dec esi
    jmp @1
@2: cmp edi,esi           //if Lo <= Hi then
    jnle  @3
    mov edx,[eax+esi*4]   //A[Lo] <-> A[Hi]
    xchg edx,[eax+edi*4]
    mov [eax+esi*4],edx
    inc edi               //Inc(Lo)
    dec esi               //Dec(Hi)
@3: cmp edi,esi           //until Lo > Hi;
    jle @r
    cmp esi,[ebp+12]      //if Hi > iLo then QuickSort(A, iLo, Hi)
    jng @4
    push edi              //zabezpieczenie lo
    push [ebp+12]
    push esi
    call @s
    pop edi
@4: cmp edi,[ebp+8]       //if Lo < iHi then QuickSort(A, Lo, iHi)
    jnl @5
    push edi
    push [ebp+8]
    call @s
@5: leave
    ret 8

@e: pop esi
    pop edi
end;
```



```
SDIMAIN.PAS.147: begin
0046ACFC 55          push ebp
0046ACFD 8BEC        mov ebp,esp
0046ACFF 83C4F4      add esp,-$0c
0046AD02 53          push ebx
0046AD03 56          push esi
0046AD04 894DF8      mov [ebp-$08],ecx
0046AD07 8955FC      mov [ebp-$04],edx
0046AD0A 8BF0        mov esi,eax
0046AD0C 8B5DF8      mov ebx,[ebp-$08]   // Lo := iLo;
0046AD0F 8B4508      mov eax,[ebp+$08]   // Hi := iHi;
0046AD12 8D1418      lea edx,[eax+ebx]  // Mid := A[(Lo + Hi) div 2];
0046AD15 D1FA        sar edx,1
0046AD17 7903        jns $0046ad1c
0046AD19 83D200      adc edx,$00
0046AD1C 8B1496      mov edx,[esi+edx*4]
0046AD1F EB01        jmp $0046ad22
0046AD21 43          inc ebx          //while A[Lo] < Mid do Inc(Lo);
0046AD22 3B149E      cmp edx,[esi+ebx*4]
0046AD25 7FFA        jnle $0046ad21
0046AD27 EB01        jmp $0046ad2a
0046AD29 48          dec eax          //while A[Hi] > Mid do Dec(Hi);
0046AD2A 3B1486      cmp edx,[esi+eax*4]
0046AD2D 7CFA        jl $0046ad29
0046AD2F 3BC3        cmp eax,ebx            //if Lo <= Hi then
0046AD31 7C14        jl $0046ad47
0046AD33 8B0C9E      mov ecx,[esi+ebx*4]    //T := A[Lo];
0046AD36 894DF4      mov [ebp-$0c],ecx
0046AD39 8B0C86      mov ecx,[esi+eax*4]    // A[Lo] := A[Hi];
0046AD3C 890C9E      mov [esi+ebx*4],ecx
0046AD3F 8B4DF4      mov ecx,[ebp-$0c]   // A[Hi] := T;
0046AD42 890C86      mov [esi+eax*4],ecx
0046AD45 43          inc ebx          //Inc(Lo);
0046AD46 48          dec eax          //Dec(Hi);
0046AD47 3BC3        cmp eax,ebx      //until Lo > Hi;
0046AD49 7DD7        jnl $0046ad22
0046AD4B 3B45F8      cmp eax,[ebp-$08]   // if Hi > iLo then
QuickSort(A, iLo, Hi);
0046AD4E 7E13        jle $0046ad63
0046AD50 8B550C      mov edx,[ebp+$0c]
0046AD53 52          push edx
0046AD54 50          push eax
0046AD55 8BC6        mov eax,esi
0046AD57 8B45F8      mov ecx,[ebp-$08]
0046AD5A 8B55FC      mov edx,[ebp-$04]
0046AD5D E89AFFFFFF  call QuickSort
0046AD62 59          pop ecx
0046AD63 3B5D08      cmp ebx,[ebp+$08]   // if Lo < iHi then
QuickSort(A, Lo, iHi);
0046AD66 7D15        jnl $0046ad7d
0046AD68 8B450C      mov eax,[ebp+$0c]
0046AD6B 50          push eax
0046AD6C 8B4508      mov eax,[ebp+$08]
0046AD6F 50          push eax
0046AD70 8BC6        mov eax,esi
0046AD72 8BCB        mov ecx,ebx
0046AD74 8B55FC      mov edx,[ebp-$04]
0046AD77 E880FFFFFF  call QuickSort
0046AD7C 59          pop ecx
0046AD7D 5E          pop esi    //end;
0046AD7E 5B          pop ebx
0046AD7F 8BE5        mov esp,ebp
0046AD81 5D          pop ebp
0046AD82 C20400      ret $0004
```

Mnożenie BCD

```
  222                5678
  5678             *    3
*    3              ------
------               5814
17034              +1122
                   ------
                   17034
```


Mnożenie BCD

```
void m_BCD(char*a,char*b ,int n, int m, char*w) {
__asm{
    pushad
    pushf
    mov     ecx, m      //liczba cyfr mnożnika - m
    mov     edi, b      //adres mnożnika
    add     edi, ecx
    mov     edx, w      //adres wyniku
    add     edx, ecx
po_m:
    push    ecx         //zabezpiecz m licznik cyfr
                        //pozostałych cyfr mnożnika
    dec     edx         //m do o cyfra wyniku
    dec     edi         //m do o cyfra mnożnika
    mov     bl, [edi]   //do bl
    mov     ecx, n      //ilość cyfr mnożnej - m
    mov     esi, a      //adres mnożnej
iloczyn:
    mov     al, [esi]   //cyfra mnożnej - od najstarszej
    mul     bl          //razy cyfra mnożnika
    aam
    push    ax          //ah - przeniesienie z
                        //mnożenia, al - wynik na stos
    inc     esi
    Loop    iloczyn
    mov     ecx, n
    pop     ax          //najmłodsze ze stosu
    add     al, [edx+ecx] //i dodajemy cyfrę do wyniku
    aaa
    mov     [edx+ecx],al
    dec     ecx
    jz      one_n       //jeśli tylko jednocyfrowa mnożna
SumaBCD:
    mov     bl, ah      //przeniesienie do bl
    pop     ax          //następna cyfra i przeniesienie ze
    stosu
    add     al, [edx+ecx] //dodajemy kolejną cyfrę wyniku
    aaa
    add     al,bl       //plus poprzednie przeniesienie
    aaa
    mov     [edx+ecx], al //i do wyniku
    loop    SumaBCD
one_n:
    mov     [edx], ah   //ostatnie przeniesienie do wyniku
    pop     ecx         //ilość pozostałych cyfr mnożnika
    loop    po_m
    popf
    popad
    }
}
// wynik musi być wyzerowany przed wywołaniem
```

Architektura procesora

Procesor 8086 - rejestry

Rejestry

- **AX (ang. Accumulator)** - jest wykorzystywany głównie do operacji arytmetycznych i logicznych.
- **BX (ang. Base Registers)** - rejestr bazowy, głównie wykorzystywany przy adresowaniu pamięci.
- **CX (ang. Counter Registers)** – rejestr często wykorzystywany jako licznik, np. przy instrukcji LOOP.
- **DX (ang. Data Register)** - rejestr danych, wykorzystywany przy operacjach mnożenia i dzielenia, a także do wysyłania i odbierania danych z portów.

Rejestry c.d.

- **SI (ang. Source Index)** - rejestr indeksujący pamięć, wskazuje obszar z którego przesyłane są dane. W połączeniu z DS tworzy adres logiczny DS:SI
- **DI (ang. Destination Index)** - rejestr indeksujący pamięć, wskazuje obszar, do którego przesyłane są dane. W połączeniu z ES, tworzy adres logiczny ES:DI
- **BP (ang. Base Pointer)** - rejestr stosowany do adresowania pamięci.
- **SP (ang. Stack Pointer)** - wskaźnik stosu.

Rejestry c.d.

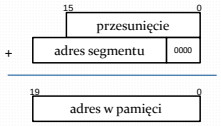
- **IP (ang. Instruction Pointer)** – zawiera adres aktualnie wykonywanej instrukcji, może być modyfikowany przez rozkazy sterujące pracą programu.
- **FLAGS** – rejestr znaczników.

Rejestry c.d. - segmentowe

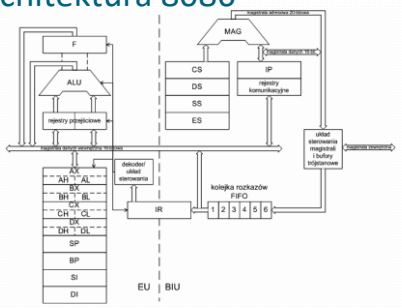
- **CS (ang. Code Segment)** - rejestr informujący o segmencie aktualnie wykonywanego rozkazu. Razem z IP tworzy adres logiczny CS:IP kolejnej instrukcji.
- **DS (ang. Data Segment)** - rejestr informujący o segmencie z danymi.
- **ES (ang. Extra Segment)** - rejestr informujący o segmencie dodatkowym np. przy operacjach przesyłania łańcuchów.
- **SS (ang. Stack Segment)** - rejestr informujący o segmencie stosu.

Adres w trybie rzeczywistym

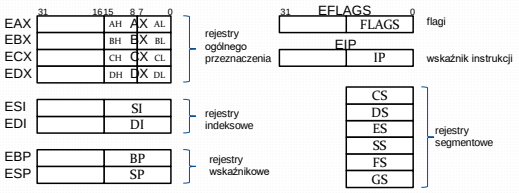
powstaje w wyniku sumowania położenia segmentu i przesunięcia w nim.



Architektura 8086



IA32- rejestry



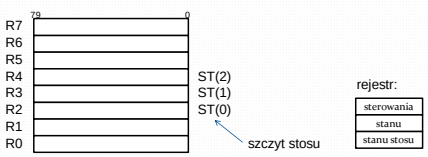
Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
1	1	zarezerwowany	
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
8	TP	flaga umożliwiająca krokowe wykonanie (trap)	X
9	IF	flaga zezwolenia na przerwania (interrupt enable)	X
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S
12, 13	IOPL	poziom uprawnień we/wy (I/O privilege level, od 286)	X
14	NT	nested task flag (od 286)	X
16	RF	flaga wznowienia (resume, od 386)	X
17	VM	flaga trybu Virtual 8086 (od 386)	X
18	AC	alignment check (od 486SX)	X
19	VIF	Virtual interrupt flag (od Pentium)	X
20	VIP	Virtual interrupt pending (od Pentium)	X
21	ID	Identification (od Pentium)	X
3, 5, 15, 22–31	0	zarezerwowany	

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

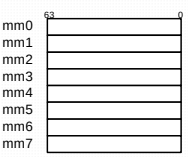
Koprocesor

stos rejestrów



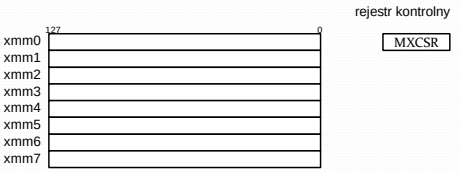
Rejestry MMX

Działają na nich instrukcje całkowitoliczbowe SIMD
Wykorzystują rejestry koprocesora

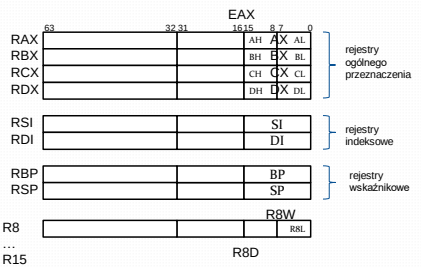


Rejestry XMM

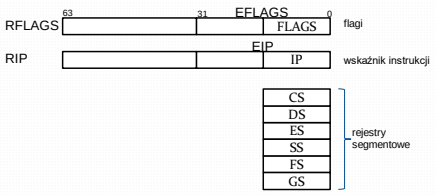
Działają na nich instrukcje zmiennoprzecinkowe SIMD



EM64T- rejestry

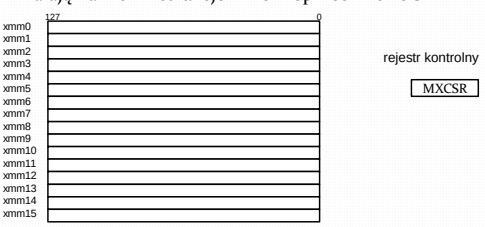


EM64T- rejestry



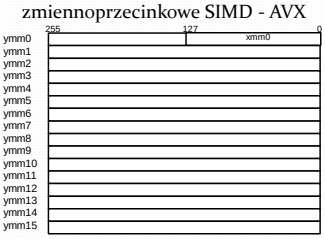
EM64T- rejestry XMM

Działają na nich instrukcje zmiennoprzecinkowe SIMD



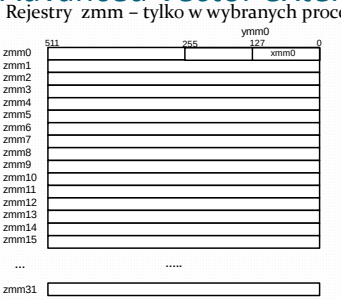
AVX-
Advanced Vector eXtensions

Rejestry ymm - działają na nich instrukcje zmiennoprzecinkowe SIMD - AVX

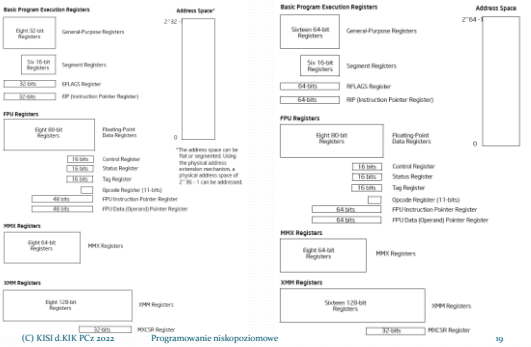


AVX512-
Advanced Vector eXtensions

Rejestry zmm – tylko w wybranych procesorach

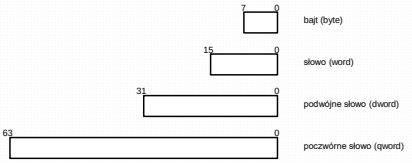


Środowisko 32 i 64 bitowe



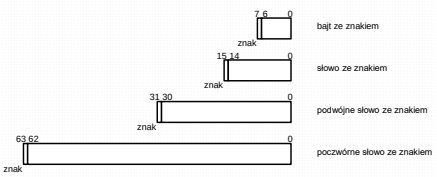
Liczbowe typy danych

Liczby całkowite bez znaku



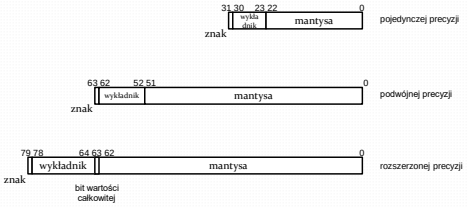
Liczbowe typy danych

Liczby całkowite ze znakiem

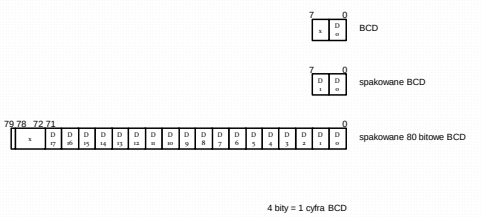


Liczbowe typy danych

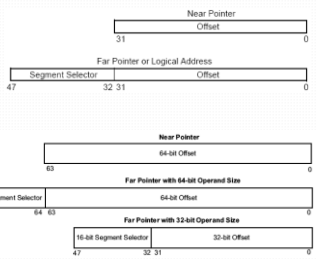
Liczby zmiennoprzecinkowe



Typy BCD

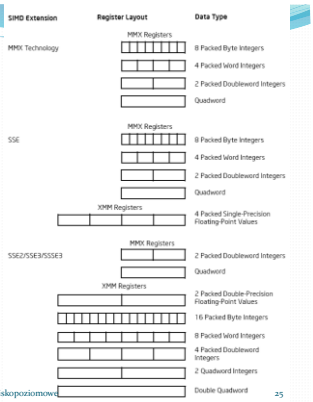


Wskaźniki w trybie 32 i 64 bitowym



SIMD

Rejestry i typy danych

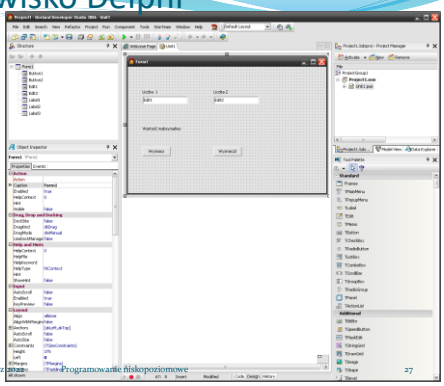


Kilka instrukcji

```
mov   eax,edx
mov   eax,[ebx]
mov   [rdx],rax
mov   eax,zmienna

push  bp
pop   eax

call  podprogram
ret
```

```
add   eax,edx
sub   rax,rbx
mul   ecx   ;edx:eax=eax*ecx
inc   ecx
dec   rcx

cmp   eax,ecx

jz    etykieta
jnz   etykieta
jc    etykieta
jnc   etykieta
```


Środowisko Delphi



Użycie rejestrów

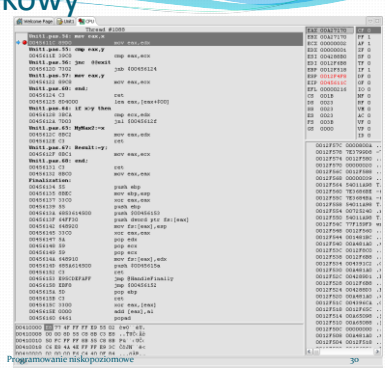
Rejestry	Windows 32	Windows 64
do użycia	EAX, ECX, EDX, ST(0)-ST(7), K0-K7, xMM0-xMM7,	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), K0-K7, xMM0-xMM5,xMM16-xMM31
do zabezpieczenia	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12-R15, xMM6-xMM15
parametry funkcji	cdecl, stdcall, pascal, Gnu C: na stosie, fastcall Microsoft/Gnu : ecx, edx fastcall Borland eax, edx, ecx thiscall Microsoft ecx	RCX, RDX, R8, R9 lub xMM0-xMM3 reszta na stosie.
zwracające wartość funkcji	EAX, EDX, ST(0)	RAX, xMM0

x = X, Y lub Z

Przykład

```
function TForm1.MyMax(x,y:integer):integer;
asm
  mov eax,x
  cmp eax,y
  jnc @@exit
  mov eax,y
@@exit:
//  mov result,eax
end;

function TForm1.MyMax2(x,y:integer):integer;
begin
  if x>y then
    MyMax2:=x
  else
    Result:=y;
end;
```


Kod wynikowy



Programowanie niskopoziomowe

Jarosław Bilski

Struktura przedmiotu

- Wykład
- Laboratorium

Literatura

- **Adam Błaszczyk - Win32ASM. Asembler w Windows, Helion 2004**
- **Randall Hyde - Asembler. Sztuka programowania, Helion 2004**
- **Stanisław Kruk - Asembler w koprocesorze, Mikom 2003**

Literatura c.d.

- **Ryszard Goczyński, Michał Tuszyński – Mikroprocesory 80286, 80386 i i486, Komputerowa Oficyna Wydawnicza „HELP" 1991**
- **Michał Tuszyński, Ryszard Goczyński – Koprocesory arytmetyczne 80287 i 80387 oraz jednostka arytmetyki zmiennoprzecinkowej mikroprocesora i486, Komputerowa Oficyna Wydawnicza „HELP" 1992**

Literatura c.d.

- **Eugeniusz Wróbel – Praktyczny kurs asemblera, wyd. II, Helion 2011**
- **Vlad Pirogow – Asembler, Podręcznik programisty, Helion 2005**

Literatura c.d.

- **Intel® 64 and IA-32 Architectures Software Developer's Manual**
  - **Basic Archtecture**
  - **Instruction Set**
  - **System Programming Guide**

www.intel.com

Narzędzia

- MASM, FASM, ...
- Delphi, Visual Studio, ...

Języki programowania

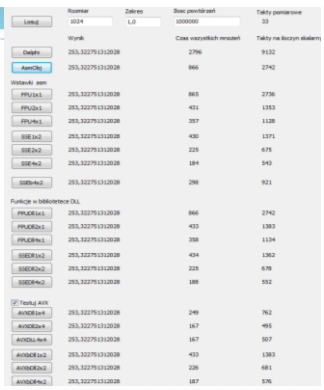
- Wysokiego poziomu – Ada, Basic, C, C++, C#, Fortran, Java, Pascal (Delphi), Python, SQL, ...
- Niskiego poziomu – asemblery – odpowiadają kodowi wykonywanemu przez procesor

Porównanie

	Języki wysokiego poziomu	Języki niskiego poziomu
Trudność programowania	niska	duża
Przenośność	duża	mała
Wykorzystanie możliwości procesora i sprzętu	średnie	duże
Programowanie zadań wymagających czasowo	słabe	dobre
Przejrzystość kodu źródłowego	duża	mała
Przejrzystość kodu wynikowego	bardzo mała	duża
Szybkość tworzenia	duża	mała
Szybkość działania	mała	duża
Rozmiar kodu źródłowego	mały	średni
Rozmiar kodu wynikowego	duży	mały
Zajętość pamięci/dysku	duża	mała

Porównanie

- Iloczyn skalarny



Porównanie

- Rekonstrukcja obrazu – tomograf komputerowy

Opis programu	Ilość wątków	1 iteracja [ms]	20k iteracji [s]	Przyśpie-szenie
Oryginalny	1	688	13760 3h49m20s	-
Asembler x64	1	8,1	162 2m42s	84,938
Asembler x64 wielowątkowy i9-7900X 10r 20t	8 10 16 20	1,119047 0,994545 1,017738 0,930187	22,38095 19,89089 20,35476 18,60375	614,809 691,774 676,009 739,636
i9-9980X 18r	16	0,68025	13,605s	1011,392
NVIDIA 1080Ti - 3584r		1,74473	34,8946	394,330
NVIDIA Titan V – 5120r		0, 70534	14,091075	975,416

Producenci procesorów

- AMD
- Cyrix
- IBM
- **Intel**
- Nec
- Siemens
- Transmeta
- VIA
- Acorn Computers
- HP
- MOS Technology
- Motorola
- Silicon Graphix
- Zilog
- Texas Instruments
- Samsung

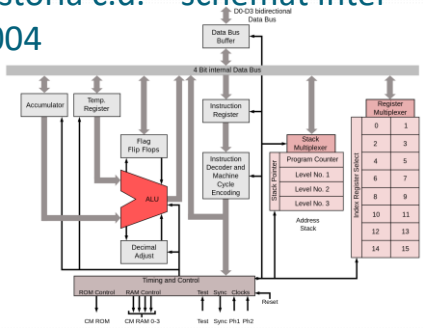
Trochę historii

- **F14 CADC (F-14A Central Air Data Computer) - mikroprocesor zaprojektowany przez Steve'a Gellera i Raya Holta na potrzeby US Navy do myśliwca F-14 Tomcat.**
  - Powstał w czerwcu 1970
  - niezwykle zaawansowany, 20-bitowy układ z techniką potokową
  - istnienie F-14 CADC zostało ujawnione dopiero w 1998 (z powodu tajemnicy wojskowej)

Historia c.d.

- **Intel 4004 - 4-bitowy mikroprocesor zaprojektowany i produkowany przez firmę Intel od 1971**
  - Powszechnie uznany za pierwszy mikroprocesor
  - Maksymalna częstotliwość taktowania - 740 kHz.
  - Osobna pamięć dla programu i danych (tzw. "architektura harwardzka").
  - 46 instrukcji.
  - 16 czterobitowych rejestrów.
  - 3-poziomowy stos.
  - 2300 tranzystorów (technologia produkcji 10 µm).

Historia c.d. – schemat Intel 4004



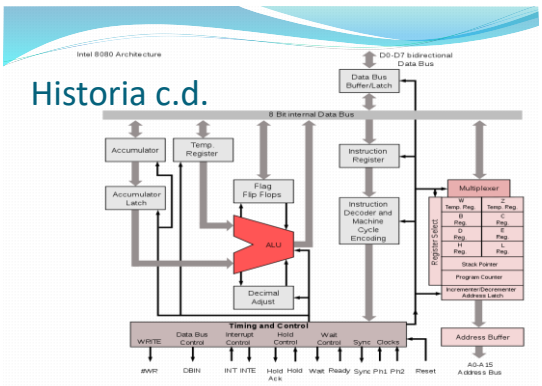
Historia c.d.

- **Intel 8008 – pierwszy mikroprocesor 8-bitowy Intela**
  - obudowa DIP18
  - 8-bitowa magistrala
  - dostęp do większej ilości RAM
  - 3-4 razy więcej mocy obliczeniowej niż procesory 4-bitowe.

Historia c.d.

- **Intel 8080**
  - wyprodukowany przez Intela w kwietniu 1974
  - 8-bitowa szyna danych, pamięć adresowana 16-bitową szyną adresową.
  - słowo 8-bitowe
  - 72 instrukcje
  - bezpośrednie adresowanie pamięci o pojemności do 64 KB
  - arytmetyka dwójkowa i dziesiętna kodowana dwójkowo (BCD)
  - 8 rejestrów programowych dostępnych dla programisty cykl pracy 2µs,
  - zegar zewnętrzny o częstotliwości 2-3 MHz (podstawowy cykl rozkazowy – 4 takty)

Historia c.d.



Historia c.d.

- **Intel 8086 - procesor 16-bitowy wprowadzony w 1978**
  - traktowany jako tymczasowy projekt przejściowy. Intel pokładał wówczas swoje nadzieje w znacznie bardziej zaawansowanym 32-bitowym układzie 8800 (iAPX 432).
  - Głównym konstruktorem był Stephen Morse, który specjalizował się w oprogramowaniu.
    "Gdyby szefostwo Intela chciało, by architektura ta przetrwała wiele generacji i przerodziła się w dzisiejsze procesory, to nigdy nie zleciliby tego zadania jednej osobie"

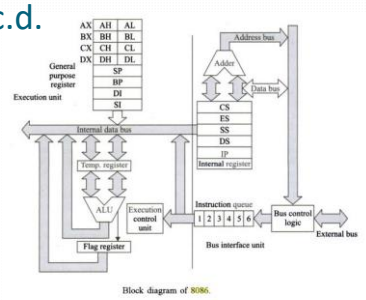
Historia c.d.

- W 1980 IBM rozpoczyna pracę nad komputerem 5150
- Microsoft ma już gotowy interpreter języka Basic, który działał na układach 8086 i 8088
- IBM 5150 staje się standardem
  "Pytaliśmy ich, czy chcą mieć komputer od International Business Machines czy od firmy, która swą nazwę wzięła od owocu"

Historia c.d.

- rozszerzenie listy rozkazów
- rozszerzenie możliwości adresowania operandów
- wprowadzenie segmentacji obszaru pamięci
- mechanizmy przyspieszenia pracy
- mechanizmy dla pracy wieloprocesorowej

Historia c.d.



Block diagram of 8086.

Historia c.d.

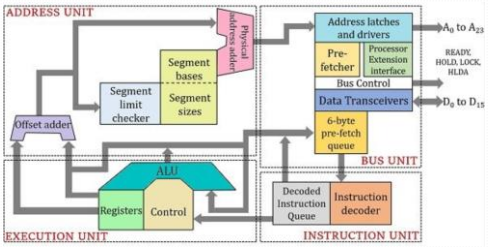
- **Intel 80186 - procesor opracowany w firmie Intel w 1982.**
- posiadał nieco większą wydajność, kilkanaście nowych rozkazów i szybszy zegar
- niektóre instrukcje były wykonywane 10-20 razy szybciej.
- procesor wykorzystywany był głównie w systemach wbudowanych jako mikrokontroler.

Historia c.d.

- **Intel 80286 - 16-bitowy procesor opracowany przez firmę Intel, pokazany po raz pierwszy 1 lutego 1982**
  - mniej więcej dwa razy bardziej wydajny w porównaniu do procesora Intel 8086
  - posiadał 24-bitową szynę adresową mógł adresować aż 16MB pamięci RAM
  - wprowadzono nowe instrukcje,
  - nowy tryb adresowania pamięci (tryb chroniony)

Historia c.d.

- **Intel 80286 - budowa**

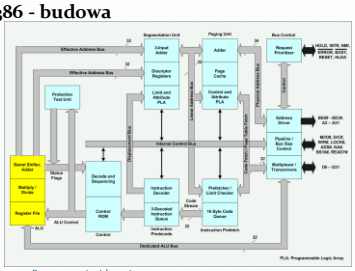


Historia c.d.

- **Intel 80386 - 32-bitowy procesor opracowany przez firmę Intel w 1986**
  - pierwszy 32-bitowy procesor z rodziny x86. Architektura tego procesora została opracowana jeszcze zanim Intel wypuścił na rynek procesory poprzedniej serii 286, jednak procesor był zbyt skomplikowany, aby go w tamtym czasie wyprodukować.
  - 32-bitowa magistrala adresowa oraz 32-bitowa magistrala danych
  - rozszerzone do 32-bitów rejestry
  - nowe tryby adresowania
  - praca w trzech trybach: rzeczywistym, chronionym i wirtualnym
  - dodanie do procesora jednostki MMU

Historia c.d.

- **Intel 80386 - budowa**



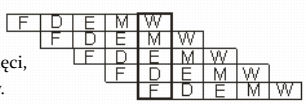
Historia c.d.

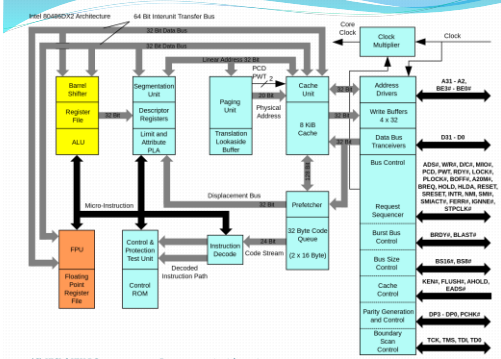
- **Intel 80486 - poprawna nazwa handlowa i486**
  - kilka (7) dodatkowych instrukcji
  - cache na dane i instrukcje
  - zintegrowany koprocesor arytmetyczny x87
  - poprawiony interfejs szyny danych
  - zastosowano pięciostopniowy potok.
  - usprawnienia spowodowały, że i486 był mniej więcej dwukrotnie szybszy od podobnie taktowanego 80386

Historia c.d.

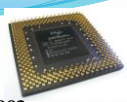
Wykonywanie instrukcji w potoku. Fazy:

- F - pobieranie instrukcji,
- D - dekodowanie,
- E - wykonanie,
- M - odwołanie do pamięci,
- W - zapisanie wyników.





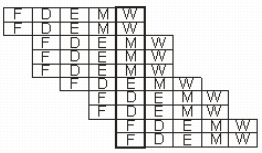
Historia c.d.



- **Pentium – mikroprocesor się 22 marca 1993**
  - architektura superskalarna (wykonywanie kilku instrukcji w kilku potokach)
  - 64-bitowa szyna danych
  - jednostka *branch prediction* do przewidywania skoków (80% skuteczność)
  - przeprojektowany koprocesor (5-6x wydajniejszy niż w i486)

Historia c.d.

- **Superskalarność** – cecha procesorów oznaczająca możliwość wykonywania kilku instrukcji (rozkazów maszynowych) jednocześnie, uzyskiwana poprzez zwielokrotnienie jednostek wykonawczych.



Historia cd.

- **Pentium Pro - mikroprocesor szóstej generacji należący do rodziny x86 (październik 1995)**
  - Podział kodu x86 na mikrorozkazy
  - Wykonywanie poza kolejnością
  - Wykonywanie spekulatywne
  - Dodatkowy potok ("pipeline") dla prostych instrukcji.

Historia cd.

Mikroprogram, mikro-kod to program implementujący listę rozkazów procesora. Rozkaz kodu maszynowego jest ciągiem mikrorozkazów (mikroinstrukcji), jest ciągiem operacji sprzętowych wykonywanych wewnątrz procesora.

Mikroprogram jest pisany przez konstruktorów procesorów w trakcie ich projektowania.

Historia cd.

Wykonywanie poza kolejnością (out-of-order execution) – zdolność mikroprocesora superskalarnego do zmiany kolejności wykonywania instrukcji, tak, aby maksymalnie wykorzystać jego jednostki wykonawcze (obliczeniowe) i równolegle, w kilku potokach, wykonać jak najwięcej instrukcji. Powoduje to przyśpieszenie wykonywania programów.

Zmiana kolejności jest możliwa tylko wówczas, gdy instrukcje są od siebie niezależne.

Historia cd.

Wykonywanie spekulatywne (speculative execution) – zdolność mikroprocesorów, przetwarzających rozkazy potokowo, do wykonywania rozkazów znajdujących się po skoku warunkowym, gdy jeszcze nie wiadomo, czy skok ten zostanie wykonany, i które rozkazy zostaną po nim wykonane. Jeśli jednak wybór był niewłaściwy, uzyskane wyniki zostaną pominięte, a potok wyczyszczony.

Wykonywanie spekulatywne występuje zwykle łącznie z mechanizmem **przewidywania skoków** (branch prediction).

Historia c.d.

- **Pentium II – mikroprocesor zaprezentowany 7 maja 1997**
  - dodatkowe instrukcje MMX (MultiMedia eXtensions lub Matrix Math eXtensions)
  - poprawiona obsługa programów 16-bitowych
  - cache pierwszego poziomu (L1) dla kodu i danych: 16 kB

Historia c.d.

- **Pentium III - procesor w 32-bitowej architekturze Intela (IA-32).**
  - architektura RISC (Reduced Instruction Set Computers)
  - rozmiar pamięci cache pierwszego poziomu (L1) dla kodu: 16 KB
  - liczba etapów przetwarzania rozkazu (w potoku): 12
  - liczba jednostek zmiennoprzecinkowych: 1 (z potokowaniem)
  - liczba jednostek całkowitoliczbowych: 6 potoków
  - liczba jednostek MMX:2
  - Instrukcje SSE (Streaming SIMD Extensions)
  - możliwość pracy w systemie wieloprocesorowym (do 2 procesorów).

Historia c.d.

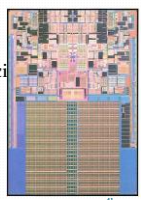
- **Pentium 4 – siódma generacja procesorów firmy Intel (od 20 listopada 2000)(wiele wersji)**
  - architektura NetBurst
  - instrukcje SSE2, w nowszych wersjach jądra – SSE3
  - niektóre wersje posiadają wbudowaną wielowątkowość (HyperThreading)
  - zwiększona pamięć poziomu L2
  - pojawia się technologia EM64T (2003)
  - pierwszy procesor dwurdzeniowy

Historia c.d.

- **Intel Core 2 to ósma generacja mikroprocesorów firmy Intel w architekturze x86**
  - mikroarchitektura Intel Core
  - wysoki współczynnik IPC (Instructions Per Cycle) - około 3,5
  - wspólna pamięć cache dla obu rdzeni procesora
  - EM64T,
  - technologia wirtualizacji,
  - XD bit (eXecute Disable – wyłącza możliwość wykonywania instrukcji z oznaczonych stron),
  - ulepszoną technologię SpeedStep,
  - wersja czterordzeniowa

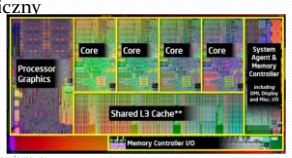
Historia c.d.

- **Intel Core i7**
- modułowa budowa
- ośmiordzeniowy Nehalem składa się z 731 milionów tranzystorów
- SSE 4.2.
- technologia współbieżnej wielowątkowości
- dynamiczne zarządzanie zasilaniem
- wbudowanie kontrolera pamięci RAM
- technologia Quick-Path



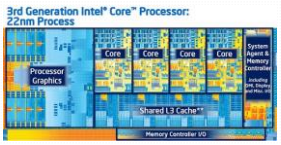
Historia c.d.

- **Intel Core i7 – 2 generacja - Sandy Bidge**
- modułowa budowa
- 32-nanometrowy proces
- 6 jednostek wykonawczych
- wbudowany układ graficzny
- instrukcje AVX
- Turbo Boost 2.0
- pamięć cache L3



Historia c.d.

- **Intel Core i7 – 3 generacja - Ivy Bridge**
- modułowa budowa
- 22-nanometrowy proces (tanzystory 3D)
- wbudowany układ graficzny Intel HD Graphics
- instrukcje AVX
- gen. liczb losowych
- PCI Express 3.0

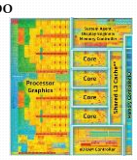


Historia c.d.

- **Intel Core i7 – 4 generacja - Haswell**
- podniesiona wydajność pamięci cache
- zwiększona wydajność i energooszczędność
- 8 jednostek wykonawczych
- instrukcje AVX2, FMA3
- rozbudowany układ graficzny
- wsparcie Direct3D 11.1 i OpenGL 4.0

Historia c.d.

- **Intel Core i7 – 5 generacja - Broadwell**
- technologia 14 nm
- zwiększona wydajność i energooszczędność
- obsługa pamięci DDR3L
- rozbudowany układ graficzny Iris Pro 6200
- 128 MB pamięci podręcznej eDRAM
- wsparcie Direct3D 11.2 i OpenGL 4.4



Historia c.d.

- **Intel Core i7 – 6 generacja - Skylake**
- technologia 14 nm
- obsługa pamięci DDR3L i DDR4
- instrukcje AVX 2
- Instrukcje AVX512 – 32 rejestry ZMM – 512 bitowe – w wersji XEON
- wsparcie Direct3D 12
- wsparcie dla Thunderbolt 3.0

Historia c.d.

- **Intel Core i7 – 7 generacja – Kaby Lake**
- technologia 14 nm+
- zwiększenie częstotliwości zegara
- zwiększenie wydajności układu graficznego
- wsparcie Intel Optane Memory storage caching

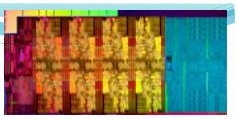
Historia c.d.

- **Intel Core i7 – 8 generacja – Coffee Lake**
- technologia 14++ nm
- zwiększenie liczby rdzeni do 6
- zwiększenie częstotliwości zegara
- zwiększenie wydajności układu graficznego

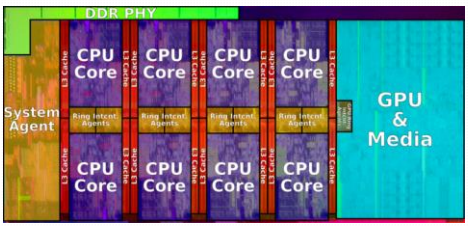
Historia c.d.

- **Intel Core i7 – 9 generacja – Coffee Lake**
- technologia 14++ nm
- zwiększenie liczby rdzeni do 8
- zwiększenie rozmiaru pamięci cache L3
- AVX512 – 32 rejestry ZMM – 512bitowe – w wersji X

Historia c.d.

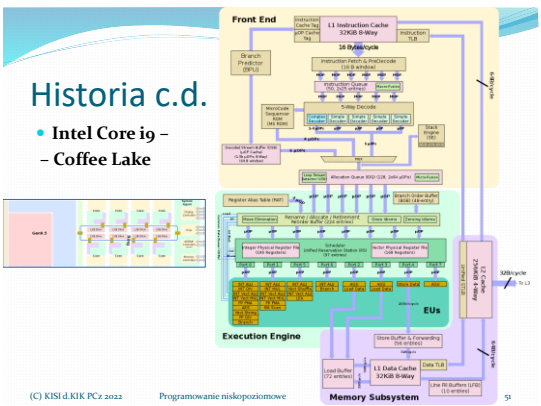


- **Intel Core i9 – 9 generacja – Coffee Lake**



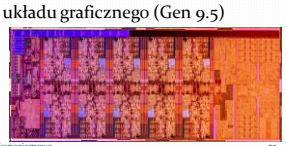
Historia c.d.

- **Intel Core i9 –**
- **– Coffee Lake**



Historia c.d.

- **Intel Core i9 – 10 generacja – Comet Lake**
- zwiększenie maksymalnej częstotliwości taktowania do 5,3 GHz
- zwiększenie liczby rdzeni do 10
- zwiększenie rozmiaru pamięci cache L3
- zwiększenie wydajności układu graficznego (Gen 9.5)



Historia c.d.

- **Intel Core i9 – 11 generacja – Rocket Lake**
- zwiększenie IPC o 10 do 19%
- instrukcje AVX512 i DL (Deep Learning)
- zwiększenie rozmiaru pamięci cache L1D – 48KiB
- nowy wydajny układ graficzny (Xe-LP Gen 12), DisplayPort 1.4a, HDMI 2.0b
- PCI Expres 4.0

Historia c.d.

- **Intel Core i9 – 12 generacja – Alder Lake**
- procesor hybrydowy – wydajne rdzenie dwuwątkowe i rdzenie energooszczędne - do 8P i 8E
- proces 10 nm SuperFin (Intel 7 proces)
- zwiększenie IPC
- 12 jednostek wykonawczych
- nowe instrukcje AVX-VNNI
- zwiększenie rozmiaru pamięci cache L3
- układ graficzny Xe-LP Gen 12.2 (32-96EU)
- PCI Expres 5.0

Co dalej?

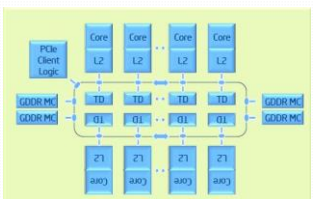
- **Raptor Lake**
- **Meteor Lake**
- **Arrow Lake**
- **Lunar Lake**
- **Nova Lake**

Zestawienie

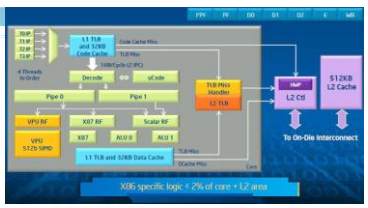
Nazwa procesora	Rok	Maks. częstotliwość taktowania (w momencie wprowadzenia, MHz)	Liczba tranzystorów (mln.) Dane szacunkowe
Intel 8086	1978	8	0,029
Intel 80186	1982	12	0,055
Intel 80286	1982	12,5	0,134
Intel 80386	1985	20	0,275
Intel i486	1989	25	1,2
Pentium	1993	66	3,1
Pentium Pro	1995	200	5,5
Pentium MMX	1995	233	4,5
Pentium II	1997	266	7
Pentium III	1999	500	8,2
Pentium 4	2000	1500	42
Pentium 4 z EM64T	2003	2200	228
Pentium D	2004	3200	230
Intel Core 2	2006	3000	321
Intel Core i7	2008	3400	731
Intel Core i7 2600K	2011	3400-3800	995
Intel Core i7 3770K	2012	3500-3900	1400
Intel Core i7 6700K	2015	4000-4200	1750
Intel Core i9 9900K	2018	3600-5000	<3000
Intel Core i9 10900K	2020	3700-5100	7400
Intel Xeon Phi KNM – 72r	2017	1500-1600	8000

Intel Xeon Phi - Knights Corner

- do 61 rdzeni połączonych w dwukierunkowy pierścień



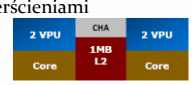
Xeon Phi

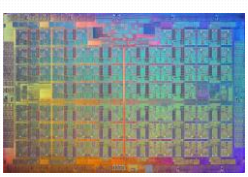


- rdzenie wykonują instrukcje SIMD - FMA3 na danych 512 bitowych, co oznacza, że jedna instrukcja przetwarza osiem zestawów danych podwójnej precyzji albo 16 pojedynczej precyzji - zwiększa to jeszcze stopień zrównoleglenia wykonywanych operacji.
- każdy rdzeń jest czterowątkowy,
- rdzenie posiadają dużą pamięć podręczną - 512KB.

Intel Xeon Phi - Knights Landing

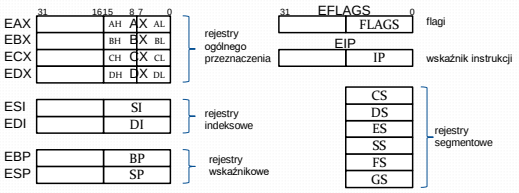
- do 72 rdzeni połączonych pierścieniami





Tryby adresowania
Instrukcje przesyłania

IA32- rejestry



Rejestr flag

Rejestr flag w architekturze Intel x86			
bit	Skrót/wartość	opis	typ
0	CF	flaga przeniesienia (carry)	S
2	PF	flaga parzystości (parity)	S
4	AF	flaga wyrównania (adjust)	S
6	ZF	flaga zera (zero)	S
7	SF	flaga znaku (sign)	S
10	DF	flaga kierunku (direction)	C
11	OF	flaga przepełnienia (overflow)	S

S: Znacznik stanu
C: Znacznik kontrolny
X: Znacznik systemowy

Format rozkazów

```
etykieta:    mnemonik    argument1, argument2    ;komentarz
etykieta:    mnemonik    cel, źródło             ;komentarz

etykieta:    mnemonik
             mnemonik    argument1
             mnemonik    argument1, argument2
Np.:
        ret
        pop     eax
        mov     edx,ecx  ;zapamiętaj licznik
        mov     rax,1001
```


Tryby adresowania - rejestrowy

Argumentem instrukcji jest rejestr:

```
push   ebx

mov    edx,ebx

inc    ecx

dec    r9
```


Tryby adresowania - prosty - natychmiastowy

Argumentem instrukcji jest wartość (zawiera się w kodzie rozkazu):

```
mov    al, 5

mov    r10d,32

mov    edi, offset tabela

jnz    petla
```

Tryby adresowania - bezpośredni

Argumentem instrukcji jest adres w pamięci (wskaźnik):

```
mov   al, [1234ec5fh]
mov   edi, tabela    ;pobiera pierwszy element
mov   zmienna, rdx
```


Tryby adresowania - pośredni - rejestrowy

Argumentem instrukcji jest rejestr – wskaźnik:

```
mov   al, [rcx]
mov   edi, [ebx]
mov   [edi], edx
```


Tryby adresowania - pośredni - bazowy

Argumentem instrukcji jest wskaźnik:

```
mov   al, [ebx+5]
mov   edi, [ebx+tablica]
mov   [rbp+8], rdx
```


Tryby adresowania - pośredni - indeksowy

Argumentem instrukcji jest rejestr – wskaźnik:

```
mov   al, [esi]
mov   edi, [esi*4+tablica]
mov   [rdi*8+tablica], rdx
```


Tryby adresowania - pośredni – bazowo-indeksowy

Argumentem instrukcji jest wskaźnik:

```
mov   al, [ebx+esi+3]
mov   edi, [ebx+eax*4]
mov   [rbp+rdi*8+tablica], rdx
```


Wielkość danych

Można określić wielkość stosowanych danych:

```
mov   al, byte ptr [ebx+esi+3]
mov   cx, word ptr [ebx+eax*4]
mov   dword ptr [ebp+edi*4+tablica], edx
mov   qword ptr [rbp+rdi*8+tablica], rdx

inc    byte ptr [ebx+esi+3]
dec    word ptr [ebx+eax*4]
inc    dword ptr [ebp+edi*4+tablica]
```

Przedrostki segmentowe

Można podać segment do danych:

```
mov   al, es:byte ptr [ebx+esi+3]
mov   cx, cs:word ptr [ebx+eax*4]
mov   ss:[ebp+4], edx
```

Przyporządkowanie rejestrów
esp, ebp: ss
eax, ebx, ecx, edx, edi, esi: ds.
eip: cs
Analogicznie rejestry 16 i 64 bitowe.

Instrukcje przesyłania

MOV	przesyła dane między rejestrami, pamięcią
XCHG	zamień
BSWAP	zamień bajty
XADD	wymień i dodaj
CMPXCHG	porównaj i wymień
CMPXCHG8(16)B	porównaj i wymień 8(16) bajtów
PUSH	wyślij na stos
POP	zdejmij ze stosu
PUSHF/PUSHFD/PUSHFQ	wyślij na stos flagi
POPF/POPFD/POPFQ	zdejmij ze stosu flagi
PUSHA/PUSHAD	wyślij rejestry na stos
POPA/POPAD	zdejmij rejestry ze stosu
CWD/CDQ/CQO	konwertuj word na dword/dword na qword
CBW/CWDE /CDQE	konwertuj byte na word/word na doubleword w rejestrze EAX/ doubleword na qouadword w RAX
MOVSX/MOVSXD	prześlij i rozszerz znakiem
MOVZX	prześlij i rozszerz zerem

Wpływa na flagi: -

Instrukcja MOV

```
mov   cel, źródło
```

Przesyła zawartość źródła do miejsca przeznaczenia (cel).

```
mov   al, bl
mov   [ebp+4], edx
mov   zmienna, eax
mov   rcx,licznik
mov   [ebp+edi*4+tablica], edx
```


Wpływa na flagi: -

Instrukcja XCHG

```
xchg      cel, źródło
```

Zamienia zawartość źródła i celu.

```
xchg      ax, zmienna
xchg      ecx, [epb+4]
xchg      rax, r12
```


Wpływa na flagi: -

Instrukcja BSWAP

```
bswap     rejestr
```

Zamienia bajty w argumencie – 32/64 bity.

```
bswap     eax
bswap     rdx
```

przed

12	c4	7f	de

po

de	7f	c4	12

Wpływa na flagi: OSZAPC

Instrukcja XADD

```
xadd   cel, źródło
```

Zamienia zawartość źródła i celu(8/16/32/64 bity), a ich sumę umieszcza w miejscu przeznaczenia (cel).

```
xadd   al,bl
xadd   eax,zmienna
xadd   edx,[ebx+esi*4]
xadd   rcx,r8
```

Wpływa na flagi: OSZAPC

Instrukcja CMPXCHG

CMPXCHG arg1, arg2

Działanie:

if acc = arg1 then
    arg1 = arg2
else
    acc = arg1

acc = al, ax, eax, rax
arg2 - rejestr

Wpływa na flagi: OSZAPC

Instrukcja CMPXCHG8(16)B

CMPXCHG8(16)B cel

Działanie:

if (E(R)DX:E(R)AX = cel) then
    cel = e(r)cx:e(r)bx
else
    e(r)dx:e(r)ax = cel

Wpływa na flagi: -

Instrukcja PUSH

push arg

Przesyła zawartość argumentu na stos.

```
push    eax
push    rdx
push    ds
push    1234
```


Wpływa na flagi: -

Instrukcja POP

pop cel

Przesyła zawartość stosu do celu.

```
pop    bx
pop    ecx
pop    rdx
pop    [edx+edi+4]
```


Wpływa na flagi: -

Instrukcja PUSHF/PUSHFD/PUSHFQ

pushf/pushfd/pushfq

Przesyła zawartość Flag/Eflag/Rflag na stos.

```
pushf
pushfd
pushfq
```


Wpływa na flagi: OSZAPC

Instrukcja POPF/POPFD/POPFQ

popf/popfd/popfq

Pobiera zawartość Flag/Eflag/Rflag ze stosu.

```
popf
popfd
popfq
```

Wpływa na flagi: -

Instrukcja PUSHA/PUSHAD

pusha/pushad

Przesyła zawartość di, si, bp, bx, dx, cx, ax / edi, esi, ebp, ebx, edx, ecx, eax na stos.

Instrukcja nie działa w trybie 64-bitowym.

```
pusha
pushad
```


Wpływa na flagi: -

Instrukcja POPA/POPAD

popa/popad

Przesyła zawartość stosu do di, si, bp, bx, dx, cx, ax / edi, esi, ebp, ebx, edx, ecx, eax.

Instrukcja nie działa w trybie 64-bitowym.

```
popa
popad
```


Wpływa na flagi: -

Instrukcja CWD/CDQ/CQO

CWD/CDQ/CQO

Konwertuje z zachowaniem znaku word na doubleword / doubleword na quadword / quadword na octaword (ax na dx:ax, eax na edx:eax, rax na rdx:rax).

```
cwd
cdq
cqo
```


Wpływa na flagi: -

Instrukcja CBW/CWDE/CDQE

CBW/CWDE

Konwertuje byte (AL) na word(AX) / word(AX) na doubleword (EAX) / doubleword (EAX) na quadword (RAX) z uwzględnieniem znaku.

```
cbw
cwde
cdqe
```


Wpływa na flagi: -

Instrukcja MOVSX/MOVSXD

movsx cel, źródło

Przesyła zawartość źródła do rejestru celu z uwzględnieniem znaku. Cel posiada 2/4/8 razy więcej bitów.

```
movsx     eax, bl
movsx     cx, al
movsxd    r8, edx     ;movsxd tylko dla 32 na 64
```


Wpływa na flagi: -

Instrukcja MOVZX

movzx cel, źródło

Przesyła zawartość źródła do rejestru celu z dopisaniem na starszych bitach zer. Cel posiada 2/4/8 razy więcej bitów. Źródło 8/16 bitów.

```
movzx     eax, bl
movzx     cx, al
```

Przykład

Wypełnienie wartościami od 0 do 255 tabeli bajtów

```
        mov   ecx, 256
        mov   eax, 0
        mov   edi, 0
p1:     mov   [edi+tabela], al
        inc   edi
        inc   eax
        dec   ecx
        jnz   p1
```


Przykład

Przepisanie wartości integer (32 bity) z tabeli tab1 do tabeli tab2.

```
        mov   rcx, 65536
        mov   rdi, 0
p1:     mov   eax, [tab1+4*rdi]
        mov   [tab2+4*rdi], eax
        inc   rdi
        dec   rcx
        jnz   p1
```


Przykład

Przepisanie wartości int (32 bity) z tabeli tab1 do tabeli tab2 z konwersją na int64. rcx – adres tab1, rdx - adres tab2, r8 – liczba elementów.

```
p1:     movsxd    rax, dword ptr[rcx+4*r8-4]
        mov       [rdx+8*r8-8], rax
        dec       r8
        jnz       p1
```


Przykład

Zamiana zawartości zmiennych a i b typu int64.

```
        mov   rax, a
        xchg  rax, b
        mov   a, rax

        push  a
        push  b
        pop   a
        pop   b
```

Instrukcje typu SIMD

Instrukcje typu SIMD

Single Instruction Multiple Data - przetwarzanych jest wiele strumieni danych przez jeden wykonywany program – cecha tzw. komputerów wektorowych.

Instrukcje SIMD dzieli się na:

- MMX (MultiMedia eXtensions lub Matrix Math eXtensions) - liczby całkowite,
- SSE (Streaming SIMD Extensions) - liczby zmiennoprzecinkowe.

Instrukcje typu MMX

MMX

- wprowadzone w 1997 przez Intela dla procesorów Pentium MMX.
- Przykłady zastosowań:
  - wyświetlanie grafiki trójwymiarowej: przekształcenia geometryczne, cieniowanie, teksturowanie;
  - dekodowanie obrazów JPEG i PNG;
  - dekodowanie i kodowanie filmów MPEG (m.in. wyznaczanie transformat DCT i IDCT);
  - filtrowanie sygnałów: obrazów statycznych, filmów, dźwięku;
  - wyświetlanie grafiki dwuwymiarowej (blue box, maskowanie, przezroczystość);
  - wyznaczanie transformat: Haara, FFT.

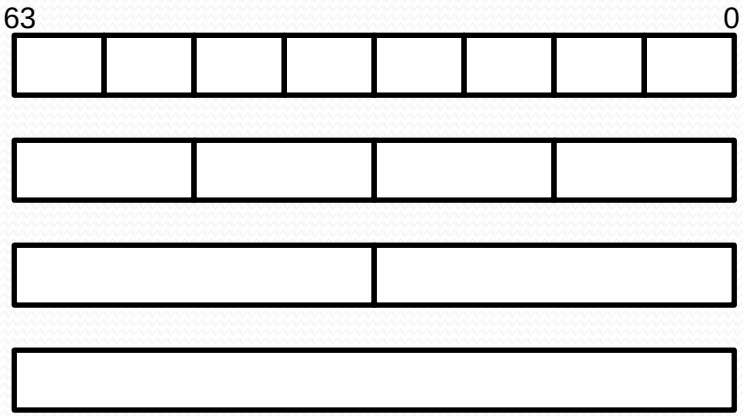
Rejestry MMX

- 8 rejestrów 64 bitowych oznaczanych jako MM0, …, MM7,
- wykorzystują rejestry koprocesora – młodsze 64 bity (mantysa),
- odczyt i zapis wartości, powoduje **wzięcie w użycie zawartości wszystkich rejestrów koprocesora**, nie można mieszać obliczeń MMX z obliczeniami w koprocesorze, po zakończeniu obliczeń MMX należy zwolnić rejestry.

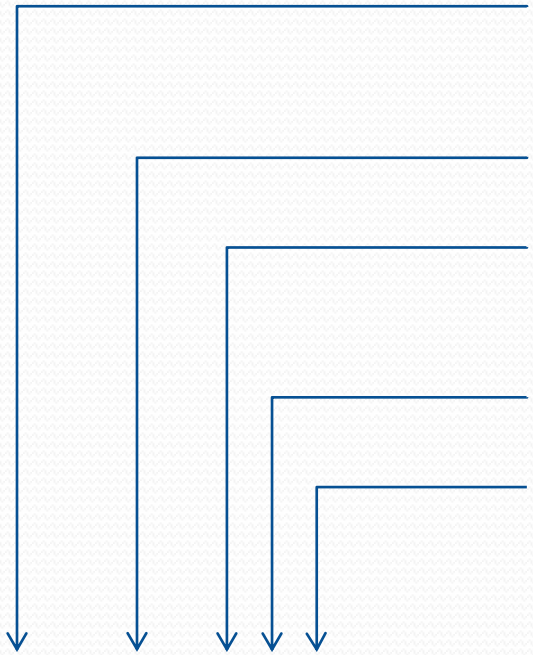
Typy danych

MMX wprowadził nowe wektorowe (macierzowe lub tablicowe) typy danych (ang. packed, czyli dosłownie spakowane, upakowane). „Spakowanie" polega traktowaniu danych 64-bitowych jako składających z odrębnych elementów o tej samej wielkości:

- 8 × 8 bitów (packed byte),
- 4 × 16 bitów (packed word),
- 2 × 32 bity (packed dword),
- 1 × 64 bity (quad word).



Budowa rozkazów



- Mnemoniki prawie wszystkich rozkazów MMX rozpoczynają się od litery **p** (od słowa packed);
- 3-4 literowy skrót wykonywanego działania (np. add, sub, mul);
- litera **s** lub **u** określa, że działanie jest wykonywane na liczbach ze znakiem (signed) lub bez znaku (unsigned);
- litera **s** oznacza operację wykonywaną z nasyceniem;
- rozmiar elementu wektora: **b** - bajt (8 bitów), **w** - słowo (16 bitów), **d** - podwójne słowo (32 bity).

paddusb

- Rozkaz **paddusb** wykonuje równoległe (p) dodawanie (add) bez znaku (u) z nasyceniem (s) liczb o rozmiarze bajtu (b)

Nasycenie

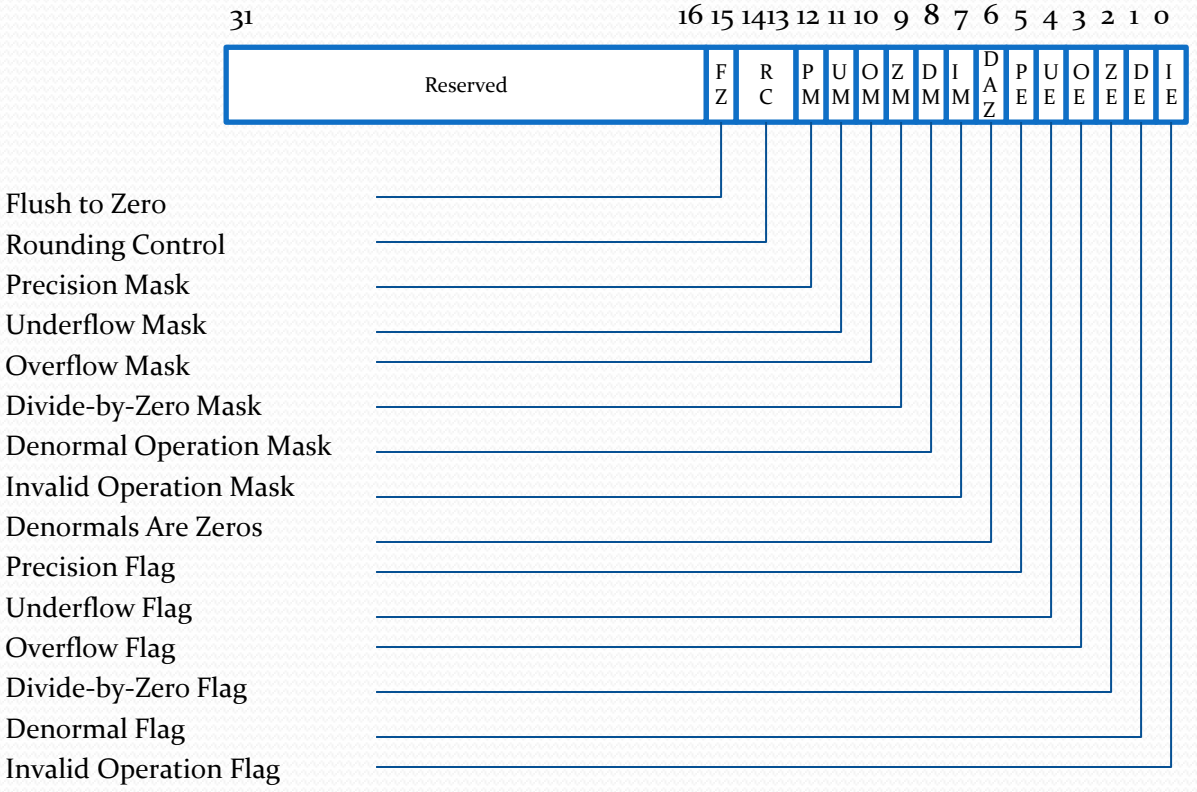
Zakresy liczb (dla 8 bitów):

- bez znaku 0 … 2^n-1 (0 … 255)
- ze znakiem – 2^{n-1} … 2^{n-1}-1 (-128 … 127)

- Jeśli wynik jest mniejszy od najmniejszej liczby z zakresu jest ustawiany na tę liczbę.
- Jeśli wynik jest większy od największej liczby z zakresu jest ustawiany na tę liczbę.

- Dla operacji bez nasycenia wynik jest obcinany do odpowiedniej ilości bitów.

Rejestr MXCSR



Operacje przesłania

- MOVD przesłanie podwójnego słowa
- MOVQ przesłanie poczwórnego słowa

movd/movq cel, źródło

Przesłanie podwójnego/poczwórnego słowa do/z rejestru mmx. Przy zapisie podwójnego słowa bity 32-63 rejestru mmx wypełniane są zerami.

Operacje konwersji

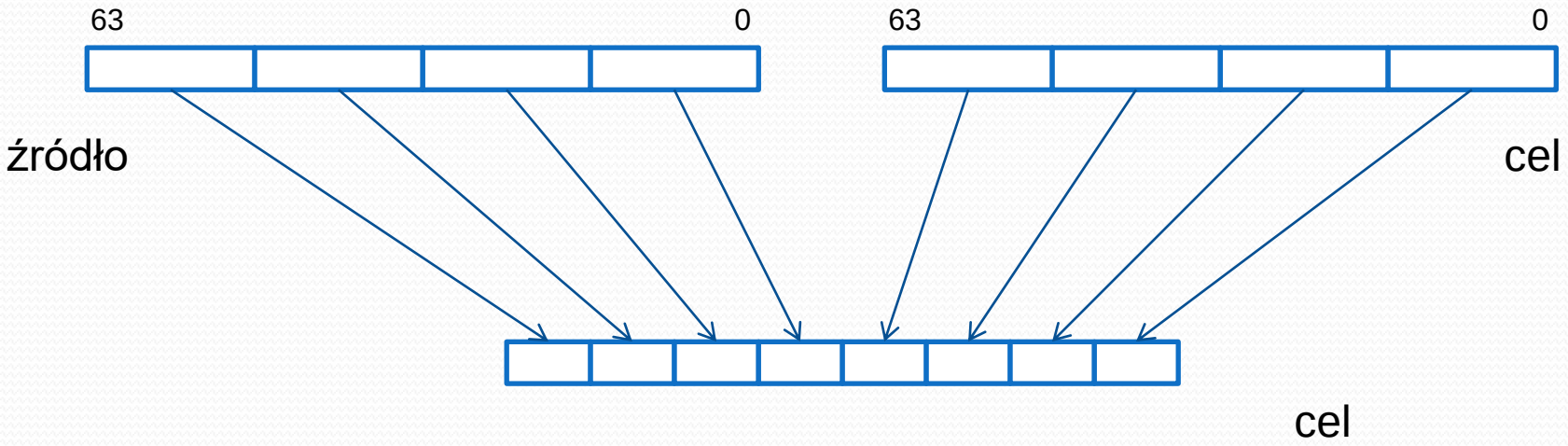
- PACKSSWB pakowanie z nasyceniem słów ze znakiem do bajtów
- PACKSSDW pakowanie z nasyceniem podwójnych słów ze znakiem do słów
- PACKUSWB pakowanie z nasyceniem słów bez znaku do bajtów
- PUNPCKHBW rozpakowanie z przeplotem starszych bajtów
- PUNPCKHWD rozpakowanie z przeplotem starszych słów
- PUNPCKHDQ rozpakowanie z przeplotem starszych podwójnych słów
- PUNPCKLBW rozpakowanie z przeplotem młodszych bajtów
- PUNPCKLWD rozpakowanie z przeplotem młodszych słów
- PUNPCKLDQ rozpakowanie z przeplotem młodszych podwójnych słów

Instrukcja PACKSSWB/PACKUSWB

PACKSSWB cel, źródło

PACKUSWB cel, źródło

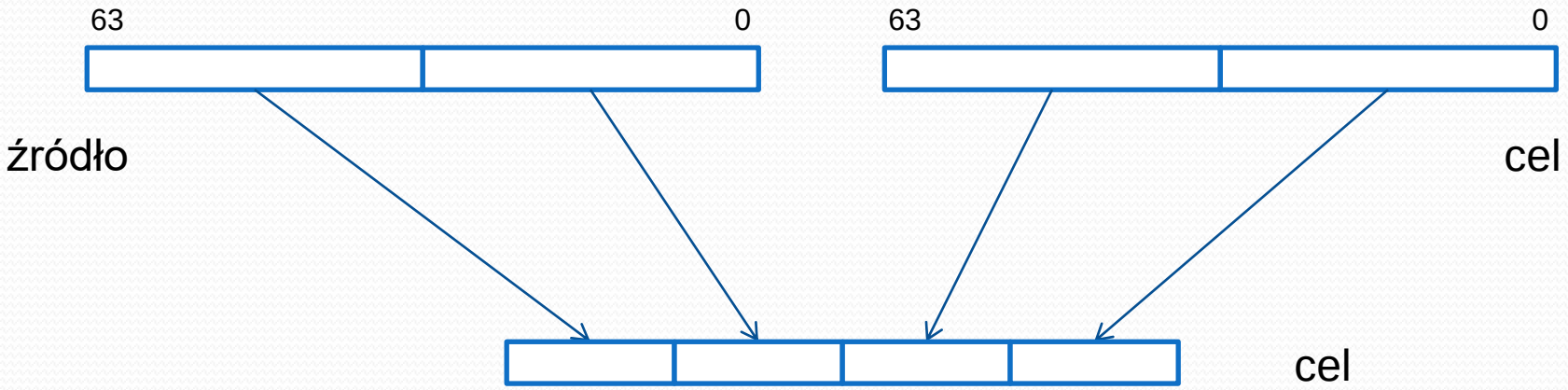
Pakowanie z nasyceniem słów ze znakiem/bez znaku do bajtów. Cel musi być rejestrem mmx.



Instrukcja PACKSSDW

PACKSSDW cel, źródło

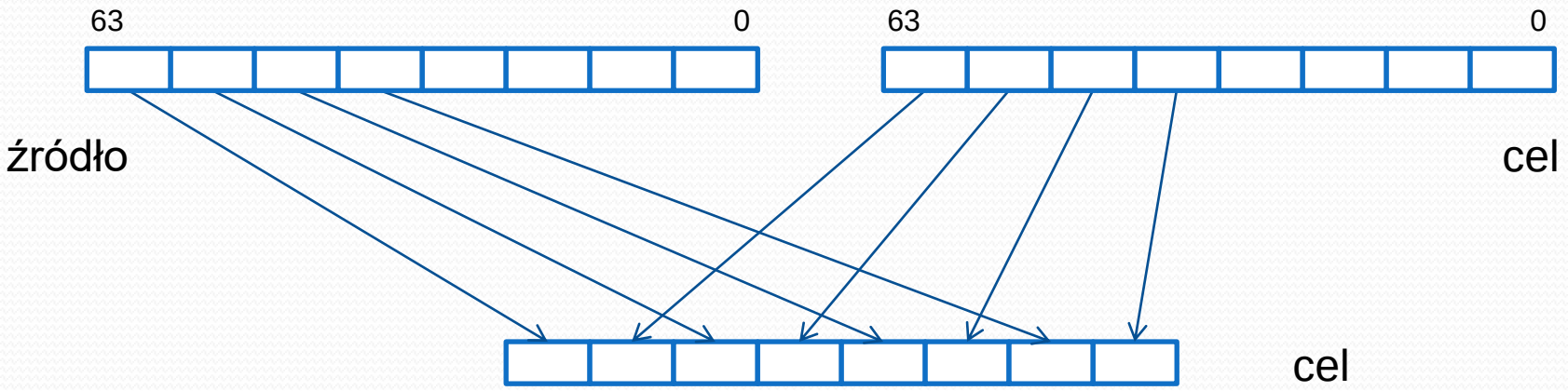
Pakowanie z nasyceniem podwójnych słów ze znakiem do słów. Cel musi być rejestrem mmx.



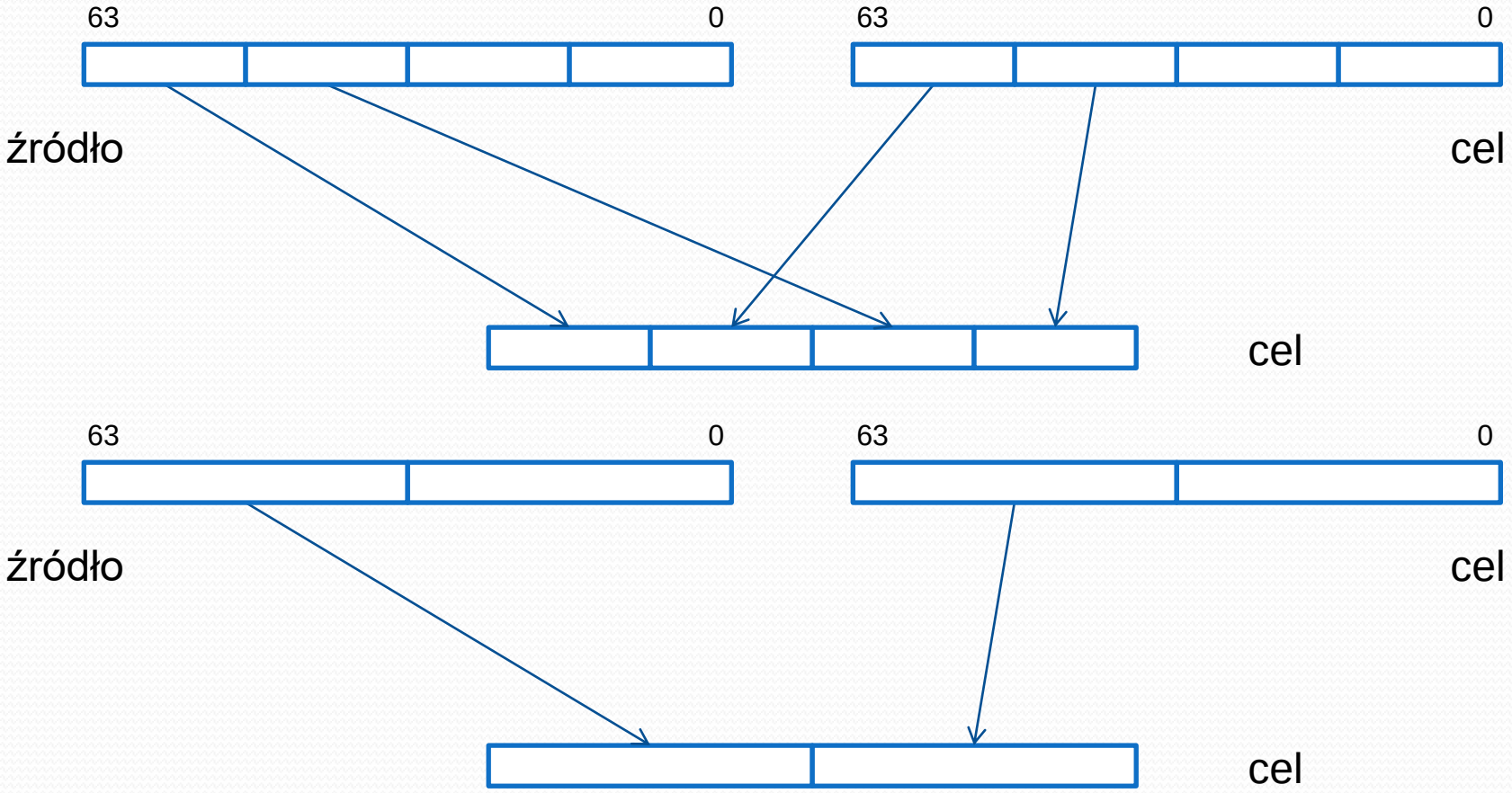
Instrukcje PUNPCKHBW PUNPCKHWD PUNPCKHDQ

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ cel, źródło

Rozpakowanie z przeplotem starszych bajtów/słów/ podwójnych słów bez znaku. Cel musi być rejestrem mmx. Jeśli źródło = 0 to następuje konwersja do słów, podwójnych i poczwórnego słowa.



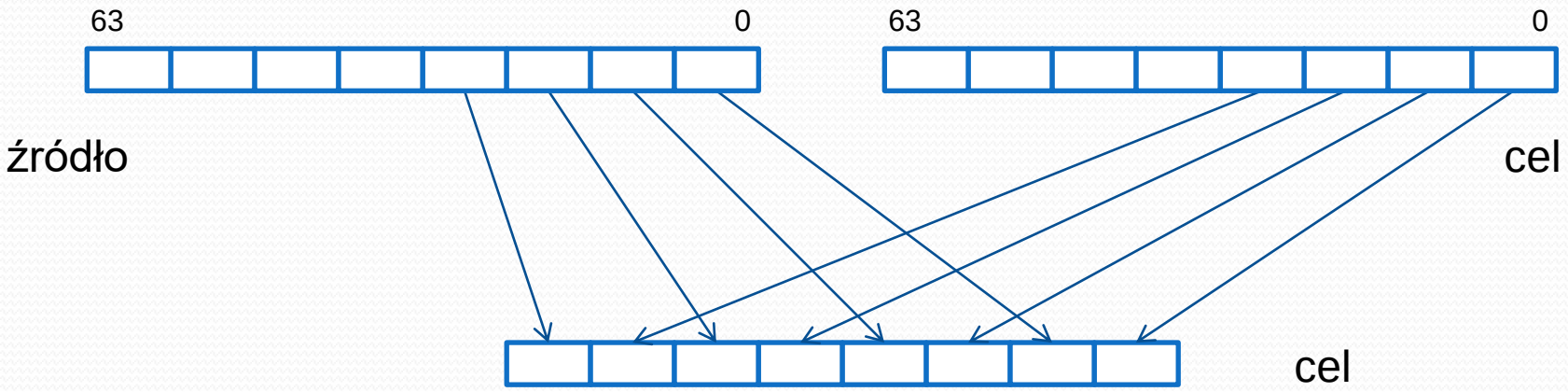
Instrukcje PUNPCKHWD PUNPCKHDQ



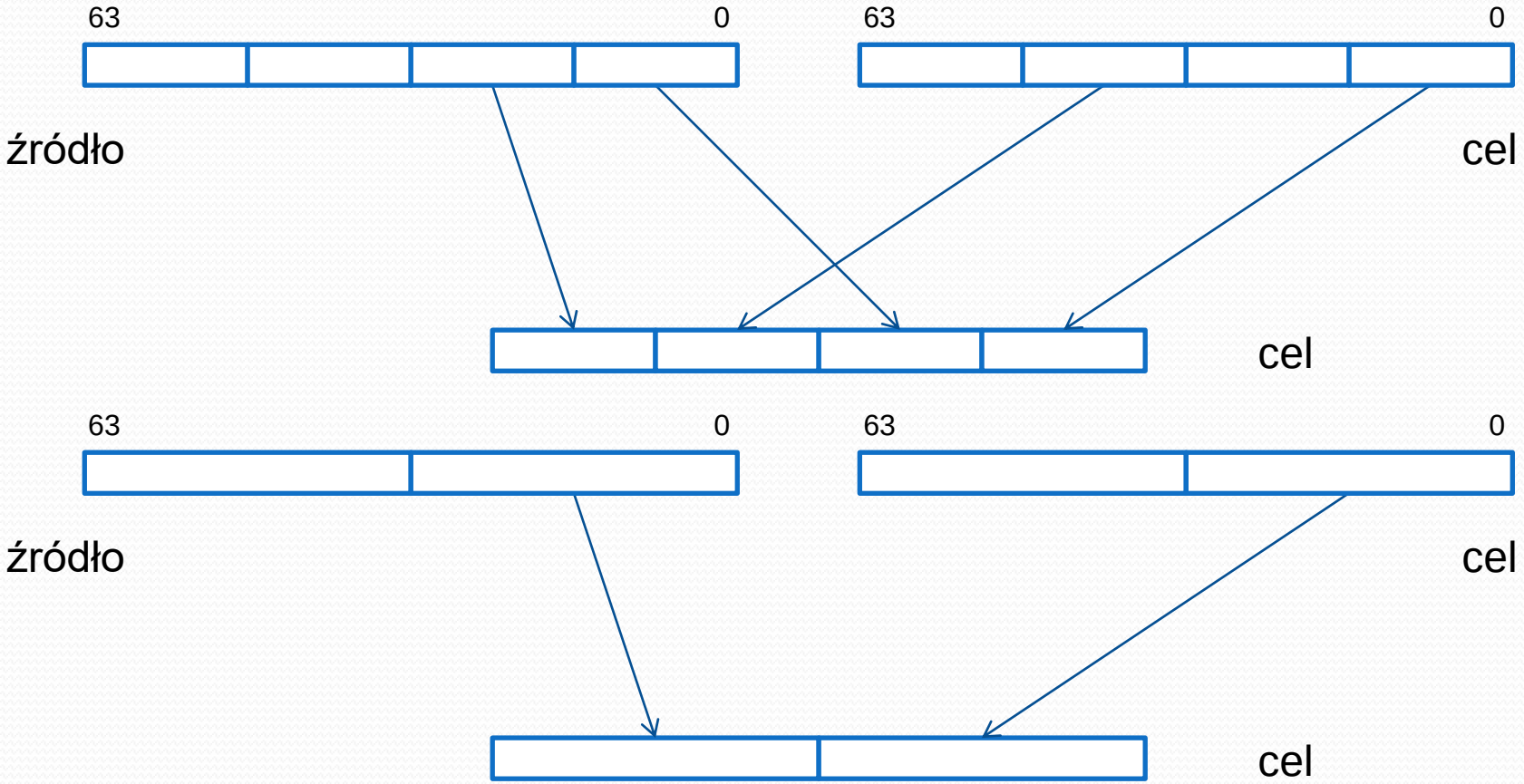
Instrukcje PUNPCKLBW PUNPCKLWD PUNPCKLDQ

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ cel, źródło

Rozpakowanie z przeplotem młodszych bajtów/słów/ podwójnych słów bez znaku. Cel musi być rejestrem mmx. Jeśli źródło = 0 to następuje konwersja do słów, podwójnych i poczwórnego słowa.



Instrukcje PUNPCKLWD PUNPCKLDQ



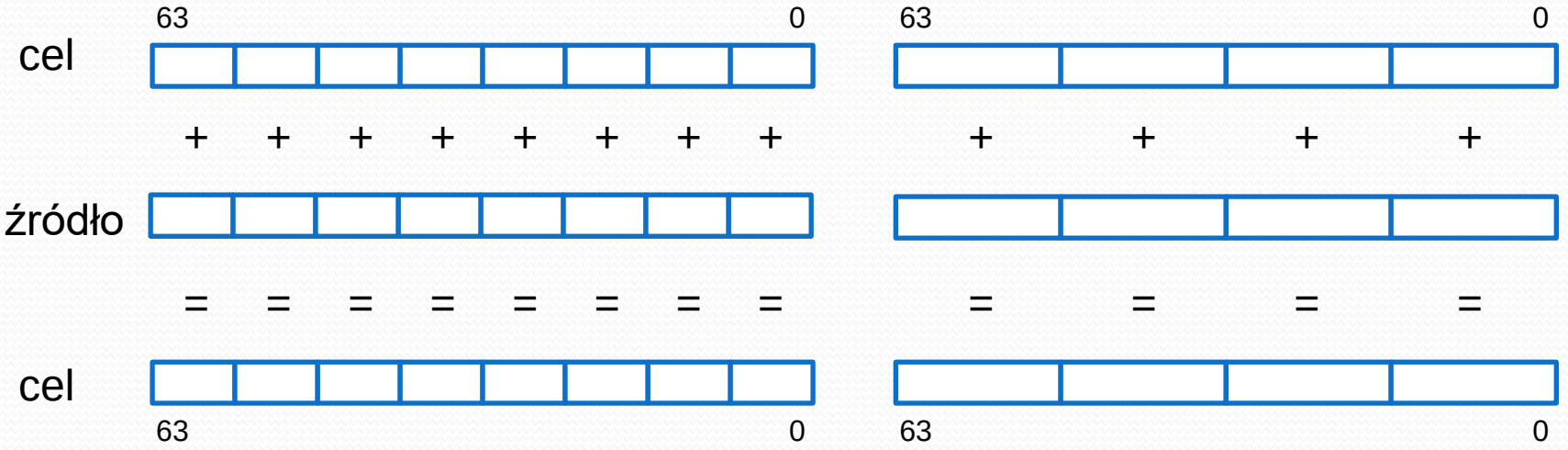
Operacje arytmetyczne

- PADDB dodawanie wektorów bajtów
- PADDW dodawanie wektorów słów
- PADDD dodawanie wektorów podwójnych słów
- PADDSB dodawanie z nasyceniem wektorów bajtów ze znakiem
- PADDSW dodawanie z nasyceniem wektorów słów ze znakiem
- PADDUSB dodawanie z nasyceniem wektorów bajtów bez znaku
- PADDUSW dodawanie z nasyceniem wektorów słów bez znaku

Instrukcje
PADDB PADDW PADDD

PADDB/PADDW/PADDD cel, źródło

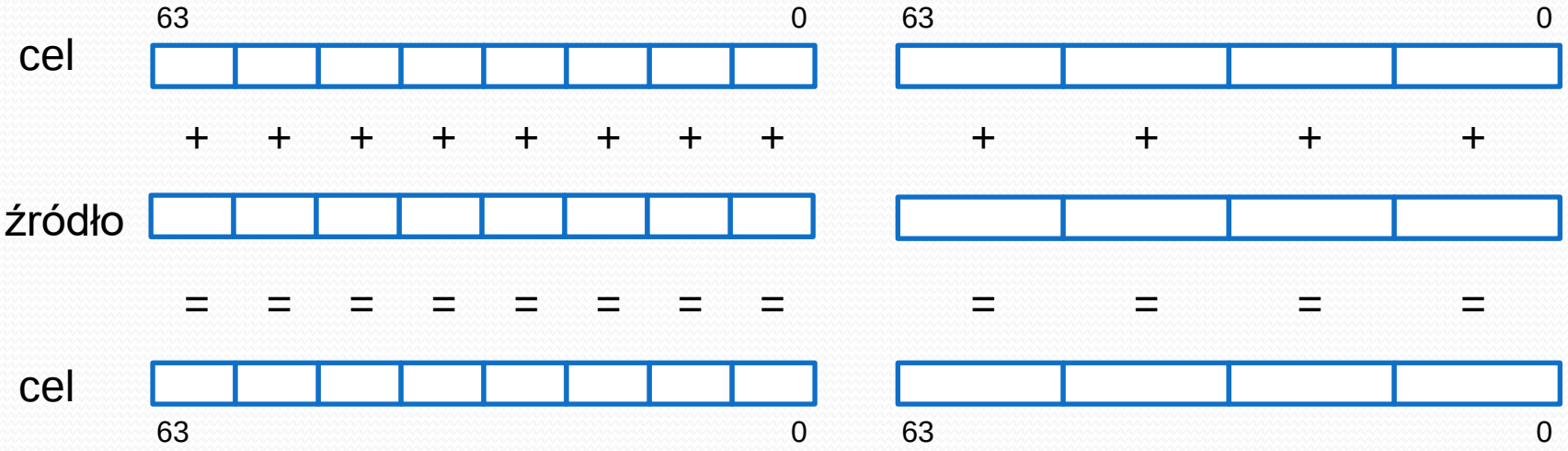
Dodawanie wektorów bajtów/słów/podwójnych słów. Cel musi być rejestrem mmx. Wynik nie uwzględnia przeniesienia.



Instrukcje PADDSB PADDSW PADDUSB PADDUSW

PADDSB/PADDSW/PADDUSB/PADDUSW cel, źródło

Dodawanie z nasyceniem wektorów bajtów/słów ze znakiem/bez znaku. Cel musi być rejestrem mmx.



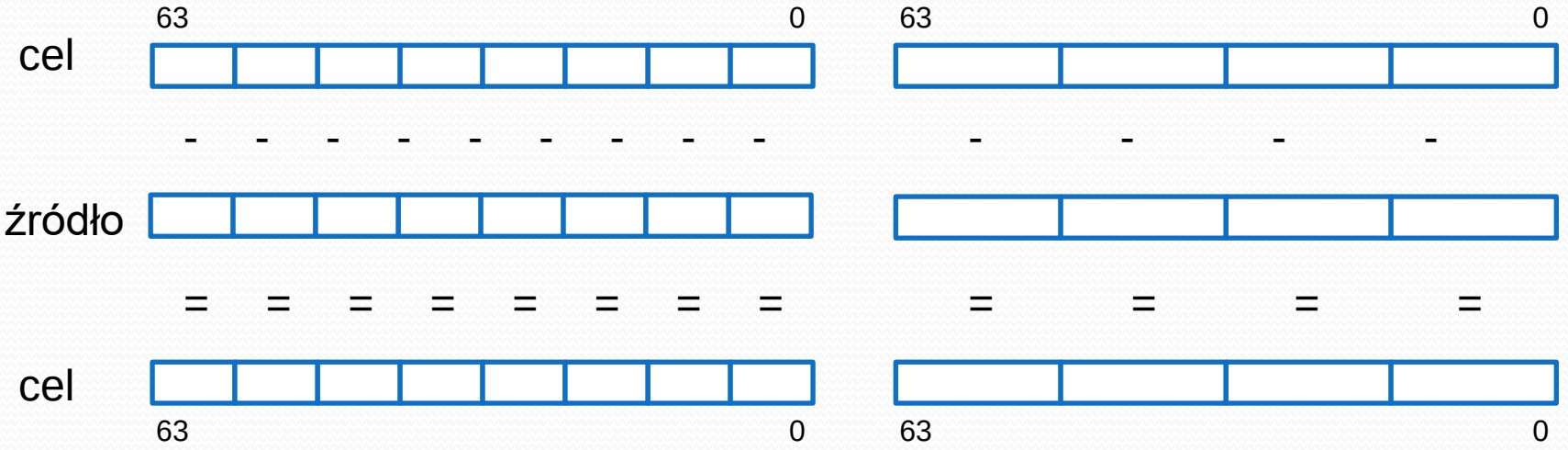
Operacje arytmetyczne

- PSUBB odejmowanie wektorów bajtów
- PSUBW odejmowanie wektorów słów
- PSUBD odejmowanie wektorów podwójnych słów
- PSUBSB odejmowanie z nasyceniem wektorów bajtów ze znakiem
- PSUBSW odejmowanie z nasyceniem wektorów słów ze znakiem
- PSUBUSB odejmowanie z nasyceniem wektorów bajtów bez znaku
- PSUBUSW odejmowanie z nasyceniem wektorów słów bez znaku

Instrukcje
PSUBB PSUBW PSUBD

PSUBB/PSUBW/PSUBD cel, źródło

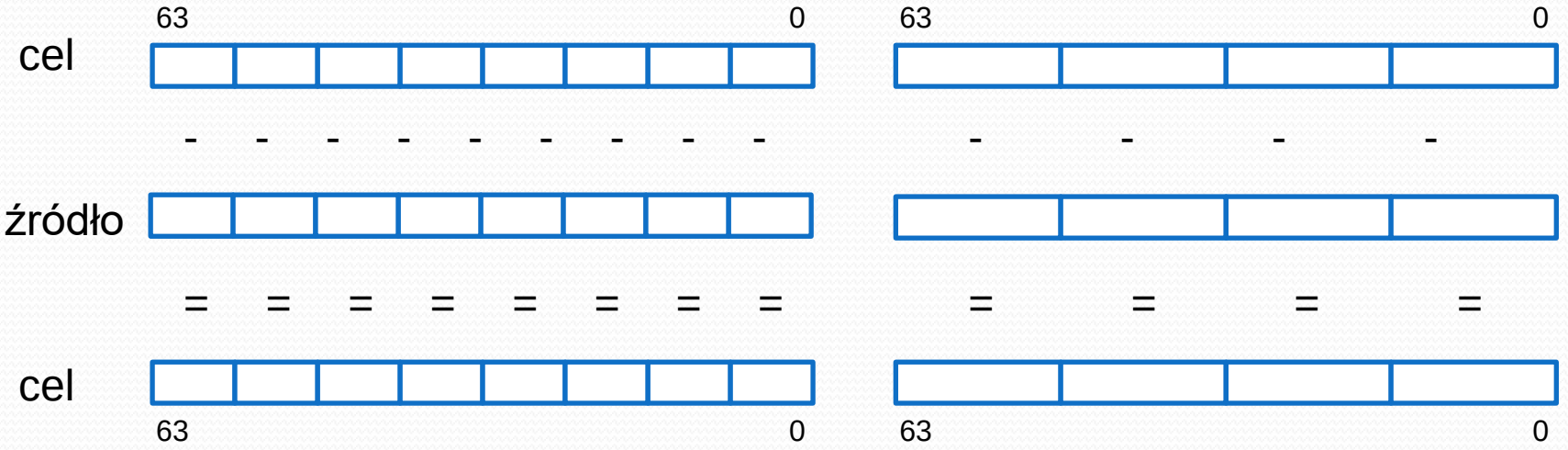
Odejmowanie wektorów bajtów/słów/podwójnych słów. Cel musi być rejestrem mmx. Wynik nie uwzględnia przeniesienia.



Instrukcje PSUBSB PSUBSW PSUBUSB PSUBUSW

PSUBSB/PSUBSW/PSUBUSB/PSUBUSW cel, źródło

Odejmowanie z nasyceniem wektorów bajtów/słów ze znakiem/bez znaku. Cel musi być rejestrem mmx.



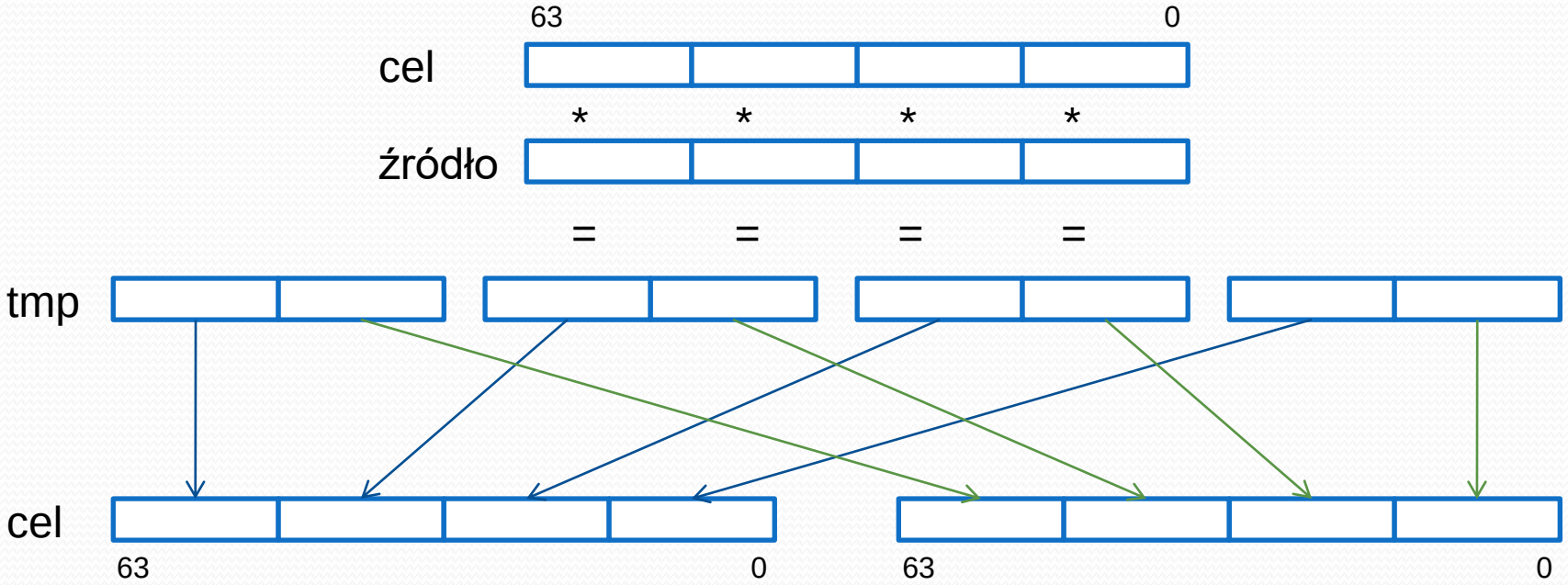
Operacje arytmetyczne

- PMULHW mnożenie wektorów słów i zapamiętanie starszych słów wyniku
- PMULLW mnożenie wektorów słów i zapamiętanie młodszych słów wyniku
- PMADDWD mnożenie i dodawanie wektorów słów

Instrukcje PMULHW PMULLW

PMULHW/PMULLW cel, źródło

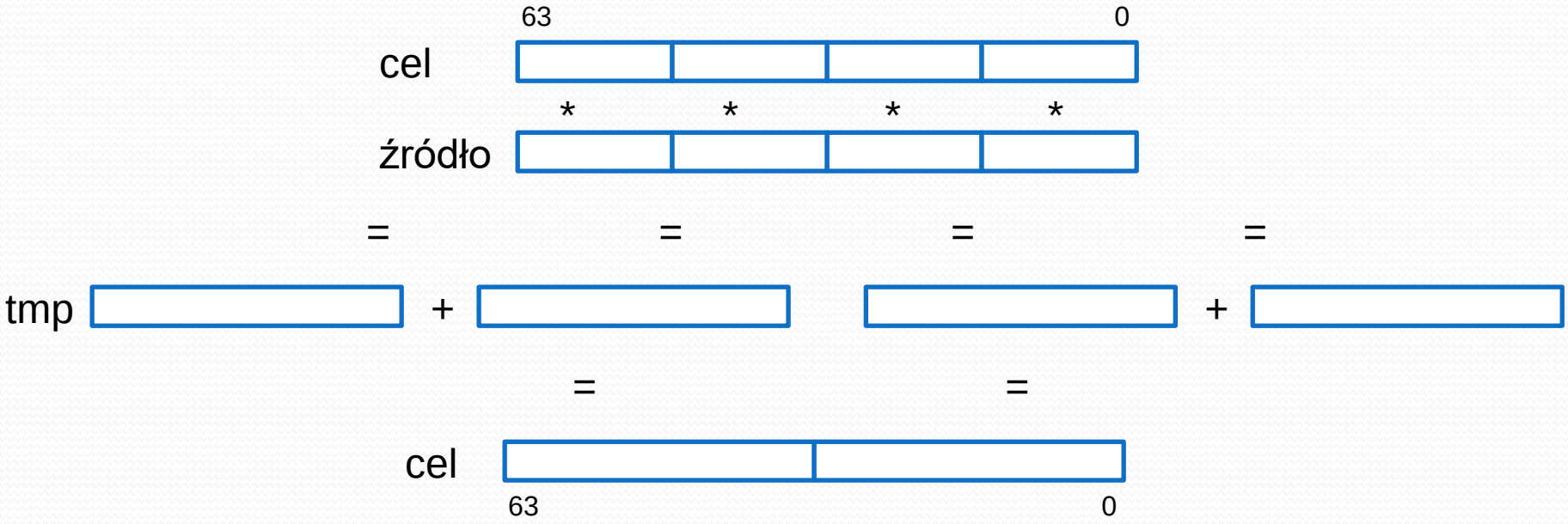
Mnożenie wektorów słów i zapamiętanie starszych/młodszych słów wyniku. Cel musi być rejestrem mmx.



Instrukcje PMADDWD

PMADDWD cel, źródło

Mnożenie i dodawanie wektorów słów. Cel musi być rejestrem mmx.



Operacje porównania

- PCMPEQB sprawdzenie równości wektorów bajtów
- PCMPEQW sprawdzenie równości wektorów słów
- PCMPEQD sprawdzenie równości wektorów podwójnych słów
- PCMPGTB sprawdzenie większości wektorów bajtów ze znakiem
- PCMPGTW sprawdzenie większości wektorów słów ze znakiem
- PCMPGTD sprawdzenie większości wektorów podwójnych słów ze znakiem

Instrukcje PCMPEQB PCMPEQW PCMPEQD

PCMPEQB/PCMPEQW/PCMPEQD cel, źródło

Sprawdzenie równości składowych wektorów bajtów/słów/podwójnych słów. Cel musi być rejestrem mmx.

if cel[i] = źródło[i] then cel[i] := offh/offffh/ffffffffh
else cel[i] := 0

Instrukcje PCMPGTB PCMPGTW PCMPGTD

PCMPGTB/PCMPGTW/PCMPGTD cel, źródło

Sprawdzenie relacji większości składowych wektorów bajtów/słów/podwójnych słów ze znakiem. Cel musi być rejestrem mmx.

if cel[i] > źródło[i] then cel[i] := offh/offffh/ffffffffh
else cel[i] := 0

Operacje logiczne

- PAND bitowy iloczyn logiczny
- PANDN bitowy iloczyn logiczny z negacją
- POR bitowa suma logiczna
- PXOR bitowa suma modulo 2

Instrukcje

PAND PANDN POR PXOR

PAND/PANDN/POR/PXOR cel, źródło

Obliczenie bitowego iloczynu logicznego/bitowego iloczynu logicznego z negacją/bitowej sumy logicznej/bitowej sumy modulo 2. Cel musi być rejestrem mmx.

cel := cel and źródło

cel := (not cel) and źródło

cel := cel or źródło

cel := cel xor źródło

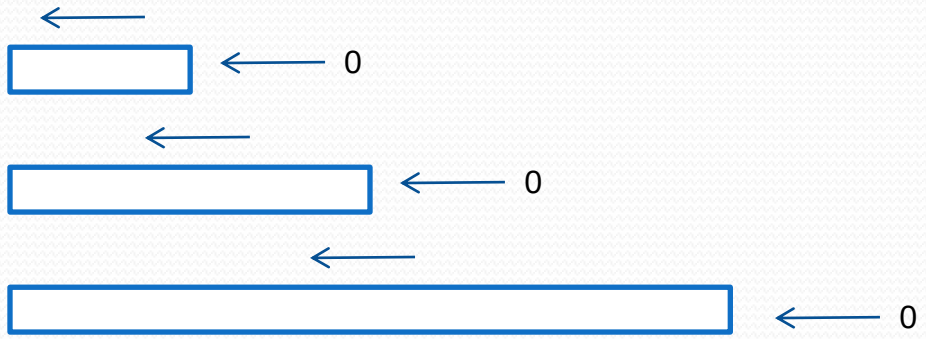
Operacje przesunięć

- PSLLW logiczne przesunięcie w lewo wektora słów
- PSLLD logiczne przesunięcie w lewo wektora podwójnych słów
- PSLLQ logiczne przesunięcie w lewo wektora poczwórnych słów
- PSRLW logiczne przesunięcie w prawo wektora słów
- PSRLD logiczne przesunięcie w prawo wektora podwójnych słów
- PSRLQ logiczne przesunięcie w prawo wektora poczwórnych słów
- PSRAW arytmetyczne przesunięcie w prawo wektora słów
- PSRAD arytmetyczne przesunięcie w prawo wektora podwójnych słów

Instrukcje
PSLLW PSLLD PSLLQ

PSLLW/PSLLD/PSLLQ cel, ile

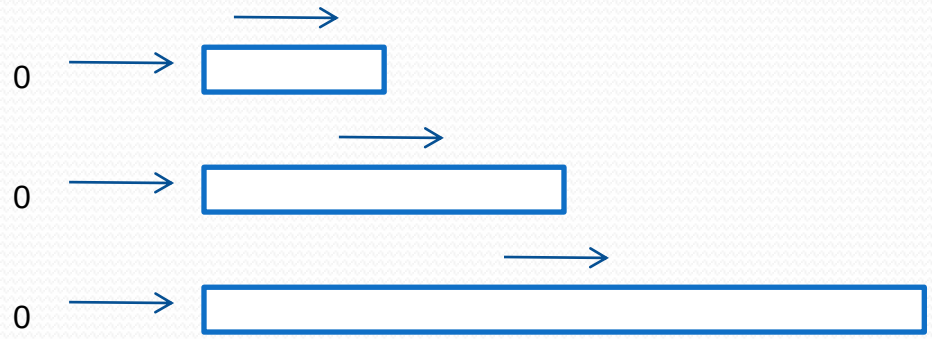
Logiczne przesunięcie w lewo elementów wektora słów/podwójnych słów/poczwórnych słów. Cel musi być rejestrem mmx, ile jest rejestrem mmx, zmienną lub stałą. Młodsze bity wypełniane są zerami.



Instrukcje
PSRLW PSRLD PSRLQ

PSRLW/PSRLD/PSRLQ cel, ile

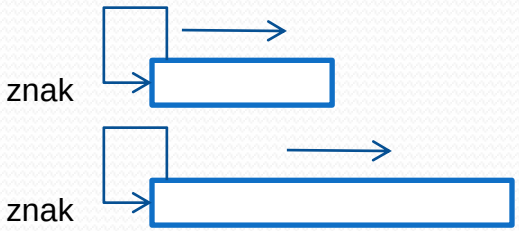
Logiczne przesunięcie w prawo elementów wektora słów/podwójnych słów/poczwórnych słów. Cel musi być rejestrem mmx, ile jest rejestrem mmx, zmienną lub stałą. Starsze bity wypełniane są zerami.



Instrukcje
PSRAW PSRAD

PSRAW/PSRAD cel, ile

Arytmetyczne przesunięcie w prawo elementów wektora słów/podwójnych słów. Cel musi być rejestrem mmx, ile jest rejestrem mmx, zmienną lub stałą. Starsze bity wypełniane są znakiem.



Operacje sterujące

- FXSAVE zapisanie stanu x87 FPU i rejestrów SIMD
- FXRSTOR wczytanie stanu x87 FPU i rejestrów SIMD
- EMMS zwalnia wszystkie rejestry koprocesora
- LDMXCSR wczytanie rejestru MXCSR
- STMXCSR zapisanie rejestru MXCSR

Instrukcje FXSAVE FXRSTOR

FXSAVE/FXRSTOR cel512

zapisanie/wczytanie stanu x87 FPU i rejestrów SIMD

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	
FPU IP				FOP	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		FPU DP				16
Reserved		ST0/MM0						32
Reserved		ST1/MM1						48
Reserved		ST2/MM2						64
Reserved		ST3/MM3						80
Reserved		ST4/MM4						96
Reserved		ST5/MM5						112
Reserved		ST6/MM6						128
Reserved		ST7/MM7						144
XMM0								160
XMM1								176
XMM2								192
XMM3								208
XMM4								224
XMM5								240
XMM6								256
XMM7								272
XMM8								288
XMM9								304
XMM10								320
XMM11								336
XMM12								352
XMM13								368
XMM14								384
XMM15								400
Reserved								416
Reserved								432
Reserved								448
Reserved								464
Reserved								480
Reserved								496

Instrukcja EMMS

EMMS

Zwalnia wszystkie rejestry koprocesora wpisując do pól TAG[i] rejestru stanu zawartości rejestrów stosu wartość 11b (rejestr pusty). Wszystkie instrukcje MMX wpisują do pól TAG[i] 0, co oznacza liczbę prawidłową!

Instrukcje LDMXCSR STMXCSR

LDMXCSR/STMXCSR zmienna

Wczytanie/zapisanie zawartości rejestru MXCSR.

Operacje MMX wprowadzone z SSE

- PAVGB — oblicza średnią z elementów wektorów bajtów bez znaku
- PAVGW — oblicza średnią z elementów wektorów słów bez znaku
- PEXTRW — wydobycie słowa
- PINSRW — wstawienie słowa
- PMAXUB — oblicza maksimum z elementów wektorów bajtów bez znaku
- PMAXSW — oblicza maksimum z elementów wektorów słów ze znakiem
- PMINUB — oblicza minimum z elementów wektorów bajtów bez znaku
- PMINSW — oblicza minimum z elementów wektorów słów ze znakiem
- PMOVMSKB — przesłanie maski bajtów
- PMULHUW — mnożenie wektorów słów bez znaku i zapamiętanie starszych słów wyniku
- PSADBW — oblicza sumę wartości bezwzględnych różnic
- PSHUFW — tasuje słowa w rejestrze MMX

Instrukcje PAVGB PAVGW

PAVGB/PAVGW cel, źródło

Oblicza średnią z elementów wektorów bajtów/słów bez znaku. Cel jest rejestrem MMX.

e_cel := (e_cel + e_źródło + 1) >> 1

Instrukcje PEXTRW PINSRW

PEXTRW cel, źródło, numer

PINSRW cel, źródło, numer

Wydobycie/wstawienie słowa o podanym numerze z/do rejestru MMX do/z rejestru ogólnego przeznaczenia lub pamięci.

Instrukcje PMAXUB PMAXSW PMINUB PMINSW

PMAXUB/PMAXSW/PMINUB/PMINSW cel, źródło

Oblicza maksimum/minimum z elementów wektorów bajtów bez znaku/słów ze znakiem.

e_cel := e_cel max/min e_źródło

Instrukcja PMOVMSKB

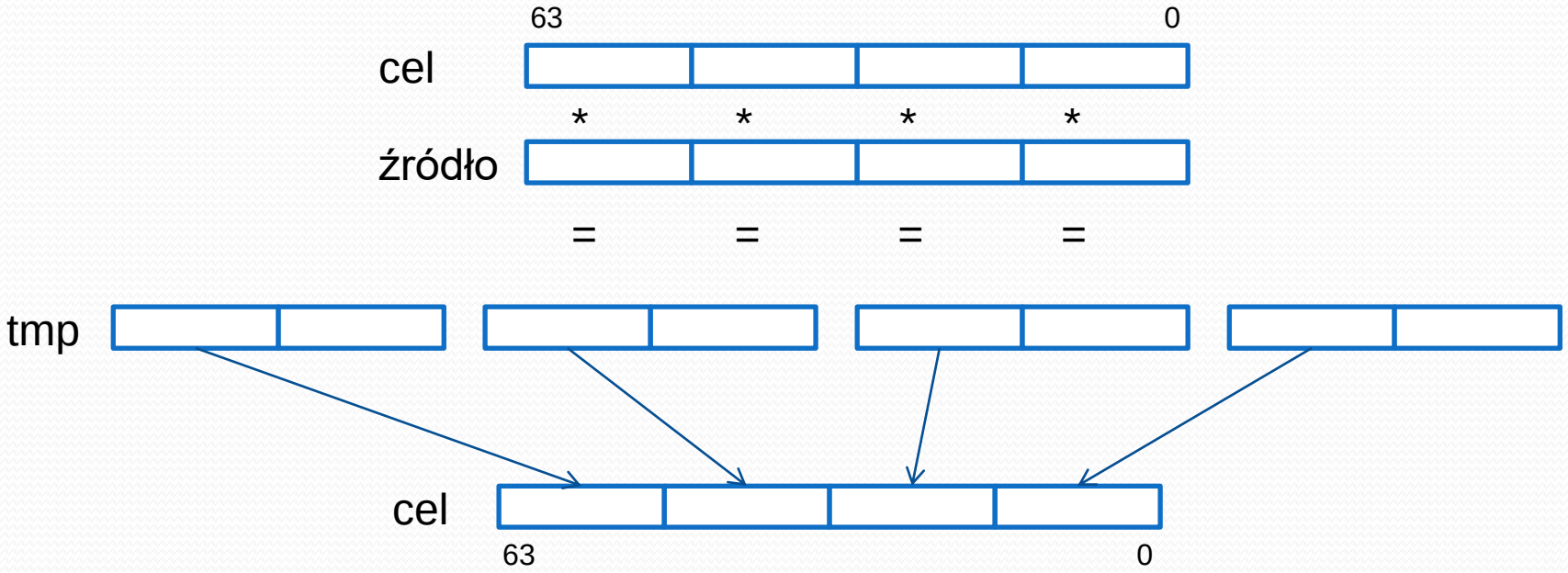
PMOVMSKB cel, źródło

Przesłanie maski bajtów. Celem jest rejestr ogólnego przeznaczenia. Najstarsze bity elementów wektora z rejestru MMX wpisywane są na bity 0 ..7 celu. Stosowane w celu określenia znaku lub sprawdzenia wyniku porównania.

Instrukcje PMULHUW

PMULHUW cel, źródło

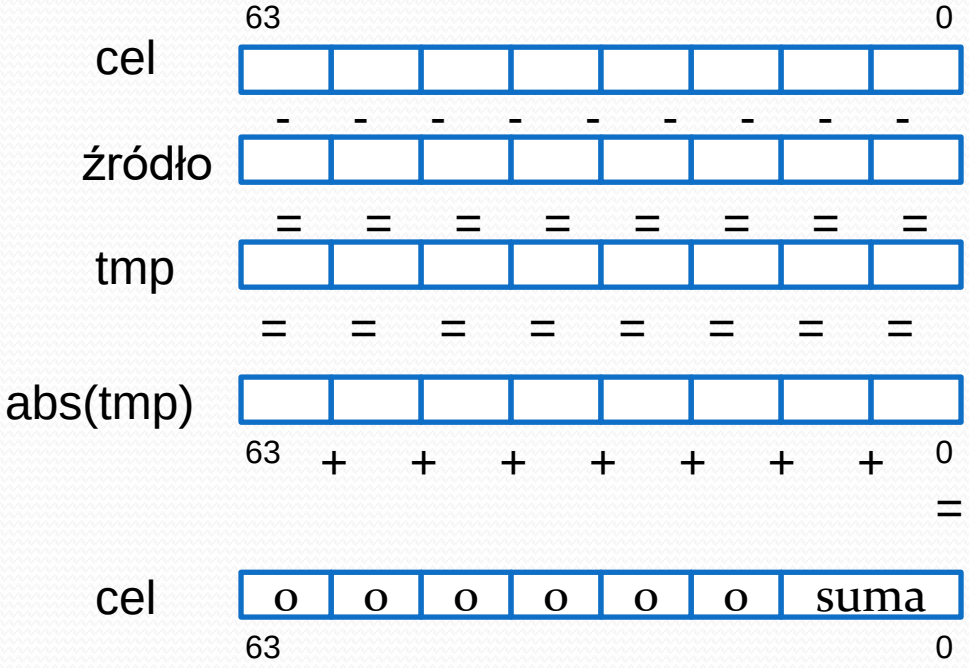
Mnożenie wektorów słów bez znaku i zapamiętanie starszych słów wyniku. Cel musi być rejestrem mmx.



Instrukcja PSADBW

PSADBW cel, źródło

Oblicza sumę wartości bezwzględnych różnic.



Instrukcja PSHUFW

PSHUFW cel, źródło, kolejność

Tasuje słowa w rejestrze celu MMX. Źródłem jest rejestr MMX lub pamięć. Bity 7,6; 5,4 ;3,2 i 1,0 stałej kolejność określają numer słowa w źródle, które zostanie umieszczone jako 3,2,1 i 0 w rejestrze celu np.

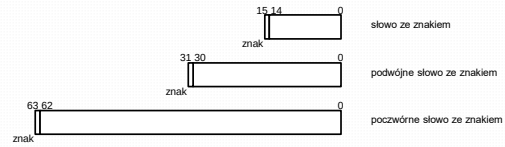
dla źródła= **0123 4567 89ab cdef**h i kolejności 00 01 10 11b

będzie: cel=0**cdef 89ab 4567 0123**h

Operacje zmiennoprzecinkowe

Liczbowe typy danych

Liczby całkowite ze znakiem

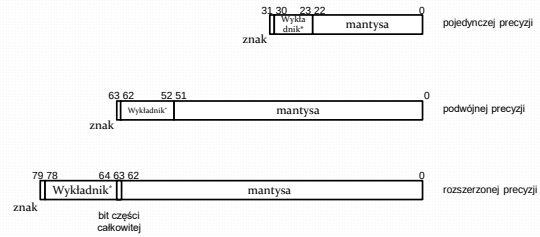


Typ BCD

4 bity = 1 cyfra BCD

Liczbowe typy danych

Liczby zmiennoprzecinkowe



Liczby zmiennoprzecinkowe

- liczba = $(-1)^{znak} * m * 2^{wykładnik}$
- Liczba znormalizowana

  $liczbaZ = 1{,}0100111010110 * 2^{wykładnik}$

  bit części całkowitej
- Liczba nieznormalizowana

  $liczbaNZ = 0{,}0110110010110 * 2^{wykładnik}$

  bit części całkowitej

Liczby zmiennoprzecinkowe

Implementacja:

- liczba = $(-1)^{znak} * m * 2^{wykładnik}$
- wykładnik = **przesunięta cecha** - przesunięcie
- m = **część całkowita** + **część ułamkowa** =

  = **bit części całkowitej** + **mantysa**
  - dla liczb pojedynczej i podwójnej precyzji część całkowita = **1** i nie jest zapamiętywana

Precyzja	Przesunięcie
pojedyncza	127
podwójna	1023
rozszerzona	16383

Wartości

poprawne i niepoprawne zawarte w rejestrach stosu koprocesora.

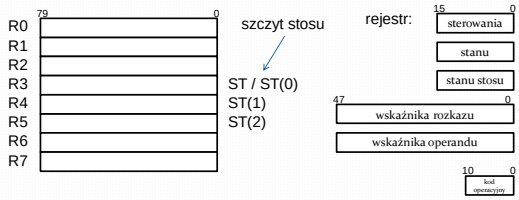
Klasa		Znak	Przesunięta cecha	Mantysa	
				Część całkowita	Część ułamkowa
Dodatnie nieliczby	Pasywne (ciche)	0 . 0	11..11 . 11..11	0	11..11 . 10..00
	Aktywne (istotne)	0 . 0	11..11 . 11..11	0	01..11 . 00..01
Dodatnie zmiennoprzecinkowe	Nieskończoność	0	11..11	0	00..00
	Znormalizowane	0 . 0	11..10 . 00..01	1	11..11 . 00..00
	Niezmormalizowane	0 . 0	11..10 . 00..01	0	11..11 . 00..00
	Pseudo-zdenormalizowane	0 . 0	00..00	1	11..11 . 00..00
	Zero	0/1	00..00	0	00..00
Ujemne zmiennoprzecinkowe	Pseudo-zdenormalizowane	1 . 1	00..00	1	11..11 . 00..00
	Niezmormalizowane	1 . 1	11..10 . 00..01	0	11..01 . 00..00
	Znormalizowane	1 . 1	11..10 . 00..01	1	11..01 . 00..00
	Minus nieskończoność	1	11..11	0	00..00
Ujemne nieliczby	Aktywne (istotne)	1 . 1	11..11 . 11..11	0	01..11 . 00..01
	Pasywne (ciche)	1 . 1	11..11 . 11..11	0	11..11 . 10..00
			←15 bits →		← 63 bits →

Liczby zmiennoprzecinkowe

	precyzja		
	pojedyncza	podwójna	rozszerzona
cyfry znaczące	6	15	18
wartość największa	3,402823466E38	1,797693134862315 8E308	1,189731495357231E4932
wartość najmniejsza	1,175494351E-38	2,225073858507 2024E-308	3,362103143112 0935E-4932

Koprocesor - budowa

stos rejestrów



Koprocesor

Rejestry R0, R1, R2, R3, R4, R5, R6, R7

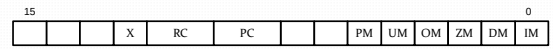
Koprocesor

rejestr stanu

B – koprocesor zajęty
Co-C3 – bity rodzaju wyniku
TOP - wskaźnik stosu
ES – znacznik błędu
SF – znacznik błędu stosu
PE –błąd niedokładności wyniku
UE –błąd niedomiaru
OE –błąd nadmiaru
ZE –błąd dzielenia przez zero
DE –błąd zdenormalizowanego argumentu
IE –błąd niedozwolonej operacji

Koprocesor

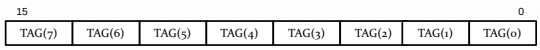
rejestr sterowania



X – interpretacja nieskończoności (tylko 287)
RC – sterowanie zaokrągleniem: do najbliższej(00), w dół (01), w górę (10), obcięcie (11)
PC – sterowanie dokładnością obliczeń(23 (00), 53 (10) i 63 (11) bity)
PM – maskowanie błędu niedokładności wyniku
UM – maskowanie błędu niedomiaru
OM – maskowanie błędu nadmiaru
ZM – maskowanie błędu dzielenia przez zero
DM – maskowanie błędu zdenormalizowanego argumentu
IM – maskowanie błędu niedozwolonej operacji

Koprocesor

rejestr stanu zawartości rejestrów stosu



TAG – pola określają zawartość poszczególnych rejestrów stosu:

00 – liczba prawidłowa
01 – zero
10 – wartość specjalna (nie liczba NaN, nieskończoność,..) lub zdenormalizowana
11 – rejestr pusty

Operacje przesyłania danych

- FLD załadowanie argumentu zmiennoprzecinkowego
- FST zapisanie wartości z wierzchołka stosu
- FSTP zapisanie wartości z wierzchołka stosu i usunięcie go za stosu
- FILD załadowanie liczby całkowitej
- FIST zapisanie liczby całkowitej
- FISTP zapisanie liczby całkowitej ze zdjęciem ze stosu
- FBLD załadowanie liczby BCD
- FBSTP zapisanie liczby BCD i zdjęcie jej ze stosu
- FXCH zamiana zawartości rejestrów
- FCMOVE przesłanie warunkowe (jeśli równe)
- FCMOVNE przesłanie warunkowe (jeśli nie równe)
- FCMOVB przesłanie warunkowe (jeśli poniżej)
- FCMOVBE przesłanie warunkowe (jeśli poniżej lub równe)
- FCMOVNB przesłanie warunkowe (jeśli nie poniżej)
- FCMOVNBE przesłanie warunkowe (jeśli nie poniżej lub równe)
- FCMOVU przesłanie warunkowe (jeśli nieuporządkowane)
- FCMOVNU przesłanie warunkowe (jeśli uporządkowane)

Instrukcja FLD

fld źródło

Przesyła liczbę zmiennoprzecinkową z rejestru st(i) lub z pamięci na wierzchołek stosu.

fpush(źródło)

```
fld   st(3)
fld   zmienna
```


Instrukcja FST

fst cel

Zapisuje liczbę zmiennoprzecinkową z wierzchołka stosu do rejestru st(i) lub pamięci.

cel := st

```
fst   st(3)
fst   zmienna
```


Instrukcja FSTP

fstp cel

Zapisuje liczbę zmiennoprzecinkową z wierzchołka stosu do rejestru st(i) lub pamięci i zdejmuje ją ze stosu.

cel := st
fpop

```
fstp  st(3)
fstp  zmienna
```


Instrukcja FILD

fild źródło

Przesyła liczbę całkowitą z pamięci na wierzchołek stosu.

fpush(źródło)

```
fild  zmienna
```

Instrukcja FIST

fist cel

Zapisuje liczbę w formacie całkowitym z wierzchołka stosu do pamięci.

cel := int(st)

fist zmienna

Instrukcja FISTP

fistp cel

Zapisuje liczbę w formacie całkowitym z wierzchołka stosu do pamięci i zdejmuje ją ze stosu.

cel := int(st)

fpop

fistp zmienna

Instrukcja FBLD

fbld źródło

Przesyła liczbę całkowitą BCD z pamięci na wierzchołek stosu.

fpush(źródło)

fbld zmienna

Instrukcja FBSTP

fbstp cel

Zapisuje liczbę w formacie całkowitym BCD z wierzchołka stosu do pamięci i zdejmuje ją ze stosu.

cel := int(st)

fpop

fbstp zmienna

Instrukcja FXCH

fxch st(i)

fxch

Zamienia liczbę z wierzchołka stosu z wartością w rejestrze celu. Bez parametru celem jest st(1).

st(i)⇔st

fxch st(5)

Instrukcja FCMOVcc

FCMOVE	przesłanie warunkowe (jeśli równe, ZF=1)
FCMOVNE	przesłanie warunkowe (jeśli nie równe, ZF=0)
FCMOVB	przesłanie warunkowe (jeśli poniżej, CF=1)
FCMOVBE	przesłanie warunkowe (jeśli poniżej lub równe, CF=1 lub ZF=1)
FCMOVNB	przesłanie warunkowe (jeśli nie poniżej, CF=0)
FCMOVNBE	przesłanie warunkowe (jeśli nie poniżej lub równe, CF=0 i ZF=0)
FCMOVU	przesłanie warunkowe (jeśli nieuporządkowane, PF=1)
FCMOVNU	przesłanie warunkowe (jeśli uporządkowane,PF=0)

fcmov*cc* st, st(i)

Jeśli jest spełniony warunek cc zapisuje do st wartość rejestru źródła st(i).

if cc then st := st(i)

fcmovnbe st, st(5)

Operacje arytmetyczne

FADD	dodawanie
FADDP	dodawanie ze zdjęciem ze stosu
FIADD	dodawanie liczby całkowitej
FSUB	odejmowanie
FSUBP	odejmowanie ze zdjęciem ze stosu
FISUB	odejmowanie liczby całkowitej
FSUBR	odejmowanie odwrotne
FSUBRP	odejmowanie odwrotne ze zdjęciem ze stosu
FISUBR	odejmowanie odwrotne liczby całkowitej
FMUL	mnożenie
FMULP	mnożenie ze zdjęciem ze stosu
FIMUL	mnożenie liczby całkowitej
FDIV	dzielenie
FDIVP	dzielenie ze zdjęciem ze stosu
FIDIV	dzielenie przez liczbę całkowitą
FDIVR	dzielenie odwrotne
FDIVRP	dzielenie odwrotne ze zdjęciem ze stosu
FIDIVR	dzielenie odwrotne liczby całkowitej
FPREM	obliczenie reszty (częściowej) z dzielenia
FPREM1	obliczenie reszty (częściowej) z dzielenia zgodne z IEEE
FABS	obliczenie wartości bezwzględnej
FCHS	zmiana znaku
FRNDINT	zaokrąglenie do liczby całkowitej
FSCALE	skalowanie przez potęgę 2
FSQRT	obliczenie pierwiastka kwadratowego
FXTRACT	obliczenie wykładnika i mantysy

Operacje arytmetyczne - formaty

- F*op* stosowy
  - argumenty niejawne: **cel – st(1), źródło – st**
  - wynik umieszczany jest w rejestrze celu, rejestr źródła zostaje zdjęty ze stosu
  - odpowiada F*op*P st(1), st
- F*op* st(i), st; F*op* st, st(i) rejestrowy
  - jednym z argumentów musi być rejestr wierzchołka stosu
- F*op*P st(i), st rejestrowy ze zdjęciem ze stosu
  - wynik umieszczany jest w st(i), argument źródłowy jest zdejmowany ze stosu
- F*op* zmienna z argumentem w pamięci
  - argumentem celu jest wierzchołek stosu
- FI*op* zmienna całk. z argumentem całkowitym w pamięci
  - argumentem celu jest wierzchołek stosu

Instrukcja FADD/FADDP/FIADD

Dodaje do rejestru celu wartość źródła. Dla FIADD źródłem jest liczba całkowita. FADDP zdejmuje wierzchołek stosu.

st(cel) := st(cel) + st(źródło)|zmienna

```
fadd   st(5), st
```


Instrukcja FSUB/FSUBP/FISUB

Odejmuje od rejestru celu wartość źródła. Dla FISUB źródłem jest liczba całkowita. FSUBP zdejmuje wierzchołek stosu.

st(cel) := st(cel) - st(źródło)|zmienna

```
fsubp  st(3), st
```


Instrukcja FSUBR/FSUBRP/FISUBR

Odejmuje od źródła rejestr celu. Wynik umieszcza w rejestrze celu. Dla FISUBR źródłem jest liczba całkowita. FSUBRP zdejmuje wierzchołek stosu.

st(cel) := st(źródło)|zmienna - st(cel)

```
fsubr  st(2), st
```


Instrukcja FMUL/FMULP/FIMUL

Mnoży rejestr celu przez wartość źródła. Dla FIMUL źródłem jest liczba całkowita. FMULP zdejmuje wierzchołek stosu.

st(cel) := st(cel) * st(źródło)|zmienna

```
fmulp st(4), st
```

Instrukcja FDIV/FDIVP/FIDIV

Dzieli rejestr celu przez wartość źródła. Dla FIDIV źródłem jest liczba całkowita. FDIVP zdejmuje wierzchołek stosu.

st(cel) := st(cel) / st(źródło)|zmienna

```
fdiv    st, st(4)
```


Instrukcja FDIVR/FDIVRP/FIDIVR

Dzieli źródło przez rejestr celu. Wynik umieszcza w rejestrze celu. Dla FIDIVR źródłem jest liczba całkowita. FDIVRP zdejmuje wierzchołek stosu.

st(cel) := st(źródło)|zmienna/st(cel)

```
fdivr   st, st(3)
```


Przykład

$y = ax^3 + bx^2 + cx + d$

```
fld     d       ;d
fld     x       ;x; d
fld     st      ;x; x; d
fmul    st, st(1)   ;xx; x; d
fld     st(1)   ;x; xx; x; d
fmul    st, st(1)   ;xxx; xx; x; d
fmul    a       ;axxx; xx; x; d
faddp   st(3), st   ;xx; x; axxx+d
fmul    b       ;b*xx; x; axxx+d
faddp   st(2), st   ; x; axxx+b*xx+d
fmul    c       ;c*x; axxx+b*xx+d
fadd            ;axxx+b*xx+c*x+d
fstp    y
```

```
;wersja zoptymalizowana
fld     x       ;x
fld     a       ;a, x
fmul    st, st(1)   ;ax, x
fadd    b       ;ax + b, x
fmul    st, st(1)   ;axx + bx, x
fadd    c       ;axx + bx + c, x
fmul            ;axxx + bxx + cx
fadd    d       ;axxx + bxx + cx + d
fstp    y
```


Przykład

$$y = x_1 \quad \cdots \quad x_n \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$$

```
        mov     ecx, n
        mov     esi, x
        mov     edi, z
        fld     [esi]       ;x
        fld     [edi]       ; z; x
        fmul                ;s:=x*z
        dec     ecx
@1:     add     esi, 8
        add     edi, 8
        fld     [esi]       ;x; s
        fmul    [edi]       ;x*z; s
        fadd                ;s:=s+x*z
        dec     ecx
        jnz     @1
```

```
        mov     esi,x
        mov     edi,y
        fld     [esi]       ;x
        mov     ecx,n
        fmul    [edi]       ;s:=x*z
        dec     ecx
@1:     add     esi, 8
        add     edi, 8
        fld     [esi]       ;x; s
        fmul    [edi]       ;x*z; s
        dec     ecx
        fadd                ;s:=s+x*z
        jnz     @1
```


Przykład

$$y = x_1 \quad \cdots \quad x_n \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$$

```
        mov     ecx,n
        fld     zero            ;s:=0
        mov     esi, x
        mov     edi, z
@1:     fld     qword ptr[esi+8*ecx-8]  ;x; s
        fmul    qword ptr[edi+8*ecx-8]  ;x*z; s
        fadd                    ;s:=s+x*z
        dec     ecx
        jnz     @1
```


Instrukcja FPREM/FPREM1

Oblicza resztę z dzielenia st/st(1), wynik umieszcza w rejestrze st. Wynik jest dokładny. Jeśli st/st(1) jest zbyt duży (c2=1) w st umieszczona zostaje częściowa reszta i trzeba powtórzyć instrukcję. Zakres reszty:

FPREM <-|st(1)|, |st(1)|>

FPREM1 <-|st(1)/2|, |st(1)/2|>

Q := int|round(st/st(1))

st :=st – Q*st(1)

```
fprem
```

Instrukcja FABS

Oblicza wartość bezwzględną liczby z wierzchołka stosu.

st := |st|

fabs

Instrukcja FCHS

Zmienia znak liczby na wierzchołku stosu.

st := -st

fchs

Instrukcja FRNDINT

Zaokrągla liczbę na wierzchołku stosu.

st := round(st)

frndint

Instrukcja FSCALE

Skalowanie przez potęgę 2. Do wykładnika st dodaje część całkowitą st(1).

$st := st*2^{int(st(1))}$

fscale

Instrukcja FSQRT

Oblicza pierwiastek kwadratowy z liczby na wierzchołku stosu.

st := sqrt(st)

fsqrt

Instrukcja FXTRACT

Oblicza wykładnik i mantysę liczby z rejestru st.

st := wykładnik(st)

fpush(mantysa(st))

fxtract

Przykład

Przekątna prostokąta $c = \sqrt{a^2 + b^2}$

```
fld a           ;a
fmul st, st     ;a*a
fld b           ;b, a*a
fmul st, st     ;b + b, a*a
fadd            ;a*a + b*b
fsqrt           ;c
fstp  c
```


Przykład

$y = \frac{\sqrt{|a-b|}}{a+b}$

```
fld a           ;a
fld st          ;a, a
fld b           ;b, a, a
fadd st(2), st  ;b, a, a+b
fsubp st(1), st ;a-b, a+b
fabs            ;|a-b|, (a+b)
fsqrt           ;sqrt(|a-b|), (a+b)
fdivr           ;sqrt(|a-b|)/(a+b)
fst y
```


Operacje ładowania stałych

- FLD1 zapisanie +1.0 na wierzchołku stosu
- FLDZ zapisanie +0.0 na wierzchołku stosu
- FLDPI zapisanie π na wierzchołku stosu
- FLDL2E zapisanie $\log_2 e$ na wierzchołku stosu
- FLDLN2 zapisanie $\log_e 2$ (ln2) na wierzchołku stosu
- FLDL2T zapisanie $\log_2 10$ na wierzchołku stosu
- FLDLG2 zapisanie $\log_{10} 2$ na wierzchołku stosu

Instrukcje zapisują stałe na wierzchołku stosu (st).

Przykład

$y = 2\pi r$

```
fldpi           ;pi
fld r           ;r, pi
fmul            ;pi*r
fadd st, st     ;2*pi*r
fstp y
```

$y = \pi r^2$

```
fldpi           ;pi
fld r           ;r, pi
fmul st, st     ;r*r, pi
fmul            ;pi*r*r
fstp y
```


Operacje funkcji przestępnych

- FSIN Oblicza sinus
- FCOS Oblicza cosinus
- FSINCOS Oblicza sinus i cosinus
- FPTAN Oblicza (częściowy) tangens
- FPATAN Oblicza (częściowy) arcus tangens
- F2XM1 Oblicza $2^x - 1$
- FYL2X Oblicza $y*\log_2 x$
- FYL2XP1 Oblicza $y*\log_2(x+1)$

Instrukcja FSIN

Oblicza sinus liczby zawartej w st(0) i wynik umieszcza w st(0). Jeśli st nie zawiera się w $<-2^{63}, 2^{63}>$, wówczas flaga C2 jest ustawiana. Argument można zredukować instrukcją FPREM z dzielnikiem 2π.

st := sin(st)

```
fsin
```

Instrukcja FCOS

Oblicza cosinus liczby zawartej w st(0) i wynik umieszcza w st(0). Jeśli st nie zawiera się w $<-2^{63},2^{63}>$, wówczas flaga C2 jest ustawiana. Argument można zredukować instrukcją FPREM z dzielnikiem 2π.

st := cos(st)

fcos

Instrukcja FSINCOS

Oblicza sinus i cosinus liczby zawartej w st(0) i wynik umieszcza w st(0) i na wierzchołku stosu. Jeśli st nie zawiera się w $<-2^{63},2^{63}>$, wówczas flaga C2 jest ustawiana. Argument można zredukować instrukcją FPREM z dzielnikiem 2π.

temp := cos(st)
st := sin(st)
fpush(temp)

fsincos

Instrukcja FPTAN

Oblicza tangens liczby zawartej w st(0) i wynik umieszcza w st(0) i 1.0 na wierzchołku stosu. Jeśli st nie zawiera się w $<-2^{63},2^{63}>$, wówczas flaga C2 jest ustawiana. Argument można zredukować instrukcją FPREM z dzielnikiem 2π.

st := tan(st)
fpush(1.0)

fptan

Instrukcja FPATAN

Oblicza arcus tangens (kąt) ilorazu st(1)/st i wynik umieszcza w st(1), a st zdejmuje z wierzchołka stosu.

st(1) := arctg(st(1)/st)
fpop

fpatan

Instrukcja F2XM1

Oblicza $2^{st} - 1$. st musi być w przedziale $<-1,1>$.

$st:=2^{st} - 1$

f2xm1

Instrukcja FYL2X

Oblicza $y*\log_2 x$. st > 0

$st(1) := st(1)* \log_2 st$
fpop

fyl2x

Instrukcja FYL2XP1

Oblicza y∗log$_2$x. Liczba w rejestrze st musi spełniać

$-(1-\sqrt{2}/2) < st < (1-\sqrt{2}/2)$

st(1) := st(1)* log$_2$(st+1)

fpop

fyl2xp1

Przykład

$a = x^y = 2^{y \cdot \log_2 x}$

```
fld y           ;y
fld x           ;x; y
fyl2x           ;y*log2x
f2xm1           ;2^ (y*log2x) - 1     !!!!!!!
fld1            ;1; 2^ (y*log2x) - 1
fadd            ;a
fstp a
```


Przykład

$a = e^x = 2^{x \cdot \log_2 e}$

```
fld   x         ;x
fldl2e          ; log2e; x
fmul            ; x*log2e
fld   st        ; x*log2e; x*log2e
frndint         ; round(x*log2e); x*log2e
fsub  st(1), st ; round(x*log2e); x*log2e - round(x*log2e)
fxch  st(1)     ; x*log2e - round(x*log2e); round(x*log2e)
f2xm1           ; 2^(x*log2e - round(x*log2e))-1; round(x*log2e)
fld1            ; 1,2^(x*log2e - round(x*log2e))-1; round(x*log2e)
fadd            ; 2^(x*log2e - round(x*log2e)); round(x*log2e)
fscale          ;2^(x*log2e - round(x*log2e))*2^round(x*log2e) ; round(x*log2e)
                ;2^( x*log2e - round(x*log2e)+round(x*log2e))= 2^( x*log2e) ; round(x*log2e)
fstp  st(1)      ;2^( x*log2e - round(x*log2e)+round(x*log2e))= 2^( x*log2e)
fstp a
```


Przykład

$a = \log_b x = \log_2 x / \log_2 b$

```
fld1            ;1
fld x           ;x; 1
fyl2x           ;log2x
fld1            ;1; log2x
fld b           ;b; 1; log2x
fyl2x           ;log2b; log2x
fdiv            ;a
fst a
```


Operacje porównania

FCOM	porównanie liczb zmiennoprzecinkowych
FCOMP	porównanie liczb zmiennoprzecinkowych i zdjęcie ze stosu
FCOMPP	porównanie liczb zmiennoprzecinkowych i podwójne zdjęcie ze stosu
FUCOM	nieuporządkowane porównanie liczb zmiennoprzecinkowych
FUCOMP	nieuporządkowane porównanie liczb zmiennoprzecinkowych i zdjęcie ze stosu
FUCOMPP	nieuporządkowane porównanie liczb zmiennoprzecinkowych i podwójne zdjęcie ze stosu
FICOM	porównanie z liczbą całkowitą
FICOMP	porównanie z liczbą całkowitą i zdjęcie ze stosu
FCOMI	porównanie liczb zmiennoprzecinkowych i ustawienie EFLAGS
FUCOMI	nieuporządkowane porównanie liczb zmiennoprzecinkowych i ustawienie EFLAGS
FCOMIP	porównanie liczb zmiennoprzecinkowych, ustawienie EFLAGS i zdjęcie ze stosu
FUCOMIP	nieuporządkowane porównanie liczb zmiennoprzecinkowych, ustawienie EFLAGS i zdjęcie ze stosu
FTST	porównanie z liczbą 0.0
FXAM	sprawdzenie liczby zmiennoprzecinkowej

Wpływa na flagi: C3 C2 C0

Instrukcja FCOM/FCOMP/FCOMPP

fcom/fcomp źródło

fcompp

Porównanie liczb zmiennoprzecinkowych st i źródła. Źródłem może być rejestr st(i) lub zmienna. Jeśli źródło nie jest podane, to jest nim st(1). Fcomp zdejmuje liczbę z wierzchołka stosu, fcompp zdejmuje dwie liczby.

st(0) <=>? źródło

fpop ;dla fcomp,fcompp

fpop ;dla fcompp

fcom st(3)

Stan flag po porównaniu

Flagi koprocesora C3, C2, Co odpowiadają flagom ZF, PF i CF procesora.

relacja \ flagi	C3 ZF	C2 PF	Co CF
st(0)>źródło	0	0	0
st(0)<źródło	0	0	1
st(0)=źródło	1	0	0
nieuporządkowane*	1	1	1

*flagi nie są ustawiane, jeśli wystąpi niezamaskowany wyjątek #IA

Wpływa na flagi: C3 C2 C0

Instrukcja FUCOM/FUCOMP/FUCOMPP

fucom/fucomp st(i)
fucompp

Porównanie liczb zmiennoprzecinkowych st i st(i). Jeśli rejestr nie jest podany, to jest nim st(1). Fucomp zdejmuje liczbę z wierzchołka stosu, fucompp zdejmuje dwie liczby. Nie zgłaszają wyjątku #IA dla nieliczb pasywnych.

st(0) <=>? st(i)

```
fpop    ;dla fucomp,fucompp
fpop    ;dla fucompp
```

fucomp st(7)

Wpływa na flagi: C3 C2 C0

Instrukcja FICOM/FICOMP

ficom/ficomp zmienna

Porównanie st z liczbą całkowitą w pamięci (16/32). Ficomp zdejmuje liczbę z wierzchołka stosu.

st(0) <=>? zmienna

```
fpop    ;dla ficomp
```

ficom liczba_c

Wpływa na flagi: ZF PF CF

Instrukcja FCOMI/FCOMIP/FUCOMI/FUCOMIP

fcomi/fcomip st(i)
fucomi/fucomip st(i)

Porównanie st z liczbą w st(i). Fcompi/fucomip zdejmuje liczbę z wierzchołka stosu. **Ustawia flagi: ZF, PF i CF.** Fucomi i fucomip nie zgłaszają wyjątku #IA dla nieliczb pasywnych

st(0) <=>? st(i)

```
fpop    ;dla fcomip/fucomip
```

fucomi st(4)

Wpływa na flagi: C3 C2 C0

Instrukcja FTST

ftst

Porównanie liczb zmiennoprzecinkowych st i 0.0.

st(0) <=>? 0.0

ftst

Wpływa na flagi: C3 C2 C1 C0

Instrukcja FXAM

fxam

Sprawdza liczbę na wierzchołku stosu. C1 = znak liczby.

znaczenie \ flagi	C3 ZF	C2 PF	Co CF
nieznormalizowana, pseudo (nie)liczba	0	0	0
nieliczba	0	0	1
znormalizowana	0	1	0
nieskończoność	0	1	1
zero	1	0	0
rejestr pusty	1	0	1
zdenormalizowana	1	1	0

Operacje sterowania koprocesorem

- FINCSTP zwiększenie rejestru wskaźnika stosu koprocesora
- FDECSTP zmniejszenie rejestru wskaźnika stosu koprocesora
- FFREE zwolnienie rejestru zmiennoprzecinkowego
- FINIT inicjalizacja koprocesora po sprawdzeniu zgłoszenia błędu numerycznego
- FNINIT inicjalizacja koprocesora bez sprawdzenia zgłoszenia błędu numerycznego
- FCLEX zerowanie flag błędów numerycznych po sprawdzeniu zgłoszenia błędu numerycznego
- FNCLEX zerowanie flag błędów numerycznych bez sprawdzenia zgłoszenia błędu numerycznego
- FSTCW zapamiętanie rejestru sterowania po sprawdzeniu zgłoszenia błędu numerycznego
- FNSTCW zapamiętanie rejestru sterowania bez sprawdzenia zgłoszenia błędu numerycznego
- FLDCW wczytanie rejestru sterowania
- FSTENV zapamiętanie środowiska koprocesora po sprawdzeniu zgłoszenia błędu numerycznego
- FNSTENV zapamiętanie środowiska koprocesora bez sprawdzenia zgł. błędu numerycznego
- FLDENV wczytanie środowiska koprocesora
- FSAVE zapamiętanie zawartości koprocesora po sprawdzeniu zgłoszenia błędu numerycznego
- FNSAVE zapamiętanie zawartości koprocesora bez sprawdzenia zgłoszenia błędu numerycznego
- FRSTOR wczytanie zawartości koprocesora
- FSTSW zapamiętanie rejestru stanu po sprawdzeniu zgłoszenia błędu numerycznego
- FNSTSW zapamiętanie rejestru stanu bez sprawdzenia zgłoszenia błędu numerycznego
- WAIT/FWAIT czekanie na koprocesor
- FNOP nic nie robi

Instrukcja FINCSTP

fincstp

Zwiększenie rejestru wskaźnika stosu koprocesora. Nie usuwa liczby ze stosu.

top := (top + 1) mod 8

fincstp

Instrukcja FDECSTP

fdecstp

Zmniejszenie rejestru wskaźnika stosu koprocesora.

top:= (top - 1) mod 8

fdecstp

Instrukcja FFREE

ffree st(i)

Zwolnienie rejestru koprocesora. Rejestr wskaźnika stosu koprocesora nie jest zmieniany.

tag(i) := 11 b

ffree st(3)

Instrukcja FINIT/FNINIT

finit/fninit

Inicjalizacja koprocesora. Fninit inicjalizacja koprocesora bez sprawdzenia zgłoszenia błędu numerycznego.

finit

rejestr stanu = 0

rejestr stanu zawartości rejestrów stosu = ffff h

rejestr sterowania = 37f h

X – interpretacja nieskończoności (tylko 287)
RC – sterowanie zaokrągleniem: do najbliższej(00), w dół (01), w górę (10), obcięcie (11)
PC – sterowanie dokładnością obliczeń(23 (00), 53 (10) i 63 (11) bity)
PM – maskowanie błędu niedokładności wyniku
UM – maskowanie błędu niedomiaru
OM – maskowanie błędu nadmiaru
ZM – maskowanie błędu dzielenia przez zero
DM – maskowanie błędu zdenormalizowanego argumentu
IM – maskowanie błędu niedozwolonej operacji

Instrukcja FCLEX/FNCLEX

fclex/fnclex

Zerowanie flag błędów numerycznych. Fnclex - bez sprawdzenia zgłoszenia błędu numerycznego.

fnclex

Instrukcja FSTCW/FNSTCW

fstcw/fnstcw cel

Zapamiętanie rejestru sterowania. Cel jest dwubajtową zmienną albo rejestrem AX. Fnstcw - bez sprawdzenia zgłoszenia błędu numerycznego.

fnstcw

Instrukcja FLDCW

fldcw źródło

Wczytanie rejestru sterowania. Źródło jest dwubajtową zmienną.

fldcw zmienna

Instrukcja FSTENV/FNSTENV

fstenv/fnstenv cel

Zapamiętanie środowiska koprocesora . Cel jest 14/28 bajtowym obszarem (tryb 16/32 bitowy). Fnstenv - bez sprawdzenia zgłoszenia błędu numerycznego.

fnstenv fsrodowisko

Instrukcja FLDENV

fldenv źródło

Wczytanie środowiska koprocesora . Źródło jest 14/28 bajtowym obszarem (tryb 16/32 bitowy).

fldenv fsrodowisko

Instrukcja FSAVE/FNSAVE

fsave/fnsave cel

Zapamiętanie zawartości koprocesora (środowisko + rejestry zmiennoprzecinkowe). Cel jest 94/108 bajtowym obszarem (tryb 16/32 bitowy). Fnsave - bez sprawdzenia zgłoszenia błędu numerycznego.

fnsave fstan

Instrukcja FRSTOR

frstor źródło

Wczytanie zawartości koprocesora . Źródło jest 94/108 bajtowym obszarem (tryb 16/32 bitowy).

frstor fstan

Instrukcja FSTSW/FNSTSW

fstsw/fnstsw cel

Zapamiętanie rejestru stanu. Cel jest dwubajtową zmienną albo rejestrem AX. Fnstsw - bez sprawdzenia zgłoszenia błędu numerycznego.

fnstsw

Instrukcja WAIT/FWAIT

wait/fwait

Czekanie przez procesor na gotowość koprocesora (na zakończenie wykonywania instrukcji).

fwait

Instrukcja FNOP

fnop

Nic nie robi.

fnop

Przykład

$$\sqrt{\Delta} = \sqrt{b^2 - 4ac}$$

```
fld  b                ;b
fld st                ;b; b
fmul                  ;bb
fld a                 ;a; bb
fmul c                ;ac; bb
fadd st, st           ;2ac; bb
fadd st, st           ;4ac; bb
fsub                  ;bb-4ac
ftst                  ;porównanie z 0,0
fstsw ax              ; ax-stan
sahf                  ;stan do flag
jp   blad
jz   rowne
jc   mniejsze
wieksze: ....
```

```
fld  b                ;b
fld st                ;b; b
fmul                  ;bb
fld a                 ;a; bb
fmul c                ;ac; bb
fadd st, st           ;2ac; bb
fadd st, st           ;4ac; bb
fsub                  ;bb-4ac
fldz                  ;0; bb-4ac
fcomip st, st(1)      ;porównanie z 0,0
jp   blad             ;flagi już ustawione
jz   rowne
jc   wieksze
mniejsze: ....
```


Przykład - maksimum

32 bity

```
;adresy 3 zmiennych na stosie
mov     eax,a
mov     edx,b
fld     qword ptr[eax]      ;a
fld     qword ptr[edx]      ;b,a
fcomi   st(1)
fcmovb  st,st(1)            ;max,a
fstp    st(1)               ;max
mov     eax,m
fstp    qword ptr[eax]
ret
;2 wartości na stosie
fld     a                   ;a
fld     b                   ;b,a
fcomi   st(1)               ;b,a
fcmovb  st,st(1)            ;max,a
fstp    st(1)               ;max
ret
```

64 bity

```
;adresy 3 zmiennych w rejestrach
fld     qword ptr[rcx]      ;a
fld     qword ptr[rdx]      ;b,a
fcomi   st(1)
fcmovb  st,st(1)            ;max,a
fstp    st(1)               ;max
fstp    qword ptr[r8]
ret
```

Instrukcje AVX

Advanced Vector Extensions

Instrukcje AVX

Część I

liczby całkowite

Instrukcje typu SIMD

Single Instruction Multiple Data (SIMD)
przetwarzanych jest wiele zestawów danych
przez jedną instrukcję

Instrukcje typu SIMD

- MMX (MultiMedia eXtensions lub Matrix Math eXtensions)
  - liczby całkowite
- SSE (Streaming SIMD Extenstions)
  - liczby całkowite i liczby zmienno-przecinkowe
- AVX (Advanced Vector Extensions)
  - zastępuje i rozszerza SSE

Instrukcje typu SIMD

Instrukcje AVX
są sposobem zrównoleglenia programów
na poziomie danych

Rejestry AVX

Rejestry AVX ymm oraz zmm

Dla rozkazów AVX dedykowano rejestry:

- 16 rejestrów 256 bitowych dla AVX oraz AVX2
  ymm/15 – ymm/0
- 32 rejestry 512 bitowe dla AVX-512
  zmm/31 – zmm/0
- Część instrukcji (większość) operuje na młodszej części rejestrów
  xmm/[15/31] – xmm/0

Rejestry wektorowe

Rejestry XMM

Przykład: VADDPD xmm1 = xmm2 + xmm3/m128

zmm/7	zmm/6	zmm/5	zmm/4	zmm/3	zmm/2	zmm/1	zmm/0
				ymm/3	ymm/2	ymm/2	ymm/0
						xmm/1	xmm/0
511			256	255	128	127 64	63 0
						xmm2/1	xmm2/0
						+	+
						xmm3/1 lub m128/1	xmm3/0 lub m128/0
						=	=
						xmm1/1	xmm1/0

Rejestry YMM

Przykład: VADDPD ymm1 = ymm2 + ymm3/m256

zmm/7	zmm/6	zmm/5	zmm/4	zmm/3	zmm/2	zmm/1	zmm/0
				ymm/3	ymm/2	ymm/2	ymm/0
						xmm/1	xmm/0
511			256	255	128	127 64	63 0
				ymm2/3	ymm2/2	xmm2/1	xmm2/0
				+	+	+	+
				ymm3/3 lub m256/3	ymm3/2 lub m256/2	xmm3/1 lub m128/1	xmm3/0 lub m128/0
				=	=	=	=
				ymm1/3	ymm1/2	xmm1/1	xmm1/0

Rejestry XMM

Przykład: VADDPD zmm1 = zmm2 + zmm3/m512

zmm/7	zmm/6	zmm/5	zmm/4	zmm/3	zmm/2	zmm/1	zmm/0
				ymm/3	ymm/2	ymm/2	ymm/0
						xmm/1	xmm/0
511			256	255	128	127 64	63 0
zmm2/7	zmm2/6	zmm2/5	zmm2/4	ymm2/3	ymm2/2	xmm2/1	xmm2/0
+	+	+	+	+	+	+	+
zmm3/7 lub m512/7	zmm3/6 lub m512/6	zmm3/5 lub m512/5	zmm3/4 lub m512/4	ymm3/3 lub m256/3	ymm3/2 lub m256/2	xmm3/1 lub m128/1	xmm3/0 lub m128/0
=	=	=	=	=	=	=	=
zmm1/71	zmm1/6	zmm1/5	zmm1/4	ymm1/3	ymm1/2	xmm1/1	xmm1/0

Typy danych AVX

Typy danych dla AVX

- Liczby całkowite (packed)

Nazwa	zakres	Oznaczenie w AVX
Bajt	8 bitów	B
Słowo	16 bitów	W
Podwójne słowo	32 bity	D
Poczwórne słowo	64 bity	Q

Typy danych dla AVX

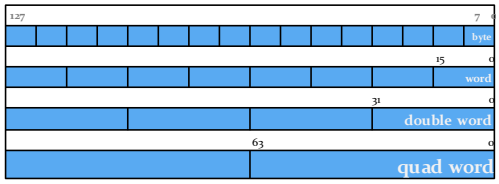
- Liczby zmienno-przecinkowe (packed i scalar)

nazwa	zakres	Oznaczenie w AVX
Pojedyncza precyzja	32 bity	SS, PS
Podwójna precyzja	64 byty	SD, PD

- Specyfikatory zmiany typu:
  xmmword ptr [adres]
  ymmword ptr [adres]

Typy danych AVX liczby całkowite

Przykład na rejestrach 128 bitowych **xmm**



Typy danych AVX

Liczby zmienno-przecinkowe

PS – *vector* liczb pojedynczej precyzji
PD - *vector* licb podwójnej precyzji
SS – *scalar* pojedynczej precyzji
SD – *scalar* podwójnej precyzji

Powyższe oznaczenia mają odzwierciedlenie w nazwach instrukcji, „mnemonikach" dla liczb zmienno-przecinkowych.

Instrukcje AVX

Instrukcje typu AVX

Pytanie: po co stosować instrukcje typu *scalar* skoro wykorzystują jedną najmłodszą część rejestru.

1. nie zawsze działamy na wektorach.
2. pominięcie koprocesora.
3. gdy liczba danych jest niepodzielna przez liczbę elementów wektora.

Jeśli AVX operuje na „skalarach", starsze części rejestru są prawie zawsze zerowane.

Instrukcje typu AVX

Instrukcje AVX podobnie jak instrukcje SSE są instrukcjami wektorowymi, jednak **nie poleca się łączenia w jednym programie/podprogramie instrukcji AVX z SSE**, ponieważ powoduje to **znaczne spowolnienie działania programu/podprogramu**.

Instrukcje typy AVX

Intel® Integrated Performance Primitives

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX,AVX2,FMA
- https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/ipp.html

Ekwiwalent np. dla **VADDPD**

```
__m128d _mm128_add_pd (__m128d a, __m128d b);
__m256d _mm256_add_pd (__m256d a, __m256d b);
__m512d _mm512_add_pd (__m512d a, __m512d b);
```


Systematyka instrukcji AVX

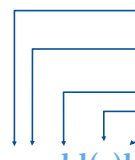
- Dokumentacja firmy Intel wyróżnia 12 kategorii instrukcji AVX.
- Na potrzeby niniejszego wykładu zastosowano podział na instrukcje AVX dotyczące liczb całkowitych, liczb zmienno-przecinkowych oraz oddzielną grupę FMA, która w AVX operuje wyłącznie na liczbach zmienno-przecinkowych. W AVX dostępna jest również grupa instrukcji szyfrujących wykorzystująca algorytm AES (ang. *Advanced Encryption Standard*)

Systematyka instrukcji AVX

- Pośród 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
- Najogólniejszą systematyką instrukcji AVX jest podział na instrukcje dla liczb całkowitych i dla liczb zmienno-przecinkowych, w tych ostatnich sytuują się instrukcje FMA.

Budowa rozkazu AVX

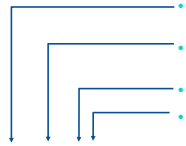
Budowa rozkazu AVX liczby całkowite



- Mnemoniki prawie wszystkich rozkazów AVX rozpoczynają się od litery **v** (od słowa Vector);
- 1 literowy skrót **p** od „packed", jednak umieszczony na początku rozkazu określa operacje na liczbach całkowitych;
- 3-4 literowy skrót wykonywanego działania (add,sub,mul...);
- niektóre instrukcje są z nasyceniem „saturation" skrót **s**;
- jednoliterowy skrót określa zakres operacji, może być (B) bytes, (W) word, (D) double word, (Q) quad word.

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

Budowa rozkazu AVX liczby zmienno-przecinkowe

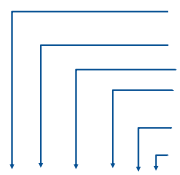


- Mnemoniki prawie wszystkich rozkazów AVX rozpoczynają się od litery **v** (od słowa Vector);
- 3-4 literowy skrót wykonywanego działania (np. add, mul, itp);
- litera **p** lub **s** określa, eng. packed lub eng. scalar;
- litera **d** lub **s** oznacza stopień precyzji: double lub single.

vaddpd

Rozkaz **vaddpd** wykonuje dodawanie (add) wektorowo/równolegle (p) liczb zmienno-przecinkowych podwójnej precyzji (d).

Budowa rozkazu AVX operacje FMA



- Mnemoniki prawie wszystkich rozkazów AVX rozpoczynają się od litery **v** (od słowa Vector);
- 2 literowy skrót dla FMA (skumulowane wyniki dodawania);
- 3-4 literowy skrót wykonywanego działania (add,sub,mul, itp);
- 3 cyfry określające kolejność mnożenia i dodawania, może być 132, 213, 231 (np. 132 mnoży 1. x 3. i dodaje 2.);
- litera p lub s określa sposób wykorzystania rejestru eng. „packed", lub „scalar";
- litera **d** lub **s** oznacza stopień precyzji „double" lub „single";

vfmadd132pd

Rozkaz **vfmadd132pd** mnoży (**m**) i dodaje (**add**) równolegle/packed (**p**) liczby zmiennoprzecinkowe podwójnej precyzji (**d**), kolejność operacji arytmetycznych jest oznaczona przez trzy cyfry **132,** czyli wg przykładu mnoży 1. i 3. argument, do iloczynu dodaje 2. argument.

Bajt sterujący imm8

Część instrukcji AVX
wykorzystuje, jako argument, **bajt sterujący** imm8

imm8[7]	imm8[6]	imm8[5]	imm8[4]	imm8[3]	imm8[2]	imm8[1]	imm8[0]
1	0	1	0	1	1	1	0

W instrukcjach bajt sterujący najczęściej jest zapisywany w postaci liczby szesnastkowej.

np. 0ech

Instrukcje AVX

dla liczb całkowitych

Instrukcje AVX dla liczb całkowitych

- Instrukcje przesłania
- Operacje matematyczne
- Operacje porównania
- Operacje przesunięcia (bitowe, arytmetyczne, logiczne)
- Instrukcje logiczne
- Instrukcje zerowania
- Instrukcje wyrównania
- Instrukcje dodatkowe (ładowanie ustawień)

Operacje przesłania AVX

- **Instrukcje przesłania:**

VMOVD, VMOVQ
VMOVDQA, VMOVDQU, VMOVNTDQA
VMOVNTDQ, VLDDQU
VPMOV[S/Z]XBW, VPMOV[S/Z]XBD
VPMOV[S/Z]XBQ, VPMOV[S/Z]XWD
VPMOV[S/Z]XWQ, VPMOV[S/Z]XDQ
VPMOVMSKB, VMASKMOVDQU, VPMASKMOV[D/Q]

Operacje przesłania AVX

- **Instrukcje kompresji/rozpakowania:**
VPACK[S/U]SWB, VPACK[S/U]SDW,
VPUNPCKHBW, VPUNPCKHWD, VPUNPCKHDQ,
VPUNPCKHQDQ, VPUNPCKLBW, VPUNPCKLWD,
VPUNPCKLDQ, VPUNPCKLQDQ

Operacje przesłania AVX

- **Instrukcje przetasowania:** VPSHUFB, VPSHUFD VPSHUFHW, VPSHUFLW
- **Instrukcje permutacji:** VPERMD, VPERMQ
- **Instrukcje mieszające:** VPBLENDB, VPBLENDW, VPBLENDD
- **Instrukcje rozgłaszania:** VPBROADCASTB, VPBROADCASTW VPBROADCASTD, VPBROADCASTQ
- **Instrukcje zbierania:** VPGATHERDD, VPGATHERQD VPGATHERDQ, VPGATHERQQ

Instrukcja przesłania
VMOV[D/Q]

vmovd xmm1, r/m32 xmm1 ← r/m32
vmovq xmm1, r/m64 xmm1 ← r/m64
vmovq xmm1, xmm2/m64 xmm1 ← xmm2/m64

vmovd r/m32, xmm1 r/m32 ← xmm1
vmovq r/m64, xmm1 r/m64 ← xmm1
vmovq xmm1/m64, xmm2 xmm1/m64 ← xmm2

VMOVD / VMOVQ **przesyła** podwójne lub poczwórne słowo z rejestru ogólnego przeznaczenia/pamięci do rejestru xmm lub odwrotnie. Używany jest jeden najmłodszy element rejestru xmm, starsze elementy są zerowane.

Instrukcja przesłania z wyrównaniem
VMOVDQ[A/U]

vmovdq[a/u] xmm1, xmm2/m128
vmovdq[a/u] ymm1, ymm2/m256

vmovdq[a/u] xmm2/m128, xmm1
vmovdq[a/u] ymm2/m256, ymm1

Przesyła **całą zawartość (128 lub 256 bitów)** źródła do celu, jeśli **celem lub źródłem** jest **pamięć**, wówczas w wersji (A) **dane w pamięci muszą być wyrównane** (ang. *aligne*) do granicy **16** (m128) lub **32** (m256) bajtów, w przeciwnym wypadku zgłaszany jest wyjątek ochrony pamięci. Aby przesłać dane do/z niewyrównanych lokalizacji pamięci należy w instrukcji zamiast (A) użyć (U) (ang. *unaligne*) .

cel (r) ← źródło (r/m) cel (r/m) ← źródło

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania z wyrównaniem
VMOVNTDQ[A]

vmovntdqa xmm1, m128 cel (r) ← źródło (m)
vmovntdqa ymm1, m256

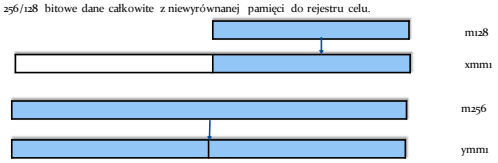
vmovntdq m128, xmm1 cel (m) ← źródło (r)
vmovntdq m256, ymm1

Przesyła **całą zawartość** z/do pamięci m128/m256 do/z rejestru xmm1/ymm1.
Dla instrukcji w wersji (A) **pamięć musi być wyrównana** (ang. *aligne*) do granicy **16** (m128) lub **32** (256) bajtów, w przeciwnym wypadku zgłaszany jest wyjątek ochrony pamięci, alternatywnie można zastosować instrukcję bez litery A. **NT** oznacza (non-temporal hint) przesłanie z pominięciem pamięci podręcznej (cache).

Bity od 128/256 do MSB są zerowane.

Instrukcja ładowanie danych z pamięci
VLDDQU

vlddqu xmm1, m128
vlddqu ymm1, m256
Ładuje 256/128 bitowe dane całkowite z niewyrównanej pamięci do rejestru celu.



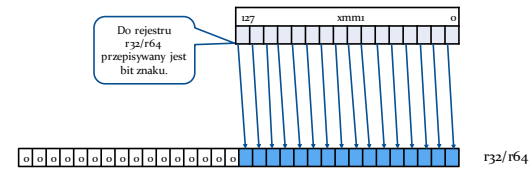
Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania
VPMOVMSKB

vpmovmaskb reg, xmm1 rejestr ← xmm1
vpmovmaskb reg, ymm1 rejestr ← ymm1 (AVX2)

Przesyła najstarsze bity (bity znaku) **każdego bajtu** rejestru xmm1/ymm1 po kolei do rejestru r32/r64, dla xmm przenosi **16 bitów** do r32, dla ymm przenosi **32 bity** do r64, pozostałe starsze bity są zerowane.

Instrukcja przesłania
VPMOVMSKB



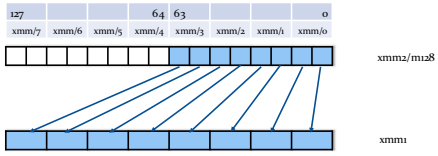
Instrukcje przesłania
z konwersją

Instrukcje przesłania z konwersją
VPMOV[S/Z]XB[W/D/Q], VPMOV[S/Z]XW[D/Q] VPMOV[S/Z]XDQ

vpmov[s/z]xbw xmm1, xmm2/m64
vpmov[s/z]xbd xmm1, xmm2/m32
vpmov[s/z]xbq xmm1, xmm2/m16
vpmov[s/z]xwd xmm1, xmm2/m64
vpmov[s/z]xwq xmm1, xmm2/m32
vpmov[s/z]xdq xmm1, xmm2/m64

Instrukcja zamienia (konwertuje) ze znakiem / bez znaku:
bajty na słowa/podwójne słowa/poczwórne słowa, **słowa na** podwójne słowa/poczwórne słowa, **podwójne słowa na** poczwórne słowa przepisując odpowiednio wartości z młodszej części rejestru xmm2/(m64||m32||m16) **do rejestru celu xmm1.**

Instrukcja przesłania z konwersją
VPMOVSXBW

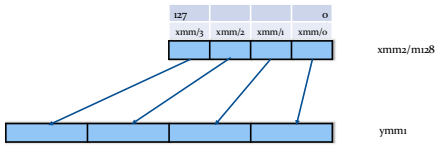


Instrukcje przesłania z konwersją
VPMOV [S/Z] XB[W/D/Q], VPMOV [S/Z] XW[D/Q], VPMOV [S/Z] XDQ (AVX2)

vpmov[s/z]xbw ymm1, xmm2/m128
vpmov[s/z]xbd ymm1, xmm2/m64
vpmov[s/z]xbq ymm1, xmm2/m32
vpmov[s/z]xwd ymm1, xmm2/m128
vpmov[s/z]xwq ymm1, xmm2/m64
vpmov[s/z]xdq ymm1, xmm2/m128

Instrukcja **zamienia** (konwertuje) ze znakiem / bez znaku:
bajty na słowa/podwójne słowa/poczwórne słowa, **słowa na** podwójne słowa/poczwórne słowa, **podwójne słowa na** poczwórne słowa przepisując odpowiednio wartości z młodszej części z rejestru ymm2/(m128||m64||m32) do rejestru celu **ymm1,** na młodszą część rejestru.

Instrukcja przesłania z konwersją
VPMOVSXDQ



Instrukcje przesłania - przykład:

```
vmovdqu ymm1,ymmword ptr[rdi]          ; rdi = int * tab1[]
vmovdqu ymm1,ymmword ptr[rdi+4*rax]    ; rdi = int * tab1[n] ; rax = n

vmovdqu ymm2, ymmword ptr[rsi]         ; rsi = int * tab2[]
vmovdqu ymm2, ymmword ptr[rsi+4*rcx]   ; rsi = int * tab2[m] ; rcx = m
```

Dla typu tablicowego ładowanie całego rejestru ymm1/ymm2 z adresu pierwszego (0) elementu tablicy oraz od kolejnych elementów (wielokrotność 4) o wielkości podwójnego słowa.

```
vmovdqa xmm6, xmmword ptr [ebx]   ; ebx = unsigned short * a
vpmovzxwd ymm5, xmm6              ; konwersja z 16 do 32 bitów
```

Ponieważ obliczenia na 16 bitach mogły by doprowadzić do przepełnienia (overflow), typ `unsigned short` konwertujemy na `unsigned int` 32 bity zachowując przy tym ilość elementów wektora równą 8.

Instrukcje przesłania

warunkowego

Instrukcja przesłania warunkowego
VMASKMOVDQU

vmaskmovdqu xmm1, xmm2

Celem jest obszar 16 bajtów pamięci adresowany przez DS: DI / EDI / RDI.
Bajty ze źródła xmm1 są przesyłane do celu pod warunkiem, że siódme bity (bity znaku) odpowiadających im bajtów z xmm2 są jedynkami.

xmm2 = maska

if xmm2[i][7] = 1 then m128[i] = xmm1[i]

i – jest numerem bajtu

Instrukcja przesłania warunkowego
VPMASKMOV[D/Q]

vpmaskmov[d/q] xmm1, xmm2, m128
vpmaskmov[d/q] ymm1, ymm2, m256
vpmaskmov[d/q] m128, xmm1, xmm2
vpmaskmov[d/q] m256, ymm1, ymm2

Przesyła podwójne/poczwórne słowa z pamięci m128/m256 lub xmm2/ymm2 do rejestru celu xmm2/ymm2 lub pamięci m128/m256 , pod warunkiem, że **bit znaku** odpowiadających wartości z rejestru maski (drugi operand) xmm2/ymm2 lub xmm1/ymm1 **jest ustawiony na jeden,** w przeciwnym wypadku zapisuje zero.

if źródło1[i][31/63] == 1 then cel[i] ← źródło2[i] else cel[i] ← 0

Bity od 128/256 do MSB są zerowane.

Instrukcje kompresji

Instrukcja kompresji

VPACK[S/U]SWB

vpack[s/u]swb xmm1, xmm2, xmm3/m128
vpack[s/u]swb ymm1, ymm2, ymm3/m256 (AVX2)

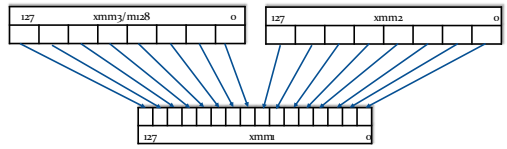
Konwertuje słowa **ze znakiem** na bajty ze znakiem (S) / bez znaku(U), z rejestru xmm2 / ymm2 wpisuje **do młodszych** części rejestru xmm1, a z xmm3/m128 / ymm3/ m256 wpisuje **do starszych** części rejestru xmm1/ ymm1. Starsza część rejestru ymm1 jest wypełniana danymi ze starszych części rejestrów hi ymm2 i hi ymm3/m256. Wynik jest zapisywany ze znakiem(S) / bez znaku(U) oraz z nasyceniem.

Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPACK[S/U]SWB

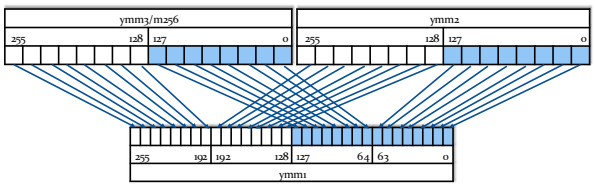
vpack[s/u]swb xmm1, xmm2, xmm3/m128



Instrukcja kompresji

VPACK[S/U]SWB

vpack[s/u]swb ymm1, ymm2, ymm3/m256 (AVX2)



Instrukcja kompresji

VPACK[S/U]SDW

vpack[s/u]sdw xmm1, xmm2, xmm3/m128
vpack[s/u]sdw ymm1, ymm2, ymm3/m256 (AVX2)

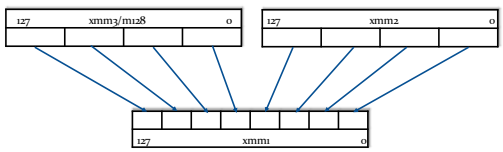
Konwertuje podwójne słowa **ze znakiem** na słowa ze znakiem(S)/bez znaku(U), z rejestru xmm2/ lo ymm2 wpisuje **do młodszych** części rejestru xmm1/ lo ymm1, z xmm3/m128 / lo ymm3/ lo m256 wpisuje **do starszych** części rejestru celu xmm1/lo ymm1. Starsza część rejestru ymm1, jest wypełniana danymi ze starszych części rejestrów hi ymm2 oraz hi ymm3/m256. Wynik jest zapisywany ze znakiem(S) / bez znaku(U) oraz z nasyceniem.

Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPACK[S/U]SDW

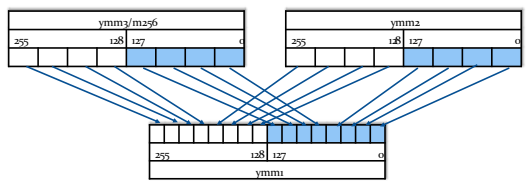
vpack[s/u]sdw xmm1, xmm2, xmm3/m128



Instrukcja kompresji

VPACK[S/U]SDW

vpack[s/u]sdw ymm1, ymm2, ymm3/m256 (AVX2)

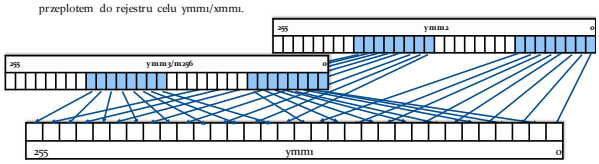


Instrukcja kompresji

VPUNPCKLBW / VPUNPCKLWD / VPUNPCKLDQ / VPUNPCKLQDQ

vpunpcklbw xmm1, xmm2, xmm3/m128
vpunpcklbw ymm1, ymm2, ymm3/m256 (AVX2)

Młodsze bajty ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.

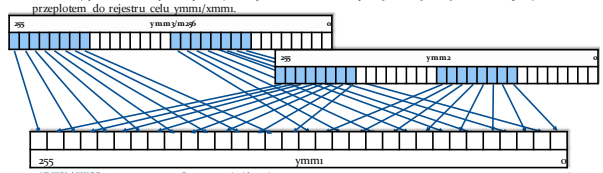


Instrukcja kompresji

VPUNPCKHBW / VPUNPCKHWD / VPUNPCKHDQ / VPUNPCKHQDQ

vpunpckhbw xmm1, xmm2, xmm3/m128
vpunpckhbw ymm1, ymm2, ymm3/m256 (AVX2)

Starsze bajty ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.

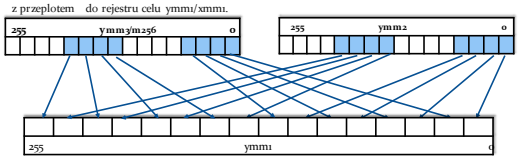


Instrukcja kompresji

VPUNPCKLBW / VPUNPCKLWD / VPUNPCKLDQ / VPUNPCKLQDQ

vpunpcklwd xmm1, xmm2, xmm3/m128
vpunpcklwd ymm1, ymm2, ymm3/m256 (AVX2)

Młodsze słowa ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.



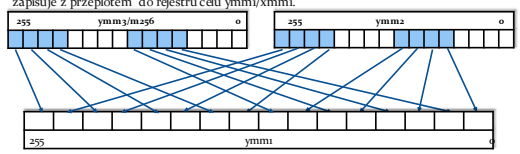
Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPUNPCKHBW / VPUNPCKHWD / VPUNPCKHDQ / VPUNPCKHQDQ

vpunpckhwd xmm1, xmm2, xmm3/m128
vpunpckhwd ymm1, ymm2, ymm3/m256 (AVX2)

Starsze słowa ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.



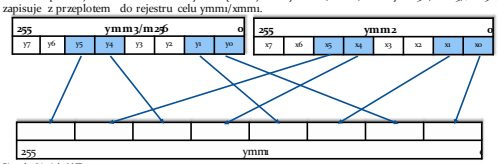
Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPUNPCKLBW / VPUNPCKLWD / VPUNPCKLDQ / VPUNPCKLQDQ

vpunpckldq xmm1, xmm2, xmm3/m128
vpunpcklqdq ymm1, ymm2, ymm3/m256 (AVX2)

Młodsze podwójne słowa ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.



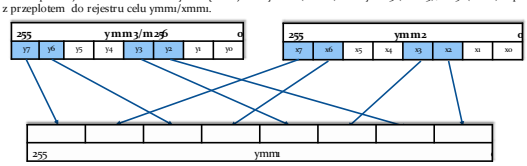
Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPUNPCKHBW / VPUNPCKHWD / VPUNPCKHDQ / VPUNPCKHQDQ

vpunpckhdq xmm1, xmm2, xmm3/m128
vpunpckhqdq ymm1, ymm2, ymm3/m256 (AVX2)

Starsze podwójne słowa ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.



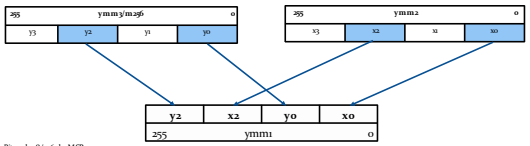
Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPUNPCKLBW / VPUNPCKLWD / VPUNPCKLDQ / **VPUNPCKLQDQ**

vpunpcklqdq xmm1, xmm2, xmm3/m128
vpunpcklqdq ymm1, ymm2, ymm3/m256 (AVX2)

Młodsze poczwórne słowa ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.



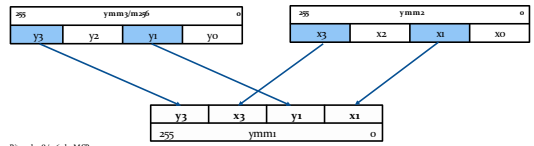
Bity od 128/256 do MSB są zerowane.

Instrukcja kompresji

VPUNPCKHBW / VPUNPCKHWD / VPUNPCKHDQ / **VPUNPCKHQDQ**

vpunpckhqdq xmm1, xmm2, xmm3/m128
vpunpckhqdq ymm1, ymm2, ymm3/m256 (AVX2)

Starsze poczwórne słowa ze 128 bitowych części rejestru ymm2 (xmm2) oraz ymm3 (xmm3)/m256(m128) zapisuje z przeplotem do rejestru celu ymm1/xmm1.



Bity od 128/256 do MSB są zerowane.

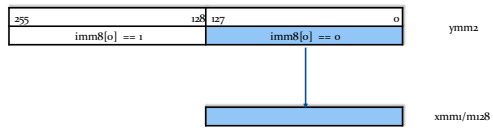
Instrukcje wyłuskiwania / wstawiania

Instrukcja wyłuskiwania

VEXTRACTI128 / VINSERTI128

vextracti128 xmm1/m128, ymm2, imm8 (AVX2)

Przepisuje 128 bitów z rejestru ymm2 do rejestru xmm1 lub m128. Jeśli zerowy bitu bajtu sterującego imm8[0] = 0 przepisuje młodszą część, a jeśli imm8[0] = 1 przepisuje starszą część rejestru ymm2.

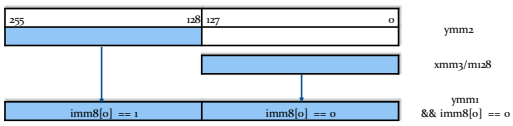


Instrukcja wstawiania

VEXTRACTI128 / **VINSERTI128**

vinserti128 ymm1, ymm2, xmm3/m128, imm8 (AVX)

Przepisuje cały rejestr ymm2 do ymm1 następnie przepisuje rejestr xmm3 lub m128 również do rejestru celu ymm1 zależnie od ustawienia bitu bajtu sterującego: imm8[0] = 0 przepisuje xmm3/m128 na młodszą część, gdy imm8[0] = 1 na starszą część celu ymm1.



Instrukcje tasowania

Instrukcja tasowania

VPSHUFB

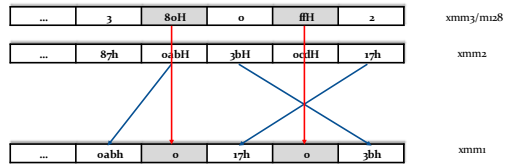
vpshufb xmm1, xmm2, xmm3/m128
vpshufb ymm1, ymm2, ymm3/m256 (AVX2)

Tasuje bajty z xmm2/ymm2, **w zależności od bitu znaku** kolejnych bajtów rejestru xmm3/m128 / ymm3/m256. Jeśli bit znaku jest ustawiony odpowiedni **bajt** rejestru celu xmm1/ymm1 **jest zerowany**, jeśli bit znaku xmm3/ymm3 / m128/m256 nie jest ustawiony wówczas z takiego bajtu jest tworzony 4 bitowy **indeks wskazujący numer bajtu ze 128-bitowej części**, który ma być przepisany z xmm2/ymm2 do właściwego xmm1/ymm1.

```
i = numer bajtu
if xmm3/m128[i][7] == 1 then xmm1[i] = 0
else { index[3..0] = xmm3/m128[i][3..0]
       xmm1[i] = xmm2[index]
     }
```


Instrukcja tasowania

VPSHUFB

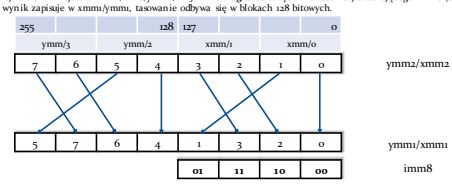


Instrukcja tasowania

VPSHUFD

vpshufd xmm1, xmm2/m128, imm8
vpshufd ymm1, ymm2/m256, imm8 (AVX2)

Tasuje podwójne słowa z rejestru xmm2/m128/ymm2/m256 **według** dwubitowych wartości bajtu sterującego imm8 (argument kolejności), wynik zapisuje w xmm1/ymm1, tasowanie odbywa się w blokach 128 bitowych.

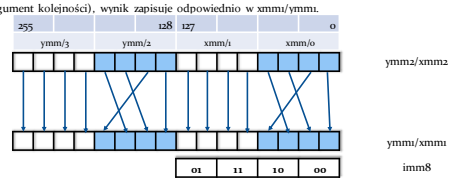


Instrukcja tasowania

VPSHUFLW

vpshuflw xmm1, xmm2/m128, imm8 (AVX)
vpshuflw ymm1, ymm2/m256, imm8 (AVX2)

Tasuje wektory **młodszych słów** z rejestru xmm2/ymm2 według **dwubitowych wartości** bajtu sterującego imm8 (argument kolejności), wynik zapisuje odpowiednio w xmm1/ymm1.

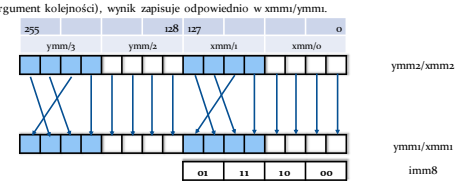


Instrukcja tasowania

VPSHUFHW

vpshuflw xmm1, xmm2/m128, imm8 (AVX)
vpshufhw ymm1, ymm2,m256, imm8 (AVX2)

Tasuje wektory **starszych słów** z rejestru xmm2/ymm2 według **dwubitowych wartości** bajtu sterującego imm8 (argument kolejności), wynik zapisuje odpowiednio w xmm1/ymm1.



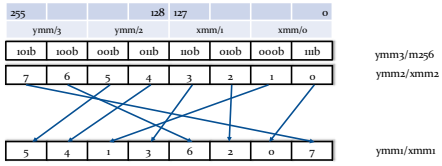
Instrukcje permutacji

Instrukcja tasowania

VPERMD

vpermd ymm1, ymm2, ymm3/m256 (AVX2)

Wykonuje permutacje wektorów podwójnych słów z rejestru **ymm3/m256** według porządku podanego w **ymm2**, najmłodsze 3 bity odpowiedniego podwójnego słowa rejestru ymm2 wyznaczają, z którego miejsca w ymm3/m256 zostanie skopiowane podwójne słowo do miejsca położenia „adresu" (ymm2). Wynik jest zapisywany w **ymm1.**

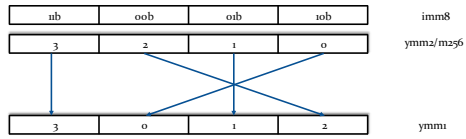


Instrukcja permutacji

VPERMQ

vpermq ymm1, ymm2/m256, imm8 (AVX2)

Wykonuje **permutacje** wektorów **poczwórnych słów** z rejestru **ymm2/m256** według porządku określonego w **imm8**, kolejne dwubitowe pola **imm8** określają, spod którego indeksu „adresu" zostaną skopiowane poczwórne słowa z **ymm2/m256.** Wynik jest zapisywany do **ymm1.**



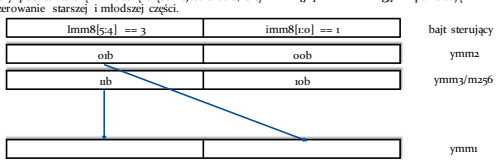
Bity od 128/256 do MSB są zerowane.

Instrukcja permutacji

VPERM2I128

vperm2i128 ymm1, ymm2, ymm3/m256, imm8 (AVX2)

Wykonuje **permutacje** dwóch wektorów 128 bitowych z rejestrów **ymm2** oraz **ymm3/m256**, bajt sterujący **imm8** odpowiada za sposób przepisania, pola imm8[5:4] i imm8[1:0] są indeksami wskazującymi skąd należy pobrać starszą i młodszą część rejestru celu, bity **imm8[7]= 1** i **imm8[3] = 1** powodują wyzerowanie starszej i młodszej części.



Instrukcja permutacji przykład: transponowanie macierzy dynamicznej

```
void Transponuj4x4(double **tab) { __asm {

push esi;
mov esi, tab
mov eax, [esi]
mov ecx, [esi + 8]
mov edx, [esi + 12]
mov esi, [esi + 4]
vmovdqu ymm0, ymmword ptr[eax]
vmovdqu ymm1, ymmword ptr[ecx]
vperm2i128 ymm2, ymm0, ymm1, 20h
vperm2i128 ymm4, ymm0, ymm1, 31h
vmovdqu ymm0, ymmword ptr[esi]
vmovdqu ymm1, ymmword ptr[edx]
vperm2i128 ymm3, ymm0, ymm1, 20h
vperm2i128 ymm5, ymm0, ymm1, 31h
vpunpckldqd ymm0, ymm2, ymm3
vpunpckhqdq ymm1, ymm2, ymm3
vpunpckldqd ymm2, ymm4, ymm5
vpunpckhqdq ymm3, ymm4, ymm5

vmovdqu ymmword ptr[eax], ymm0
vmovdqu ymmword ptr[esi], ymm1
vmovdqu ymmword ptr[ecx], ymm2
vmovdqu ymmword ptr[edx], ymm3

pop esi;
}
}
```


Instrukcje mieszające

Instrukcja mieszająca

VPBLENDVB

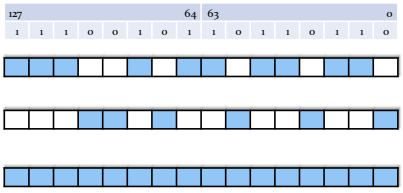
vpblendvb xmm1, xmm2, xmm3/m128, xmm4 (AVX)
vpblendvb ymm1, ymm2, ymm3/m256, ymm4 (AVX2)

Miesza wektory bajtów z rejestru xmm2/ymm2 oraz xmm3/ymm3 lub m128/m256 **według bitu znaku** każdego bajtu w xmm4/ymm4, wynik zapisuje w xmm1/ymm1.

```
i <0, 15> lub <0, 31> - indeks bajtu
if źródło3 [i][7] = 1   => cel[i] = źródło2[i]
else  cel[i] = źródło1[i]

if xmm4 [i][7] = 1   => xmm1[i] = xmm3/m128[i]
else  xmm1[i] = xmm2[i]

if  ymm4[i][7] = 1   => ymm1[i] = ymm3/m256[i]
else  ymm1[i] = ymm2[i]
```

Bity od 128/256 do MSB są zerowane.

Instrukcja mieszająca
VPBLENDVB



Instrukcja mieszająca
VPBLENDW

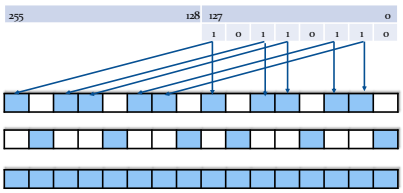
vpblendw xmm1, xmm2, xmm3/m128, imm8 (AVX)
vpblendw ymm1, ymm2, ymm3/m256, imm8 (AVX2)

W oparciu o bajt kontrolny miesza wektory słów; wybiera elementy wektora z rejestru xmm3/ymm3 lub m128/m256 dla imm8[i] = 1, albo elementy wektora xmm2/ymm2 dla imm8[i] = 0. Dla indeksu 8-15 należy wziąć imm8[i-8]. Wynik zapisuje w xmm1/ymm1.

```
i <0, 7> lub <0, 15> - indeks słowa
if imm8[i modulo 8] = 1 then cel[i] = źródło2[i]
else cel[i] = źródło1[i]

if imm8[i]= 1 then xmm1[i] = xmm3/m128[i]
else xmm1[i] = xmm2[i]

if imm8[i modulo 8] = 1 then ymm1[i] = ymm3/m256[i]
else ymm1[i] = ymm2[i]
```

Bity od 128/256 do MSB są zerowane.

Instrukcja mieszająca
VPBLENDW



Instrukcja mieszająca
VPBLENDD

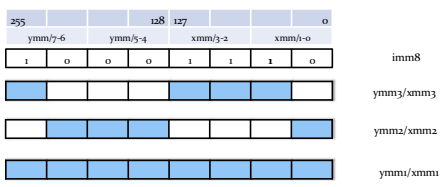
vpblendd ymm1, ymm2, ymm3/m256, imm8 (AVX2)

Miesza wektory podwójnych słów z rejestru ymm2 oraz ymm3 lub m256, w oparciu o specyfikację z bajtu kontrolnego imm8, wynik zapisuje w ymm1.

```
i <0, 7> - indeks podwójnego słowa
if imm8[i] = 1 then cel[i] = źródło2[i]
else cel[i] = źródło1[i]

if imm8[i] = 1 then ymm1[i] = ymm3/m256[i]
else ymm1[i] = ymm2[i]
```

Bity od 128/256 do MSB są zerowane.

Instrukcja mieszająca
VPBLENDD



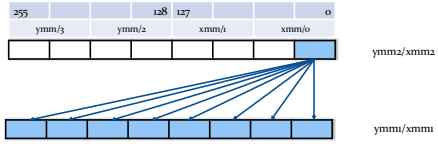
Instrukcje rozgłaszające

Instrukcja rozgłaszania

VPBROADCAST[B/W/D/Q]

vpbroadcast[b/w/d/q] xmm1, xmm2/m8/m16/m32/m64 (AVX2)
vpbroadcast[b/w/d/q] ymm1, ymm2/m8/m16/m32/m64 (AVX2)

Rozgłasza tę samą wartość 8/16/32/64 bitową ze źródła xmm2/ymm2 lub pamięci m8/m16/m32/m64 do wszystkich elementów wektora rejestru celu xmm1/ymm1. Jeśli operand źródłowy jest rejestrem wartość rozgłaszana w najmłodszym elemencie wektora. Bity od 128/256 do MSB są zerowane.



Instrukcje zbierania

Instrukcja zbierania

VPGATHER[D/Q][D/Q]

Drugie [D/Q] dotyczy typu danych wartości pobranych z pamięci.

Pierwsze [D/Q] dotyczy adresu, jednocześnie określa maksymalną ilość pobieranych elementów z pamięci.

vpgatherdd xmm1, vm32x, xmm3
vpgatherqd xmm1, vm64x, xmm3

vpgatherdd ymm1, vm32y, ymm3
vpgatherqd ymm1, vm64y, ymm3

vpgatherdq xmm1, vm32x, xmm3
vpgatherqq xmm1, vm64x, xmm3

Instrukcja kompletuje wektor xmm1/ymm1 używając **adresów w postaci** podwójnych/poczwórnych słów zdefiniowanych w vm32[x/y]/vm64[x/y] używając jako indeksów podwójnych/poczwórnych słów zapisanych w xmm2/ymm2 do wskazanej lokalizacji pamięci skąd pobierane są wartości podwójnego/poczwórnego słowa.

vpgatherqq xmm1, vm32y, xmm3
vpgatherqq ymm1, vm64y, ymm3

Pobierane z pamięci wartości są zapisywane do rejestru celu xmm1/ymm1 tylko wówczas gdy najstarsze bity odpowiednich elementów wektora maski xmm3/ymm3 są równe 1.

Bity od 128/256 do MSB są zerowane.

Instrukcja zbierania (szczegółowo) AVX2

VPGATHER[D/Q][D/Q]

adres_fizyczny[i] = adres_bazowy + index[i]*skalowanie + przesunięcie

adres_bazowy – adres danych, określa użyty rejestr GPR

index[i] – i-ty element rejestru xmm2/ymm2 (z xmm2/ymm2 używane są jedynie indeksy)

skalowanie – określa rozmiar danych (1, 2, 4, 8)

przesunięcie – wartość w bajtach

Instrukcja zbierania (szczegółowo) AVX2

VPGATHER[D/Q][D/Q]

Adresowanie cd.

W opisie instrukcji vm32x wskazuje wektor czterech 32-bitowych wartości adresów dla konkretnego xmm, vm32y wektor ośmiu 32- bitowych wartości indeksów dla konkretnego ymm.

Notacja vm64x i vm64y wskazuje analogicznie na maksymalnie dwa lub cztery adresy.

Instrukcja zbierania (szczegółowo) AVX2

VPGATHER[D/Q][D/Q]

Działanie instrukcji gather

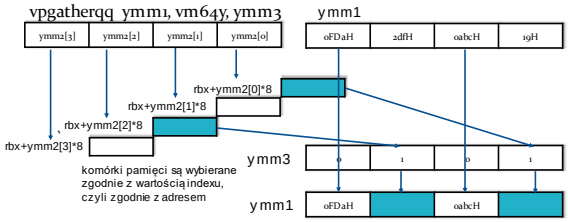
Pobiera z pamięci o wskazanej lokalizacji określonej tu jako adres_fizyczny wartości podwójnych/poczwórnych słów i zapisuje je do rejestru celu ymm1/xmm1 tylko wówczas gdy bit znaku odpowiadającego elementu maski ymm3/xmm3 jest równy jeden, jeśli bit znaku jest równy zero w rejestrze celu zostaje wartość poprzednia. Po wykonaniu operacji pobierania z pamięci elementy maski są zerowane.

if xmm3[i][63/31] then xmm1[i] ← [adres_fizyczny(xmm2[i])]

if ymm3[i][63/31] then ymm1[i] ← [adres_fizyczny(ymm2[i])]

Instrukcja zbierania (przykład) AVX2 VPGATHERQQ

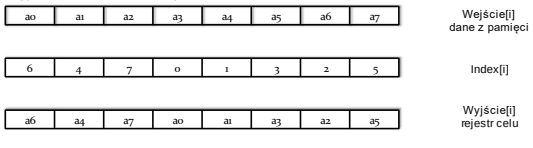
vpgatherqq ymm1, [rbx+ymm2*8], ymm3



Instrukcja zbierania (ogólnie) AVX2

VPGATHER[D/Q][D/Q]

Wyjście[i] = Wejście[Index[i]] Gather AVX2
Wyjście[Index[i]] = Wejście[i] Scather AVX-512



Model instrukcji Gather

Instrukcja zbierania (ogólnie) AVX2

VPGATHER[D/Q][D/Q]

Różnica pomiędzy instrukcjami
gather a *blend/perm/shuf*

Instrukcje *blend/perm/shuf* operują na rejestrach, zatem najpierw należy załadować dane z pamięci do rejestru ymm/xmm.
Instrukja *gather* pobiera dane bezpośrednio z pamięci, jednak wcześniej trzeba przygotować rejestr indeksów (rejestr porządku).

Operacje arytmetyczne

Operacje arytmetyczne AVX

- Dodawanie: VPADDB, VPADDW, VPADDD, VPADDQ
  VPADDSB, VPADDSW, VPADDUSB, VPADDUSW
  VPHADDW, VPHADDD, VPHADDSW
- Odejmowanie: VPSUBB, VPSUBW, VPSUBD, VPSUBQ
  VPSUBSB, VPSUBSW, VPSUBUSB, VPSUBUSW
  VPHSUBW, VPHSUBD, VPHSUBSW, VPSADBW
- Mnożenie: VPMULLW, VPMULLD,VPMULHUW
  VPMULHW, VPMULHRSW, VPMULDQ, VPMILUDQ
  VPCLMULQDQ
- Mnożenie i dodawanie: VPMADDWD, VPMADDUBSW

Instrukcje dodawania

Instrukcja dodawania
VPADD[B/W/D/Q]

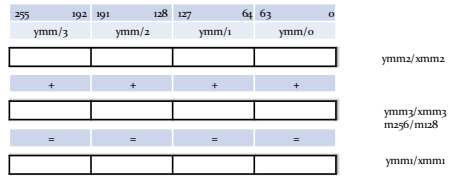
vpadd[b/w/d/q] xmm1, xmm2, xmm3/m128
vpadd[b/w/d/q] ymm1, ymm2, ymm3/m256 (AVX2)

Do wartości bajtów/słów/podwójnych słów/poczwórnych słów z rejestru **xmm2/ymm2** są dodawane równolegle odpowiednie wartości z rejestru **xmm3/ymm3** lub z pamięci **m128/m256**, wynik jest zapisywany w rejestrze **xmm1/ymm1.**

cel[i] = źródło1[i] + źródło2[i]
xmm1[i] = xmm2[i] + xmm3/m128[i]
ymm1[i] = ymm2[i] + ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPADDQ



Instrukcja dodawania
VPADDS[B/W]

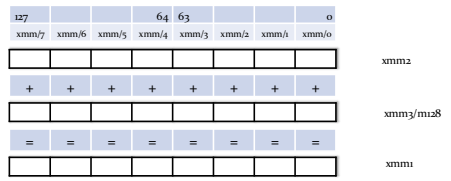
vpaddsb xmm1, xmm2, xmm3/m128
vpaddsb ymm1, ymm2, ymm3/m256 (AVX2)

Dodawanie ze znakiem wektorów bajtów/słów z rejestru xmm2/ymm2 oraz xmm3/ymm3 lub pamięci m128/m256, wynik jest zapisywany **z nasyceniem** w rejestrze xmm1/ymm1.

cel[i] = źródło1[i] + źródło2[i]
xmm1[i] = xmm2[i] + xmm3/m128[i]
ymm1[i] = ymm2[i] + ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPADDSW



Instrukcja dodawania
VPADDUS[B/W]

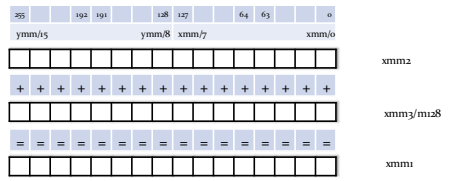
vpaddusb xmm1, xmm2, xmm3/m128
vpaddusb ymm1, ymm2, ymm3/m256 (AVX2)

Dodawanie bez znaku wektorów bajtów/słów rejestru xmm2/ymm2 i xmm3/ymm3 lub pamięci m128/m256, **wynik** jest zapisywany **z nasyceniem** w rejestrze xmm1/ymm1.

cel[i] = źródło1[i] + źródło2[i]
xmm1[i] = xmm2[i] + xmm3/m128[i]
ymm1[i] = ymm2[i] + ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPADDUSW

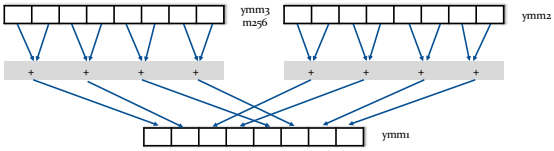


Instrukcja dodawania

VPHADDW / **VPHADDD** / VPHADDSW

vphaddd xmm1, xmm2,ymm3/m128
vphaddd ymm1, ymm2,ymm3/m256

Horyzontalne dodawanie sąsiednich słów/podwójnych słów i zapisywanie wyniku z przeplotem po 64 bity. Jako najmłodsze są zapisywane sumy z rejestru xmm2. Ostatnia w/w instrukcja jest dodawaniem słów z nasyceniem.



Instrukcja dodawania - przykład:

```
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; adres tablicy t1
        mov edx, t2; adres tablicy t2
        mov edi, t3; adres tablicy t3
```

```
cd.
        petla:

        sub ecx, 32; podwójne słowo
        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;
}
}
```


Instrukcje odejmowania

Instrukcja odejmowania

VPSUB[B/W/D/Q]

vpsub[b/w/d/q] xmm1, xmm2, xmm3/m128
vpsub[b/w/d/q] ymm1, ymm2, ymm3/m256 (AVX2)

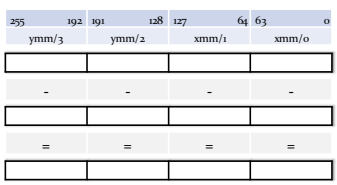
Od wartości bajtu/słowa/podwójnego słowa/poczwórnego słowa z rejestru **xmm2/ymm2** są odejmowane równolegle odpowiednie wartości z rejestru **xmm3/ymm3** lub z pamięci **m128/m256**, wynik jest zapisywany w rejestrze **xmm1/ymm1.**

cel[i] = źródło1[i] - źródło2[i]
xmm1[i] = xmm2[i] - xmm3/m128[i]
ymm1[i] = ymm2[i] - ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowania

VPSUBQ



Instrukcja odejmowania

VPSUB[U]S[B/W]

vpsub[u]s[b/w] xmm1, xmm2, xmm3/m128
vpsub[u]s[b/w] ymm1, ymm2, ymm3/m256 (AVX2)

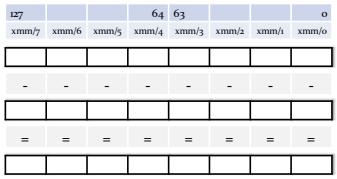
Od wartości ze znakiem/ bez znaku (U) wektorów bajtów/słów rejestru **xmm2/ymm2** są **odejmowane** odpowiednie wartości rejestru **xmm3/ymm3** lub pamięci **m128/m256**, wynik jest zapisywany w rejestrze **xmm1/ymm1** z **nasyceniem.**

cel[i] = źródło1[i] - źródło2[i]
xmm1[i] = xmm2[i] - xmm3/m128[i]
ymm1[i] = ymm2[i] - ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowanie

VPSUBSW



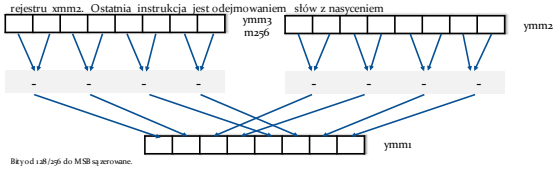
Instrukcja odejmowania

VPHSUBW / VPHSUBD / VPHSUBSW

vphsubd xmm1, xmm2, xmm3/m128

vphsubd ymm1, ymm2, ymm3/m256

Horyzontalne odejmowanie sąsiednich słów/podwójnych słów w ramach 128-bitowych części. Od młodszego elementu wektora jest odejmowany starszy, oraz jako najmłodsze są zapisywane różnice z rejestru xmm2. Ostatnia instrukcja jest odejmowaniem słów z nasyceniem



Bity od 128/256 do MSB są zerowane.

Instrukcja sumowanie modułów różnic

VPSADBW

vpsadbw xmm1, xmm2, xmm3/m128

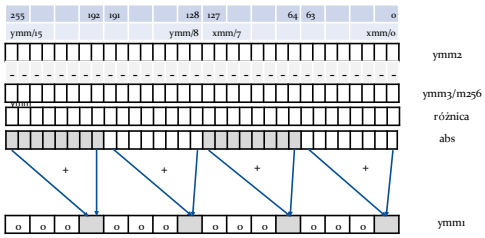
vpsadbw ymm1, ymm2, ymm3/m256 (AVX2)

Od wartości wektorów bajtów rejestru **xmm2/ymm2** są **odejmowane** odpowiednie wartości rejestru **xmm3/ymm3** lub pamięci **m128/m256**, następnie obliczane są wartości absolutne i ich sumy po 8 elementów, wynik jest zapisywany w rejestrze **xmm1/ymm1** dla zestawów 8-elementowych

Bity od 128/256 do MSB są zerowane.

Instrukcja sumowanie modułów różnic

VPSADBW



Instrukcje mnożenia

Instrukcja mnożenia

VPMULL[W/D]

vpmull[w/d] xmm1, xmm2, xmm3/m128

vpmull[w/d] ymm1, ymm2, ymm3/m256 (AVX2)

Mnożenie wektorów słów/podwójnych słów ze znakiem z rejestru xmm2/ymm2 przez odpowiadające im wartości z xmm3/m128 / ymm3/m256, **iloczyny są podwójnymi/poczwórnymi słowami**, jednak do rejestru celu xmm1/ymm1 są zapisywane **jedynie młodsze słowa/podwójne słowa iloczynów.**

cel[i] = lo (źródło1[i] * źródło2[i])

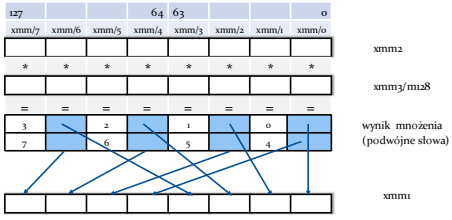
xmm1[i] = lo (xmm2[i] * xmm3/m128[i])

ymm1[i] = lo (ymm2[i] * ymm3/m256[i])

Bity od 128/256 do MSB są zerowane.

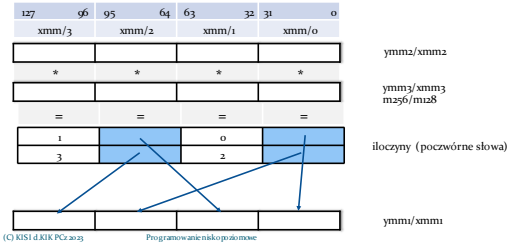
Instrukcja mnożenia

VPMULLW



Instrukcja mnożenia

VPMULLD



Instrukcja mnożenia

VPMUH[U]W

vpmulh[u]w xmm1, xmm2, xmm3/m128

vpmulh[u]w ymm1, ymm2, ymm3/m256 (AVX2)

Mnożenie wektorów słów bez znaku/ze znakiem (U) z rejestru xmm2/ymm2 przez odpowiadające im wartości z xmm3/m128 / ymm3/m256, **iloczyny są podwójnymi słowami**, jednak do rejestru celu xmm1/ymm1 są zapisywane **jedynie starsze słowa,** iloczynów.

cel[i] = hi (źródło1[i] x źródło2[i])

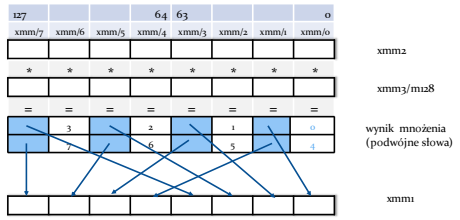
xmm1[i] = hi (xmm2[i] x xmm3/m128[i])

ymm1[i] = hi (ymm2[i] x ymm3/m256[i])

Bity od 128/256 do MSB są zerowane.

Instrukcja mnożenia

VPMULHW



Instrukcja mnożenia

VPMULHRSW

vpmulhrsw xmm1, xmm2, xmm3/m128

vpmulhrsw ymm1, ymm2, ymm3/m256 (AVX2)

Mnoży wektory słów ze znakiem ze skalowaniem i zaokrągleniem, wartości z rejestru xmm2/ymm2 przez wartości z rejestru xmm3/m128 / ymm3/m256, podwójne słowa iloczynów zostają przesunięte w prawo o 14 bitów oraz zostaje dodana jedynka **w celu zaokrąglenia wartości**. Bity od 1 do 16 są zapisywane w celu.

cel[i] = ((źródło1[i] * źródło2[i] >> 14) + 1) >> 1

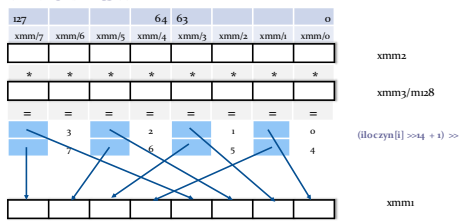
xmm1[i] = ((xmm2[i] * xmm3/m128[i] >> 14) + 1) >> 1

ymm1[i] = ((ymm2[i] * ymm3/m256[i] >> 14) + 1) >> 1

Bity od 128/256 do MSB są zerowane.

Instrukcja mnożenia

VPMULHRSW



Instrukcja mnożenia

VPMUL[U]DQ

vpmul[u]dq xmm1, xmm2, xmm3/m128

vpmul[u]dq ymm1, ymm2, ymm3/m256 (AVX2)

Mnożenie **co drugich** elementów wektora podwójnych słów ze znakiem/bez znaku (U) xmm2/ymm2 **z co drugimi** elementami podwójnych słów ze znakiem xmm3/m128 / ymm3/m256, iloczyny są zapisywane w xmm1/ymm1 **jako wektor poczwórnych słów ze znakiem.**

cel[i] = źródło1[2i] * źródło2[2i]

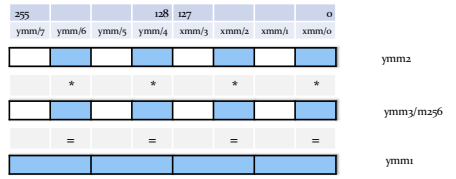
xmm1[i] = xmm2[2i] * xmm3/m128[2i]

ymm1[i] = ymm2[2i] * ymm3/m256[2i]

Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowanie

VPMULDQ



Instrukcja mnożenia

VPCLMULQDQ

vpclmulqdq xmm1, xmm2, xmm3/m128, imm8 (AVX)

Mnożenie poczwórnego słowa z xmm2 przez poczwórne słowo z xmm3/m128, iloczyn jest zapisywany w xmm1. Bity imm8[0] i imm8[4] wybierają młodsze lub starsze (0 lub 1) poczwórne słowa z rejestrów xmm2 i xmm3/m128, które zostaną pomnożone.

if imm8[0] = 0||1 && imm8[4] = 0||1 => cel <- źródło1[0||1] * źródło2[0||1]

if imm8[0] = 0 && imm8[4] = 0 => xmm1 <- xmm2[63:0] * xmm3/m1228[63:0]

if imm8[0] = 0 && imm8[4] = 1 => xmm1 <- xmm2[63:0] * xmm3/m1228[127:64]

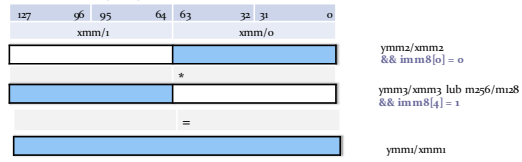
if imm8[0] = 1 && imm8[4] = 0 => xmm1 <- xmm2[127:64] * xmm3/m1228[63:0]

if imm8[0] = 1 && imm8[4] = 1 => xmm1 <- xmm2[127:64] * xmm3/m1228[127:64]

Bity od 128/256 do MSB są zerowane.

Instrukcja mnożenia

VPCLMULQDQ



Instrukcje mnożenia z dodawaniem

Dla liczb zmienno-przecinkowych odpowiednikiem bardziej zaawansowanym są instrukcje FMA

Instrukcja mnożenia i dodawania

VPMADDWD

vpmaddwd xmm1, xmm2, xmm3/m128

vpmaddwd ymm1, ymm2, ymm3/m256 (AVX2)

Mnoży słowa z rejestru xmm2/ymm2 przez słowa z rejestru xmm3/m128 / ymm3/m256, iloczyny są podwójnymi słowami, następnie kolejne podwójne słowa dodaje horyzontalnie i zapisuje jako podwójne słowa w rejestrze celu xmm1/ymm1.

cel[i] = źródło1[2i] * źródło2[2i] + źródło1[2i+1] * źródło2[2i+1]

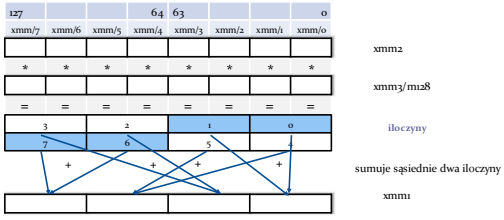
xmm1[i] = xmm2[2i] * xmm3/m128[2i] + xmm2[2i+1] * xmm3/m128[2i+1]

ymm1[i] = ymm2[2i] * ymm3/m128[2i] + ymm2[2i+1] * ymm3/m128[2i+1]

Bity od 128/256 do MSB są zerowane.

Instrukcja mnożenia

VPMADDWD



Instrukcja mnożenia i dodawania

VPMADDUBSW

vpmaddubsw xmm1, xmm2, xmm3/m128

vpmaddubsw ymm1, ymm2, ymm3/m256 (AVX2)

Mnoży **bajty bez znaku z** rejestru xmm2/ymm2 przez bajty z rejestru xmm3/m128 / ymm3/m256, iloczyny są słowami, następnie dwa kolejne słowa dodaje horyzontalnie i zapisuje z nasyceniem, jako słowa w rejestrze celu xmm1/ymm1.

cel[i] = źródło1[2i] * źródło2[2i] + źródło1[2i+1] * źródło2[2i+1]

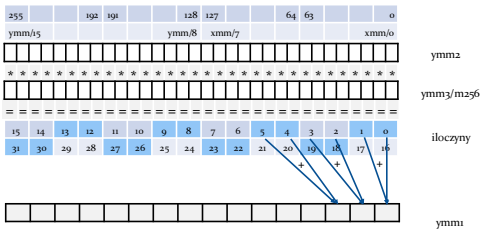
xmm1[i] = xmm2[2i] * xmm3/m128[2i] + xmm2[2i+1] * xmm3/m128[2i+1]

ymm1[i] = ymm2[2i] * ymm3/m128[2i] + ymm2[2i+1] * ymm3/m128[2i+1]

Bity od 128/256 do MSB są zerowane.

Instrukcja sumowanie modułów

VPMADDUBSW



Operacje arytmetyczne AVX cd.

- **Wartość maksymalna:** VPMAXUB, VPMAXUW VPMAXUD, VPMAXSB, VPMAXSW, VPMAXSD
- **Wartość minimalna:** VPMINUB, VPMINUW, VPMINUD VPMINSB, VPMINSW, VPMINSD, VPHMINPOSUW
- **Średnia:** VPAVGB, VPAVGW
- **Wartość bezwzględna liczby:** VPABSB, VPABSW, VPABSD
- **Negacja / zero / zachowanie:** VPSIGNB, VPSIGNW, VPSIGND

Instrukcje wartość maksymalna

Instrukcja wartość maksymalna

VPMAX[U/S][B/W/D]

vpmax[u/s][b/w/d] xmm1, xmm2, xmm3/m128

vpmax[u/s][b/w/d] ymm1, ymm2, ymm3/m256 (AVX2)

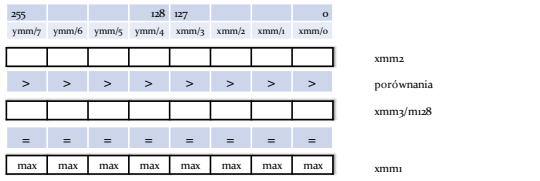
Porównuje **bez znaku/ze znakiem** wartości w wektorach bajtów/słów/podwójnych słów rejestru **xmm2/ymm2** z odpowiednimi wartościami **xmm3/ymm3** lub pamięci **m128/m256**, **wektory wartości maksymalnych** są zapisywane w rejestrze **xmm1/ymm1.**

if źródło1[i] > źródło2[i] **then** cel[i] = źródło1[i] **else** cel[i] = źródło2[i]

if xmm2/ymm2[i] > xmm3||m128[i] / ymm3||m256[i]
then xmm1/ymm1[i] = xmm2/ymm2[i]
else xmm1/ymm1[i] = xmm3||m128[i] / ymm3||m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPMAX[U/S][B/W/D]



Instrukcja wartość maksymalna
VPMIN[U/S][B/W/D]

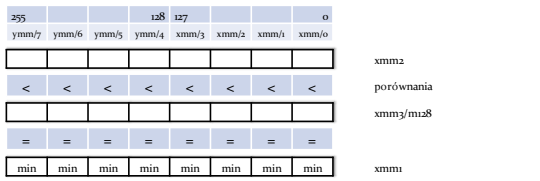
vpmax[u/s][b/w/d] xmm1, xmm2, xmm3/m128
vpmax[u/s][b/w/d] ymm1, ymm2, ymm3/m256 (AVX2)
Porównuje **bez znaku/ze znakiem** wartości w wektorach bajtów/słów/podwójnych słów rejestru **xmm2/ymm2** z odpowiednimi wartościami **xmm3/ymm3** lub pamięci **m128/m256**, **wektory wartości minimalnych** są zapisywane w rejestrze **xmm1/ymm1**.

if źródło1[i] < źródło2[i] **then** cel[i] = źródło1[i] **else** cel[i] = źródło2[i]

if xmm2/ymm2[i] < xmm3||m128[i] / ymm3||m256[i]
then xmm1/ymm1[i] = xmm2/ymm2[i]
else xmm1/ymm1[i] = xmm3||m128[i] / ymm3||m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPMIN[U/S][B/W/D/Q]



Instrukcje wartość średnia

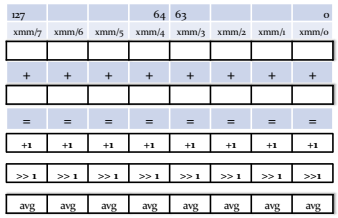
Instrukcja wartość średnia
VPAVG[B/W]

vpavg[b/w] xmm1, xmm2, xmm3/m128
vpavg[b/w] ymm1, ymm2, ymm3/m256 (AVX2)
Zwraca **średnią dwóch wartości bez znaku** z wektorów bajtów/słów, dodaje wektory rejestru **xmm2/ymm2** z odpowiednimi wartościami **xmm3/ymm3** lub pamięci **m128/m256**, **sumę zaokrągla** jedynką oraz **dzieli przez dwa** poprzez przesuniecie bitowe o jeden w prawo, **wynik** zapisuje w rejestrze **xmm1/ymm1**.

cel[i] = (źródło1[i] + źródło2[i] + 1) >> 1
xmm1[i] = (xmm2[i] + xmm3/m128[i] + 1) >> 1
ymm1[i] = (ymm2[i] + ymm3/m256[i] + 1) >> 1

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPAVG[B/W]



Instrukcje wartości bezwzględnej

Instrukcja wartość bezwzględna

VPABS[B/W/D]

vpabsb xmm1, xmm2

vpabsb ymm1, ymm2 (AVX2)

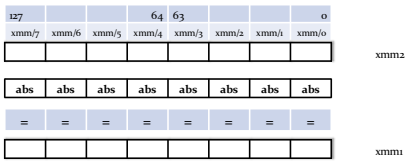
Oblicza **wartość bezwzględną** od wartości bajtów/słów/podwójnych słów rejestru xmm2/ymm2, wynik zapisuje w rejestrze xmm1/ymm1 **baz znaku.**

cel[i] = abs(źródło1[i])
xmm1[i] = abs(xmm2[i])
ymm1[i] = abs(ymm2[i])

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania

VPABS[B/W/D]



Instrukcje znaku

pozostawienie / zerowanie / negacja

Instrukcja znaku

VPSIGN[B/W/D]

vpsign[b/w/d] xmm1, xmm2, xmm3/m128
vpsign[b/w/d] ymm1, ymm2, ymm3/m256 (AVX2)

Zapisuje bajty/słowa/podwójne słowa do xmm1/ymm1 wartościami z rejestru xmm2 w zależności od znaku odpowiadającej wartości wektora w rejestrze xmm3/m128.

if źródło2[i] > 0; cel[i] = źródło1[i]
if źródło2[i] = 0; cel[i] = 0
if źródło2[i] < 0; cel[i] = - źródło1[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja znaku

VPSIGN[B/W/D]

ymm

if ymm3/m256[i] > 0 then ymm1[i] = ymm2[i]

if ymm3/m256[i] = 0 then ymm1[i] = 0

if ymm3/m256[i] < 0 then ymm1[i] = - ymm2[i]

xmm

if xmm3/m128[i] > 0 then xmm1[i] = xmm2[i]

if xmm3/m128[i] = 0 then xmm1[i] = 0

if xmm3/m128[i] < 0 then xmm1[i] = - xmm2[i]

Instrukcja dodawania

VPSIGN[B/W/D]

127			64	63			0
xmm/7	xmm/6	xmm/5	xmm/4	xmm/3	xmm/2	xmm/1	xmm/0
8	7	6	5	4	3	2	1

xmm2

-4	5	0	-9	2	6	0	-3

xmm3/m128
odczytanie bitu znaku

-8	7	0	-5	4	3	0	-1

xmm1

Operacje porównania AVX

- **Porównanie liczb:** VPCMPEQB, VPCMPEQW VPCMPEQD, VPCMPGTB, VPCMPGTW VPCMPGTD, VPCMPGTQ
- **Porównanie ciągów:** VPCMPESTRI, VPCMPISTRI, VPCMPESTRM VPCMPISTRM

Instrukcje porównania

Instrukcja porównania liczb

VPCMPEQ[B/W/D/Q]

vpcmpeq[b/w/d/q] xmm1, xmm2, xmm3/m128
vpcm0eq[b/w/d/q] ymm1, ymm2, ymm3/m256 (AVX2)

Porównuje wartości współrzędnych wektorów bajtów/słów/podwójnych słów/poczwórnych słów z rejestru xmm3/ymm3 lub pamięci m128/m256 z odpowiednimi wartościami współrzędnych rejestru xmm2/ymm2, jeśli wartości są równe, odpowiednie współrzędne rejestru celu xmm1/ymm1 są ustawiane na -1, jeśli nie na 0.

if źródło1[i] = źródło2[i] then cel[i] = -1 else cel[i] = 0;
if xmm3/m128[i] = xmm2[i] then xmm1[i] = -1 else xmm1[i] = 0;
if ymm3/m128[i] = ymm2[i] then ymm1[i] = -1 else ymm1[i] = 0;

Bity od 128/256 do MSB są zerowane.

Instrukcja porównania liczb

VPCMPEQ[B/W/D/Q]

127			64	63			0
xmm/7	xmm/6	xmm/5	xmm/4	xmm/3	xmm/2	xmm/1	xmm/0
2	4	9	7	6	5	3	2
EQ	EQ	EQ	EQ	EQ	EQ	EQ	EQ
5	0	1	7	6	4	3	2

xmm3/m128

xmm2

=	=	=	=	=	=	=	=
0	0	0	-1	-1	0	-1	-1

xmm1

Instrukcja porównania liczb

VPCMPGT[B/W/D/Q]

vpcmpgt[b/w/d] xmm1, xmm2, xmm3/m128
vpcmpgt[b/w/d] ymm1, ymm2, ymm3/m128 (AVX2)

Porównuje wartości współrzędnych wektorów bajtów/słów/podwójnych słów/poczwórnych słów z rejestru xmm3/ymm3 lub pamięci m128/m256 z odpowiednimi wartościami współrzędnych rejestru xmm2/ymm2, jeśli wartości xmm2/ymm2 są **większe niż** wartości xmm3/ymm3 lub m128/m256 wówczas odpowiednie współrzędne rejestru celu xmm1/ymm1 są ustawiane są na -1, w przeciwnym wypadku na 0.

if źródło1[i] > źródło2[i] then cel[i] = -1 else cel[i] = 0
if xmm2[i] > xmm3/m128[i] => xmm1[i] = -1 else xmm1[i] = 0
if ymm2[i] > ymm3/m256[i] then ymm1[i] = -1 else ymm1[i] = 0

Bity od 128/256 do MSB są zerowane.

Instrukcja porównania liczb
VPCMPGT[B/W/D/Q]

127				64	63				0
xmm7	xmm6	xmm5	xmm4	xmm3	xmm2	xmm1	xmm0		
7	2	3	4	6	8	9	-1	xmm3/m128	
GT	GT	GT	GT	GT	GT	GT	GT		
0	3	4	6	5	7	0	1	xmm2	
=	=	=	=	=	=	=	=		
0	-1	-1	-1	0	0	0	-1	xmm1	

Instrukcje porównania ciągów znakowych

- Operacje porównania ciągów znakowych porównują w istocie liczby całkowite.
- Instrukcje te można podzielić na porównujące ciągi znakowe o, ustalonej (znanej) w rejestrach [R/E]AX i [R/E]DX oraz nieznanej, długości.
- Instrukcje CMP na wyjściu tworzą indeks lub maskę, ale wynik porównania jest zapisywany w [R/E]CX/xmm0 (brak w wywołaniu instrukcji).
- W instrukcjach tego typu istotne zadanie pełni bajt sterujący imm8, gdzie można zdefiniować to złożone i wieloetapowe porównanie i pełni on w istocie funkcję algorytmu instrukcji
- Instrukcje porównania jako nieliczne w AVX ustawiają flagi

Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]

Bajt sterujący **imm8** (1/4)

7	6	5	4	3	2	1	0

Typ danych na wejściu
imm8[1:0] = {**00b** bajty bez znaku, **01b** słowa bez znaku,
10b bajty ze znakiem, **11b** słowa ze znakiem}

Operacja (sposób porównania)
imm8[3:2] = {**00b** porównanie arytmetyczne, czy w ciągu występują podane bajty/słowa,
01b porównanie arytmetyczne większe lub równe dla parzystych elementów wektora lub mniejsze lub równe dla nieparzystych elementów wektora
10b porównuje arytmetycznie, czy odpowiadające sobie wartości są równe
11b porównuje arytmetycznie, czy równy (w kolejności) }

Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]

Bajt sterujący **imm8** (2/4)

7	6	5	4	3	2	1	0

Polaryzacja (nadawanie znaku)
imm8[5:4] = {**00b** pozytywna polaryzacja (bez zmiany znaku),
01b negatywna polaryzacja (ze zmianą znaku),
10b stosowanie maski (bez zmiany znaku),
11b stosowanie maski (ze zmianą maski) dla elementów nieważnych w reg/mem1 są przepisywane, w przeciwnym wypadku nieważne są negowane }

Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]

Bajt sterujący **imm8** (3/4)

7	6	5	4	3	2	1	0

Wynik zwracany w postaci **indeksu** lub **maski** jest tworzony etapami:
1. logiczne porównanie każdy z każdym (bajt z bajtem / słowo ze słowem)
2. intermediate result 1 (pośrednie zagregowane wyniki porównania - **prawdziwe**)
3. intermediate result 2 etap poprzedni jest negowany logicznie
4. zwracany jest albo najbardziej/najmniej znaczący bit porównania pkt. 3 (index) albo całe porównanie z pkt. 3 w opcji rozszerzenia zerami lub rozszerzenia do bajtu/słowa (maska)

Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]

Bajt sterujący **imm8** (4/4)

7	6	5	4	3	2	1	0

Wybór Wyjścia
Dla indeksu (I) = wynik do ECX/RCX
imm8[6] = {ob z wyniku jest pobierany **najmłodszy bit**
ib z wyniku jest pobierany **najstarszy bit** }

Dla maski (M) = wynik do xmm0
imm8[6] = {ob zwracany wynik uzupełniany jest zerami,
ib wynik jest rozszerzany do bajtu/słowa (z samymi zerami lub jedynkami) }

Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]

Instrukcje porównania ciągów znakowych operują na rejestrach **xmm** oraz w sposób określony przez bajt sterujący **imm8**, który jest częścią kodującą instrukcji.

Instrukcje ustawiają flagi arytmetyczne ZF, CF, SF, OF, AF, PF (wyjątek w AVX), jednak znaczenia flag zostały przeciążone z ich zwykłego znaczenia celem dostarczenia dodatkowych informacji o relacji pomiędzy dwoma wejściami.

Instrukcje typu PCMPxSTRx wykonują porównania arytmetyczne między wszystkimi możliwymi parami bajtów lub słów, po jednym z każdego wektora źródłowego. Wartości logiczne tych porównań są następnie agregowane w celu uzyskania wyniku końcowego.

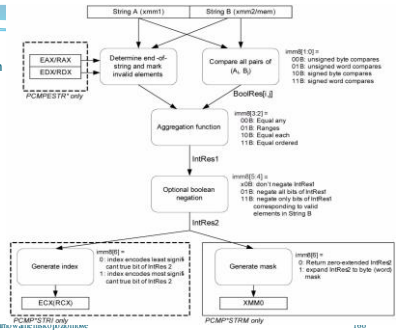
Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]

Algorytm instrukcji definiowany w bajcie sterującym:
- Ustawienie źródła
- Operacje porównania i agregacji (wyniki pośrednie)
- Polaryzacja
- Wybór wyjścia dla wyniku końcowego

Bajt kontrolny określa spodziewany wynik i kontroluje następujące atrybuty:
- format danych bajt/słowo, ze znakiem/bez znaku imm8[]
- koduje tryb operacji porównania
- określa przetwarzanie pośrednie
- określa operację tworzenia wyjścia zależnie, czy index, czy maska.

Zatem instrukcje porównują ciągi znakowe bajtów lub słów, **wynikiem jest:**
- **maska w xmm0 lub index w [R/E]CX**
- **ustawione flagi**.

Instrukcja porównania ciągów znakowych
VPCMP[E/I]STR[I/M]



Instrukcja porównania ciągów znakowych
VPCMPESTRI

vpcmpestri xmm1, xmm2/m128, imm8 (AVX)

Porównuje łańcuchy o ustalonej długości z rejestru xmm1 i xmm2/m128, na wyjściu tworzy index, wynik zapisuje w rejestrze ogólnego przeznaczenia ECX
E (Explicit) – oznacza łańcuchy o ustalonej długości
I (Index) – oznacza, że instrukcja tworzy na wyjściu index

Operand 1	Operand 2	Długość 1	Długość 2	Wynik
xmm1	xmm2/m128	[R/E]AX	[R/E]DX	ECX

CFlag - Reset if IntRes2 is equal to zero, set otherwise
ZFlag - Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag - Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag - IntRes2[0]
AFlag - Reset
PFlag - Reset

Instrukcja porównania ciągów znakowych
VPCMPISTRI

vpcmpistri xmm1, xmm2/m128, imm8 (AVX)

I (Implicit) - oznacza łańcuch o nieoznaczonej długości.
I (Index) – ożnacza, że instrukcja tworzy na wyjściu index
Porównuje **łańcuchy o nie ustalonej długości** z rejestru xmm1 i xmm2/m128, na wyjściu tworzy index, wynik zapisuje w rejestrze ogólnego przeznaczenia ECX.

Operand 1	Operand 2	Wynik
xmm1	xmm2/m128	ECX

CFlag - Reset if IntRes2 is equal to zero, set otherwise
ZFlag - Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag - Set if any byte/word of xmm1 is null, reset otherwise
OFlag -IntRes2[0]
AFlag - Reset
PFlag - Reset

Instrukcja porównania ciągów znakowych
VPCMPESTRM

vpcmpestrm xmm1, xmm2/m128, imm8 (AVX)

E (Explicit) – oznacza łańcuchy o ustalonej długości
M (Mask) – oznacza, że instrukcja tworzy na wyjściu maskę
Porównuje łańcuchy o ustalonej długości z rejestru xmm1 i xmm2/m128, na wyjściu tworzy maskę, wynik zapisuje w rejestrze **xmm0**. (nie jest umieszczany w definicji).

Operand 1	Operand 2	Długość 1	Długość 2	Wynik
xmm1	xmm2/m128	[R/E]AX	[R/E]DX	xmm0

CFlag - Reset if IntRes2 is equal to zero, set otherwise
ZFlag - Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag - Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag - IntRes2[0]
AFlag - Reset
PFlag - Reset

Instrukcja porównania ciągów znakowych
VPCMPISTRM

vpcmpistrm xmm1, xmm2/m128, imm8 (AVX)

I (Implicit) - oznacza łańcuch o nieoznaczonej długości.
M (Mask) – oznacza, że instrukcja tworzy na wyjściu maskę

Porównuje **łańcuchy o nie ustalonej długości** z rejestru xmm1 i xmm2/m128, na wyjściu tworzy maskę, wynik zapisuje w rejestrze **xmm0** (nie jest umieszczany w definicji).

Operand 1	Operand 2	Wynik
xmm1	xmm2/m128	xmm0

CFlag - Reset if IntRes2 is equal to zero, set otherwise
ZFlag - Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag - Set if any byte/word of xmm1 is null, reset otherwise
OFlag -IntRes2[0]
AFlag - Reset
PFlag - Reset

Operacje przesunięć AVX liczby całkowite

- **Przesunięcie w lewo:**

VPSLL[W/D/Q]
VPSLLDQ
VPSLLV[W/D/Q]

- **Przesunięcie w prawo:**

VPSRL[W/D/Q]
VPSRLDQ
VPSRLV[D/Q]
VPSRA[W/D/Q]
VPSRAV[W/D/Q]

Instrukcje przesunięć bitowych

Instrukcja przesunięcia w prawo
VPSRL[W/D/Q]

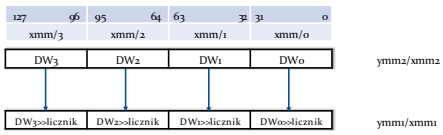
vpsrl[w/d/q] xmm1, xmm2, xmm3/m128 lub imm8
vpsrl[w/d/q] ymm1, ymm2, ymm3/m256 lub imm8 (AVX2)

Przesuwalogicznie **słowa/podwójne słowa /poczwórne słowa w prawo logicznie (całe)** z rejestru xmm2/ymm2 o wartość wskazaną przez rejestr xmm3/ymm3 lub m128/m256. Podczas przesunięcia starsze bity są zerowane. Jeśli wartość licznika jest większa niż 15 dla słów, 31 dla podwójnych słów, 63 dla poczwórnych słów, wówczas wszystkie bity są zerowane.

cel[i] = źródło1[i] >> źródło2[0] lub imm8
xmm1[i] = xmm2[i] >> xmm3/m128[0] lub imm8
ymm1[i] = ymm2[i] >> ymm3/m128[0] lub imm8

Bity od 128/256 do MSB są zerowane.

Instrukcja przesunięcia w prawo
VPSRLD



Licznik może być zapisany w rejestrze **xmm3/m128** (ymm3/m256) albo w bajcie sterującym **imm8**

Instrukcja przesunięcia logicznego w prawo
VPSRLDQ

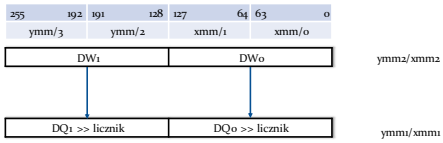
vpsrldq xmm1, xmm2, imm8
vpsrldq ymm1, ymm2, imm8 (AVX2)

Przesuwa **logicznie w prawo** podwójne poczwórne słowo z rejestru xmm2/ymm2 **o liczbę bajtów** określoną przez imm8. Podczas przesunięcia starsze bity są zerowane.

cel[i] = źródło1[i] >> imm8
xmm1[i] = xmm2[i] >> imm8
ymm1[i] = ymm2[i] >> imm8

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania
VPSRDQ



Licznik jest określony w **imm8**.

Instrukcja przesunięcia w prawo
VPSRLV[D/Q]

vpsrlv[d/q] xmm1, xmm2, xmm3/m128 (AVX2)
vpsrlv[d/q] ymm1, ymm2, ymm3/m256 (AVX2)

Przesuwa **logicznie w prawo** bity podwójne słowa/poczwórne słowa z rejestru xmm2/ymm2 **o liczbę bitów** wskazaną przez odpowiednie elementy rejestru xmm3/ymm3 lub m128/m256 . Podczas przesunięcia starsze bity są zerowane. Jeśli wartość licznika jest większa niż 31 dla podwójnych słów, 63 dla poczwórnych słów, wówczas wszystkie bity są zerowane.

cel[i] = źródło1[i] >> źródło2[i]
xmm1[i] = xmm2[i] >> xmm3/m128[i]
ymm1[i] = ymm2[i] >> ymm3/m128[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja przesunięcia arytmetycznego w prawo
VPSRA[W/D]

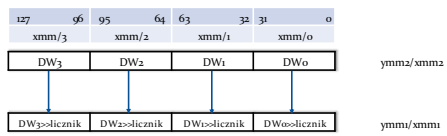
vpsra[w/d] xmm1, xmm2, xmm3/m128 lub imm8
vpsra[w/d] ymm1, ymm2, ymm3/m256 lub imm8 (AVX2)

Przesuwa **arytmetycznie w prawo z powieleniem bitu znaku** słowa/podwójne słowa z rejestru xmm2/ymm2 **(całe)** określoną przez wartość zapisaną w rejestrze xmm3/ymm3 lub m128/m256 lub przez bajt sterujący imm8. Starsze bity są ustawiane na bit znaku. Jeśli wartość licznika jest większa niż 15 dla słów, 31 dla podwójnych słów, 63 dla poczwórnych słów, wówczas wszystkie bity są ustawiane na bit znaku.

cel[i] = źródło1[i] >> źródło2[0] lub źródło2
xmm1[i] = xmm2[i] >> xmm3/m128[0] lub imm8
ymm1[i] = ymm2[i] >> ymm3/m128[0] lub imm8

Bity od 128/256 do MSB są zerowane.

Instrukcja przesunięcia arytmetycznego w prawo
VPSRAD



Licznik może być zapisany w rejestrze **xmm3/m128** (ymm3/m256) albo w bajcie sterującym **imm8**

Instrukcja przesunięcia arytmetycznego w prawo
VPSRAVD

vpsravd xmm1, xmm2, xmm3/m128 (AVX2)
vpsravd ymm1, ymm2, ymm3/m256 (AVX2)

Przesuwa **arytmetycznie w prawo z powieleniem bitu znaku** słowa/podwójne słowa z rejestru xmm2/ymm2 **o liczbę bitów** określoną przez wartości zapisane w rejestrze xmm3/ymm3 lub m128/m256. Starsze bity są ustawiane na bit znaku. Jeśli wartość licznika jest większa niż 15 dla słów, 31 dla podwójnych słów, wówczas wszystkie bity są ustawiane na bit znaku.

cel[i] = źródło1[i] >> źródło2[i]
xmm1[i] = xmm2[i] >> xmm3/m128[i]
ymm1[i] = ymm2[i] >> ymm3/m156[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja przesunięcia logicznego w lewo
VPSLL[W/D/Q]

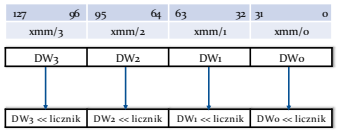
vpsll[w/d/q] xmm1, xmm2, xmm3/m128 lub imm8
vpsll[w/d/q] ymm1, ymm2, ymm3/m256 lub imm8 (AVX2)

Przesuwa **logicznie w lewo** słowo/podwójne słowo/poczwórne słowo (w całości) z rejestru xmm2/ymm2 **o wartość** określoną przez rejestr xmm3/ymm3 lub m128/m256 lub przez bajt sterujący imm8. Młodsze bity są zerowane. Jeśli wartość licznika jest większa niż 15 dla słów, większa niż 31 dla podwójnych słów, większa niż 63 dla poczwórnych słów, wówczas bity danego elementu są zerowane.

cel[i] = źródło1[i] << źródło2[0] lub imm8
xmm1[i] = xmm2[i] << xmm3/m128[0] lub imm8
ymm1[i] = ymm2[i] << ymm3/m128[0] lub imm8

Bity od 128/256 do MSB są zerowane.

Instrukcja przesunięcia logicznego w lewo
VPSLLD



Licznik może być zapisany w rejestrze **xmm3/m128** (ymm3/m256) albo w bajcie sterującym **imm8**

Instrukcja przesunięcia logicznego w lewo
VPSLLV[D/Q]

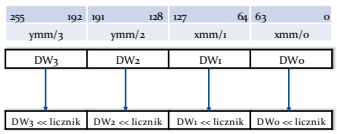
vpsllv[d/q] xmm1, xmm2, xmm3/m128
vpsllv[d/q] ymm1, ymm2, ymm3/m256

Przesuwa **logicznie w lewo** podwójne słowo/poczwórne słowo z rejestru xmm2/ymm2 **o liczbę bitów** określoną przez rejestr xmm3/ymm3 lub m128/m256. Młodsze bity są zerowane. Jeśli wartość licznika jest większa niż 31 dla podwójnych słów, większa niż 63 dla poczwórnych słów, wówczas wszystkie bity danego elementu są zerowane.

cel[i] = źródło1[i] << źródło2[i]
xmm1[i] = xmm2[i] << xmm3/m128[i]
ymm1[i] = ymm2[i] << ymm3/m128[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja przesunięcia arytmetycznego w lewo
VPSLLVQ



Liczniki są określone w **xmm3/ymm3 lub m128/m256**

Operacje logiczne / inne AVX liczby całkowite

- **Instrukcje logiczne:**
  VPAND, VPANDN, VPOR, VPXOR
- **Instrukcje zerowania:**
  VZEROALL, VZEROUPPER
- **Instrukcje dodatkowe:**
  VLDMXCSR / VSTMXCSR
- **Instrukcja wyrównywania:**
  VPALIGNR

Instrukcje logiczne

Instrukcje logiczne koniunkcja
VPAND / VPANDN

vpand xmm1, xmm2, xmm3/m128
vpand ymm1, ymm2, ymm3/m256 (AVX2)

Oblicza **iloczyn logiczny bit po bicie** dla wszystlich bitów rejestrów xmm2/ymm2 oraz xmm3/ymm3 lub m128/m256, wynik zapisuje w xmm1.

cel[i] źródło1[i] and źródło2[i]

vpandn xmm1, xmm2, xmm3/m128 (AVX)
vpandn ymm1, ymm2, ymm3/m256 (AVX2)

Oblicza **iloczyn logiczny bit po bicie** operendu2 xmm3/ymm3 lub m128/m256 oraz negacji operendu1 xmm2/ymm2, wynik zapisuje w xmm1/ymm1.

cel[i] = (not źródło1[i]) and źródło2[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje logiczne alternatywa
VPOR / VPXOR

vpor xmm1, xmm2, xmm3/m128
vpor ymm1, ymm2, ymm3/m256 (AVX2)

Oblicza **sumę logiczną bit po bicie** dla wszystkich bitów rejestrów xmm2/ymm2 oraz xmm3/ymm3 lub m128/m256, wynik zapisuje w xmm1/ymm1.

cel[i] = źródło1[i] or źródło2[i]

vpxor xmm1, xmm2, xmm3/m128 (AVX)
vpxor ymm1, ymm2, ymm3/m256 (AVX2)

Oblicza **alternatywę wykluczającą bit po bicie** dla wszystkich bitów rejestrów xmm2/ymm2 oraz xmm3/ymm3 lub m128/m256, wynik zapisuje w zmm1/ymm1.

cel[i] = źródło1[i] xor źródło2[i]

Bity od 128/256 do MSB są zerowane.

Bity od 128/256 do MSB są zerowane.

Instrukcje dodatkowe

Instrukcje dodatkowe
VZEROALL / VZEROUPPER

- vzeroall (AVX)

Zeruje wszystkie rejestry ymm/0 - ymm/15

- vzeroupper (AVX)

Zeruje bity od 128. do ostatniego rejestrów ymm/0 – ymm/15 / zmm/0 – zmm/15.

W trybie 32 bitowym zeruje tylko pierwsze 8 rejestrów.

Bity od 128/256 do MSB są zerowane.

Instrukcje dodatkowe
VLDMXCSR / VSTMXCSR

- vldmxcsr m32 (AVX)

Ładuje zawartość operandu źródłowego m32 do rejestru kontrolnego i statusu (MXCSR Control and Status Register), jest to **ładowanie ustawień.**

- vstmxcsr m32 (AVX)

Przesyła zawartość rejestru kontrolnego i statusu (MXCSR Control and Status Register) do operandu źródłowego m32, jest to **kopiowanie ustawień.**

Instrukcja łączenia dodatkowe
VPALIGNR

vpalignr xmm1, xmm2, xmm3/m128, imm8 (AVX)
vpalignr ymm1, ymm2, ymm3/m256, imm8 (AVX2)

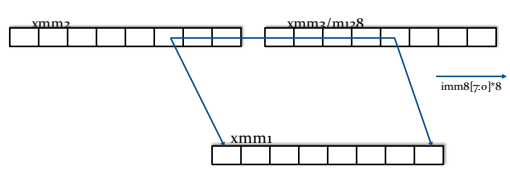
Łączy (konkatenacja) rejestry źródła xmm2/ymm2 i xmm3/ymm3 lub m128/m256, na podstawie bajtu sterującego przesuwa 128 bitowe części o imm8*8 i zapisuje 128 bitowe części do rejestru celu xmm1/ymm1.

cel = (źródło1+źródło2) >> źródło3
xmm1 = (xmm2+xmm3/m128) >> imm8[7:0]*8
hi ymm1 = (hi ymm2 +hi ymm3/m256) >> imm8[7:0]*8
lo ymm1 = (lo ymm2 + lo ymm3/m256) >> imm8[7:0]*8

Bity od 128/256 do MSB są zerowane.

Instrukcja łączenia dodatkowe
VPALIGNR

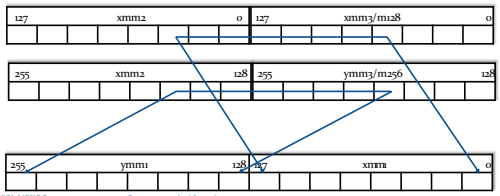
Sposób łączenia rejestrów xmm



Instrukcja łączenia dodatkowe

VPALIGNR

Sposób łączenia rejestrów xmm



Instrukcje szyfrujące

Operacje szyfrujące AVX

- Instrukcje szyfrujące

VAESENC, VAESENCLAST
VAESDEC, VAESDECLAST
VAESIMC, VAESKEYGENASSIST

Instrukcje szyfrujące

Algorytm AES (Advanced Encryption Standard) (1/2)

Instrukcje typu AVX udostępniają szyfrowanie danych z zastosowaniem algorytmu AES jedynie w wersji 128 bitowej.

Algorytm AES występuje w trzech wariantach

- AES-128 używa klucza 128 bitowego (możliwe 10 rund szyfrowania)
- AES-192 używa klucza 192 bitowego (możliwe 12 rund szyfrowania)
- AES-256 używa klucza 256 bitowego (możliwe 14 rund szyfrowania)

to jednak podstawową jednostką do zaszyfrowania/odszyfrowania jest blok danych 128 bitowy wykorzystując odpowiednio klucz 128, 192, 256 bitowy.

Instrukcje szyfrujące

Algorytm AES (Advanced Encryption Standard) (2/2)

AES jest algorytmem symetrycznym to znaczy, że ten sam klucz jest stosowany do zaszyfrowania i odszyfrowania danych.

Dane podlegają trzem rodzajom przekształceń:

- podstawianie (substitution),
- transponowanie (transposition)
- mieszanie (mixing)

po czym następuje zestawienie przekształconych danych (alternatywa wykluczająca) z kluczem.

Instrukcja szyfrująca AES (Advanced Encryption Standard)

VAESENC

vaesenc xmm1, xmm2, xmm3/m128

Szyfruje jedną rundą (jednokrotnie) dane (blok danych) całego rejestru xmm2 (128 bitów) z wykorzystaniem 128 bitowego klucza symetrycznego (ten sam klucz do szyfrowania i odszyfrowania) zapianego w rejestrze xmm3/m128, zaszyfrowane dane zapisuje w rejestrze xmm1.

```
[cel = aes(źródło1) xor źródło2 (key)]
for (unsigned int i =0; i < [1-9]; i++)
{
    xmm1 = aes(xmm2) xor xmm3/m128
}
```

Bity od 128/256 do MSB nie są modyfikowane.

Instrukcja szyfrująca AES (Advanced Encryption Standard)

VAESENCLAST

vaesenclast xmm1, xmm2, xmm3/m128

Szyfruje jedną ale ostatnią rundą dane (blok danych) całego rejestru xmm2 (128 bitów) z wykorzystaniem 128 bitowego klucza zapianego w rejestrze xmm3/m128, zaszyfrowane dane zapisuje w rejestrze xmm1.

```
cel = aes(źródło1) xor źródło2 (key)
xmm1 = aes(xmm2) xor xmm3/m128
```

Bity od 128/256 do MSB nie są modyfikowane.

Instrukcja szyfrująca AES (Advanced Encryption Standard)

VAESDEC

vaesdec xmm1, xmm2, xmm3/m128

Odszyfrowuje jedną rundą (jednokrotnie) blok danych całego rejestru xmm2 (128 bitów) z wykorzystaniem 128 bitowego klucza wcześniej użytego do zaszyfrowania zapianego w rejestrze xmm3/m128, odszyfrowane dane zapisuje w rejestrze xmm1.

```
for (unsigned int i = 0; i < [1-9]; i++)
{
    [cel = aes(źródło1) xor źródło2 (key)]
    xmm1 = aes(xmm2) xor xmm3/m128
}
```

Bity od 128/256 do MSB nie są modyfikowane.

Instrukcja szyfrująca AES (Advanced Encryption Standard)

VAESDECLAST

vaesdeclast xmm1, xmm2, xmm3/m128

Odszyfrowuje jedną ale ostatnią rundą dane (blok danych) całego rejestru xmm2 (128 bitów) z wykorzystaniem 128 bitowego klucza zapianego w rejestrze xmm3/m128, odszyfrowane dane zapisuje w rejestrze xmm1.

```
cel = aes(źródło1) xor źródło2 (key)
xmm1 = aes(xmm2) xor xmm3/m128
```

Bity od 128/256 do MSB nie są modyfikowane.

Instrukcja szyfrująca AES (Advanced Encryption Standard)

VAESIMC

vaesimc xmm1, xmm2/m128

Dokonuje **przekształcenia** 128 bitowego **klucza** zapisanego w xmm2/m128 poprzez odwróconą funkcję mieszania kolumn InvMixColumns(), wynik zapisuje w xmm1.

Funcja InvMixColumns() jest odwrotnością funkcji MixColumns().

```
cel = InvMixColumn(źródło-key)
xmm1 = InvMixColumn(xmm2/m128)
```

Bity od 128/256 do MSB nie są modyfikowane.

Instrukcja szyfrująca AES (Advanced Encryption Standard)

VAESKEYGENASSIST

vaeskeygenassist xmm1, xmm2/m128 , imm8

Asystuje w rozszerzeniu **klucza**, poprzez obliczanie kroków w kierunku wygenerowania nowego klucza do zaszyfrowania, używając RoundConstant (pełni funkcję klucza klucza) ma sposób zdefiniowany w bajcie sterującym imm8, wynik zapisuje w rejestrze celu xmm1.

```
cel = szyfrowanie(źródło1-key), źródło2
xmm1 = syfrowanie(xmm2/m128), imm8
```

Bity od 128/256 do MSB nie są modyfikowane.

Instrukcje AVX

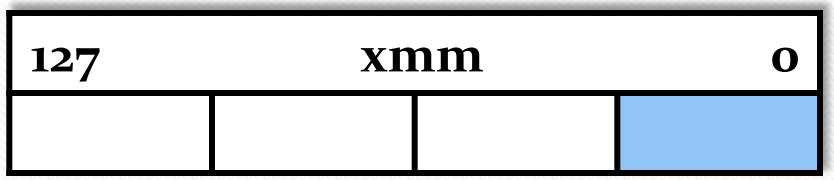
Advanced Vector Extensions

Instrukcje AVX

Część II

liczby zmienno-przecinkowe

Operacje na skalarach



SS skalar / **pojedynczej precyzji**

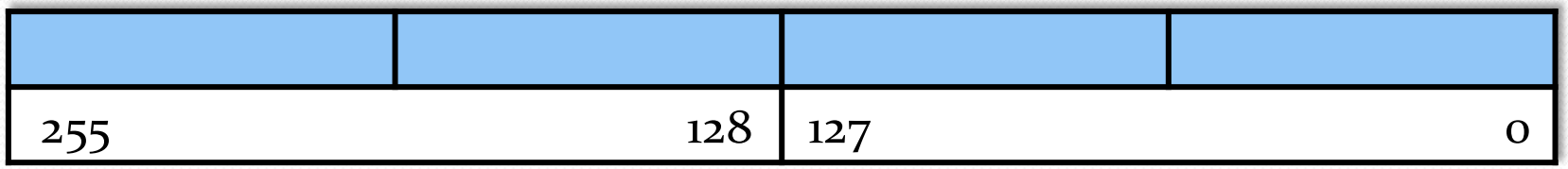
SD skalar / **podwójnej precyzji**

Operacje na wektorach

PS wektor / **pojedynczej precyzji**

PD wektor / **podwójnej precyzji**

PS wektor / **pojedynczej precyzji**



PD wektor / **podwójnej precyzji**

Operacje AVX zmienno-przecinkowe

- Instrukcje przesłania
- Instrukcje arytmetyczne (w tym FMA)
- Instrukcje porównania
- Instrukcje logiczne
- Instrukcje konwersji

Instrukcje przesłania AVX

- Instrukcje przesłania

VMOVS[S/D], VMOV[U/A]P[S/D]

VMOVNTP[S/D], VMOV[H/L]P[S/D]

VMOV[HL/LH]PS, VMOV[D/SH/SL]DUP

VMASKMOVP[S/D]

- Instrukcje konwersji: VUNPACK[H/L]P[S/D]
- Instrukcje wstawiania: VINSERTPS, VINSERTF128
- Instrukcje wyciągania: VEXTRACTPS, VEXTRACTF128

Instrukcje przesłania

Instrukcja przesłania

VMOVSS

1. vmovss xmm1, xmm2, xmm3

Przepisuje z xmm2 do xmm1 liczbę pojedynczej precyzji pozostałe uzupełnia z xmm3.

xmm1[31:0] ← xmm3[31:0]

xmm1[127:32] ← xmm2[127:32}

2. vmovss xmm1, m32

Przepisuje liczbę rzeczywistą pojedynczej precyzji z pamięci m32 do rejestru xmm1.

xmm1 ← m32

xmm1[127:32] ← 0

3. vmovss m32, xmm1

Przesyła liczbę rzeczywistą pojedynczej precyzji (scalar) z xmm1 do pamięci m32.

m32 ← xmm1[31:0]

Bity od 32/128/256 do MSB są zerowane.

Instrukcja przesłania

VMOVSD

1. vmovsd xmm1, xmm2, xmm3

Przepisuje z xmm2 do xmm1 liczbę podwójnej precyzji pozostałe uzupełnia z xmm3.

$$xmm1[63:0] \leftarrow xmm2[63:0]$$
$$xmm1[127:64] \leftarrow xmm3[127:64]$$

2. vmovsd xmm1, m64

Przepisuje liczbę rzeczywistą podwójnej precyzji z pamięci m64 do rejestru xmm1.

$$xmm1[63:0] \leftarrow m64[63:0]$$
$$xmm1[127:64] \leftarrow 0$$

Bity od 64/128/256 do MSB są zerowane.

3. vmovsd m64, xmm1

Przesyła liczbę rzeczywistą pojedynczej precyzji z xmm1 do pamięci m64.

$$m64 \leftarrow xmm1[64:0]$$

Instrukcja przesłania

VMOV[U/A]P[S/D]

vmov[u/a]p[s/d] xmm1, xmm2/m128

vmov[u/a]p [s/d] ymm1, ymm2/m256

Przesyła wektory liczb rzeczywistych pojedynczej/podwójnej precyzji **bez wyrównania / z wyrównaniem** (U – unaligned / A – aligned) z xmm2/ymm2 lub m128/m256 do xmm1/ymm1.

cel ← źródło | xmm1/ymm1 ← xmm2/ymm2 lub m128/m256

vmov[u/a]p[s/d] xmm2/m128, xmm1

vmov[u/a]p[s/d] ymm2/m256, ymm1

Przesyła wektory liczb zmienno-przecinkowych pojedynczej/podwójnej precyzji **bez wyrównania / z wyrównaniem** (U – unaligned / A - aligned) z xmm1/ymm1 do xmm2/ymm2 lub m128/m256.

cel ← źródło | xmm2/ymm2 lub m128/m256 ← xmm1/ymm1

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania

VMOVNTP[S/D]

vmovntp[s/d] m128, xmm1

vmovntp[s/d] m256, ymm1 (AVX2)

Przesyła wektor liczb rzeczywistych pojedynczej / podwójnej precyzji z rejestru xmm1/ymm1 do pamięci m128/m256.

cel ← źródło

m128 ← xmm1

m256 ← ymm1

NT oznacza (non-temporal hint) przesłanie z pominięciem pamięci podręcznej (cache).

Bity od 128/256 do MSB są zerowane.

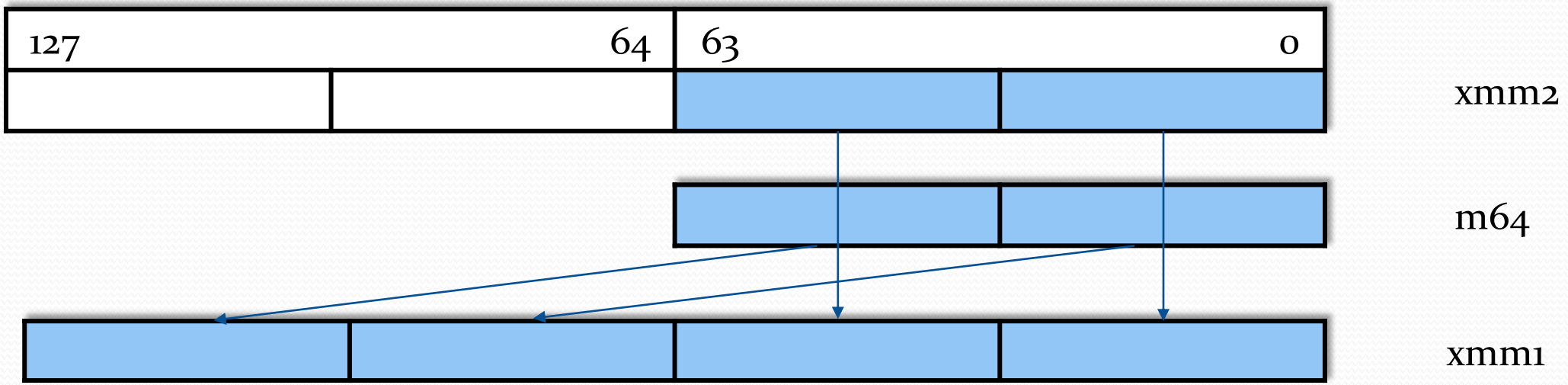
Instrukcja przesłania

VMOVHPS

1. vmovhps xmm1, xmm2, m64

Przesyła **dwie wartości** rzeczywiste pojedynczej precyzji z młodszej połowy xmm2 do młodszej połowy xmm1 oraz dwie wartości rzeczywiste z pamięci m64 do starszej połowy rejestru celu xmm1.

$$xmm1[63:0] \leftarrow xmm2[63:0] \;\&\&\; xmm1[127:64] \leftarrow m64[63:0]$$



Bity od 128/256 do MSB są zerowane.

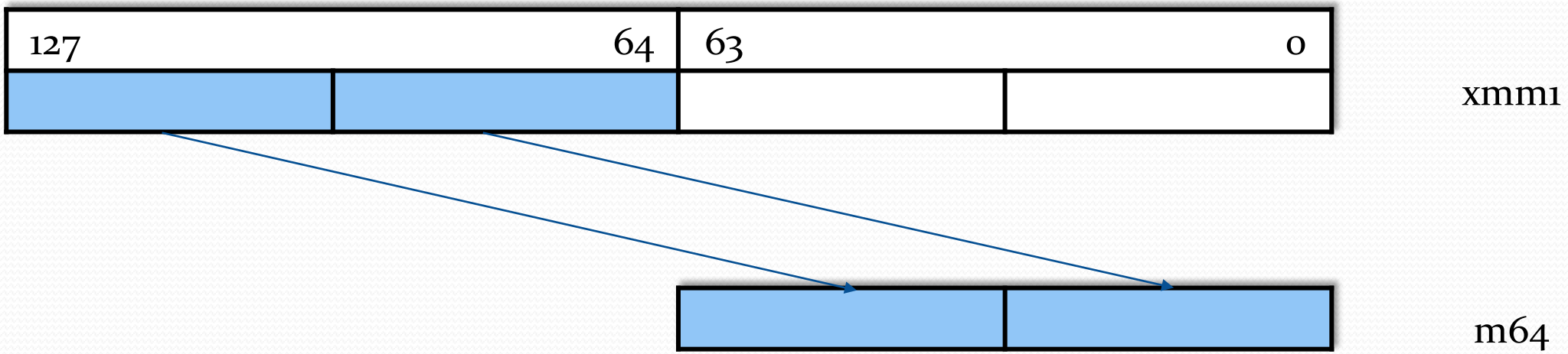
Instrukcja przesłania

VMOVHPS

2. vmovhps m64, xmm1

Przesyła **dwie wartości** rzeczywiste pojedynczej precyzji ze starszej połowy xmm1 do pamięci m64.

m64 ← xmm1[127:64]



Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania

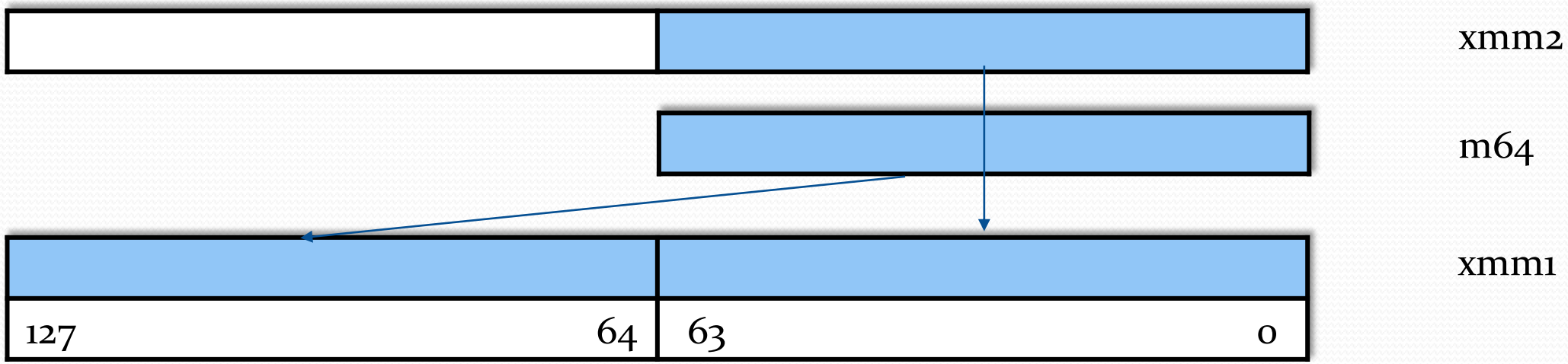
VMOVHPD

1. vmovhpd xmm2, xmm1, m64

Kopiuje wartości liczb rzeczywistych podwójnej precyzji z młodszej połowy rejestru xmm1 oraz z pamięci m64, wynik zapisuje w xmm2.

xmm1[63:0] ← xmm2[63:0] && xmm1[127:64] ← m64[63:0]

.



Bity od 128/256 do MSB są zerowane.

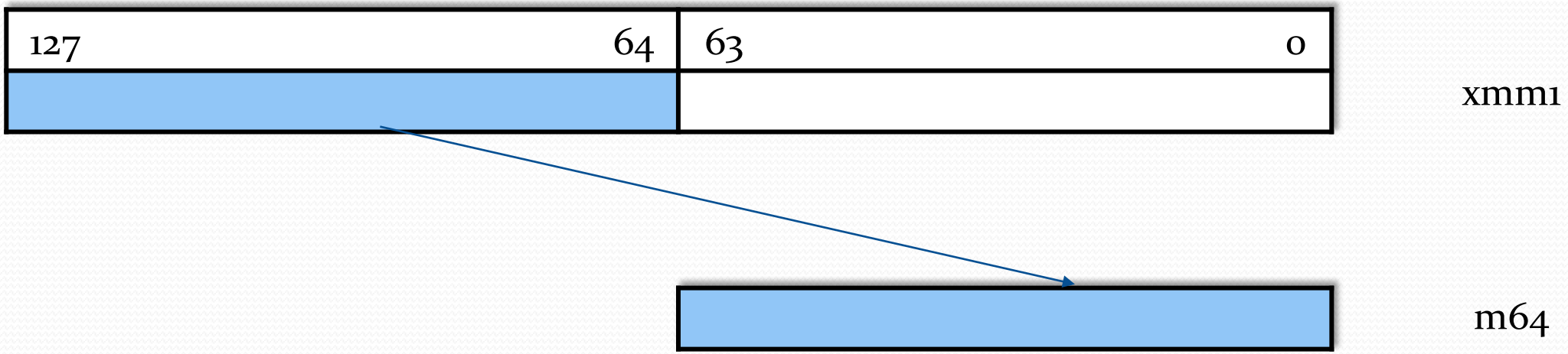
Instrukcja przesłania

VMOVHPD

2. vmovhpd m64, xmm1

Kopiuje liczbę rzeczywistą podwójnej precyzji ze starszej połowy xmm1 do pamięci m64.

m64 ← xmm1[127:64]



Bity od 128/256 do MSB są zerowane.

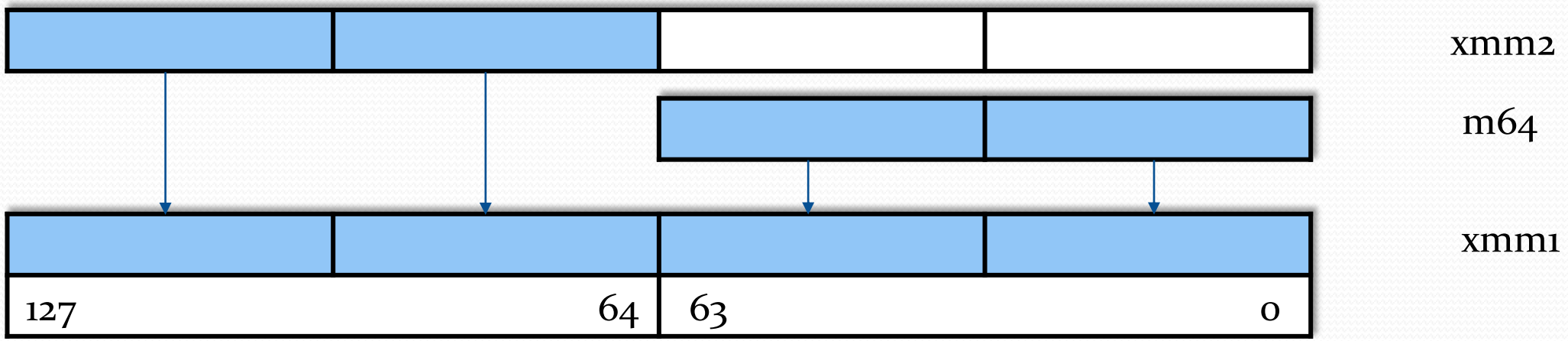
Instrukcja przesłania

VMOVLPS

1. vmovlps xmm1, xmm2, m64

Przepisuje po dwie wartości rzeczywiste pojedynczej precyzji ze starszej połowy rejestru xmm2 do xmm1 oraz z pamięci m64, wynik zapisuje w xmm1.

$$xmm1[63:0] \leftarrow m64[63:0] \;\&\&\; xmm1[127:64] \leftarrow xmm2[127:64]$$



Bity od 128/256 do MSB są zerowane.

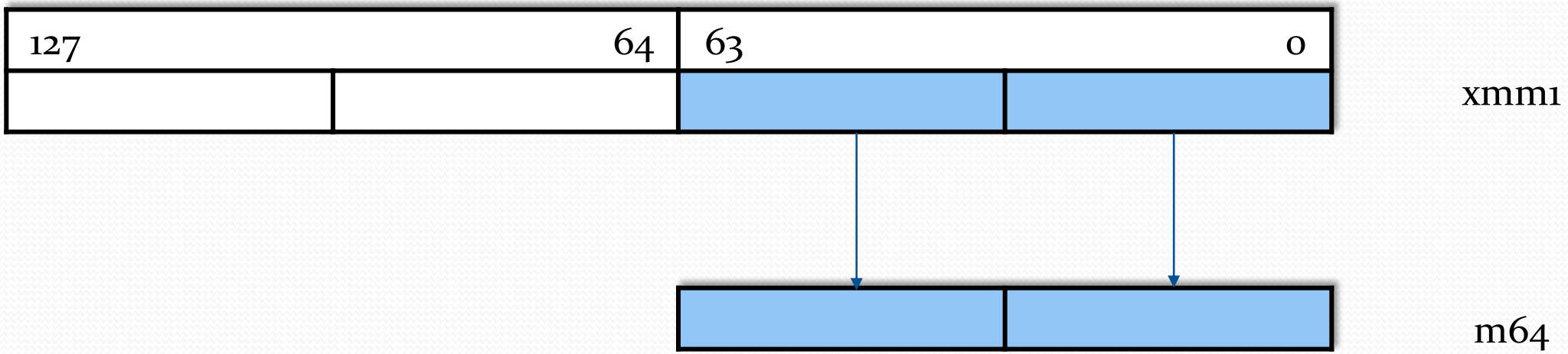
Instrukcja przesłania

VMOVLPS

2. vmovlps m64, xmm1

Przesyła dwie wartości liczb rzeczywistych pojedynczej precyzji z młodszej połowy xmm1 do pamięci m64.

m64[63:0] ← xmm1[63:0]



Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania

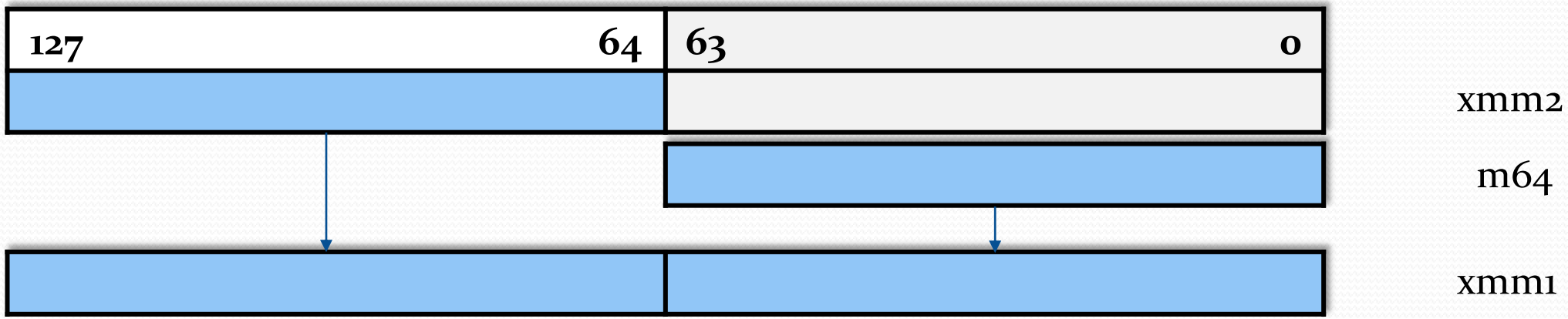
VMOVLPD

1. vmovlpd xmm1, xmm2, m64

Przesyła liczbę rzeczywistą podwójnej precyzji z pamięci m64 oraz starszą **połową** rejestru xmm2, wynik zapisuje w xmm1.

$$xmm1[63:0] \leftarrow m64[63:0] \;\&\&\; xmm1[127:64] \leftarrow xmm2[127:64]$$

.



Bity od 128/256 do MSB są zerowane.

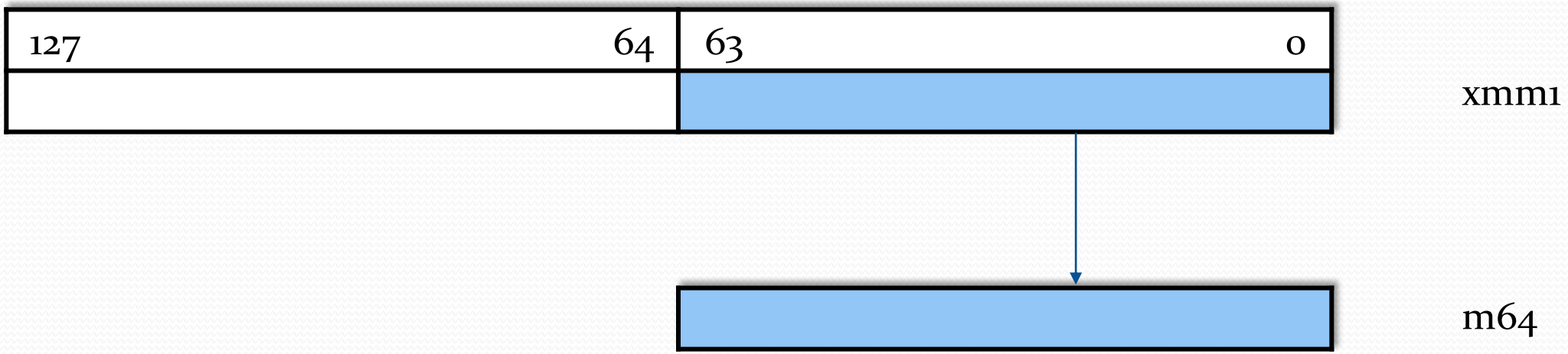
Instrukcja przesłania

VMOVLPD

2. vmovlpd m64, xmm1

Przesyła **liczbę** rzeczywistą podwójnej precyzji z młodszej połowy xmm1 do pamięci m64.

m64[63:0] ← xmm1[63:0]



Bity od 128/256 do MSB są zerowane.

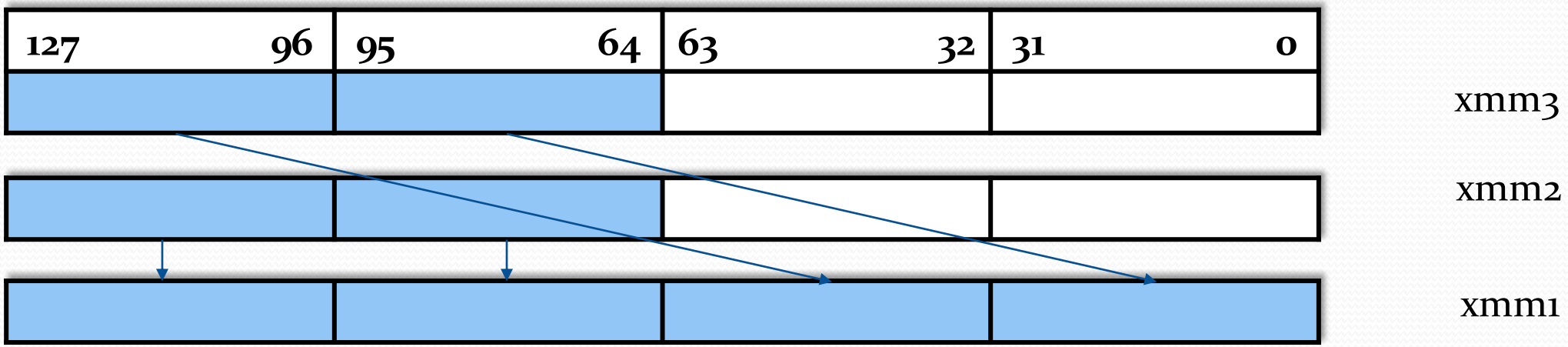
Instrukcja przesłania

VMOVHLPS

vmovhlps xmm1, xmm2, xmm3

Przesyła ze starszej połowy rejestru xmm3 do młodszej połowy xmm1 oraz z starszej połowy rejestru xmm2 do starszej połowy rejestru celu po dwie **liczby rzeczywiste pojedynczej precyzji**, wynik zapisuje w xmm1.

xmm1[63:0] ← xmm3[127:64] && xmm1[127:64] ← xmm2[127:64]



Bity od 128/256 do MSB są zerowane.

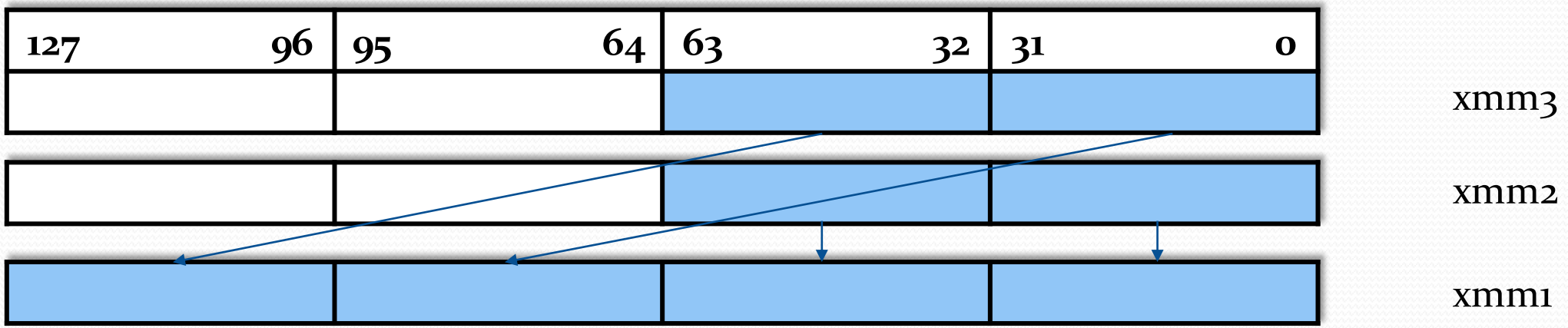
Instrukcja przesłania

VMOVLHPS

vmovlhps xmm1, xmm2, xmm3

Przesyła z młodszej połowy rejestru xmm2 do młodszej połowy rejestru celu oraz ze młoszej połowy rejestru xmm3 do starszej połowy rejestru celu po dwie **liczby rzeczywiste pojedynczej precyzji**, wynik zapisuje w xmm1.

xmm1[63:0] ← xmm2[63:0] && xmm1[127:64] ← xmm3[127:63]



Bity od 128/256 do MSB są zerowane.

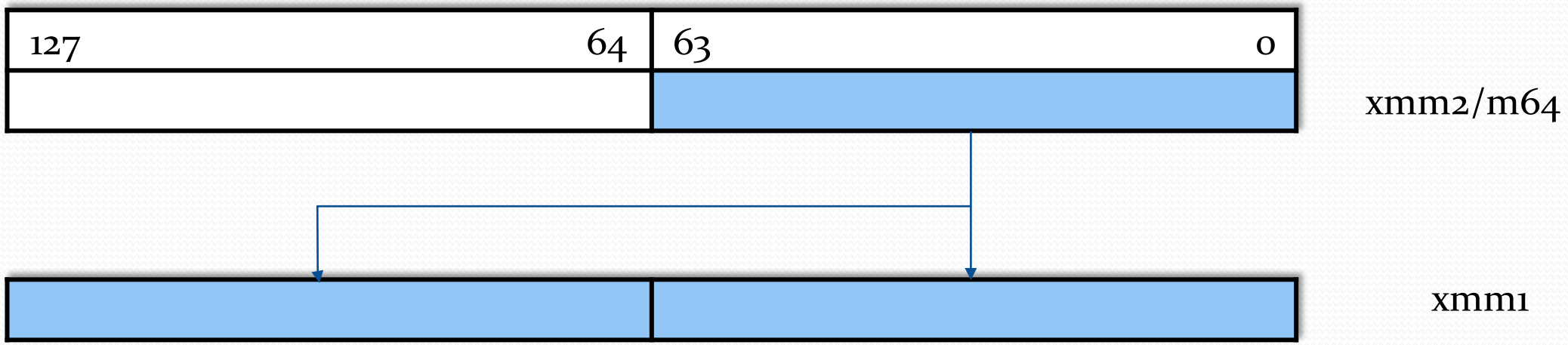
Instrukcja przesłania

VMOVDDUP

vmovddup xmm1, xmm2/m64

Kopiuje liczbę rzeczywistą podwójnej precyzji z rejestru xmm2 lub pamięci m64 do obu części rejestru xmm1.

$$xmm1[63:0] \leftarrow xmm2/m64[63:0] \;\&\&\; xmm1[127:64] \leftarrow xmm2/m64[63:0]$$



Bity od 128/256 do MSB są zerowane.

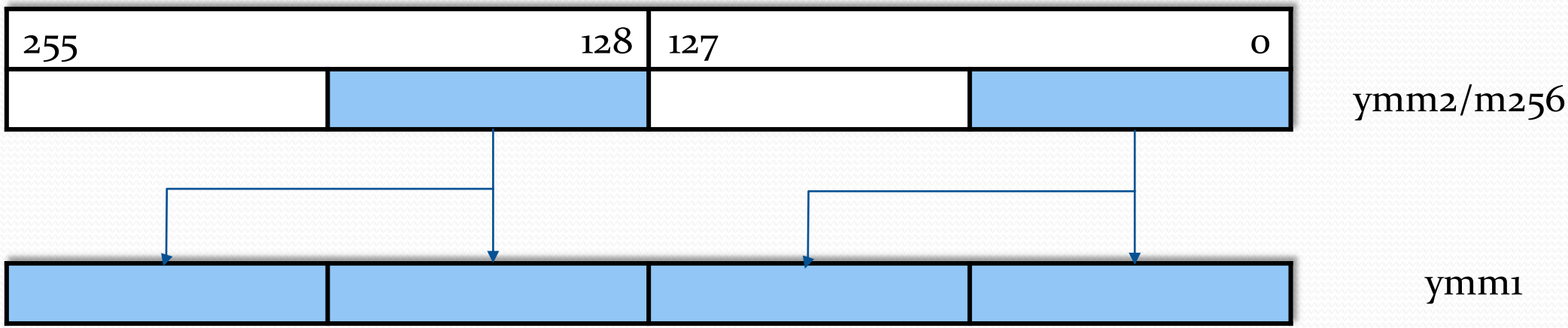
Instrukcja przesłania

VMOVDDUP

vmovddup ymm1, ymm2/m256

Kopiuje liczby rzeczywiste podwójnej precyzji o parzystych indeksach z rejestru ymm2 lub pamięci m256 do ymm1.

$$ymm1[2i] \leftarrow ymm2/m256[2i] \;\&\&\; ymm1[2i+1] \leftarrow ymm2/m256[2i]$$



Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania

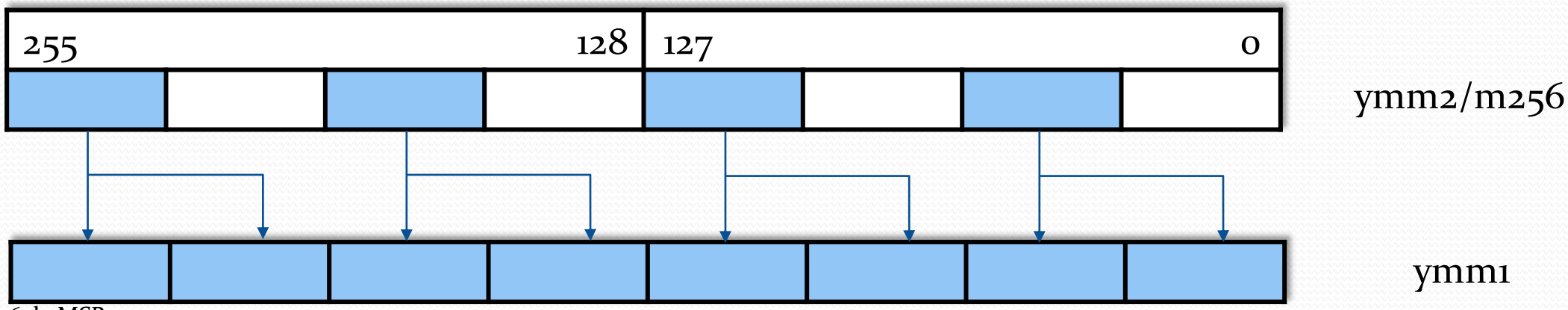
VMOVSHDUP

vmovshdup xmm1, xmm2/m128

vmovshdup ymm1, ymm2/m256

Kopiuje z powieleniem wartości liczb rzeczywistych pojedynczej precyzji o **nieparzystych indeksach** xmm2/ymm2 lub m128/m256 i zapisuje do xmm1/ymm1.

ymm1[2i] ← ymm2/m256[2i+1] && ymm1[2i+1] ← ymm2/m256[2i+1]



Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania

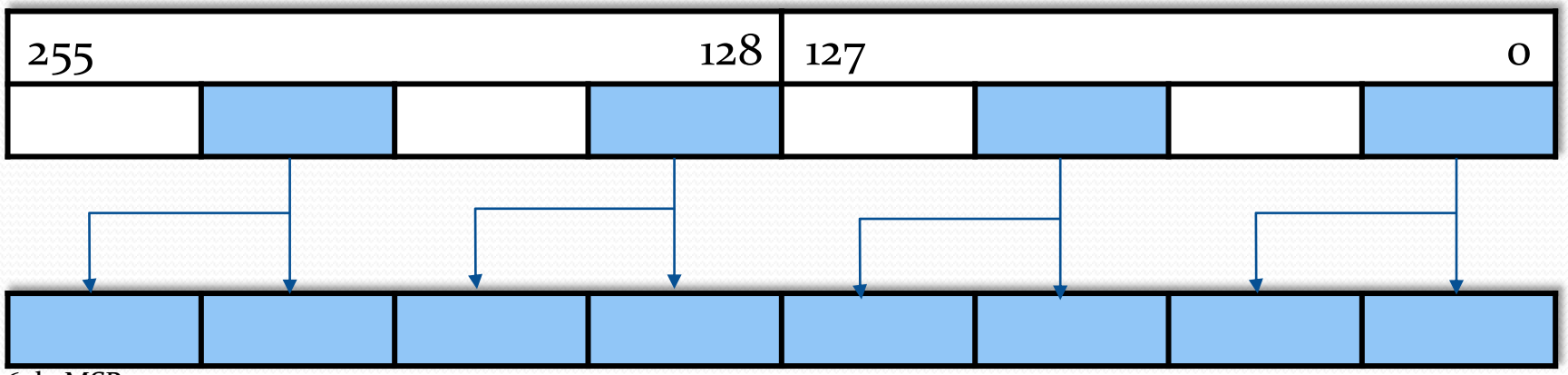
VMOVSLDUP

vmovsldup xmm1, xmm2/m128

vmovsldup ymm1, ymm2/m256

Kopiuje z powieleniem wartości liczb rzeczywistych pojedynczej precyzji o **parzystych indeksach** xmm2/ymm2 lub m128/m256 i zapisuje do xmm1/ymm1.

xmm1[2i] ← xmm2/m128[2i] && ymm1[2i+1] ← ymm2/m256[2i]



Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPS (1/4)

vmaskmovps xmm1, xmm2, m128

vmaskmovps ymm1, ymm2, m256

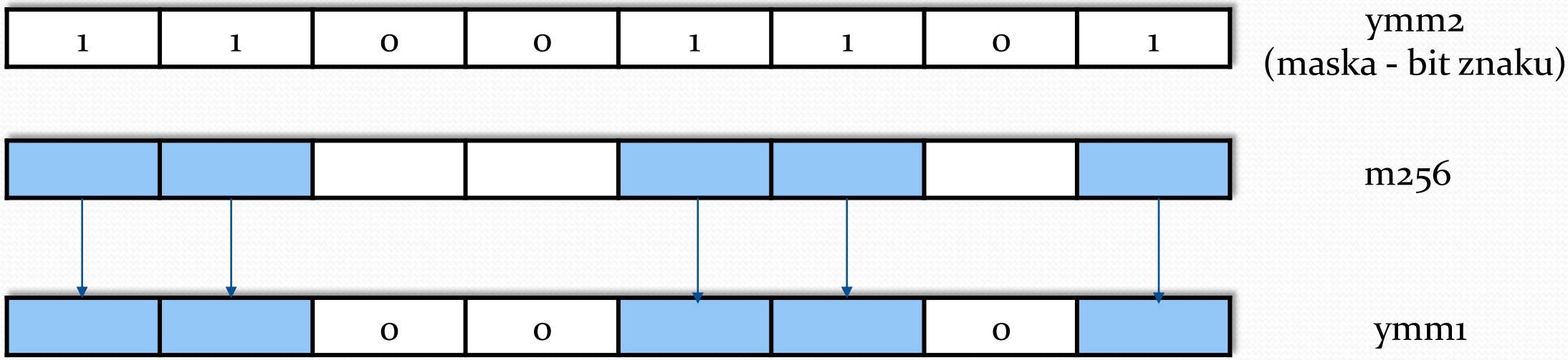
Przesyła liczby rzeczywiste pojedynczej precyzji z pamięci m128/m256 do rejestru celu xmm1/ymm1, pod warunkiem, że bit znaku odpowiadających wartości z rejestru maski (drugi operand) xmm2/ymm2 lub xmm1/ymm1 **jest ustawiony na jeden**, w przeciwnym wypadku zapisuje zero.

```
if xmm2[i][31] then xmm1[i] ← m128[i]
        else xmm1[i] = 0

if ymm2[i][31] then ymm1[i] ← m256[i]
        else ymm1[i] = 0
```

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPS (2/4)



Model przesłania warunkowego z pamięci

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPS (3/4)

vmaskmovps m128, xmm1, xmm2

vmaskmovps m256, ymm1, ymm2

Przesyła liczby rzeczywiste pojedynczej precyzji z rejestru xmm2/ymm2 do pamięci m128/m256 pod warunkiem, że bit znaku odpowiadających wartości z rejestru maski (drugi operand) xmm1/ymm1 **jest ustawiony na jeden,** w przeciwnym wypadku zapisuje zero.

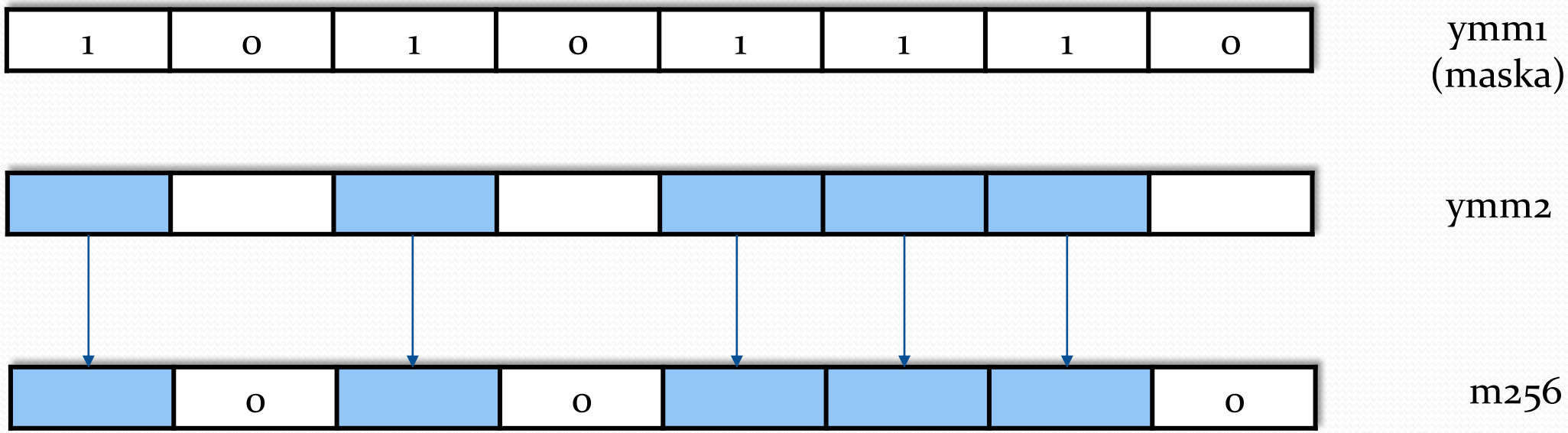
if xmm1[i][31] then m128[i] ← xmm2[i]

if ymm1[i][31] then m256[i] ← ymm2[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPS (4/4)



Model przesłania warunkowego do pamięci

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPD (1/4)

vmaskmovpd xmm1, xmm2, m128

vmaskmovpd ymm1, ymm2, m256

Pobiera kolejne elementy pamięci **podwójnej precyzji z m128/m256**, wynik zapisuje warunkowo w ymm1/xmm1 według **maski** (bit znaku) zdefiniowanej w **ymm2/xmm2.**

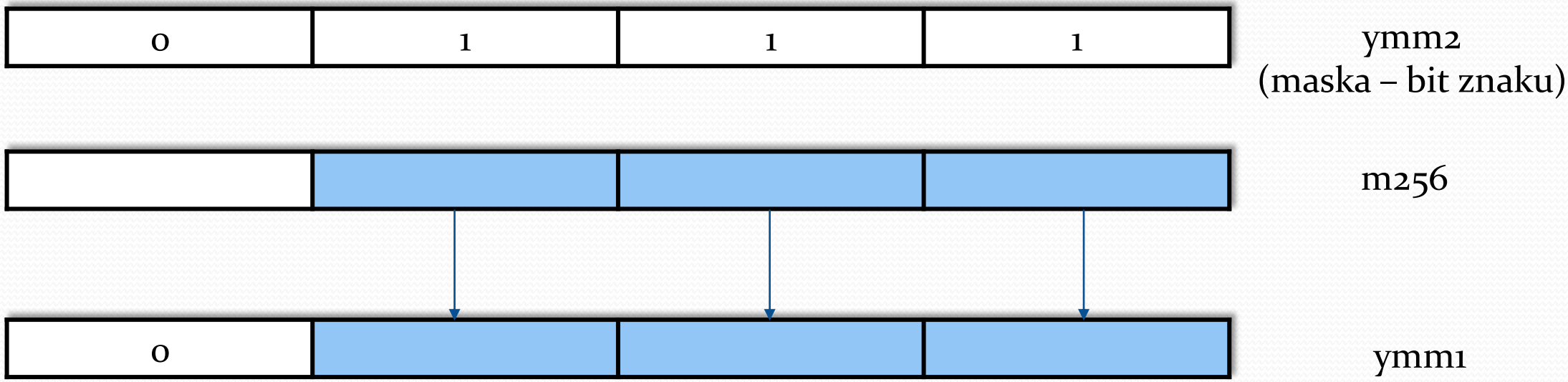
if xmm2[i][63] then xmm1[i] ← m128[i]
else xmm1[i] = 0

if ymm2[i][63] then ymm1[i] ← m256[i]
else ymm1[i] = 0

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPD (2/4)



Model przesłania warunkowego z pamięci

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPD (3/4)

vmaskmovpd m128, xmm1, xmm2

vmaskmovpd m256, ymm1, ymm2

Zapisuje do pamięci m128/m256 kolejne elementy wektora **podwójnej precyzji** z xmm2/ymm2, według maski (bit znaku) zdefiniowanej w ymm1/ xmm1.

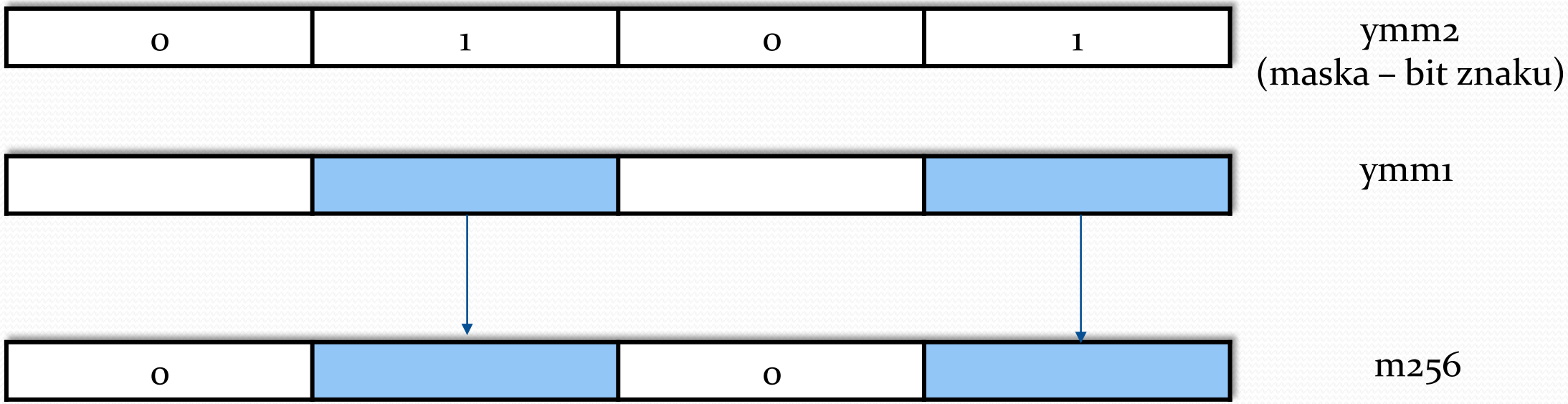
if xmm1[i][63] then m128[i] ← xmm2[i]

if ymm1[i][63] then m256[i] ← ymm2[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja przesłania warunkowego

VMASKMOVPD (4/4)



Model przesłania warunkowego do pamięci

Bity od 128/256 do MSB są zerowane.

Instrukcje dekompresji

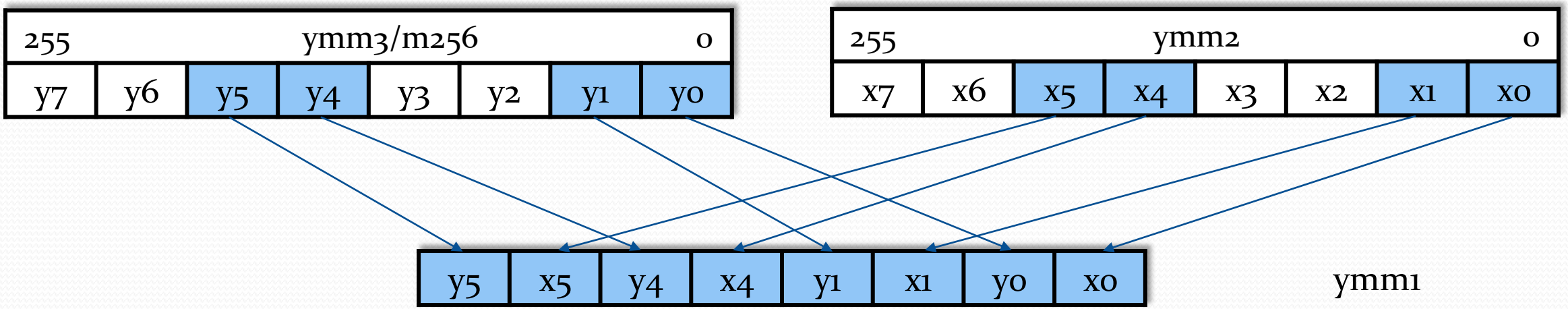
Instrukcja dekompresji

VUNPCKLPS

vunpcklps xmm1, xmm2, xmm3/m128

vunpcklps ymm1, ymm2, ymm3/m256

Przepisuje liczby rzeczywiste pojedynczej precyzji. Młodsze dwie liczby ze 128-bitowych części rejestru xmm2/ymm2 i xmm3/m128 / ymm3/m256 są zapisywane **z przeplotem** jako **liczby rzeczywiste pojedynczej precyzji** do rejestru xmm1/ymm1.



Bity od 128/256 do MSB są zerowane.

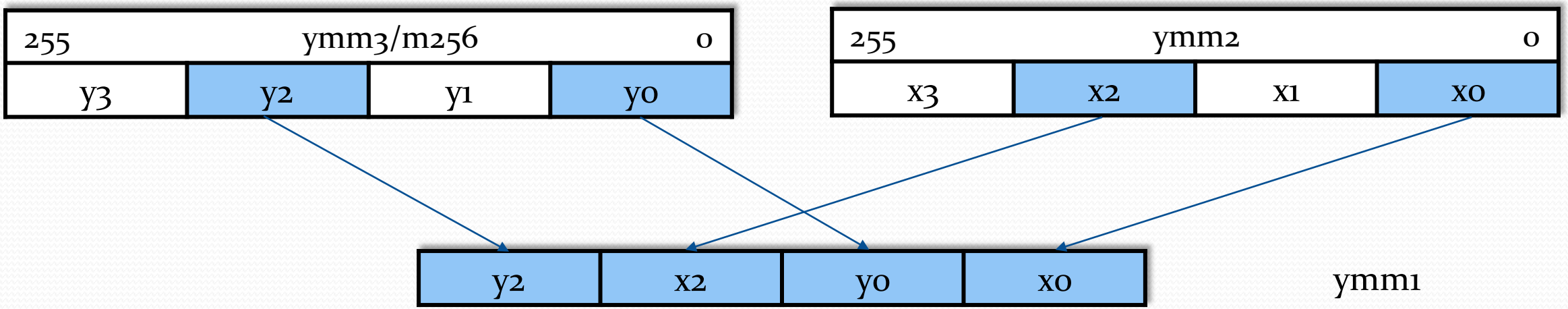
Instrukcja dekompresji

VUNPCKLPD

vunpcklpd xmm1, xmm2, xmm3/m128

vunpcklpd ymm1, ymm2, ymm3/m256

Przepisuje liczby rzeczywiste podwójnej precyzji. Młodsze liczby ze 128-bitowych części rejestru xmm2/ymm2 i xmm3/m128 / ymm3/m256 są zapisywane **z przeplotem** jako **liczby rzeczywiste podwójnej precyzji** do rejestru xmm1/ymm1.



Bity od 128/256 do MSB są zerowane.

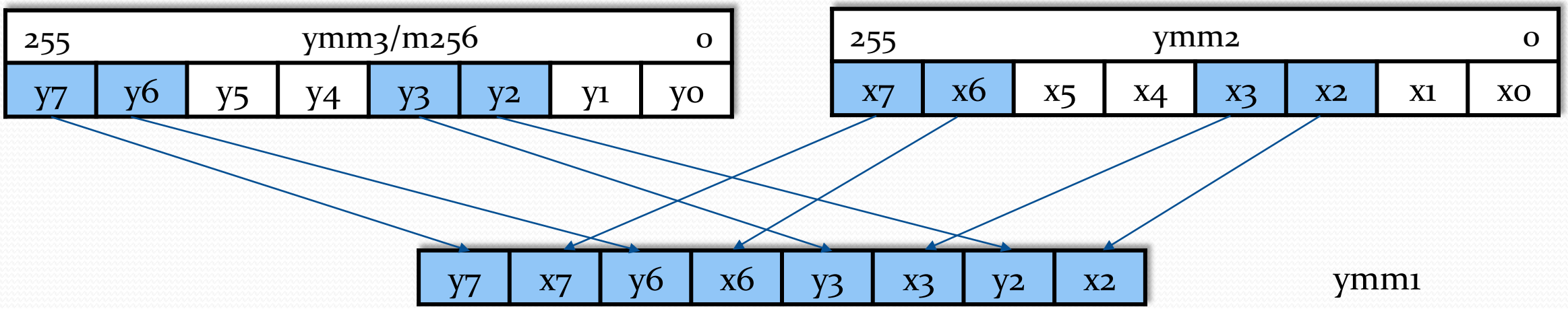
Instrukcja dekompresji

VUNPCKHPS

vunpckhps xmm1, xmm2, xmm3/m128

vunpckhps ymm1, ymm2, ymm3/m256

Przepisuje liczby rzeczywiste pojedynczej precyzji. Starsze dwie liczby ze 128-bitowych części rejestru xmm2/ymm2 i xmm3/m128 / ymm3/m256 są zapisywane **z przeplotem** jako **liczby rzeczywiste pojedynczej precyzji** do rejestru celu xmm1/ymm1.



Bity od 128/256 do MSB są zerowane.

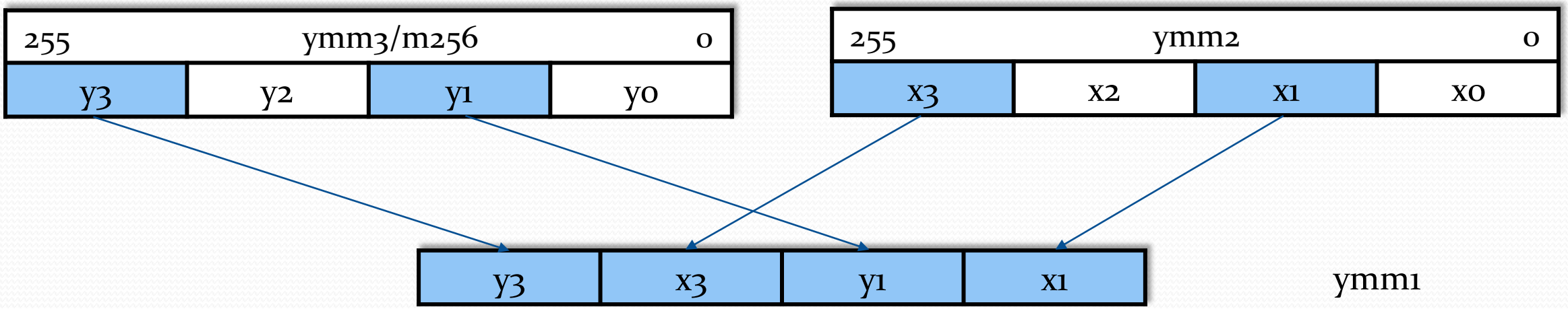
Instrukcja dekompresji

VUNPCKHPD

vunpckhpd xmm1, xmm2, xmm3/m128

vunpckhpd ymm1, ymm2, ymm3/m256

Przepisuje liczby rzeczywiste podwójnej precyzji. Starsze liczby ze 128-bitowych części rejestru xmm2/ymm2 i xmm3/m128 / ymm3/m256 są zapisywane **z przeplotem** jako **liczby rzeczywiste podwójnej precyzji** do rejestru celu xmm1/ymm1.



Bity od 128/256 do MSB są zerowane.

Instrukcje wstawiania

Instrukcja wstawiania

VINSERTPS (1/3)

vinsertps xmm1, xmm2, xmm3/m32, imm8

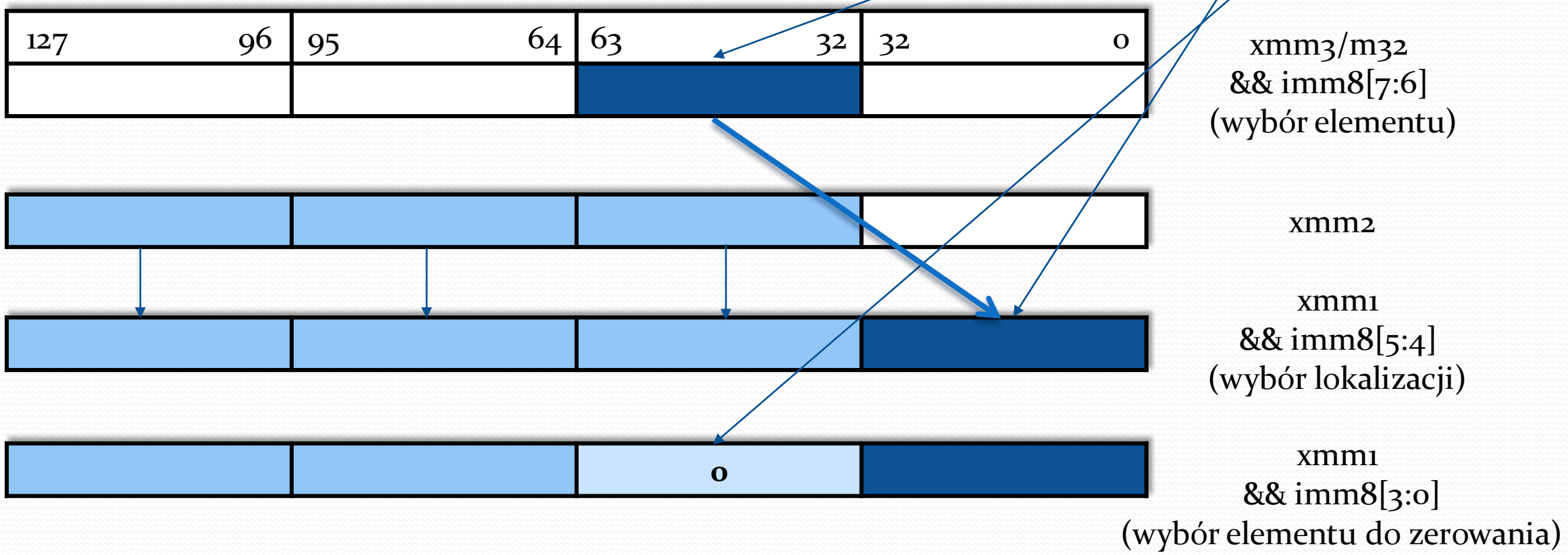
1. Kopiuje zawartość xmm2 do xmm1 oraz

2. Kopiuje element o indeksie imm8[7:6] z xmm3/m32
   do rejestru xmm1 pod index imm8[5:4]

3. Zeruje elementy wektora xmm1 jeśli odpowiadające im bity imm8[3:0] są równe 1

Instrukcja wstawiania

VINSERTPS (3/3)



Bity od 128/256 do MSB są zerowane.

Instrukcja wstawiania

VINSERTF128

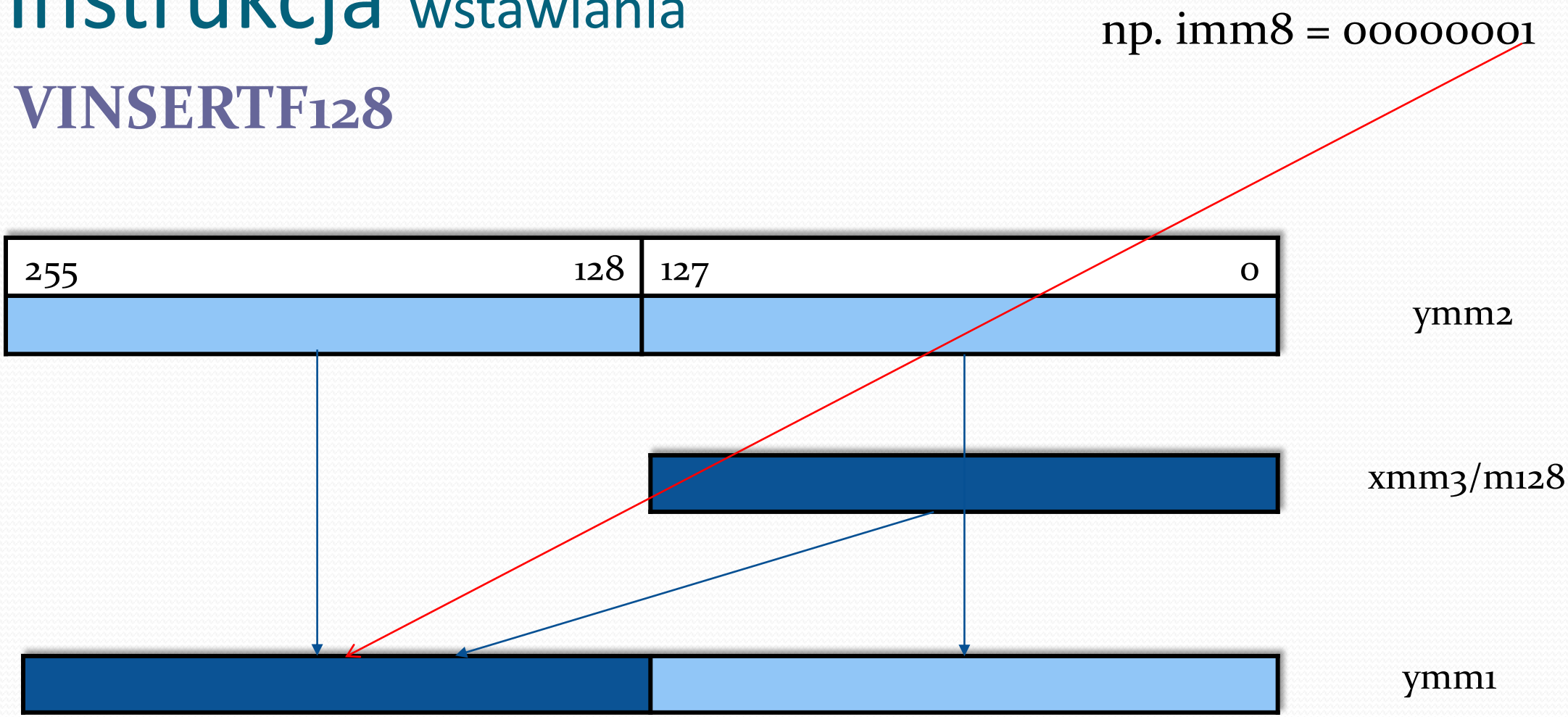
vinsertf128 ymm1, ymm2, xmm3/m128, imm8

Kopiuje połowę rejestru ymm2 oraz cały rejestr xmm3/m128 liczb rzeczywistych zależnie od najmłodszego bitu bajtu sterującego imm8[0].

```
if imm8[0] == 0 then
        ymm1 = ymm2
        ymm1[0] =  xmm3/m128
else
        ymm1 = ymm2
        ymm1[1] =  xmm3/m128
```

Instrukcja wstawiania

VINSERTF128



Instrukcje wybierania

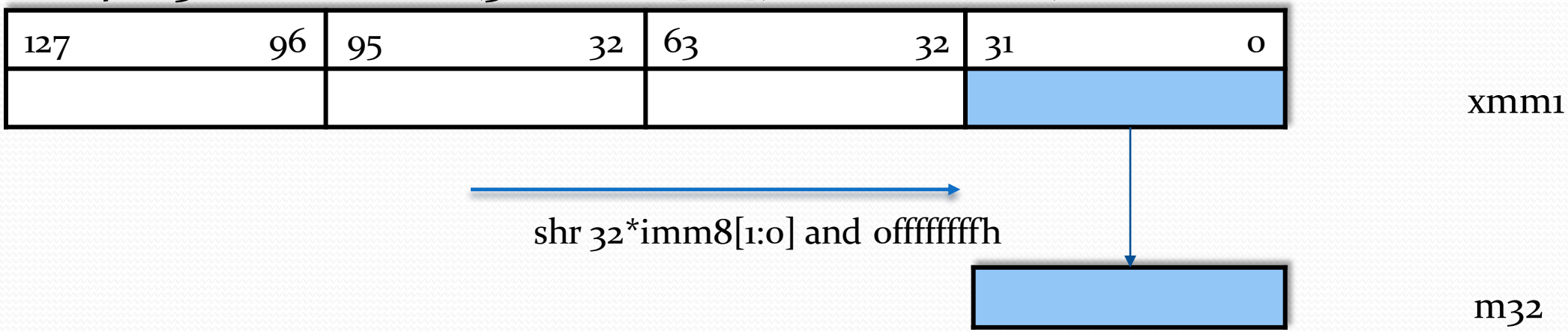
Instrukcja wybierania

VEXTRACTPS

vextractps reg/m32, xmm1, imm8

Wybiera z rejestru xmm1, liczbę rzeczywistą pojedynczej precyzji w oparciu o dwubitową wartość imm8[1:0] stanowiącą offset (wielokrotność przesunięcia bitowego) i **przesyła do rejestru ogólnego przeznaczenia**, (jeśli rejestr ma 64 bity. Wówczas starsza jego część jest zerowana) lub do pamięci.

m64/m32 = xmm1 >> (32*imm8[1:0]) and ffffffffh)



Instrukcja wybierania

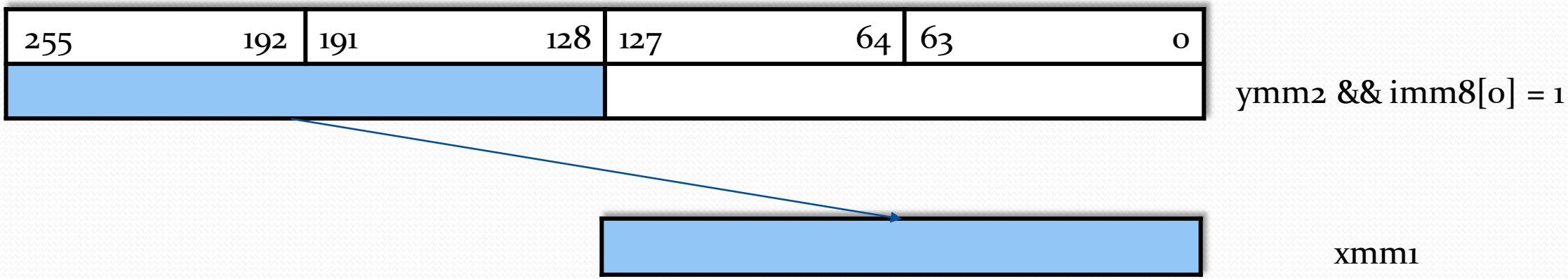
VEXTRACTF128

vextractf128 xmm1/m128, ymm2, imm8

Przesyła połowę rejestru (128 bitów) ymm2 liczb rzeczywistych pojedynczej lub podwójnej precyzji do rejestru xmm1 lub do pamięci m128 według bajtu sterującego imm8.

if imm8[0] = 0 then xmm1/m128 = ymm2[127:0]

else xmm1/m128 = ymm2[255:128]



Operacje przesłania AVX cd.

- Instrukcje mieszające: VBLENDP[S/D], VBLENDVP[S/D]
- Instrukcje rozgłaszania: VBROADCASTS[S/D], VBROADCASTF128
- Instrukcje zbierania: VGATHER[D/Q]P[S/D]
- Instrukcje permutacji: VPERMP[S/D], VPERMILP[S/D], VPREM2F128
- Instrukcje tasowania: VSHUFP[S/D]

Instrukcje mieszające

Instrukcja mieszająca

VBLENDP[S/D]

vblendp[s/d] xmm1, xmm2, xmm3/m128, imm8

vblendp[s/d] ymm1, ymm2, ymm3/m256, imm8

Wybiera komplementarnie elementy wektorów liczb rzeczywistych pojedynczej/podwójnej precyzji, z xmm3/ymm3 lub m128/m256 oraz xmm2/ymm2 według bajtu sterującego imm8, wynik zapisuje w xmm1/ymm1. Kolejne bity imm8 odpowiadają kolejnym 32/64 bitom rejestrów/pamięci i pełnią zadanie przełącznika.

i <0, 7> dla PS lub i <0, 3> dla PD

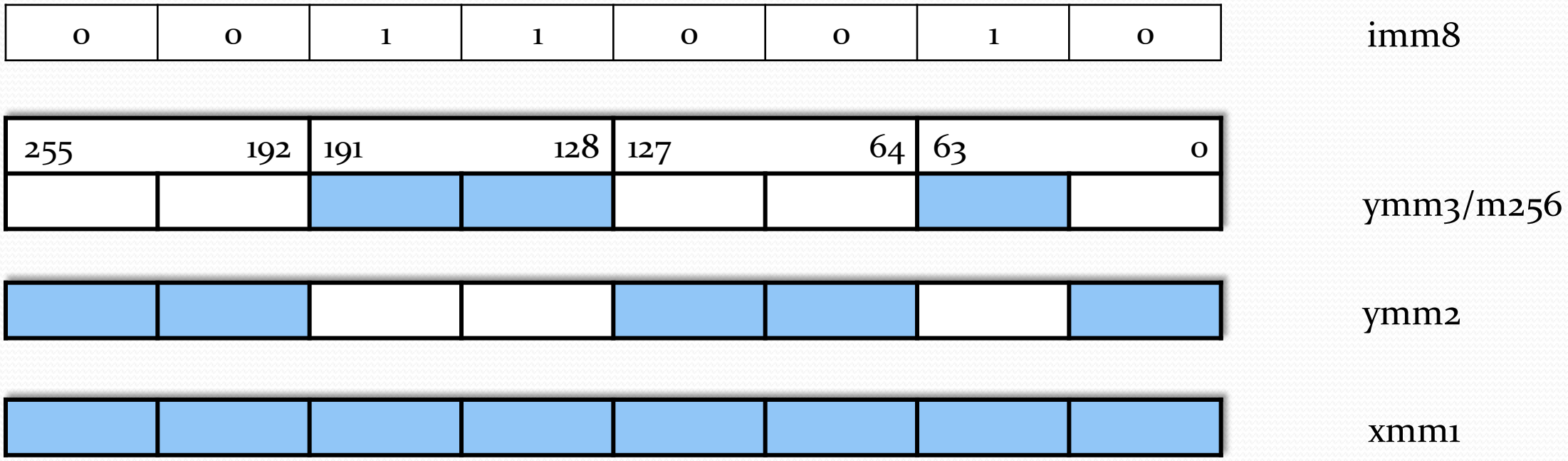
if imm8[i] = 0 then xmm1/ymm1[i] = xmm2/ymm2[i]

else xmm1/ymm1[i] = xmm3/ymm3[i] lub m128/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja mieszająca

VBLENDPS



Instrukcja mieszająca

VBLENDVP[S/D]

vblendvp[s/d] xmm1, xmm2, xmm3/m128, xmm4

vblendvp[s/d] ymm1, ymm2, ymm3/m256, ymm4

Warunkowo kopiuje elementy wektorów liczb rzeczywistych pojedynczej/podwójnej precyzji, z xmm3/ymm3 lub m128/m256 oraz xmm2/ymm2 według maski rejestru xmm4/ymm4, wynik zapisuje w xmm1/ymm1. Maską są odpowiadające poszczególnym wektorom bity znaku xmm4/ymm4.

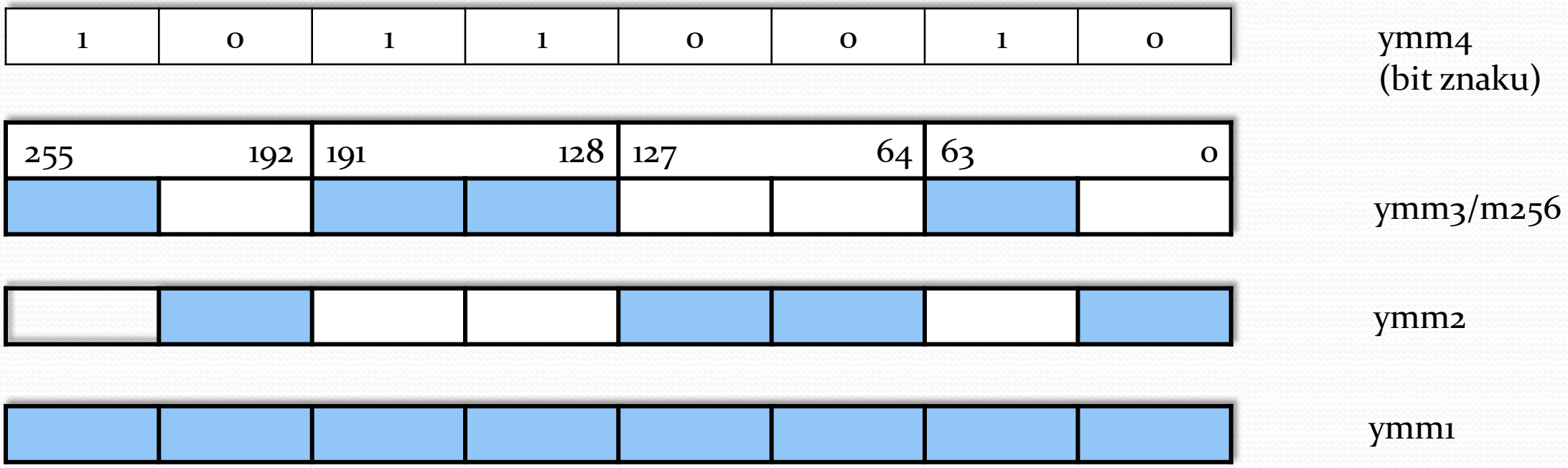
```
if xmm4/ymm4[i][31/63] = 0
        then xmm1/ymm1[i] = xmm2/ymm2[i]
else
        xmm1/ymm1[i] = xmm3/ymm3[i] lub m128/m256[i]
```

Bity od 128/256 do MSB są zerowane.

Instrukcja mieszająca

VBLENDVPS

Maska w rejestrze ymm4



Instrukcje rozgłaszania

Instrukcja rozgłaszania

VBROADCASTS[S/D] / VBROADCASTF128

vbroadcastss xmm1, m32/xmm2

vbroadcastss ymm1, m32/xmm2

Przesyła liczbę rzeczywistą pojedynczej precyzji z pamięci m32 lub najmłodszą z rejestru xmm2 do całego rejestru celu xmm1/ymm1.

vbroadcastsd ymm1, m64

vbroadcastsd ymm1, xmm2

Przesyła liczbę rzeczywistą podwójnej precyzji z pamięci m64 lub najmłodszą z rejestru xmm2 do całego rejestru celu xmm1/ymm1.

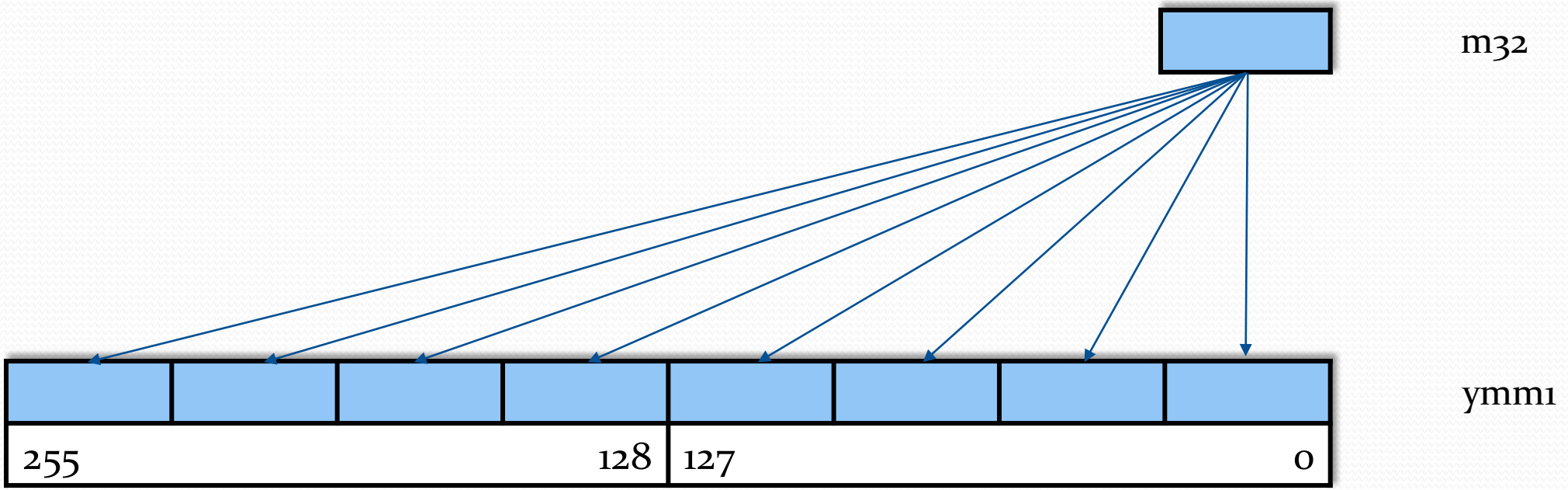
vbroadcastf128 ymm1, m128

Przesyła zawartość pamięci m128 do całego rejestru celu ymm1.

Bity od 128/256 do MSB są zerowane.

Instrukcja rozgłaszania

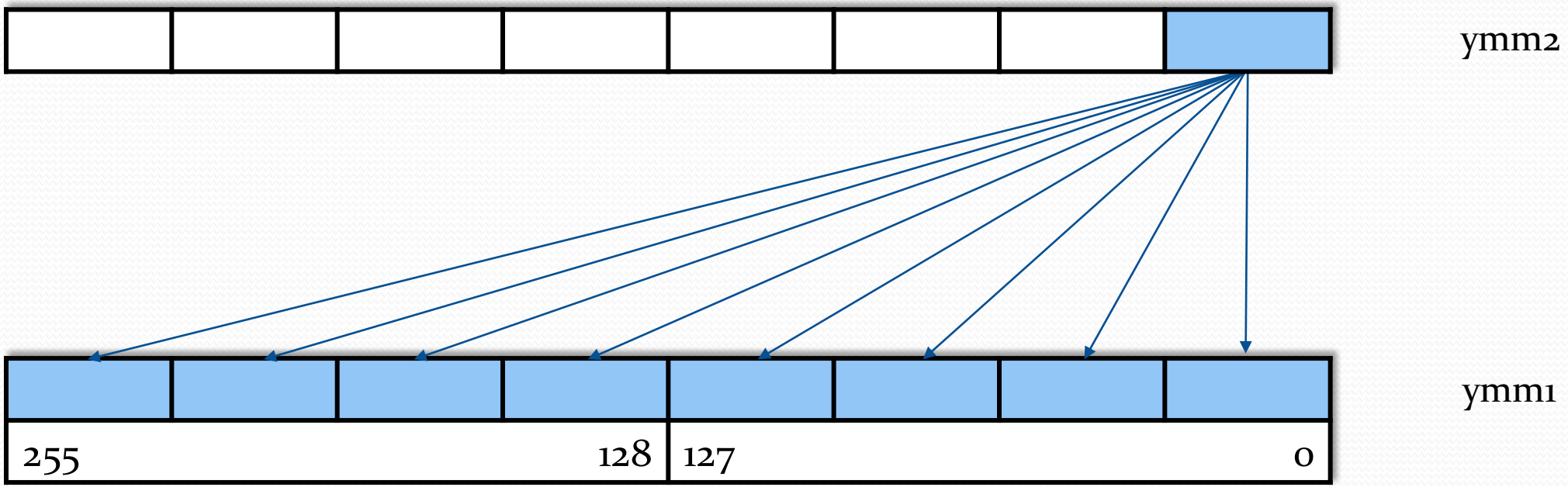
VBRODCASTSS



Bity od 128/256 do MSB są zerowane.

Instrukcja rozgłaszania

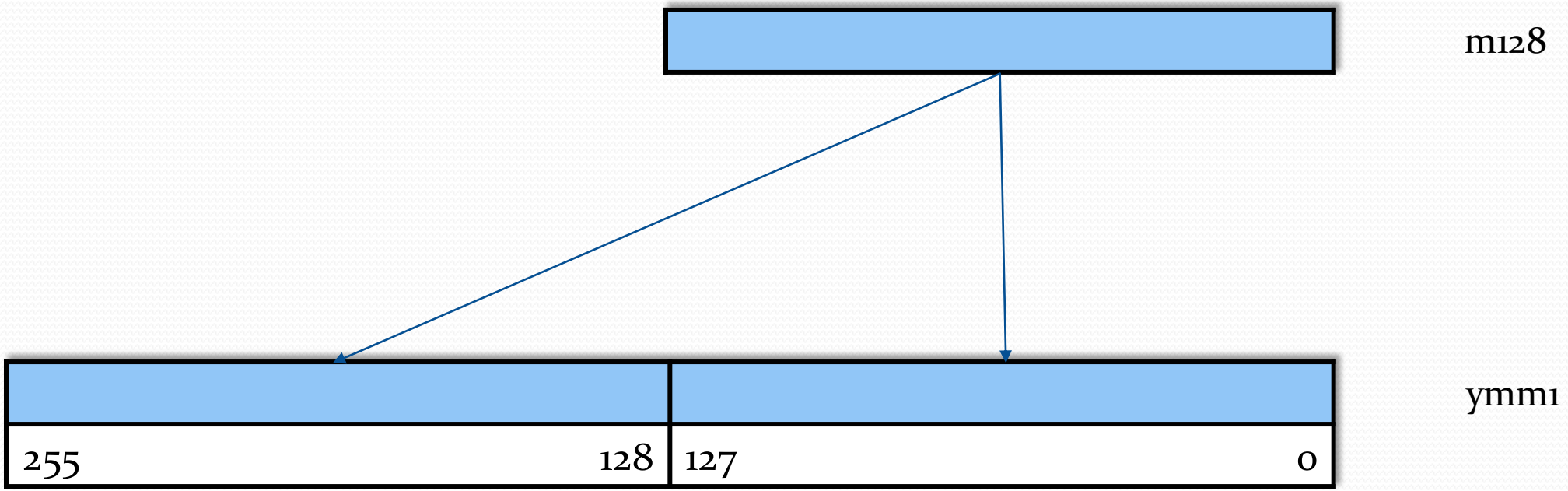
VBRODCASTSS



Bity od 128/256 do MSB są zerowane.

Instrukcja rozgłaszania

VBRODCASTF128



Bity od 128/256 do MSB są zerowane.

Instrukcje zbierania

Instrukcja zbierania

VGATHER[D/Q]P[S/D]

Dotyczy typów danych

Dotyczy adresów

vgather[d/q]p[s/d] xmm1, vm[32/64]x, xmm3 (AVX2)
vgather[d/q]p[s/d] ymm1, vm[32/64]y, ymm3 (AVX2)

Instrukcja kompletuje wektor xmm1/ymm1 używając **adresów w postaci** podwójnych/poczwórnych słów zdefiniowanych w vm32[x/y]/vm64[x/y] używając jako indeksów podwójnych/poczwórnych słów zapisanych w xmm2/ymm2 do wskazanej lokalizacji pamięci, skąd pobierane są liczby rzeczywiste pojedynczej/podwójnej precyzji.

Pobierane z pamięci wartości są zapisywane do rejestru celu xmm1/ymm1 tylko wówczas gdy najstarsze bity odpowiednich elementów wektora maski xmm3/ymm3 są równe 1.

Bity od 128/256 do MSB są zerowane.

Instrukcja zbierania

VGATHER[D/Q]P[S/D]

vgather[d/q]p[s/d] xmm1, vm[32/64]x, xmm3 (AVX2)
vgather[d/q]p[s/d] ymm1, vm[32/64]y, ymm3 (AVX2)

adres_fizyczny[i] = adres_bazowy + index[i]*skalowanie + przesunięcie

adres_bazowy – adres danych, określa rejestr GPR ma zostać użyty

index[i] – i-ty element rejestru xmm2/ymm2 (z xmm2/ymm2 używane są jedynie indeksy)

skalowanie – określa rozmiar danych (1, 2, 4, 8)

przesunięcie – wartość w bajtach

Bity od 128/256 do MSB są zerowane.

Instrukcja zbierania

VGATHER[D/Q]P[S/D]

vgather[d/q]p[s/d] xmm1, vm[32/64]x, xmm3 (AVX2)
vgather[d/q]p[s/d] ymm1, vm[32/64]y, ymm3 (AVX2)

Adresowanie cd.

W opisie instrukcji vm32x wskazuje wektor czterech 32-bitowych indeksów zapisanych w xmm2, vm32y wektor ośmiu 32- bitowych indeksów dla ymm2.

Notacja vm64x i vm64y wskazuje analogicznie na maksymalnie dwa lub cztery indeksy.

Bity od 128/256 do MSB są zerowane.

Instrukcja zbierania

VGATHER[D/Q]P[S/D]

vgather[d/q]p[s/d] xmm1, vm[32/64]x, xmm3 (AVX2)
vgather[d/q]p[s/d] ymm1, vm[32/64]y, ymm3 (AVX2)

Działanie instrukcji gather:

Pobiera z pamięci o wskazanej lokalizacji określonej tu jako adres_fizyczny liczby pojedynczej/podwójnej precyzji i zapisuje je do rejestru celu ymm1/xmm1 tylko wówczas gdy bit znaku odpowiadającego elementu maski ymm3/xmm3 jest równy jeden, jeśli bit znaku jest równy zero w rejestrze celu zostaje wartość poprzednia. Po wykonaniu operacji pobierania z pamięci elementy maski są zerowane.

if xmm3[i][63/31] then xmm1[i] ← [adres_fizyczny(xmm2[i])]

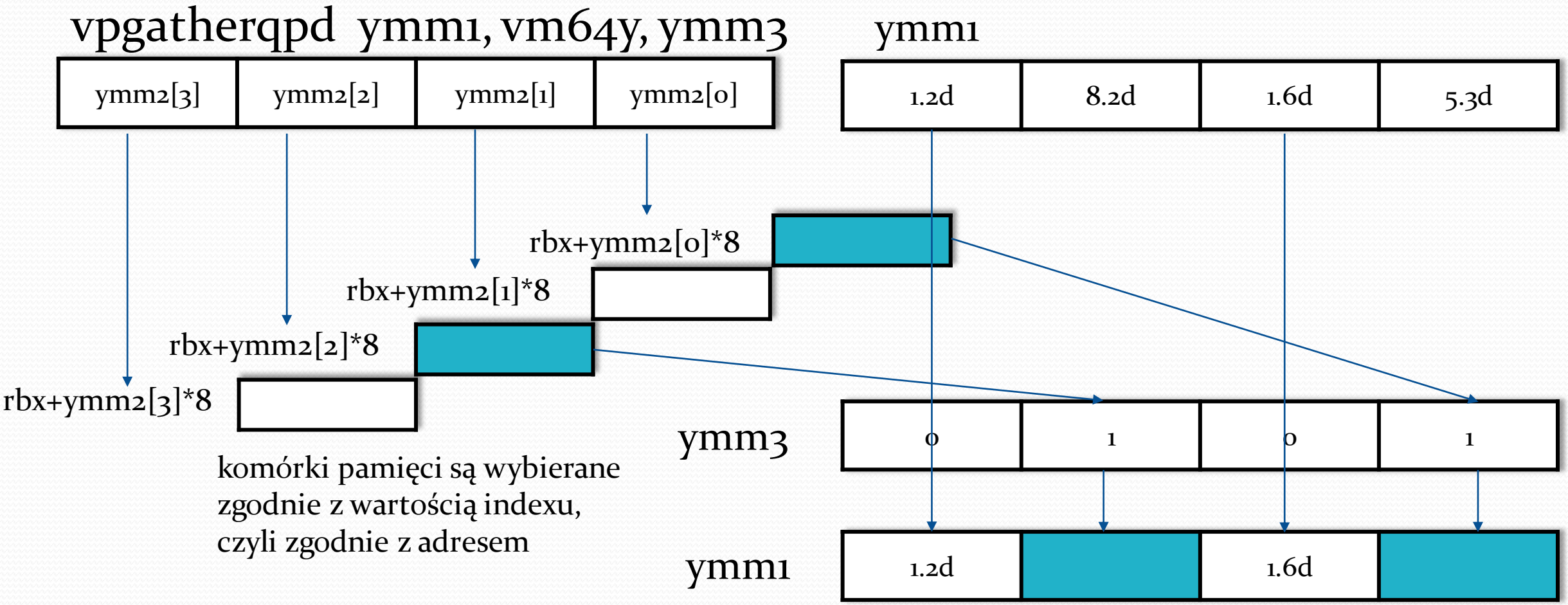
if ymm3[i][63/31] then ymm1[i] ← [adres_fizyczny(ymm2[i])]

Bity od 128/256 do MSB są zerowane.

Instrukcja zbierania (przykład) AVX2 VPGATHERQPD

vpgatherqpd ymm1, [rbx+ymm2*8], ymm3

vm64y = [rbx+ymm2*8]



Instrukcje permutacji

Instrukcja permutacji

VPERMP[S/D]

vpermps ymm1, ymm2, ymm3/m256

Wybiera liczby rzeczywiste pojedynczej precyzji z ymm3/m256 według wskazań ymm2 (trzy najmłodsze bity każdego elementu wektora stanowią indeks), wynik zapisuje w ymm1.

ymm1[i] = ymm3/m256[ymm2[i][2:0]]

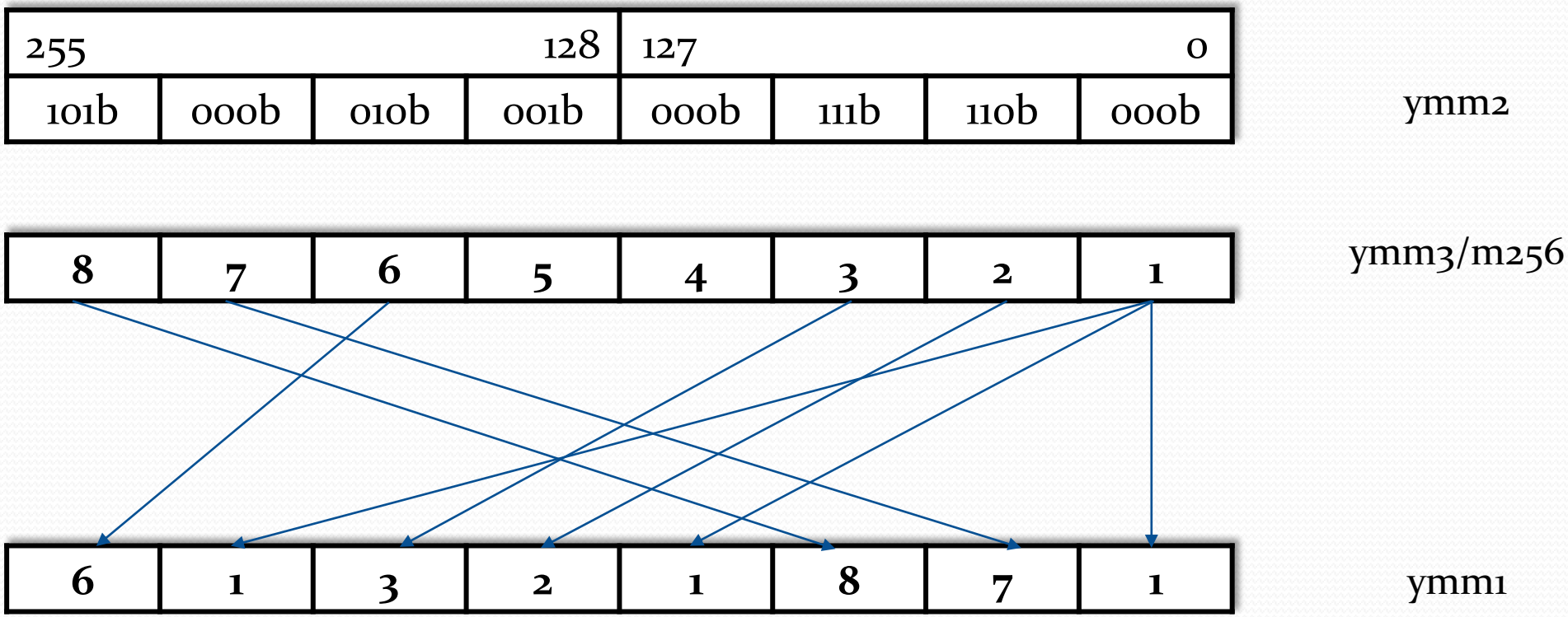
vpermpd ymm1, ymm2/m256, imm8

Wybiera liczby rzeczywiste podwójnej precyzji z ymm2/m256 według bajtu sterującego imm8, (kolejne dwa bity), wynik zapisuje w ymm1.

ymm1[i] = ymm2/m256[imm8[2i+1:2i]]

Bity od 128/256 do MSB są zerowane.

Instrukcja permutacji VPERMPS

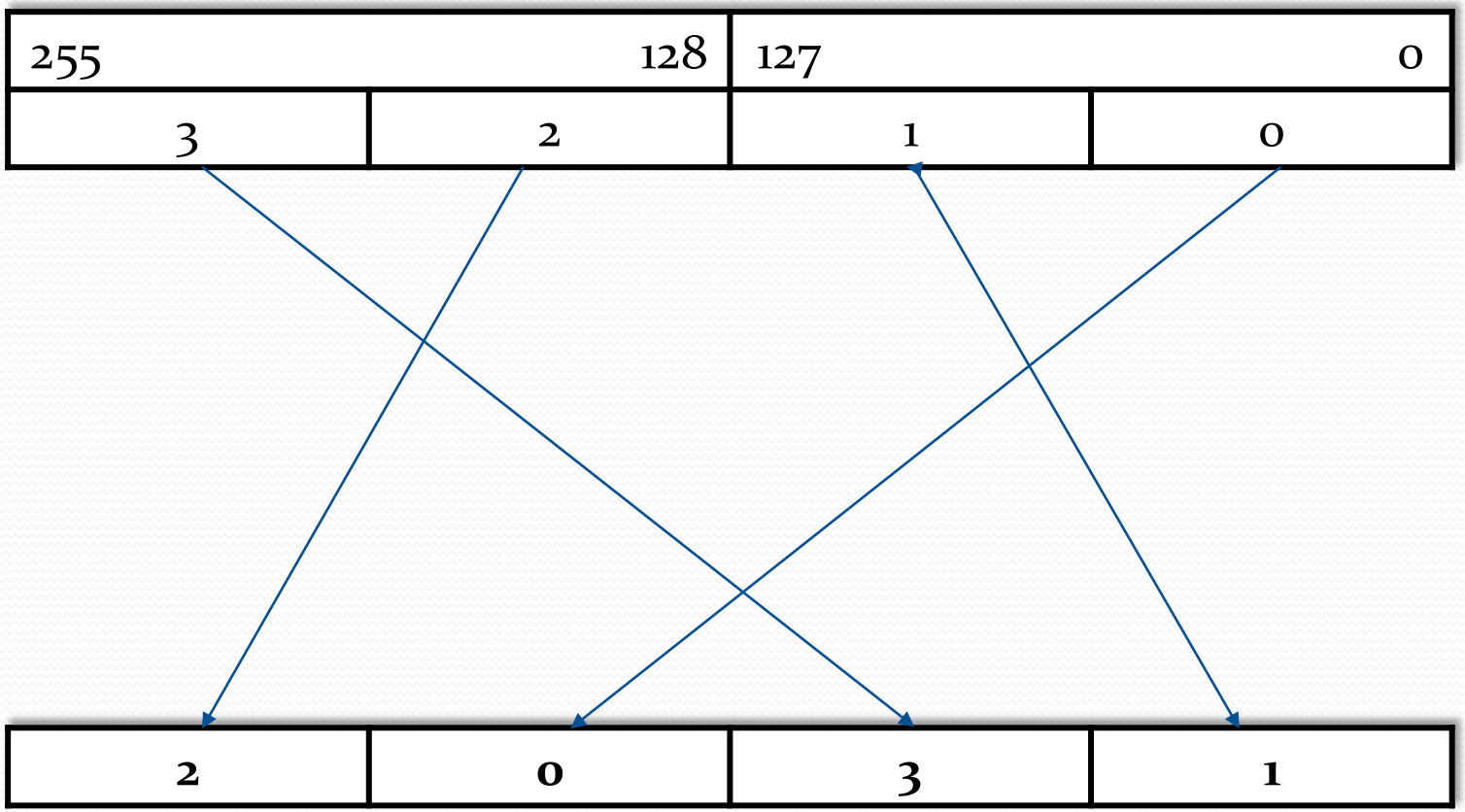


Bity od 128/256 do MSB są zerowane.

Instrukcja permutacji VPERMPD

np. imm8 = 10 00 11 01

Indeks dla kolejnych elementów wektora licząc od elementu zerowego



ymm2

ymm1

Przykład dla imm8 = 10001101 zamienia wektor 0123 na wektor 1302

Instrukcja permutacji

VPERMILPS

vpermilps xmm1, xmm2, xmm3/m128
vpermilps xmm1, xmm2/m128, imm8
vpermilps ymm1, ymm2, ymm3/m256
vpermilps ymm1, ymm2/m256, imm8

Wybiera liczby rzeczywiste pojedynczej precyzji z xmm2/ymm2 (/m128 /m256) według wskazania odpowiadających dwóch najmłodszych bitów xmm3/m128 lub odpowiednich bitów imm8, wynik zapisuje w xmm1/ymm1.

xmm1[i] = xmm2[xmm3/m128[i][1:0]] lub xmm2[imm8[2i+1:2i]]
if i>3 ymm1[i] = ymm2[ymm3/m256[i][1:0]] lub ymm2[4+imm8[2i-7:2i-8]]

Bity od 128/256 do MSB są zerowane.

Instrukcja permutacji

VPERMILPD

vpermilpd xmm1, xmm2, xmm3/m128
vpermilpd xmm1, xmm2/m128, imm8
vpermilpd ymm1, ymm2, ymm3/m256
vpermilpd ymm1, ymm2/m256, imm8

Wybiera liczby rzeczywiste podwójnej precyzji z xmm2/ymm2 (/m128 /m256) według wskazania każdego **drugiego bitu(o indeksie 1)** z xmm3/m128, wynik zapisuje w xmm1/ymm1.

xmm1[i] = xmm2[xmm3/m128[i][1]] lub xmm2[imm8[i]]
if i>1 ymm1[i] = xmm2[2+ymm3/m256[i][1]] lub ymm2[2+imm8[i]]

Bity od 128/256 do MSB są zerowane.

Instrukcja permutacji

VPERM2F128

vperm2f128 ymm1, ymm2, ymm3/m256, imm8

Wybiera odpowiednio 128 bitów liczb rzeczywistych z ymm2 oraz ymm3/m256 według wskazania bajtu sterującego imm8, wynik zapisuje w xmm1/ymm1.

```
if imm8[1:0] = 0 then ymm1[127:0] = ymm2[127:0]
if imm8[1:0] = 1  then ymm1[127:0] = ymm2[255:128]
if imm8[1:0] = 2 then ymm1[127:0] = ymm3/m256[127:0]
if imm8[1:0] = 3 then ymm1[127:0] = ymm3/m256[255:128]

if imm8[5:4] = 0 then ymm1[255:128] = ymm2[127:0]
if imm8[5:4] = 1  then ymm1[255:128] = ymm2[255:128]
if imm8[5:4] = 2 then ymm1[255:128] = ymm3/m256[127:0]
if imm8[5:4] = 3 then ymm1[255:128] = ymm3/m256[255:128]
```

Bity od 128/256 do MSB są zerowane.

imm8[3] => ymm1[127:0] ← 0
imm8[7] => ymm1[255:128] ← 0

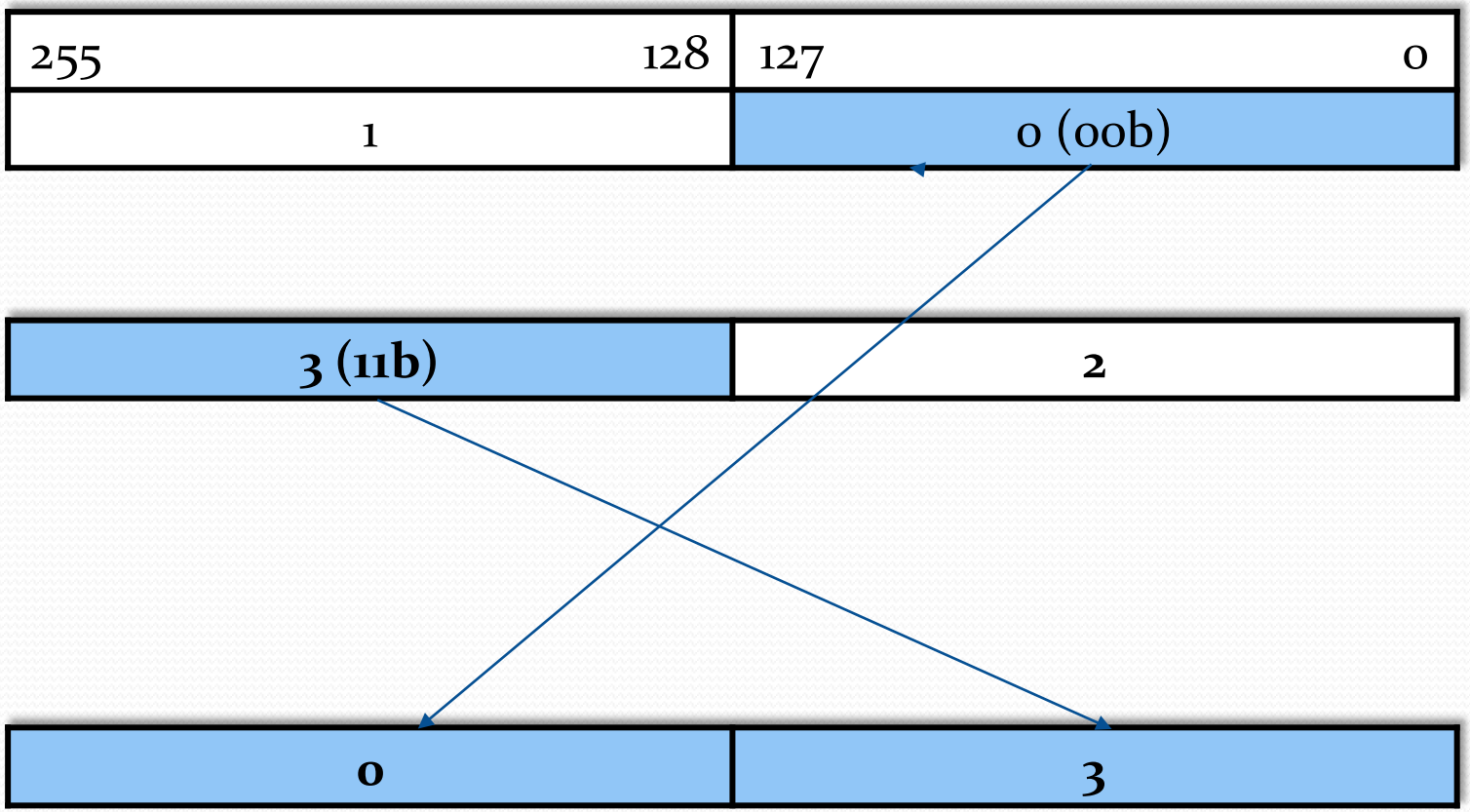
Instrukcja permutacji VPERM2F128

Bit zerowania

Bit zerowania

np. imm8 = **0** 0 **00** **0** 0 **11**

Wartość odpowiada elementowi źródła (3) a usytuowanie (0) elementowi celu



Instrukcje tasowania

Instrukcja tasowania

VSHUFPS

vshufps xmm1, xmm2, xmm3/m256, imm8

vshufps ymm1, ymm2, ymm3/m256, imm8

Wybiera liczby rzeczywiste pojedynczej precyzji z xmm2/ymm2 oraz xmm3/ymm3 lub m128.m256 po dwie z każdego źródła według bajtu sterującego imm8 i zapisuje w xmm1/ymm1 po dwie wartości przeplatając źródła pochodzenia.

if i=(0,1,4,5) then s=2 else s=3

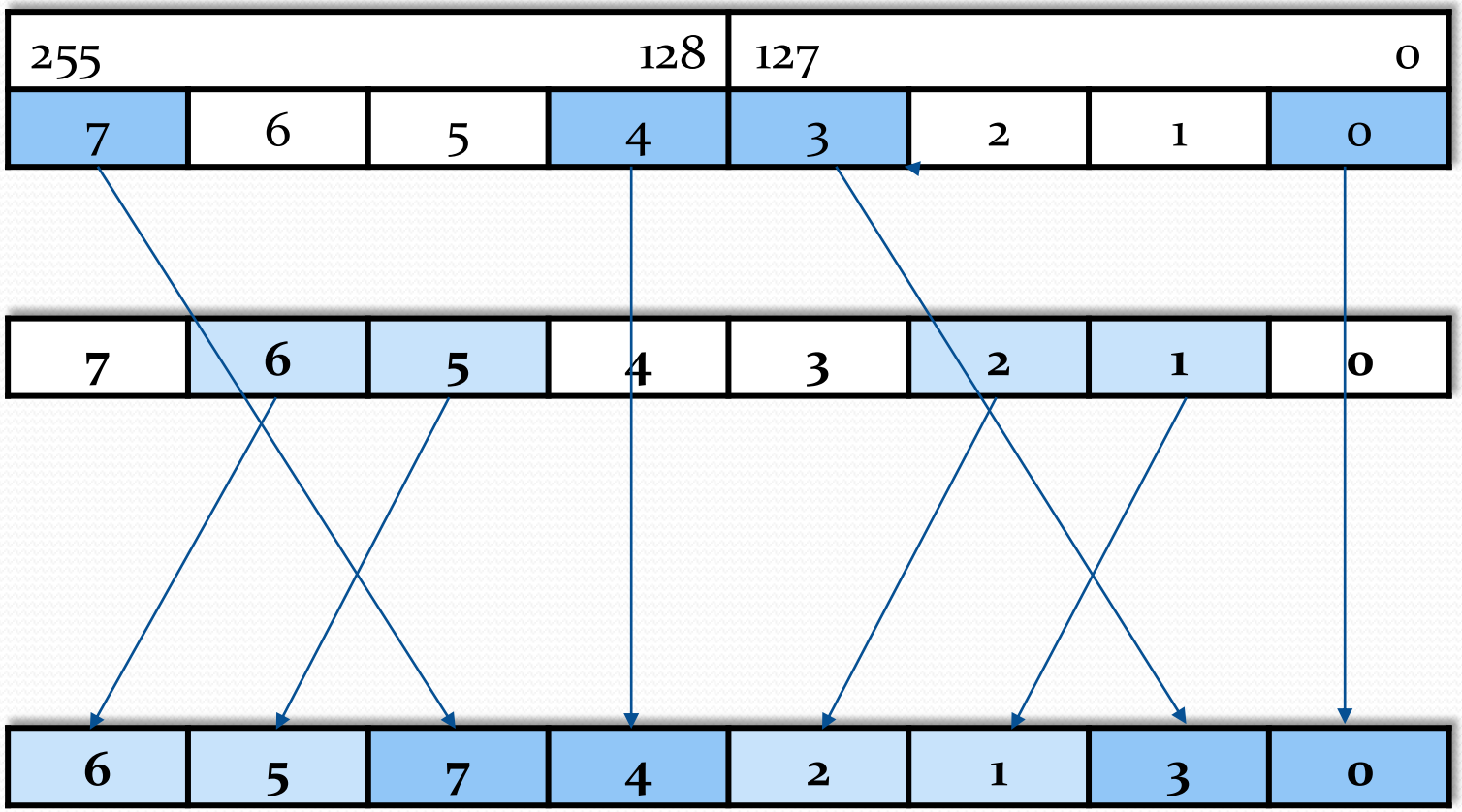
xmm[i] = xmms[imm8[2i+1:2i]] i=(0..3)

ymm[i] = ymms[imm8[2(i-4)+1:2(i-4)]] i=(4..7)

Bity od 128/256 do MSB są zerowane.

Instrukcja tasowania VSHUFPS

np. imm8 = 10 **01 11 00**



Bajt koduje połowę rejestru, drugą powiela według tej samej sekwencji.

Wartości [3:2],[1:0] określają elementy źródła 1, wartości [7:6],[5:4] określają elementy źródła 2, usytuowanie dwóch bitów determinuje element celu.

Instrukcja tasowania

VSHUFPD

vshufpd ymm1, ymm2, ymm3/m256, imm8

Tasuje liczby rzeczywiste podwójnej precyzji z xmm2/ymm2 oraz xmm3/ymm3 lub m128 według bajtu sterującego imm8. Wynik zapisuje w xmm1/ymm1 przeplatając źródła pochodzenia.

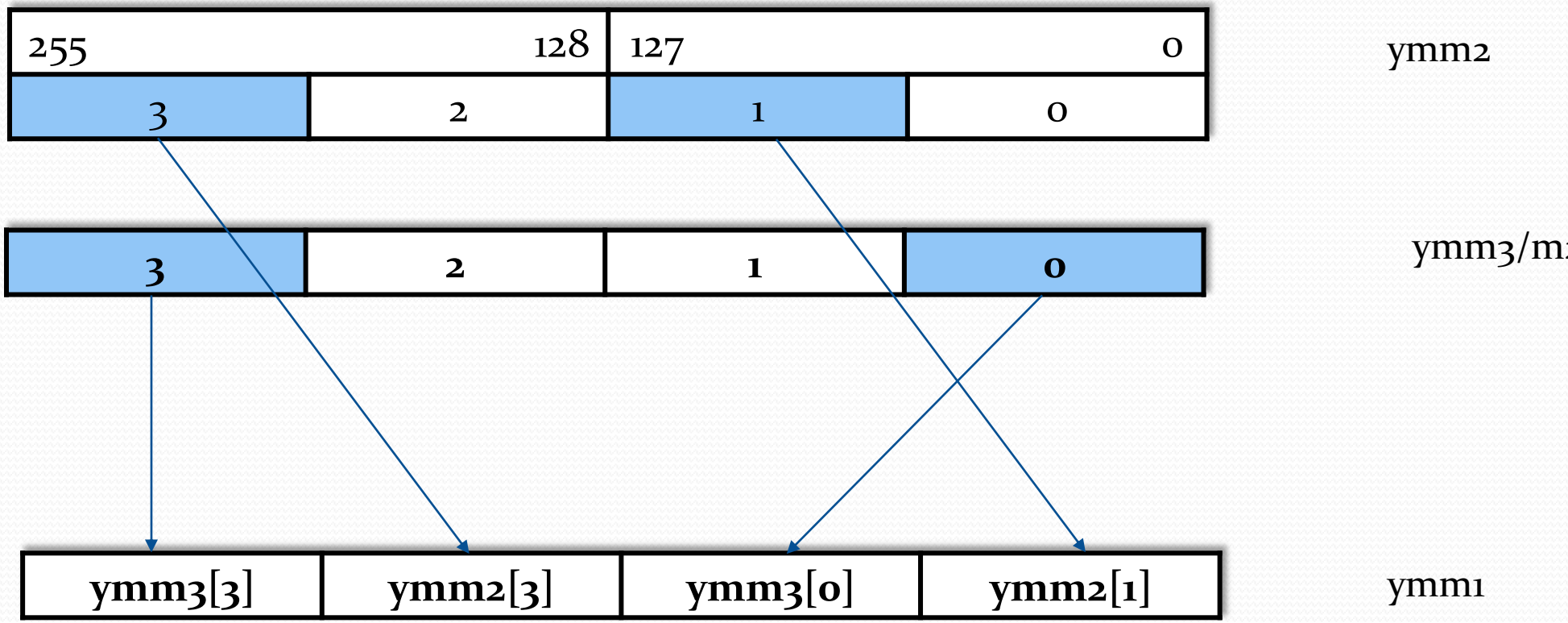
```
xmm1[0] = xmm2[imm8[0]]
xmm1[1] = xmm3[imm8[1]]
ymm1[2] = ymm2[2+imm8[2]]
ymm1[3] = ymm3[2+imm8[3]]
```

Bity od 128/256 do MSB są zerowane.

Instrukcja tasowania

VSHUFPD

np. imm8 = 0000 1 1 0 1



Operacje arytmetyczne AVX

Operacje arytmetyczne AVX

- Instrukcje dodawania: VADDS[S/D], VADDP[S/D], VHADDP[S/D]
- Instrukcje odejmowania: VSUBS[S/D], VSUBP[S/D], VHSUBP[S/D]
- Instrukcje dodawania i odejmowania: VADDSUBP[S/D]
- Instrukcje mnożenia: VMULS[S/D], VMULP[S/D]
- Instrukcje mnożenia z sekwencyjnym dodawaniem: VDPP[S/D]
- Instrukcje dzielenia: VDIVS[S/D], VDIVP[S/D]

Instrukcje dodawania

Instrukcja dodawania
VADDS[S/D], VADDP[S/D]

vadds[s/d] xmm1, xmm2, xmm3/m32/m64

vaddp[s/d] xmm1, xmm2, xmm3/m128

vaddp[s/d] ymm1, ymm2, ymm3/m256

Dodaje skalary/wektory liczb rzeczywistych pojedynczej/podwójnej precyzji z rejestru xmm2/ymm2 i xmm3/ymm3 lub m32/m64/m128/m256, wynik zapisuje w xmm1/ymm1. Dla skalarów pozostałe elementy są przepisywane ze źródła 1.

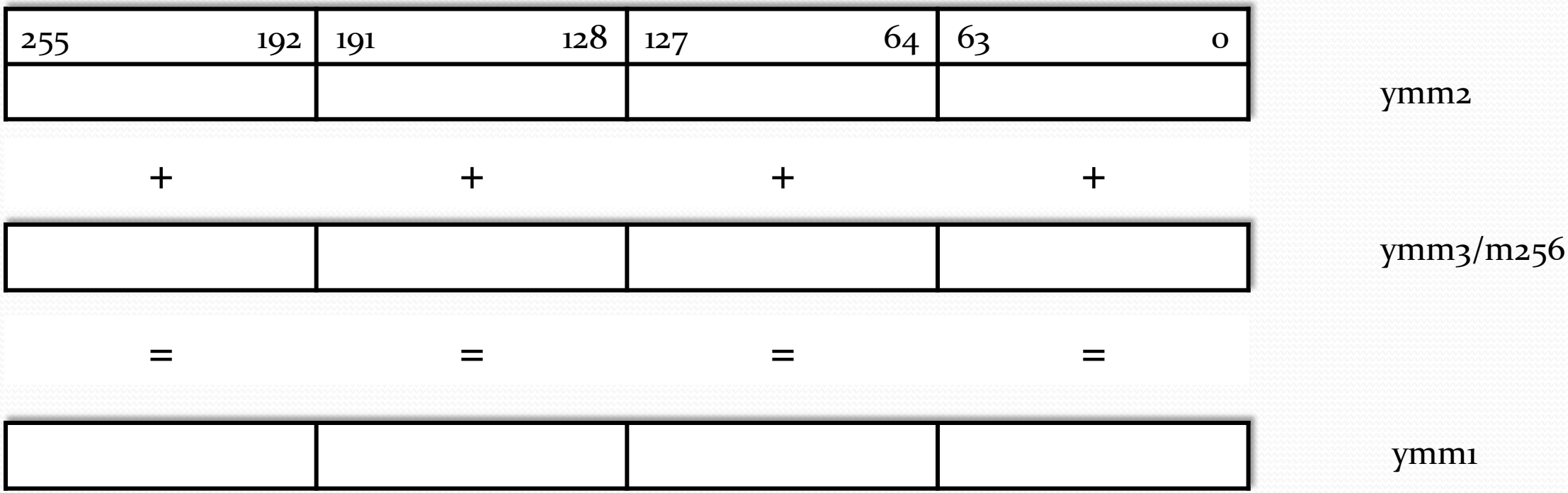
xmm1[i] = xmm2[i] + xmm3/m128[i]

ymm1[i] = ymm2[i] + ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania

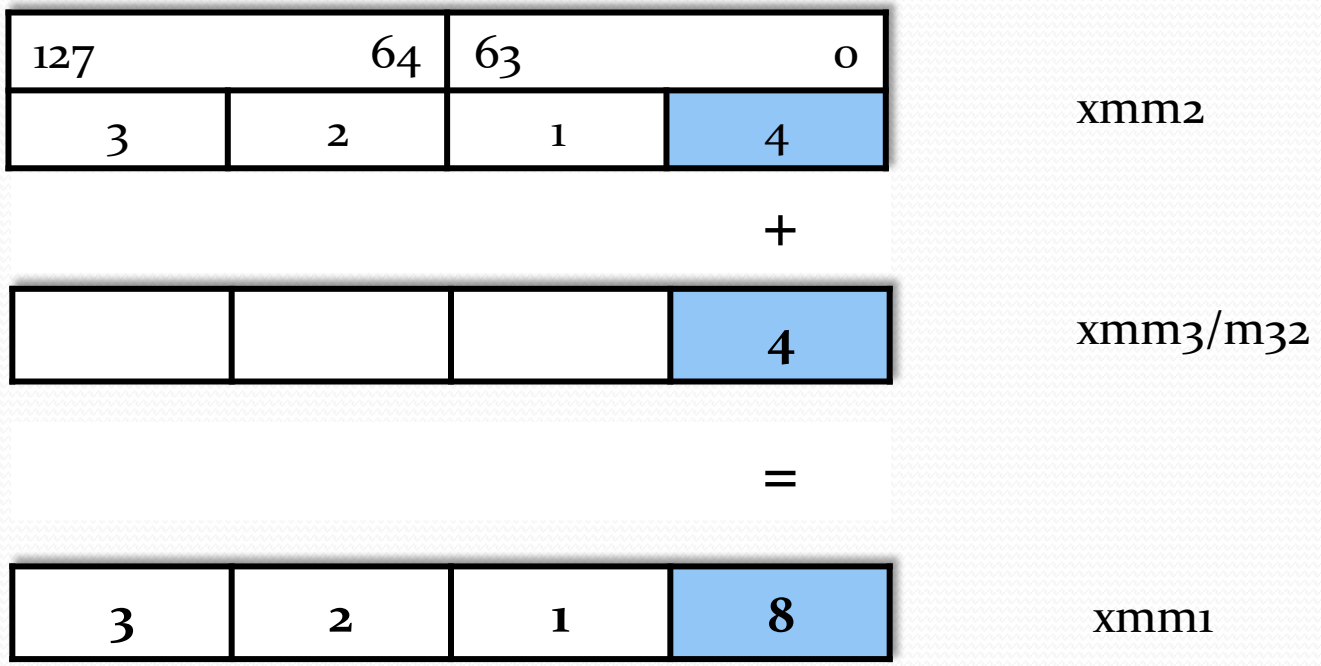
VADDPD



Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania

VADDSS



Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania

VHADDPS

vhaddps xmm1, xmm2, xmm3/m128

vhaddps ymm1, ymm2, ymm3/m256

Horyzontalne dodawanie sąsiednich liczb rzeczywistych pojedynczej precyzji i zapisywanie wyniku z przeplotem co 64 bity.

ymm1/xmm1[31:0] = ymm2/xmm2[63:32] + ymm2/xmm2[31:0]
ymm1/xmm1[63:32] = ymm2/xmm2[127:96] + ymm2/xmm2[95:64]
ymm1/xmm1[95:64] = ymm3/xmm3[63:32] + ymm3/xmm3[31:0]
ymm1/xmm1[127:96] = ymm3/xmm3[127:96] + ymm3/xmm3[95:64]

ymm1[159:128] = ymm2[191:160] + ymm2[159:128]
ymm1[191:160] = ymm2[255:224] + ymm2[223:192]
ymm1[223:192] = ymm3[191:160] + ymm3[159:128]
ymm1[255:224] = ymm3[255:224] + ymm3[223:192]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania

VHADDPD

vhaddpd xmm1, xmm2, xmm3/m128

vhaddpd ymm1, ymm2, ymm3/m256

Horyzontalne dodawanie sąsiednich liczb rzeczywistych podwójnej precyzji i zapisywanie wyniku z Przeplotem co 64 bity.

xmm1[0] = xmm2[1] + xmm2[0]

xmm1[1] = xmm3[1] + xmm3[0]

ymm1[2] = ymm2[3] + ymm2[2]

ymm1[3] = ymm3[3] + ymm 3[2]

Bity od 128/256 do MSB są zerowane.

Instrukcje odejmowania

Instrukcja odejmowania

VSUBS[S/D], VSUBP[S/D]

vsubs[s/d] xmm1, xmm2, xmm3/m32/m64

vsubp[s/d] xmm1, xmm2, xmm3/m128

vsubp[s/d] ymm1, ymm2, ymm3/m256

Od zawartości rejestru xmm2/ymm2 **odejmuje** liczby rzeczywiste pojedynczej/podwójnej precyzji odpowiednio z xmm3/ymm3 lub m32/m64/m128/m256, wynik zapisuje w xmm1/ymm1. Dla skalarów pozostałe elementy są przepisywane ze źródła 1.

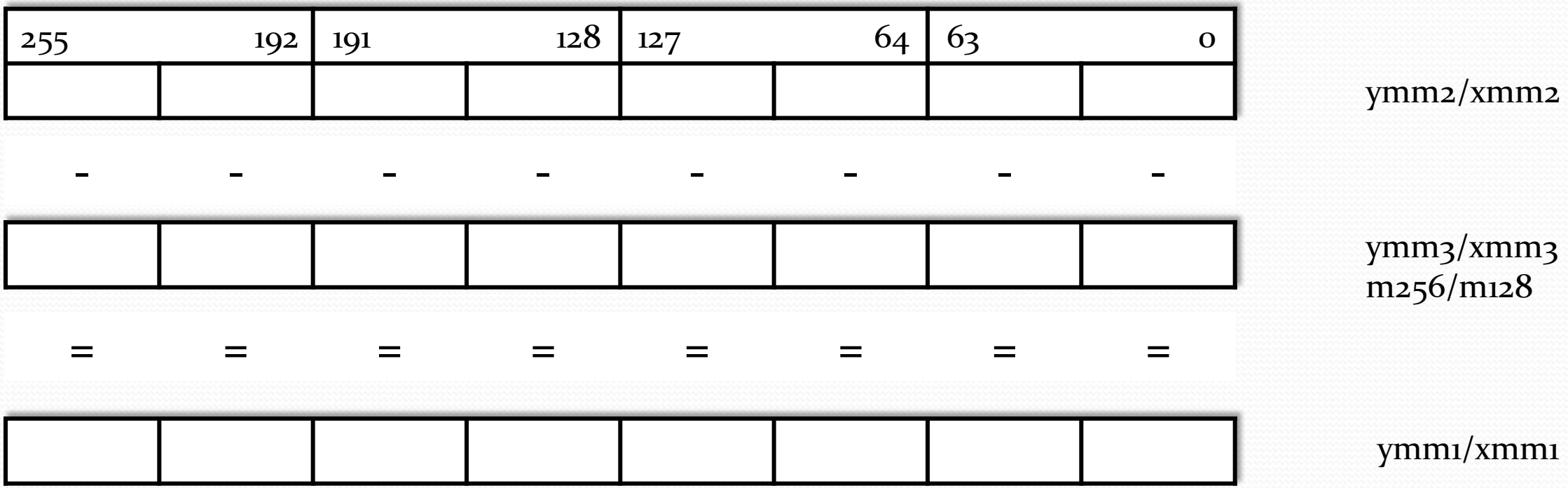
xmm1[i] = xmm2[i] – xmm3/m128[i]

ymm1[i] = ymm2[i] – ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowania

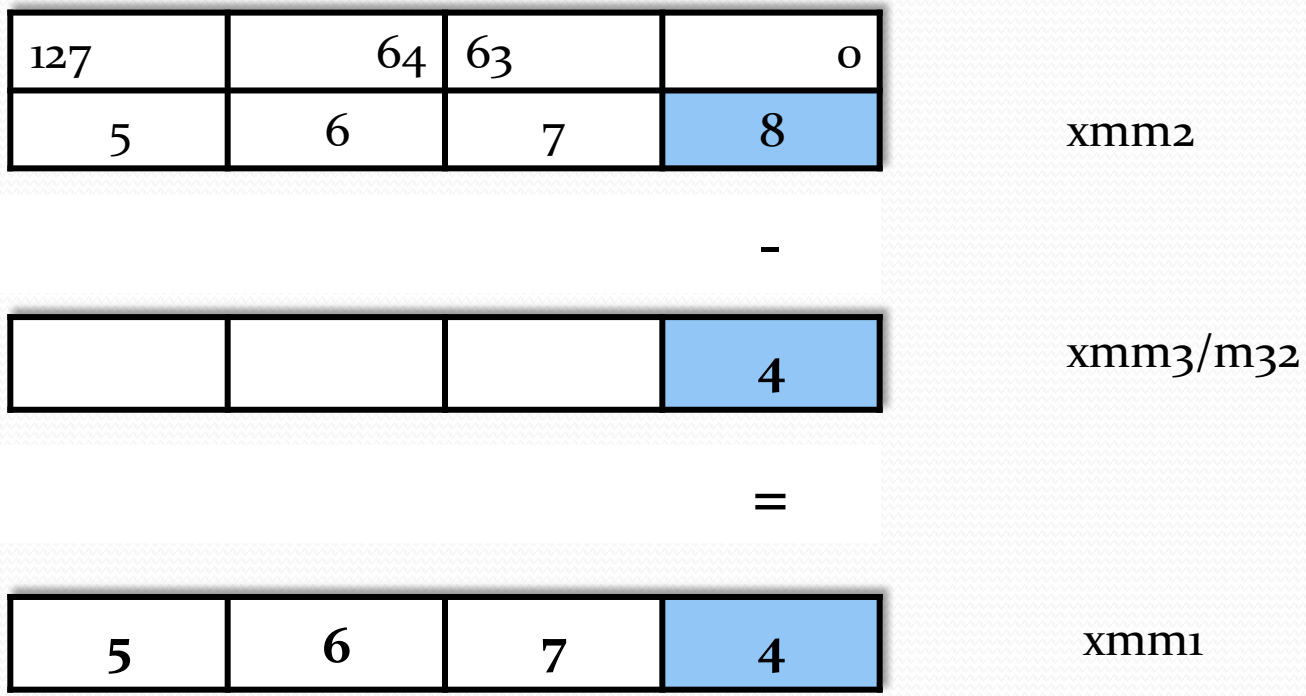
VSUBPS



Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowania

VSUBSS



Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowania horyzontalnego

VHSUBPS

vhsubps xmm1, xmm2, xmm3/m128

vhsubps ymm1, ymm2, ymm3/m256

Horyzontalne odejmowanie sąsiednich liczb rzeczywistych pojedynczej precyzji i zapisywanie wyniku z przeplotem co 64 bity.

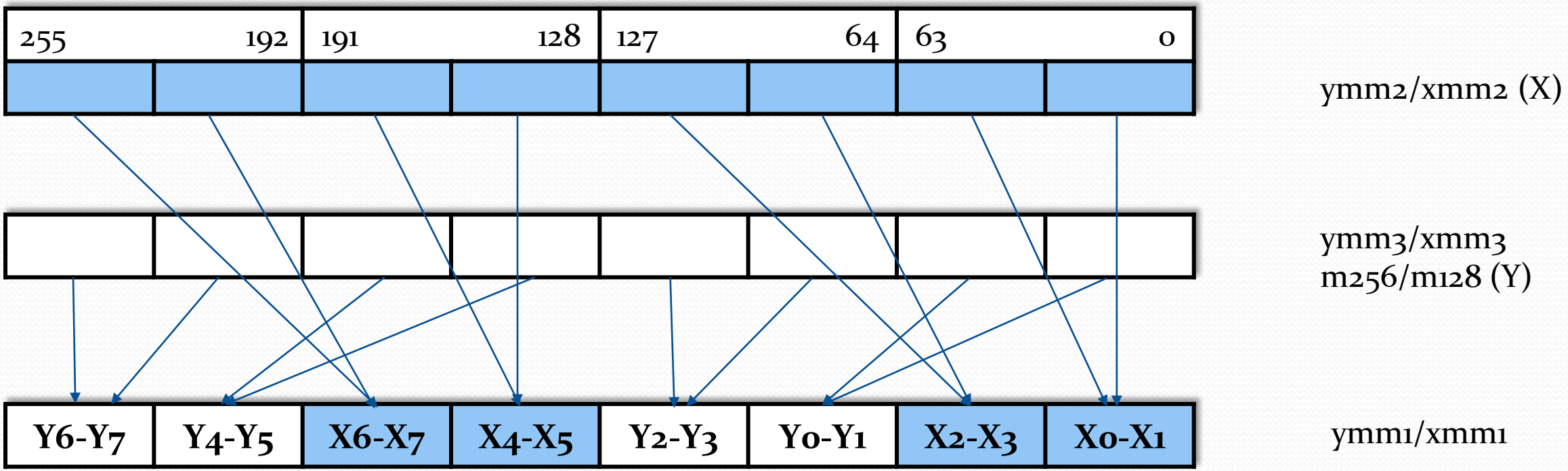
xmm1[0] = xmm2[0] - xmm2[1]
xmm1[1] = xmm2[2] - xmm2[3]
xmm1[2] = xmm3[0] - xmm3[1]
xmm1[3] = xmm3[2] - xmm3[3]

ymm1[4] = ymm2[4] - ymm2[5]
ymm1[5] = ymm2[6] - ymm2[7]
ymm1[6] = ymm3[4] - ymm3[5]
ymm1[7] = ymm3[6] - ymm3[7]

Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowania

VHSUBPS



Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowania horyzontalnego

VHSUBPD

vhsubpd xmm1, xmm2, xmm3/m128

vhsubpd ymm1, ymm2, ymm3/m256

Horyzontalne odejmuje sąsiednie liczby rzeczywiste podwójnej precyzji i zapisuje wynik z przeplotem z przeplotem co 64 bity.

xmm1[0] = xmm2[0] - xmm2[1]

xmm1[1] = xmm3[0] - xmm3[1]

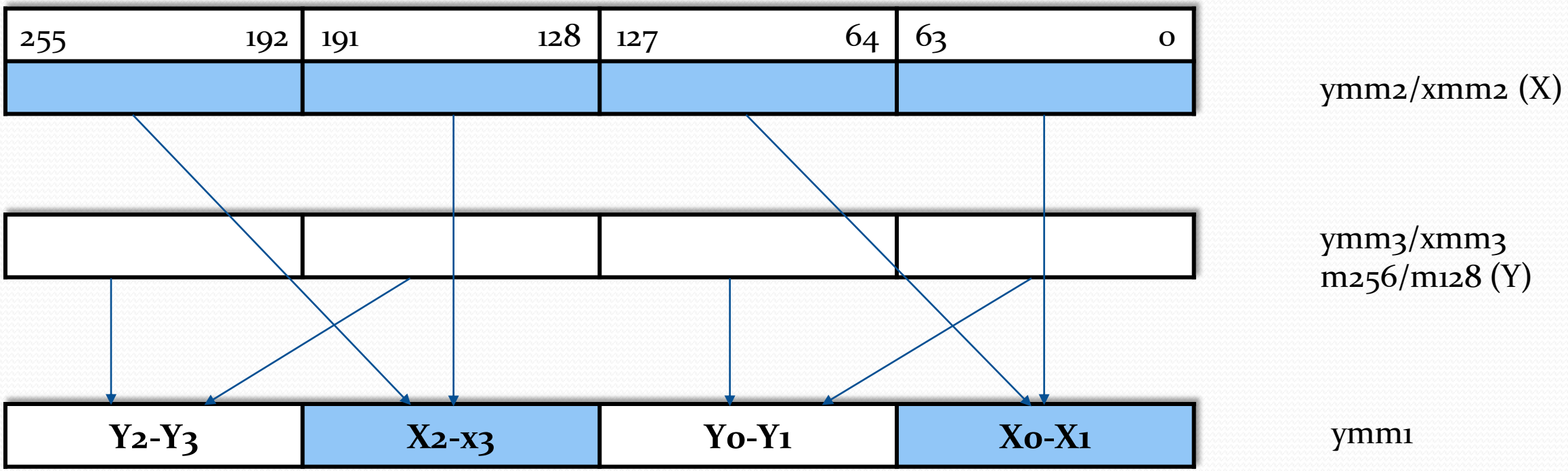
ymm1[2] = ymm2[2] - ymm2[3]

ymm1[3] = ymm3[2] - ymm3[3]

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania

VHSUBPD



Bity od 128/256 do MSB są zerowane.

Instrukcja odejmowanie macierzy – przykład:

```
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; mnożenie przez 4
petlaM:
sub ecx, 32; ps
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;
}
}
```

Instrukcje dodawania i odejmowania

Instrukcja dodawania i odejmowania

VADDSUBP[S/D]

vaddsubp[s/d] xmm1, xmm2, xmm3/m128

vaddsubp[s/d] ymm1, ymm2, ymm3/m256

Naprzemiennie odejmuje i dodaje wektory liczb rzeczywistych pojedynczej/podwójnej precyzji od/do zawartości rejestru xmm2/ymm2 **odejmuje** / **dodaje** odpowiadające wartości xmm3/ymm3 lub m128/m256, wynik zapisuje w xmm1/ymm1.

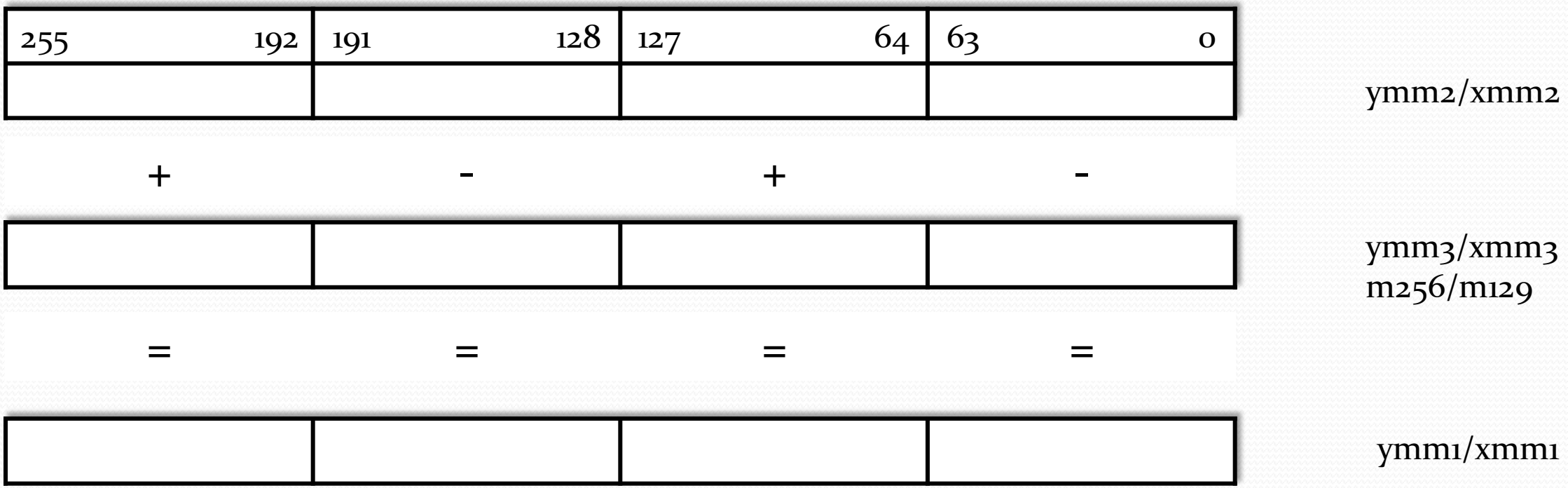
$$ymm1[2i] = ymm2[2i] - ymm3/m256[2i]$$

$$ymm1[2i+1] = ymm2[2i+1] + ymm3/m256[2i+1]$$

Bity od 128/256 do MSB są zerowane.

Instrukcja dodawania i odejmowania

VADDSUBPD



Bity od 128/256 do MSB są zerowane.

Instrukcje mnożenia

Instrukcja mnożenia

VMULS[S/D], VMULP[S/D]

vmuls[s/d] xmm1, xmm2, xmm3/m32/m64

vmulp[s/d] xmm1, xmm2, xmm3/m128

vmulp[s/d] ymm1, ymm2, ymm3/m256

Mnoży liczby rzeczywiste pojedynczej/podwójnej precyzji z rejestru xmm2/ymm2 odpowiednio przez xmm3/ymm3 lub m32/m64/m128/m256, wynik zapisuje w xmm1/ymm1. Dla skalarów pozostałe elementy pochodzą ze źródła 1.

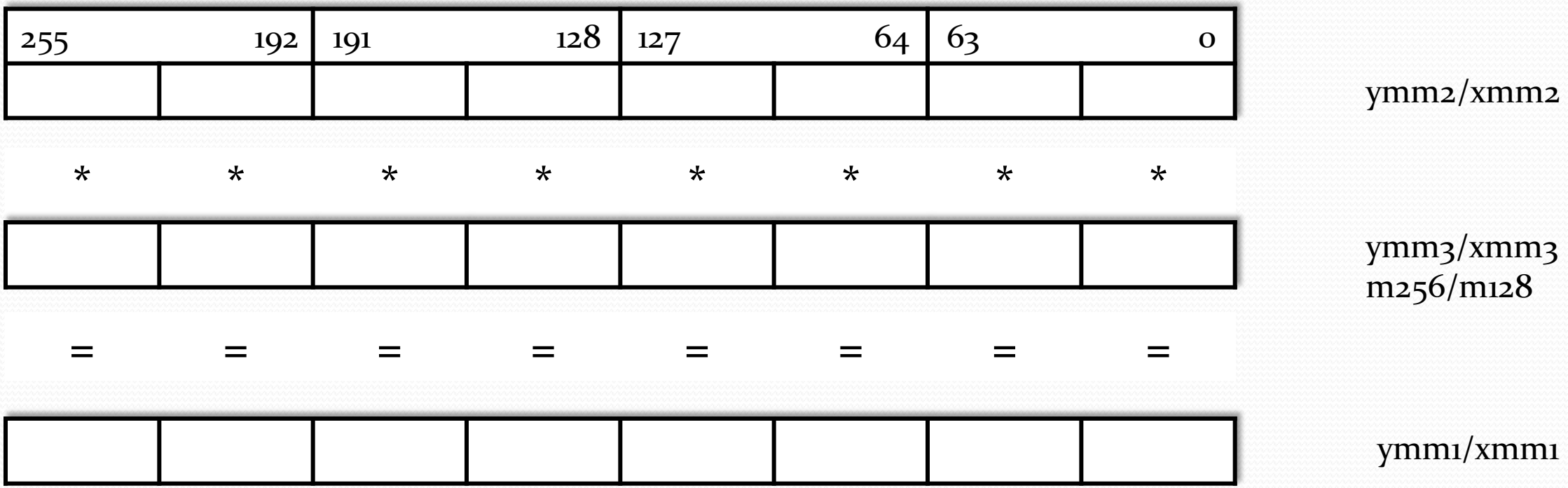
xmm1[i] = xmm2[i] * xmm3/m128[i]

ymm1[i] = ymm2[i] * ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja mnożenia

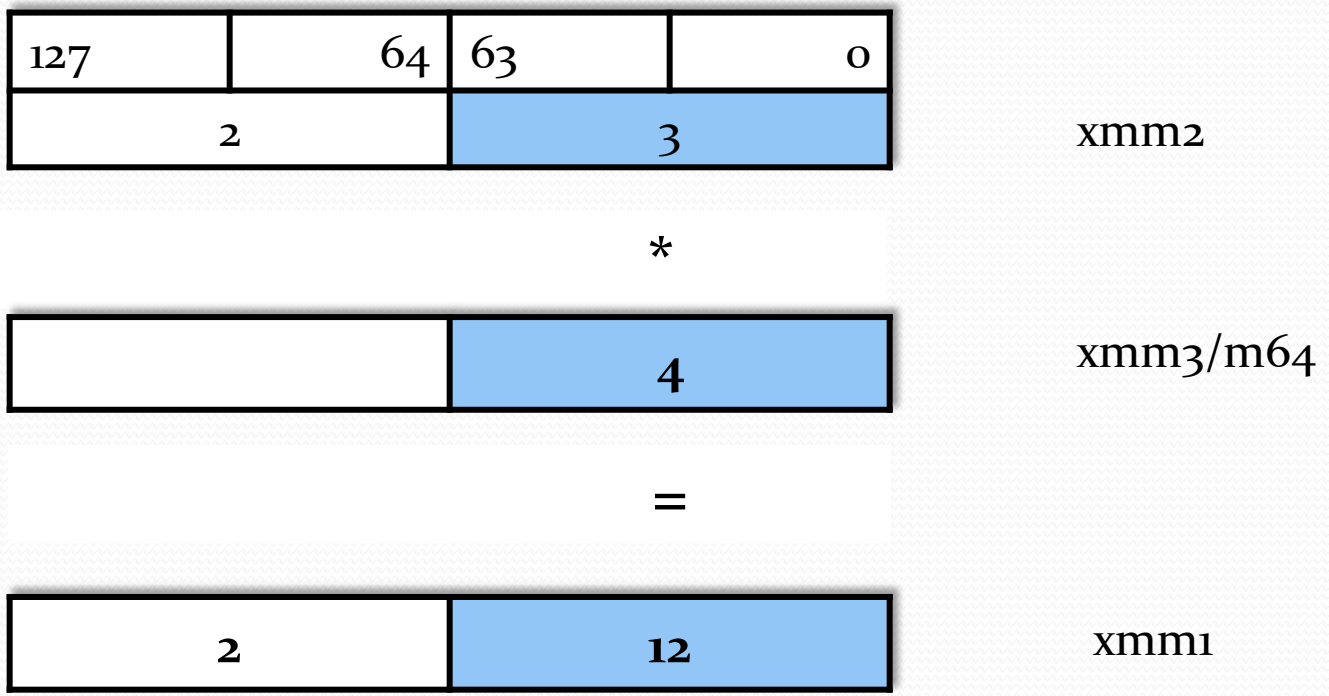
VMULPS



Bity od 128/256 do MSB są zerowane.

Instrukcja mnożenia

VMULSD



Bity od 128/256 do MSB są zerowane.

Instrukcja przykład: iloczyn skalarny

```
double iloczynVektor(double* tab1,
double *tab2, int n) {
double sum = 0;
double zerod = 0;
__asm {

push esi
push edi
vbroadcastsd ymm0, zerod;
mov ecx, n;
mov esi, tab1;
mov edi, tab2;
shl ecx, 3

p2:
sub ecx, 32
vmovupd ymm1, ymmword ptr[esi + ecx]
vmulpd ymm1, ymm1, ymmword ptr[edi + ecx]
vaddpd ymm0, ymm0, ymm1
jnz p2

vperm2f128 ymm1, ymm0, ymm0, 1; lo 1=hi 0
vaddpd ymm0, ymm0, ymm1
vpermilpd ymm1, ymm0, 1
vaddpd ymm0, ymm0, ymm1
vmovsd sum, xmm0;

pop edi
pop esi

} return sum; }
```

Instrukcja mnożenia , dodawania przykład:

```
.code
WIELOMIAN PROC public

push rbx
mov rax, rcx    ; wskaźnik na x w rcx
mov rbx, rdx    ; wskaźnik na wynik w rdx

movsxd r8, r8d ; movsxd r8, ilość w r8d

sub r8, 8 ; -1 bo zliczanie od 0
loop:
        vmovups ymm0, [rax+r8*4] ; x
        vmulps ymm1, ymm0, ymm0 ; x^2

        vmulps ymm2, ymm1, ymm0 ; x^3

        vaddps ymm0, ymm0, ymm1 ; x+x^2
        vaddps ymm0, ymm0, ymm2 ; x+x^2+x^3
        vmovups [rbx+r8*4], ymm0    ; wynik

        sub r8, 8 ; dekrementacja
        jge loop

        vzeroupper
        pop rbx

        ret
WIELOMIAN ENDP

end
```

Instrukcja mnożenia , dodawania przykład:

```
double wielomian_v1(double a, double b,
double c, double d, double x)
{
double wynik = 0;
__asm
{
vmovsd xmm0, a;
vmovsd xmm1, b;
vmovsd xmm2, c;
vmovsd xmm3, d;
vmovsd xmm4, x;
vmulsd xmm0, xmm0, xmm4     ; ax
vmulsd xmm2, xmm2, xmm4     ; cx
vmulsd xmm4, xmm4, xmm4     ; xx
vmulsd xmm0, xmm0, xmm4     ; axxx
vaddsd xmm0, xmm0, xmm3     ; axxx+d
vmulsd xmm1, xmm1, xmm4     ; bxx
vaddsd xmm0, xmm0, xmm2     ; axxx+cx+d
vaddsd xmm0, xmm0, xmm1     ; axxx+bxx+cx+d
vmovsd wynik, xmm0;

}
return wynik;
}
```

Instrukcja iloczyn skalarny

VDPPS

vdpps xmm1, xmm2, xmm3/m128, imm8

vdpps ymm1, ymm2, ymm3/m256, imm8

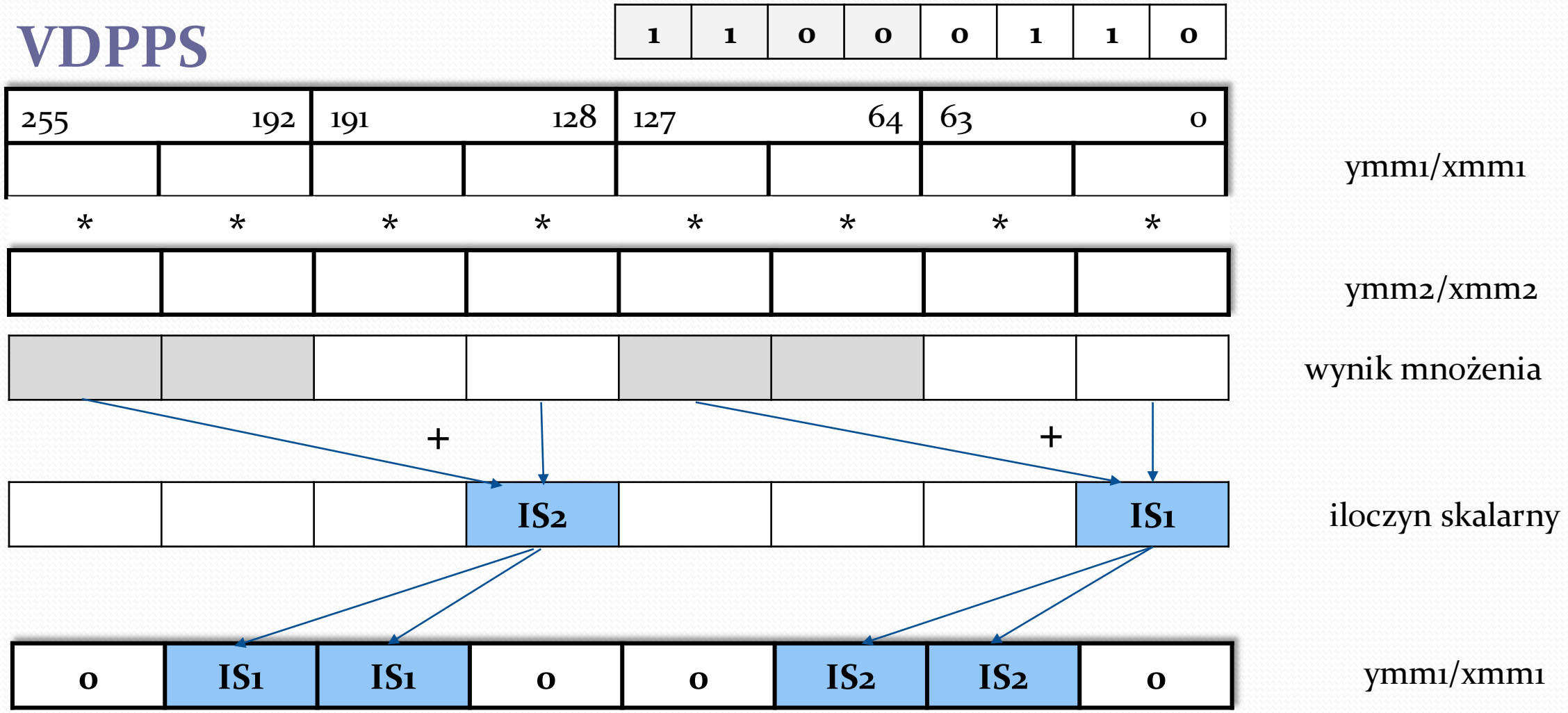
Oblicza iloczyn skalarny wektorów mnożąc warunkowo elementy rejestru xmm2 przez xmm3/mem, a następnie warunkowo (zależnie od ustawień imm8[3...0]) zapisuje wynik lub 0 w xmm1.

```
is = 0
if imm8[i+4] then
        is += xmm2[i]*xmm3/m128[i]

if imm8[i]
        xmm1[i] =  is
else
        xmm1[i] = 0;
```

```
is1 = 0; is2 = 0
if imm8[i+4] then
        is1 += xmm2[i]*xmm3/m128[i]
        is2 += ymm2[i+4]*ymm3/m256[i+4]
if imm8[i]
        xmm1[i] =  is1; ymm1[i+4] =  is2
else
        xmm1[i] =  0; ymm1[i+4] =  0
```

Bity od 128/256 do MSB są zerowane.

Instrukcja iloczyn skalarny



Instrukcja iloczyn skalarny

VDPPD

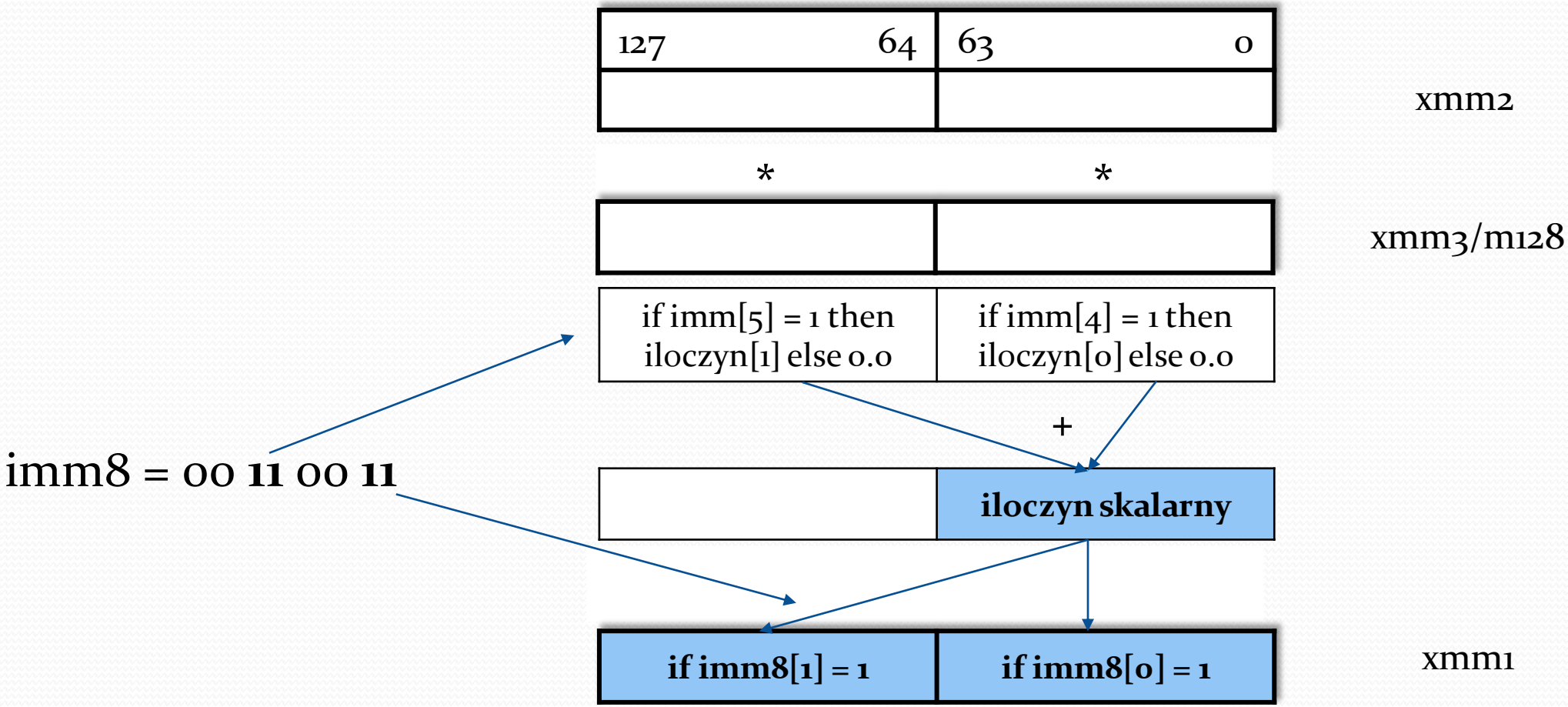
vdppd xmm1,xmm2, xmm3/m128, imm8 (tylko na xmm)

Oblicza iloczyn skalarny wektorów mnoży warunkowo elementy rejestru xmm2 przez xmm3/m128, a następnie sumuje iloczyny ustalając iloczyn skalarny, wynik, w zależności od bajtu sterującego zapisuje w podanych w imm8[1,0] lokalizacjach xmm1.

```
is = 0
if imm8[i+4] then
        is += xmm2[i]*xmm3/m128[i]
if imm8[i]=1 then
        xmm1[i] =  is
else
        xmm[i] = 0
```

Bity od 128/256 do MSB są zerowane.

Instrukcja iloczyn skalarny
VDPPD (tylko dla xmm)



Instrukcje dzielenia

Instrukcja dzielenia

VDIVS[S/D]

vdivs[s/d] xmm1, xmm2, xmm3/m32/m64

Dzieli liczby (skalary) rzeczywiste pojedynczej/podwójnej precyzji z rejestru xmm2/ymm2 odpowiednio przez xmm3/ymm3 lub m32/m64, wynik zapisuje w xmm1/ymm1. Pozostałe elementy przepisuje z xmm2.

xmm1[0] = xmm2[0] / xmm3[0]|m32|m64

Bity od 128/256 do MSB są zerowane.

Instrukcja dzielenia

VDIVP[S/D]

vdivp[s/d] xmm1, xmm2, xmm3/m128

vdivp[s/d] ymm1, ymm2, ymm3/m256

Dzieli wektory liczb rzeczywistych pojedynczej/podwójnej precyzji z rejestru xmm2/ymm2 odpowiednio przez xmm3/ymm3 lub m32/m64/m128/m256, wynik zapisuje w xmm1/ymm1.

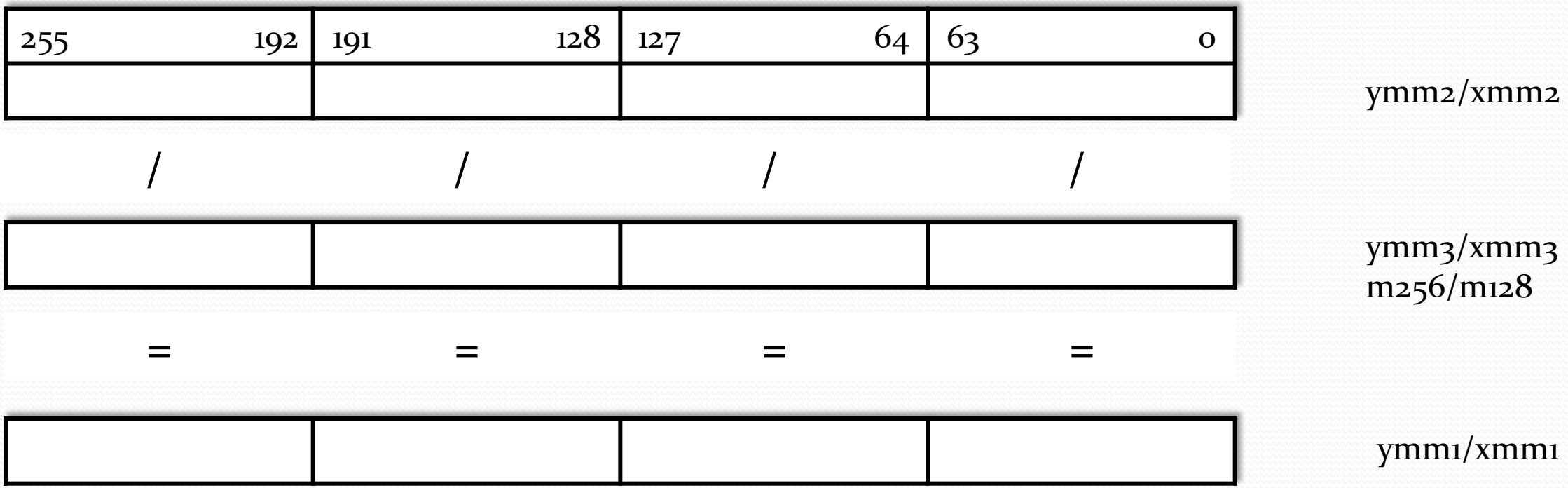
xmm1[i] = xmm2[i] / xmm3/m128[i]

ymm1[i] = ymm2[i] / ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

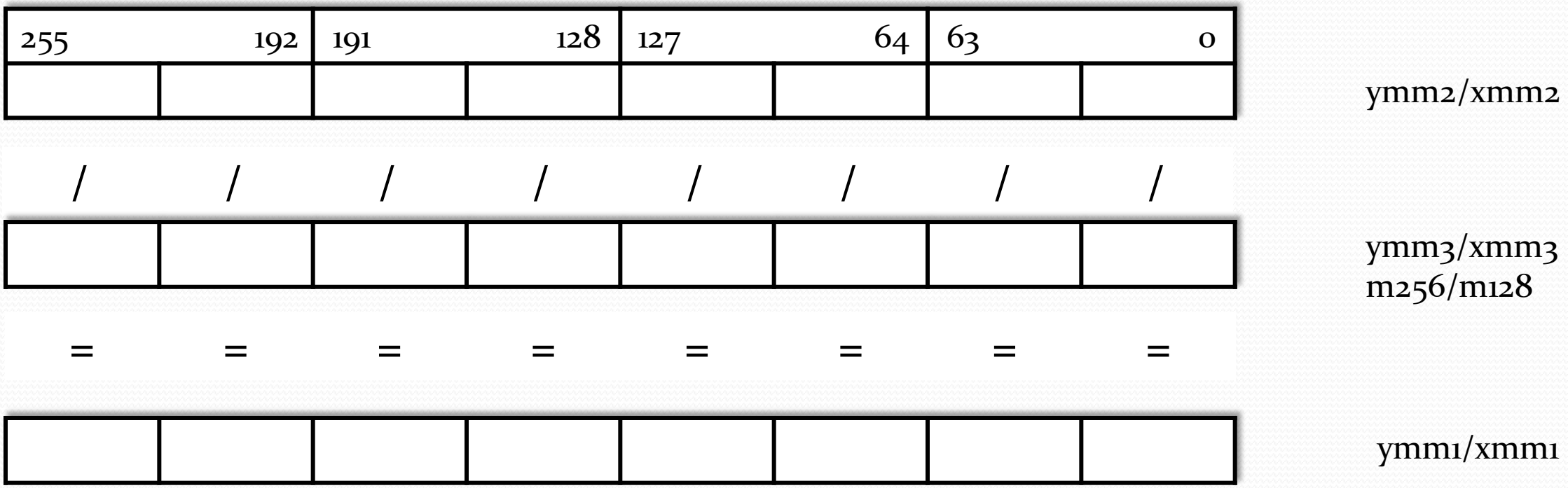
Instrukcja dzielenia

VDIVPD



Instrukcja mnożenia

VDIVPS



Bity od 128/256 do MSB są zerowane.

Instrukcja dzielenia przykład:

```
.code
dzielw proc public
; wektor liczb 1 w rcx; wektor liczb 2 w rdx; wektor liczb wynikowych w r8; liczba elementów w r9
        shl r9, 2    ; r9 rozmiar danych w bajtach
loopd:
        sub r9, 32
        vmovups ymm0, [rcx+r9]
        vmovups ymm1, [rdx+r9]
        vdivps ymm0, ymm0, ymm1
        vmovups [r8+r9], ymm0
        jnz loopd
        ret
dzielw endp
end
```

Operacje arytmetyczne AVX

- Instrukcje maksimum: VMAX[S/P][S/D]
- Instrukcje minimum: VMIN[S/P][S/D]
- Instrukcje zaokrąglenia: VROUND[S/P][S/D]
- Instrukcje odwróconej wartości przybliżonej: VRCPS[S/S]
- Instrukcje odwrócony pierwiastek przybliżony: VRSQRT[S/P]S
- Instrukcje pierwiastkowania: VSQRT[S/P][S/D]

Instrukcja
wartość maksymalna

Instrukcja wartość maksymalna

VMAXS[S/D], VMAXP[S/D]

vmaxs[s/d] xmm1, xmm2, xmm3/m32/m64

vmaxp[s/d] xmm1, xmm2, xmm3/m128

vmaxp[s/d] ymm1, ymm2, ymm3/m256

Zapisuje wartość maksymalną z porównania liczb rzeczywistych pojedynczej/podwójnej precyzji z rejestrów xmm2/ymm2 i xmm3/ymm3 lub m32/m64/m128/m256 do xmm1/ymm1. Dla skalarów pozostałe elementy pochodzą z xmm2.

if xmm2/ymm2[i] > xmm3/ymm3 lub m32/m64/m128/m256[i]

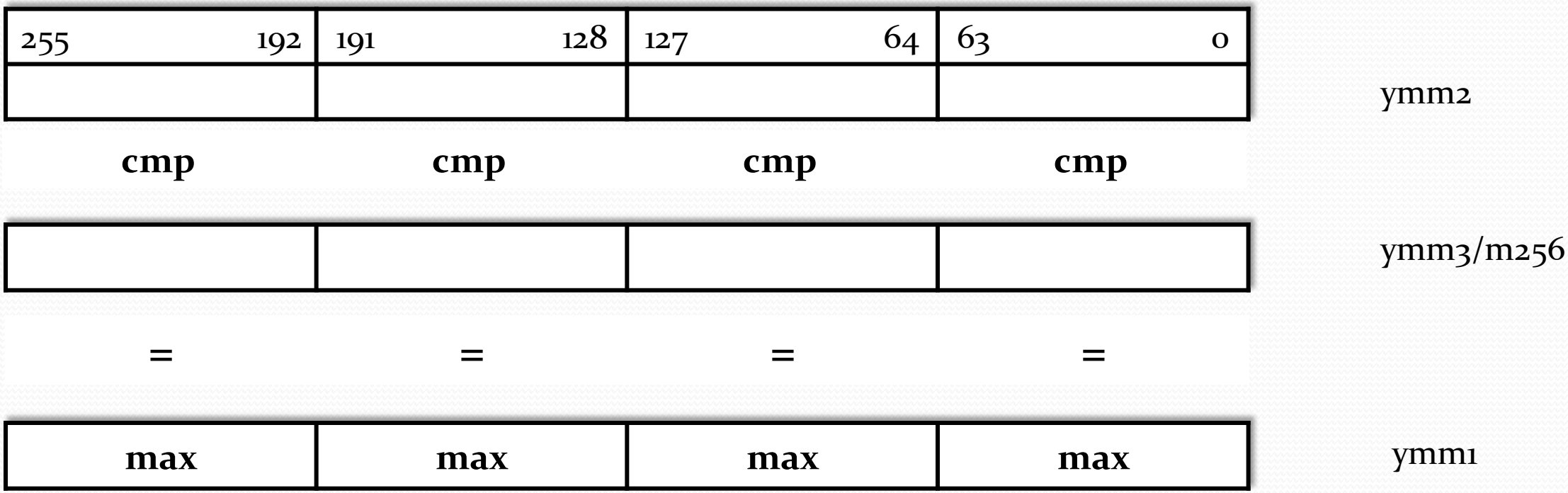
then xmm1/ymm1[i] = xmm2/ymm2[i]

else xmm1/ymm1[i] = xmm3/ymm3 lub m32/m64/m128/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja wartość maksymalana

VMAXPD



Bity od 128/256 do MSB są zerowane.

Instrukcja
wartość minimalna

Instrukcja wartość minimalna

VMIN[S/P][S/D]

vmins[s/d] xmm1, xmm2, xmm3/m32/m64

vminp[s/d] xmm1, xmm2, xmm3/m128

vminp[s/d] ymm1, ymm2, ymm3/m256

Zwraca wartość minimalną z liczb rzeczywistych pojedynczej/podwójnej precyzji odpowiednio skalary/wektory dla rejestrów xmm2/ymm2 i xmm3/ymm3 lub m32/m64/m128/m256, wynik zapisuje do xmm1/ymm1. Dla skalarów pozostałe elementy pochodzą z xmm2.

if xmm2/ymm2[i] < xmm3/ymm3 lub m32/m64/m128/m256[i]

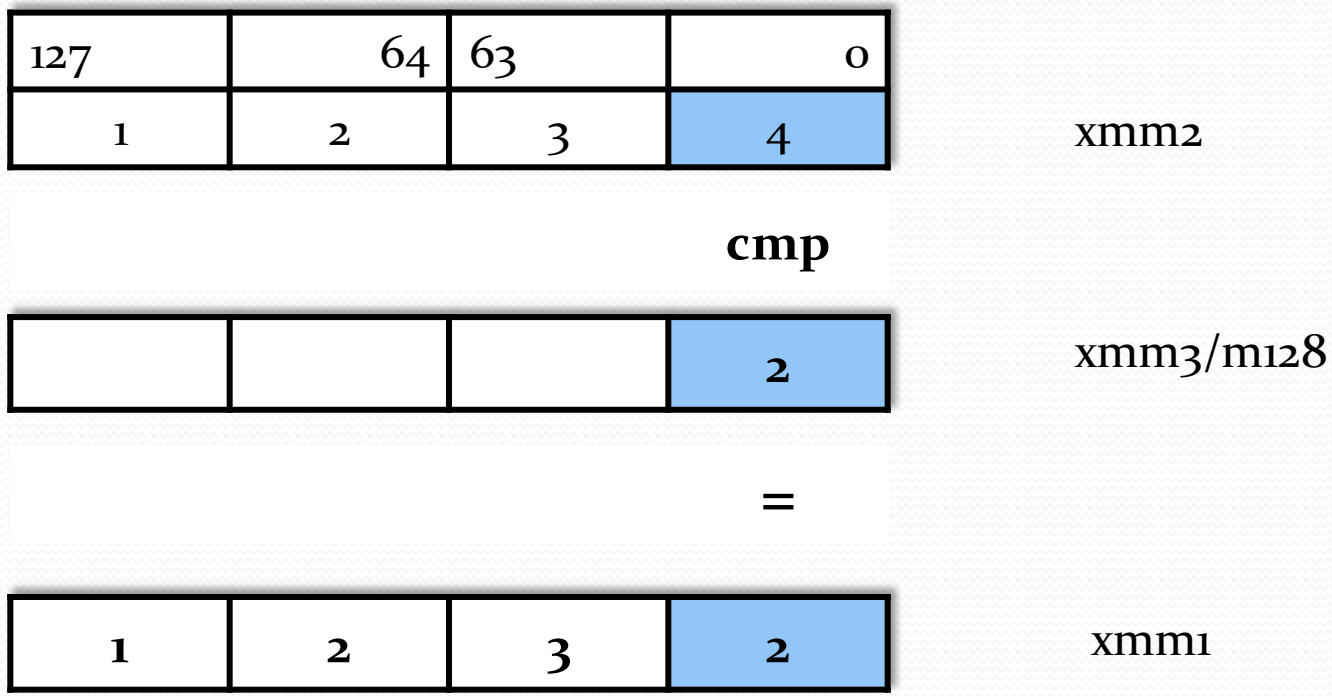
then xmm1/ymm1[i] = xmm2/ymm2[i]

else xmm1/ymm1[i] = xmm3/ymm3 lub m32/m64/m128/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja wartość minimalna

VMINSS



Bity od 128/256 do MSB są zerowane.

Instrukcje
wartość przybliżona

Instrukcja zaokrąglenia

VROUND[S/P][S/D]

vroundss xmm1, xmm2, xmm3/m32, imm8

vroundsd xmm1, xmm2, xmm3/m64, imm8

Zaokrągla najmłodszą liczbę rzeczywistą pojedynczej/podwójnej precyzji z xmm3 lub m32/m64 do wartości podwójnego/poczwórnego słowa (integer), wynik zapisuje w xmm1 jako liczbę rzeczywistą pojedynczej precyzji (z przecinkiem i zerami po nim), pozostałe elementy pochodzą z xmm2, sposób zaokrąglenia jest zdeterminowany bajtem sterującym imm8.

vroundp[s/d] xmm1, xmm2/m128, imm8

vroundp[s/d] ymm1, ymm2/m256, imm8

Zaokrągla wektor liczb rzeczywistych pojedynczej/podwójnej precyzji z xmm2/ymm2 lub m128/m256 do liczb całkowitych podwójnych/poczwórnych słów, wynik zapisuje jako liczby rzeczywiste w xmm1/ymm1, zaokrąglenie odbywa się według bajtu sterującego imm8.

Bity od 128/256 do MSB są zerowane.

Instrukcja zaokrąglenia

VROUND[S/P][S/D]

np. imm8 = 0000 **0 0 00**

P Precision Mask; 0. normal 1. inexact (niedokładny)

RS Rounding select (wybór zaokrąglenia) 1. MXCSR.RC 0. imm8.RC

RC Rounding mode (sposób zaokrąglenia)

Precision:
if imm8[3] = 0;

if imm8[3] = 1
then ustawienie precyzji jest ignorowane;

RC Rounding mode:
- 00 - Zaokrąglij do najbliższej (parzystej)
- 01 - Zaokrąglij w dół (w kierunku $-\infty$)
- 10 – Zaokrąglij w górę (w kierunku $+\infty$)
- 11 - Zaokrąglij do zera (obetnij)

Bity od 128/256 do MSB są zerowane.

Instrukcja wartości odwrotności przybliżonej

VRCP[SS/PS]

vrcpss xmm1, xmm2, xmm3/m32

Oblicza **przybliżoną** wartość **odwrotności** liczby rzeczywistej pojedynczej precyzji z xmm3/m32 i wynik umieszcza w xmm1, dodatkowo przepisuje starsze elementy xmm2 do xmm1.

(Relative Error ≤ 1,5 *2^-12.)

xmm1[31:0] ← 1.0/xmm3/m32[31:0]

xmm1[127:32] ← xmm12[127:32]

vrcpps xmm1, xmm2/m128

vrcpps ymm1, ymm2/m256

Oblicza przybliżoną wartość odwrotności elementów wektora liczb rzeczywistych pojedynczej precyzji z xmm2/ymm2 lub m128/m256 , wynik umieszcza w xmm1/ymm1. (Relative Error ≤ 1,5 *2^-12.)

xmm1[i] ← 1.0/xmm3/m128[i]

ymm1[i] ← 1.0/ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcja wartości odwrotności przybliżonej pierwiastka

VRSQRTSS / VRSQRTPS

vrsqrtss xmm1, xmm2, xmm3/m32

Oblicza **przybliżoną odwrotność pierwiastka** z liczby rzeczywistej pojedynczej precyzji, wynik umieszcza w xmm1, dodatkowo przepisuje starsze elementy xmm2 do xmm1.(Relative Error ≤ 1,5 *2^-12.)

xmm1[31:0] ← 1.0/sqrt(xmm3/m32[31:0])
xmm1[127:32] ← xmm12[127:32]

vrsqrtps xmm1, xmm2/m128

vrsqrtps ymm1, ymm2/m256

Oblicza **przybliżoną odwrotność pierwiastka** z elementów wektora liczb rzeczywistych pojedynczej precyzji xmm2/ymm2 lub m128/m256, wynik umieszcza w xmm1/ymm1. (Relative Error ≤ 1,5 *2^-12.)

xmm1[i] ← 1.0/sqrt(xmm3/m128[i])
ymm1[i] ← 1.0/sqrt(ymm3/m256[i])

Bity od 128/256 do MSB są zerowane.

Instrukcje pierwiastkowania

Instrukcja wartości odwrotności przybliżonej pierwiastka

VSQRT[S/P][S/D]

vsqrtss xmm1, xmm2, xmm3/m32

vsqrtsd xmm1, xmm2, xmm3/m64

Oblicza wartość pierwiastka kwadratowego z liczby rzeczywistej pojedynczej/podwójnej precyzji xmm3/m32, wynik umieszcza w xmm1, dodatkowo przepisuje starsze elementy xmm2 do xmm1.

xmm1[0] ← sqrt(xmm3[0]/m32/m64)

xmm1[i] ← xmm2[i]

vsqrtp[s/d] xmm1, xmm2/m128

vsqrtp[s/d] ymm1, ymm2/m256

Oblicza wartość pierwiastka kwadratowego z elementów wektora liczb rzeczywistych pojedynczej /podwójnej precyzji xmm2/ymm2 lub m128/m256, wynik zapisuje w xmm1/ymm1.

xmm1[i] ← sqrt(xmm3/m128[i])

ymm1[i] ← sqrt(ymm3/m256[i])

Bity od 128/256 do MSB są zerowane.

Instrukcje FMA
Fused Multiply Add

Operacje arytmetyczne FMA

- Instrukcje mnożenie i dodawanie: VFMADD[123/213/231][S/P][S/D]
- Instrukcje mnożenie i odejmowanie: VFMSUB[123/213/231][S/P][S/D]
- Instrukcje mnożenie i odejmowanie z dodawaniem: VFMADDSUB[123/213/231]P[S/D]
- Instrukcje mnożenie i odejmowanie z dodawaniem: VFMSUBADD[123/213/231]P[S/D]
- Instrukcje mnożenie z negacją i dodawanie:
- VFNMADD[123/213/231][S/P][S/D]
- Instrukcje mnożenie z negacją i odejmowanie: VFNMSUB[123/213/231][S/P][S/D]

Instrukcje FMA

VFMADD[132/213/231][S/P][S/D]

vfmadd[132/231/231]S[S/D] xmm1, xmm2, xmm3/m32/m64

vfmadd[132/231/231]P[S/D] xmm1, xmm2, xmm3/m128

vfmadd[132/231/231]P[S/D] ymm1, ymm2, ymm3/m256

Mnoży skalary / wektory liczb rzeczywistych pojedynczej / podwójnej precyzji dwóch rejestrów i **dodaje** odpowiednie wartości trzeciego rejestru w zależności od podanej kolejności, pierwsze dwie cyfry oznaczają czynniki iloczynu, trzecia cyfra jest składnikiem sumy, wynik zapisuje w xmm1/ymm1.

132

ymm1[i] = ymm1[i] * ymm3/m256[i] + ymm2[i]

231

ymm1[i] = ymm2[i] * ymm3/m256[i] + ymm1[i]

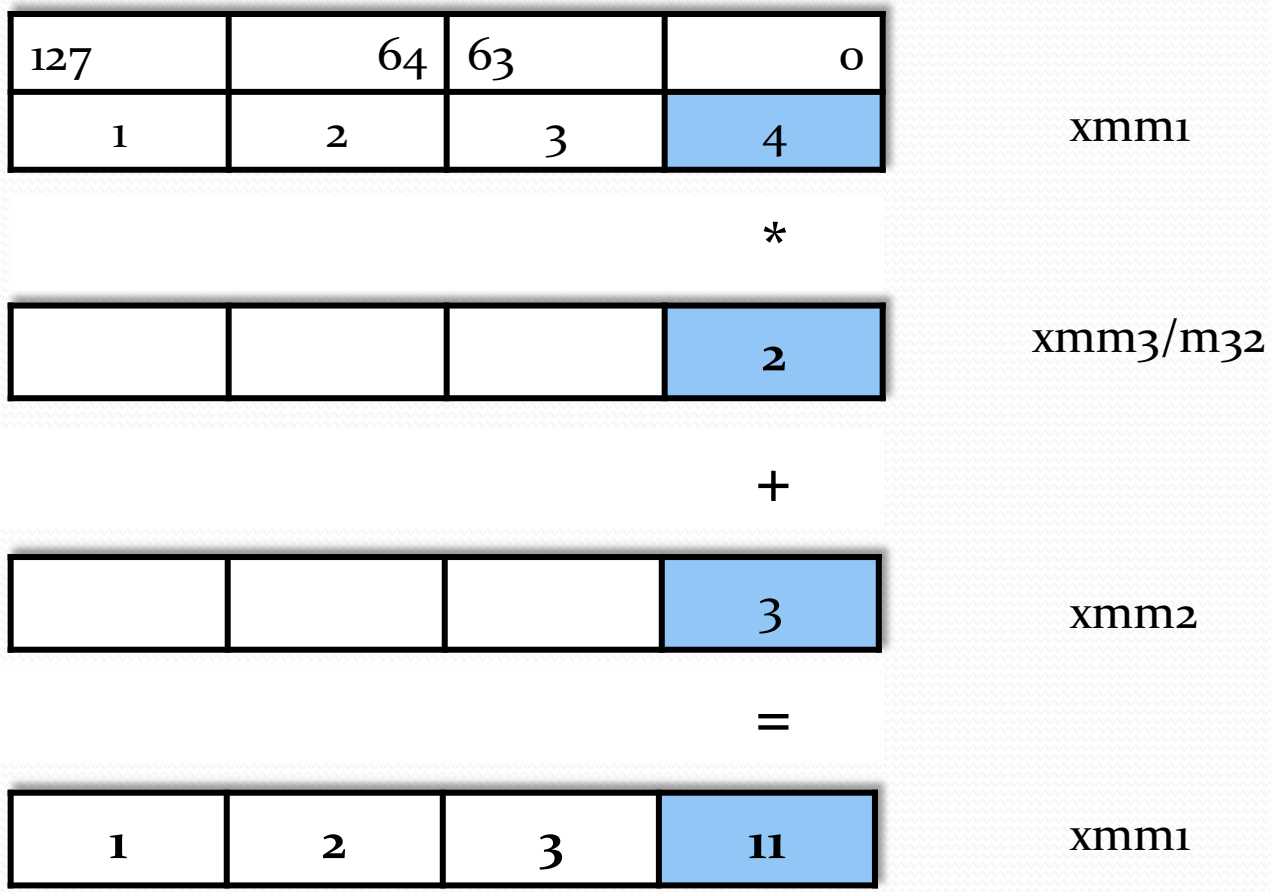
213

ymm1[i] = ymm2[i] * ymm1[i] + ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje FMA

vfmadd132ss xmm1, xmm2, xmm3/m32



Instrukcje FMA

VFMSUB[132/213/231][S/P][S/D]

vfmsub[132/231/231]**S**[S/D] xmm1, xmm2, xmm3/m32/m64

vfmsub[132/231/231]**P**[S/D] xmm1, xmm2, xmm3/m128

vfmsub[132/231/231]**P**[S/D] ymm1, ymm2, ymm3/m256

Mnoży skalary / wektory liczb rzeczywistych pojedynczej / podwójnej precyzji dwóch rejestrów i **odejmuje** odpowiednie wartości trzeciego rejestru w zależności od podanej kolejności, pierwsze dwie cyfry oznaczają czynniki iloczynu, trzecia cyfra jest elementem różnicy, wynik zapisuje w xmm1/ymm1.

132

ymm1[i] = ymm1[i] * ymm3/m256[i] - ymm2[i]

231

ymm1[i] = ymm2[i] * ymm3/m256[i] - ymm1[i]

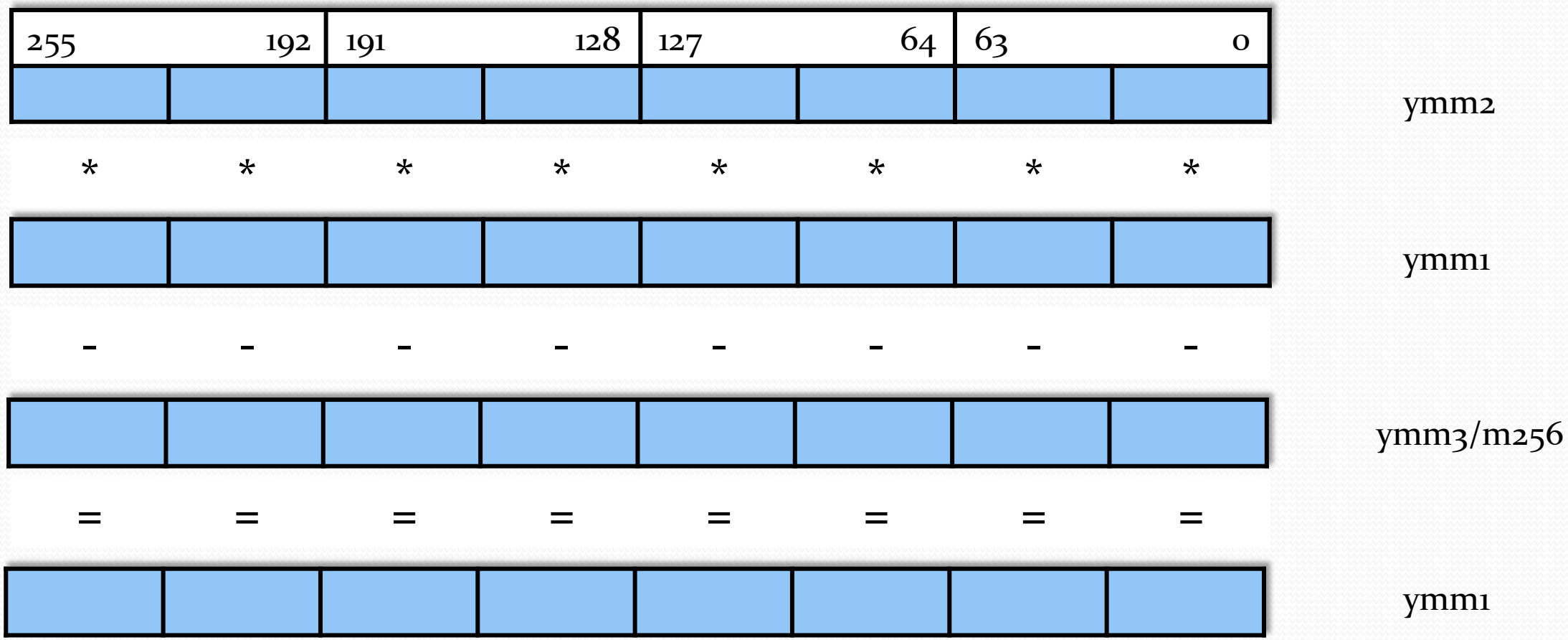
213

ymm1[i] = ymm2[i] * ymm1[i] - ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje FMA

vfmsub213ps ymm1, ymm2, ymm3/m256



Instrukcje FMA

VFMADDSUB[132/213/231]P[S/D]

vfmaddsub[132/231/231]p[s/d] xmm1, xmm2, xmm3/m128

vfmaddsub[132/231/231]p[s/d] ymm1, ymm2, ymm3/m256

Mnoży wektory liczb rzeczywistych pojedynczej / podwójnej precyzji dwóch rejestrów i naprzemiennie **odejmuje i dodaje** odpowiednie wartości trzeciego rejestru w zależności od podanej kolejności, pierwsze dwie cyfry oznaczają czynniki iloczynu, trzecia cyfra jest elementem różnicy/sumy, wynik zapisuje w xmm1/ymm1.

132

ymm1[2i] = ymm1[2i] * ymm3/m256[2i] – ymm2[2i]

ymm1[2i+1] = ymm1[2i+1] * ymm3/m256[2i+1] + ymm2[2i+1]

231

ymm1[2i] = ymm2[2i] * ymm3/m256[2i] - ymm1[2i]

ymm1[2i+1] = ymm2[2i+1] * ymm3/m256[2i+1] + ymm1[2i+1]

213

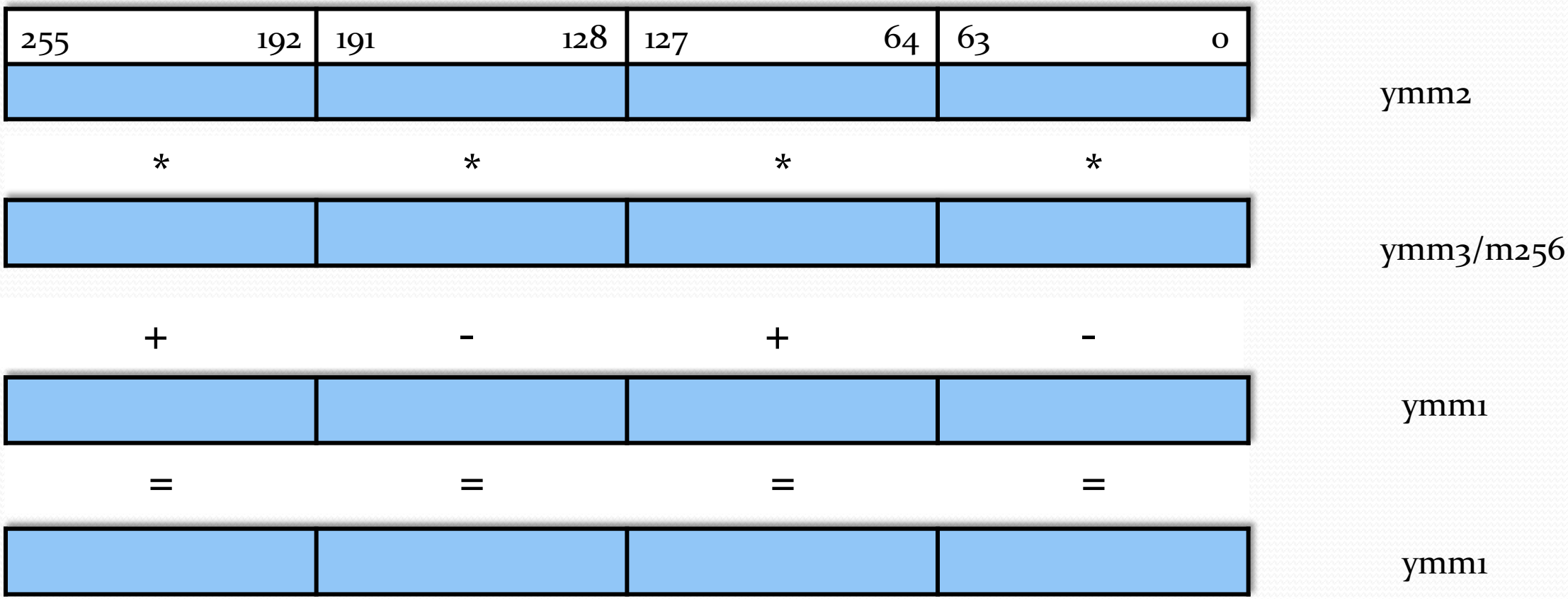
ymm1[2i] = ymm2[2i] * ymm1[2i] – ymm3/m256[2i]

ymm1[2i+1] = ymm2[2i+1] * ymm1[2i+1] + ymm3/m256[2i+1]

Bity od 128/256 do MSB są zerowane.

Instrukcja FMA

vfmaddsub231pd ymm1, ymm2, ymm3/m256



Instrukcje FMA

VFMSUBADD[132/213/231]P[S/D]

vfmsubadd[132/231/231]p[s/d] xmm1, xmm2, xmm3/m128

vfmsubadd[132/231/231]p[s/d] ymm1, ymm2, ymm3/m256

Mnoży wektory liczb rzeczywistych pojedynczej / podwójnej precyzji dwóch rejestrów i naprzemiennie **dodaje i odejmuje** odpowiednie wartości trzeciego rejestru w zależności od podanej kolejności, pierwsze dwie cyfry oznaczają czynniki iloczynu, trzecia cyfra jest elementem sumy/różnicy, wynik zapisuje w xmm1/ymm1.

132

ymm1[2i] = ymm1[2i] * ymm3/m256[2i] + ymm2[2i]

ymm1[2i+1] = ymm1[2i+1] * ymm3/m256[2i+1] - ymm2[2i+1]

231

ymm1[2i] = ymm2[2i] * ymm3/m256[2i] + ymm1[2i]

ymm1[2i+1] = ymm2[2i+1] * ymm3/m256[2i+1] - ymm1[2i+1]

213

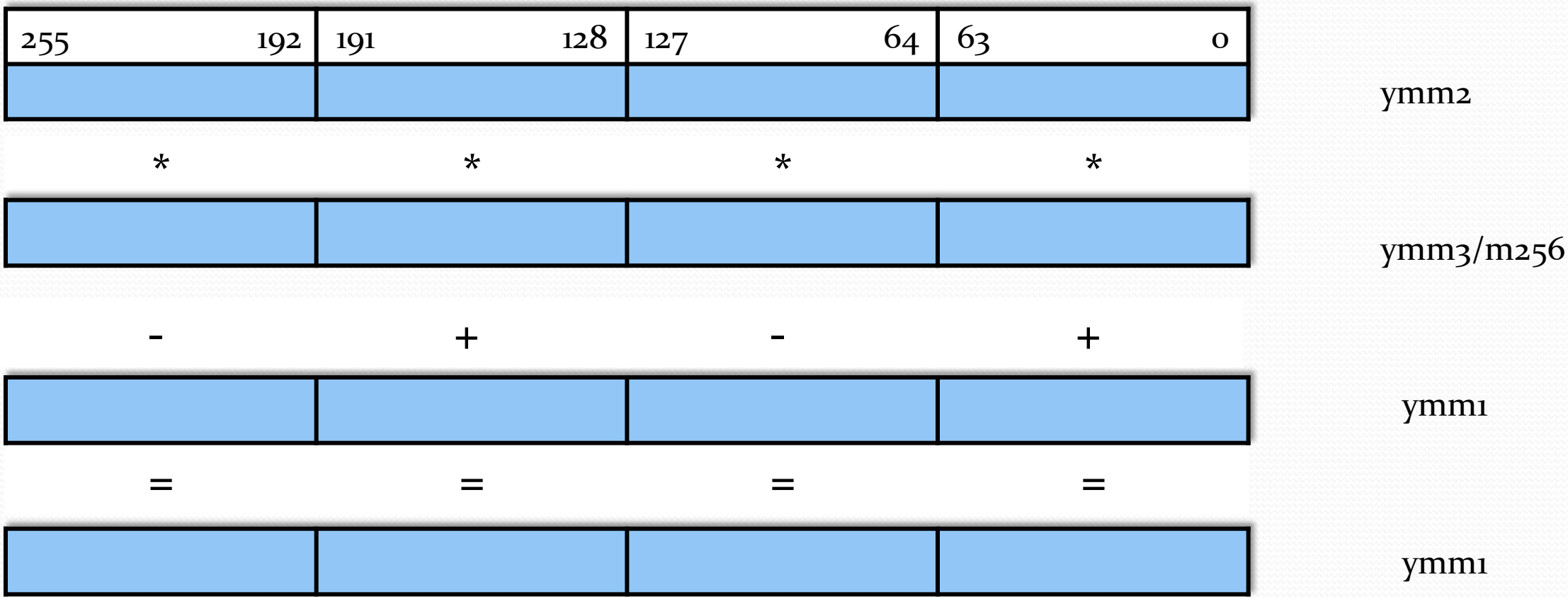
ymm1[2i] = ymm2[2i] * ymm1[2i] + ymm3/m256[2i]

ymm1[2i+1] = ymm2[2i+1] * ymm1[2i+1] - ymm3/m256[2i+1]

Bity od 128/256 do MSB są zerowane.

Instrukcja FMA

vfmsubadd231pd ymm1, ymm2, ymm3/m256



Instrukcje FMA

VFNMADD[132/213/231][S/P][S/D]

vfnmadd[132/231/231]S[S/D] xmm1, xmm2, xmm3/m32/m64

vfnmadd[132/231/231]P[S/D] xmm1, xmm2, xmm3/m128

vfnmadd[132/231/231]P[S/D] ymm1, ymm2, ymm3/m256

Mnoży skalary / wektory liczb rzeczywistych pojedynczej / podwójnej precyzji dwóch rejestrów, **zmienia znak iloczynów** i **dodaje** odpowiednie wartości trzeciego rejestru w zależności od podanej kolejności, pierwsze dwie cyfry oznaczają czynniki iloczynu, trzecia cyfra jest składnikiem sumy, wynik zapisuje w xmm1/ymm1.

132

ymm1[i] = -ymm1[i] * ymm3/m256[i] + ymm2[i]

231

ymm1[i] = -ymm2[i] * ymm3/m256[i] + ymm1[i]

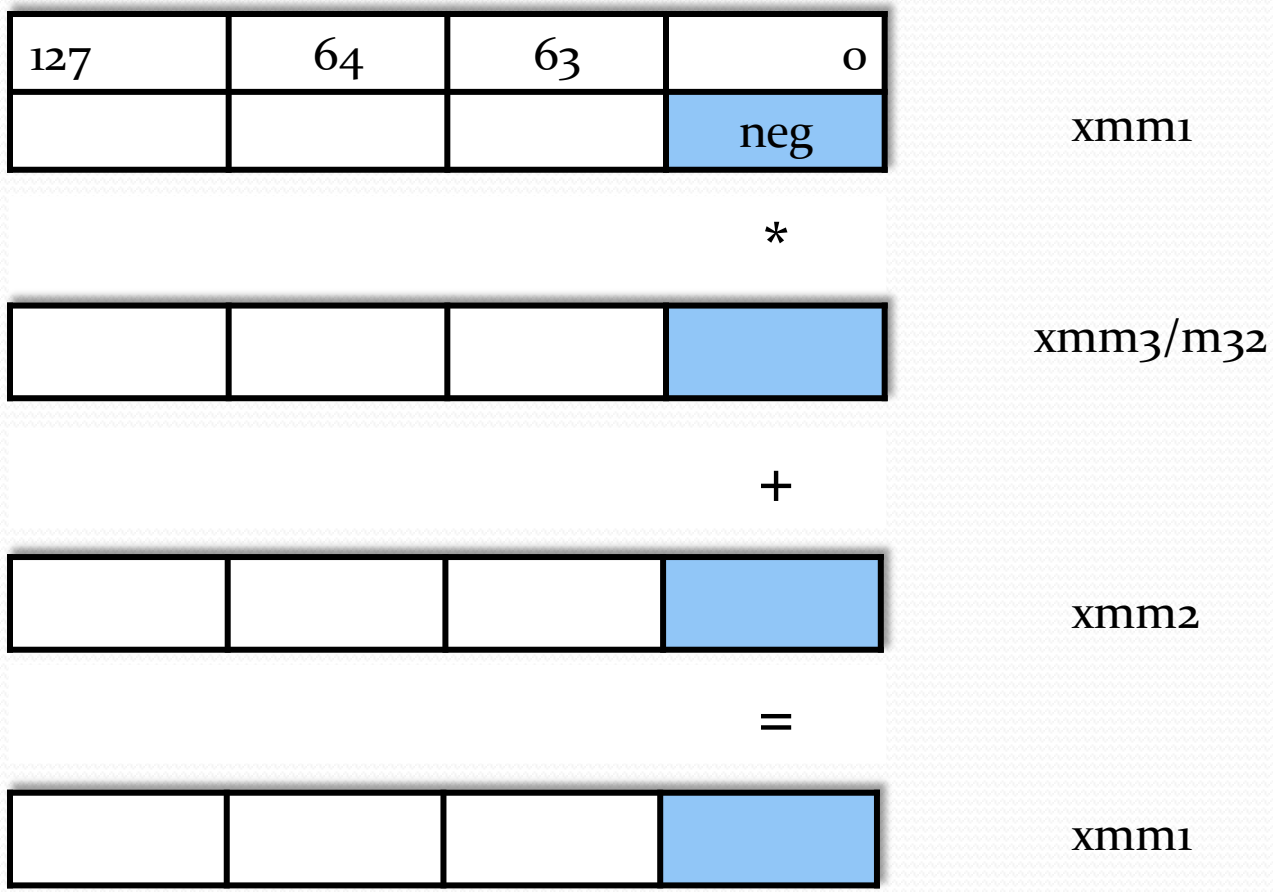
213

ymm1[i] = -ymm2[i] * ymm1[i] + ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje FMA

vfnmadd132ss ymm1, ymm2, ymm3/m256



Instrukcje FMA

VFNMSUB[132/213/231][S/P][S/D]

vfnmsub[132/231/231]**S**[S/D] xmm1, xmm2, xmm3/m32/m64

vfnmsub[132/231/231]**P**[S/D] xmm1, xmm2, xmm3/m128

vfnmsub[132/231/231]**P**[S/D] ymm1, ymm2, ymm3/m256

Mnoży skalary / wektory liczb rzeczywistych pojedynczej / podwójnej precyzji dwóch rejestrów, **zmienia znak iloczynów** i **odejmuje** odpowiednie wartości trzeciego rejestru w zależności od podanej kolejności, pierwsze dwie cyfry oznaczają czynniki iloczynu, trzecia cyfra jest elementem różnicy, wynik zapisuje w xmm1/ymm1.

132

ymm1[i] = -ymm1[i] * ymm3/m256[i] - ymm2[i]

231

ymm1[i] = -ymm2[i] * ymm3/m256[i] - ymm1[i]

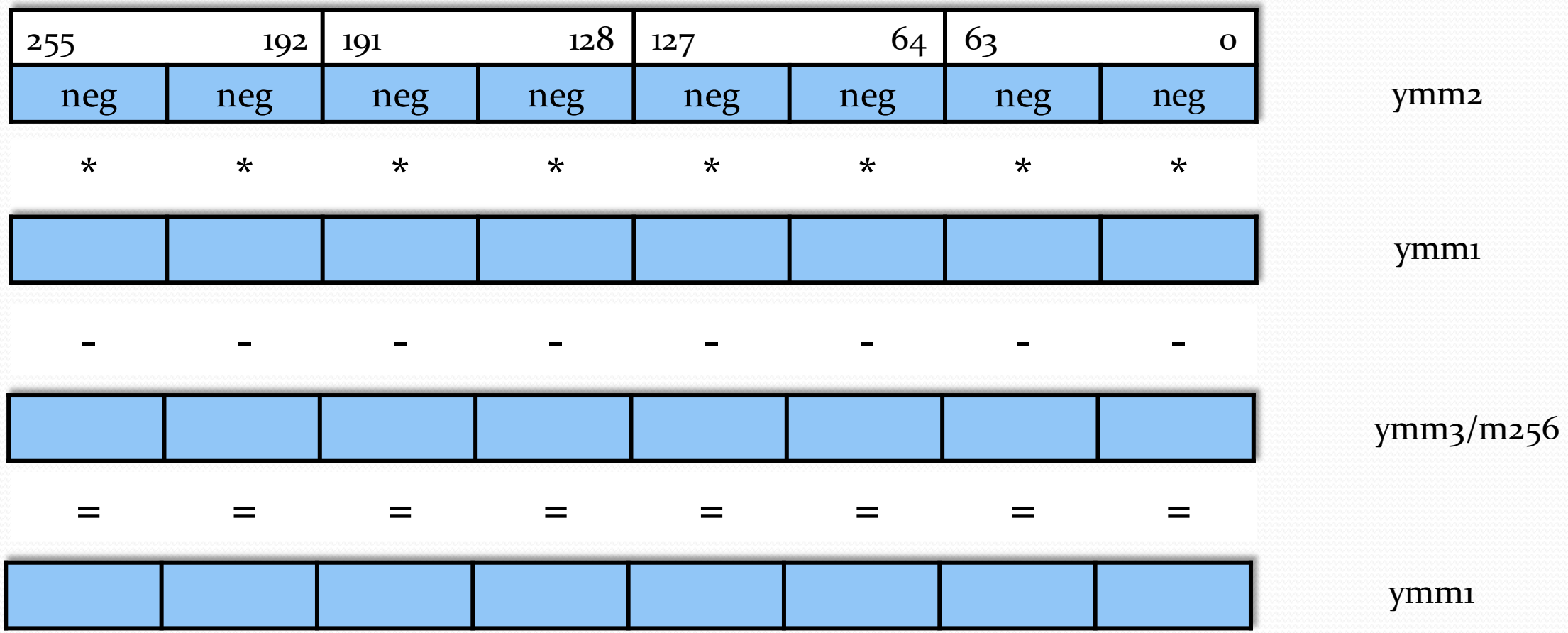
213

ymm1[i] = -ymm2[i] * ymm1[i] - ymm3/m256[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje FMA

vfnmsub213ps ymm1, ymm2, ymm3/m256



Operacje porównania

Operacje porównania:

- Instrukcje porównania:

VCMP[S/P][S/D]

VCOMIS[S/D]

VUCOMIS[S/D]

Instrukcje porównania

VCMPS[S/D], VCMPP[S/D]

vcmps[s/d] xmm1, xmm2 xmm3/m32/m64, imm8

vcmpp[s/d] xmm1, xmm2 xmm3/m128, imm8

vcmpp[s/d] ymm1, ymm2, ymm3/m256, imm8

Porównuje skalary/wektory liczb rzeczywistych pojedynczej/podwójnej precyzji xmm2/ymm2 i xmm3/ymm3 lub m32/m64/m128/m256 według funktora zapisanego na bitach imm8[4:0] (łącznie 32 funktory), wynik **jako liczbę całkowitą** -1 lub 0 zapisuje w xmm1/ymm1. Dla skalarów pozostałe elementy są kopiowane ze źródła1.

Bity od 128/256 do MSB są zerowane.

Instrukcje porównania VCMPP[S/D]

funktor	imm8	opis	>	<	=	Unordered	#IA on QNAN
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	f	f	t	f	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	f	t	f	f	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	f	t	t	f	Yes
UNORD_Q (UNORD)	3H	Unordered (no-signaling)	f	f	f	t	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling	t	t	f	t	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	t	f	t	t	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	t	f	f	t	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	t	t	t	f	No
EQ_UQ	8H	Equal (unordered, non-signaling)	f	f	t	t	No
NGE_US (NGE)	9H	Not-greater-than-orequal (unordered, signaling)	f	t	f	t	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	f	t	t	t	Yes
FALSE_OQ (FALSE)	BH	False (ordered, non-signaling)	f	f	f	f	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	t	t	f	f	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	t	f	t	f	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	t	f	f	f	Yes
TRUE_UQ(TRUE)	FH	True (unordered, nonsignaling)	t	t	t	t	No
EQ_OS	10H	Equal (ordered, signaling)	f	f	t	f	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	f	t	f	f	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	f	t	t	f	No
UNORD_S	13H	Unordered (signaling)	f	f	f	t	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	t	t	f	t	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	t	f	t	t	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	t	f	f	t	No
ORD_S	17H	Ordered (signaling)	t	t	t	f	Yes
EQ_US	18H	Equal (unordered, signaling)	f	f	t	t	Yes
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	f	t	f	t	No
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	f	t	t	t	No
FALSE_OS	1BH	False (ordered, signaling)	f	f	f	f	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	t	t	f	f	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	t	f	t	f	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	t	f	f	f	No
TRUE_US	1FH	True (unordered,signaling)	t	t	t	t	Yes

Instrukcje porównania VCMP[S/P][S/D]

Lp.	Pseudo-operacja	Implementacja
1.	VCMP**EQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 0*
2.	VCMP**LT**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1*
3.	VCMP**LE**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 2*
4.	VCMP**UNORD**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 3*
5.	VCMP**NEQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 4*
6.	VCMP**NLT**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 5*
7.	VCMP**NLE**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 6*
8.	VCMP**ORD**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 7*
9.	VCMP**EQ_UQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 8*
10.	VCMP**NGE**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 9*
11.	VCMP**NGT**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 0AH*
12.	VCMPFALSE[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 0BH*
13.	VCMP**NEQ_OQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 0CH*
14.	VCMP**GE**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 0DH*
15.	VCMP**GT**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 0EH*

Lp.	Pseudo-operacja	Implementacja
16.	VCMP**TRUE**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, **0FH***
27.	VCMP**EQ_OS**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 10H*
18.	VCMP**LT_OQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 11H*
19.	VCMP**LE_OQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 12H*
20.	VCMP**UNORD_S**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 13H*
21.	VCMPN**EQ_US**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 14H*
22.	VCMP**NLT_UQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 15H*
23.	VCMP**NLE_UQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 16H*
24.	VCMP**ORD_S**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 17H*
25.	VCMP**EQ_US**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 18H*
26.	VCMP**NGE_UQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 19H*
27.	VCMP**NGT_UQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1AH*
28.	VCMP**FALSE_OS**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1BH*
29.	VCMP**NEQ_OS**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1CH*
30	VCMP**GE_OQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1DH*
31	VCMP**GT_OQ**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1EH*
32	VCMP**TRUE_US**[S/P][S/D] *reg1, reg2, reg3*	VCMP[S/P][S/D] *reg1, reg2, reg3, 1FH*

Instrukcje porównania

VCOMIS[S/D] / VUCOMIS[S/D]

vcomis[s/d] xmm1, xmm2/m32/m64

Porównuje pojedyncze liczby rzeczywiste pojedynczej/podwójnej precyzji z rejestru xmm2 lub pamięci m32/m64 i xmm1 wynikiem jest **ustawienie odpowiednich flag procesora.** Instrukcja VCOMISD sygnalizuje wyjątek nieprawidłowej operacji zmiennoprzecinkowej SIMD (#I) gdy operandem źródłowym jest QNaN lub SNaN.

Instrukcje porównania

VCOMIS[S/D] / VUCOMIS[S/D]

vcomis[s/d] xmm1, xmm2/m32/m64

Porównuje pojedyncze liczby rzeczywiste pojedynczej/podwójnej precyzji z rejestru xmm2 lub pamięci m32/m64 i xmm1, wynikiem jest ustawienie odpowiednich flag procesora. Instrukcja VUCOMISD sygnalizuje wyjątek nieprawidłowej operacji tylko wtedy, gdy operandem źródłowym jest SNaN.

Operacje logiczne

Operacje logiczne

- Koniunkcja: VANDP[S/D]
- Koniunkcja z zaprzeczeniem: VANDNP[S/D]
- Alternatywa: VORP[S/D]
- Alternatywa wykluczająca: VXORP[S/D]
- Instrukcja Test: VTESTP[S/D]

Instrukcje logiczne koniunkcja

VANDP[S/D] / VANDNP[S/D]

vandp[s/d] xmm1, xmm2, xmm3/m128

vandp[s/d] ymm1, ymm2, ymm3/m256

Oblicza bitową **koniunkcję wektorów** liczb rzeczywistych pojedynczej/podwójnej precyzji z xmm2/ymm12 i xmm3/ymm3 lub m128/m256, wynik zapisuje w xmm1/ymm1.

cel[i] = źródło1[i] and źródło2[i]

vandnp[s/d] xmm1, xmm2, xmm3/m128

vandnp[s/d] ymm1, ymm2, ymm3/m256

Oblicza bitową **koniunkcję z negacją wektorów** liczb rzeczywistych pojedynczej /podwójnej precyzji z xmm2/ymm12 i xmm3/ymm3 lub m128/m256, wynik zapisuje w xmm1/ymm1.

cel[i] = (not źródło1[i]) and źródło2[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje logiczne alternatywa

VORP[S/D] / VXORP[S/D]

vorp[s/d] xmm1, xmm2, xmm3/m128

vorp[s/d] ymm1, ymm2, ymm3/m256

Oblicza bitową **alternatywę wektorów liczb rzeczywistych pojedynczej/podwójnej precyzji** rejestru xmm2/ymm2 (źródło 1) i xmm3/ymm3 lub m128/m256 (źródło 2), wynik zapisuje w w xmm1/ymm1.

cel[i] = źródło1[i] or źródło2[i]

vxorp[s/d] xmm1, xmm2, xmm3/m128

vxorp[s/d] ymm1, ymm2, ymm3/m256

Oblicza bitową **alternatywę wykluczającej wektorów liczb rzeczywistych pojedynczej/podwójnej precyzji** rejestru xmm2/ymm2 (źródło 1) i xmm3/ymm3 lub m128/m256 (źródło 2), wynik zapisuje w w xmm1/ymm1.

cel[i] = źródło1[i] xor źródło2[i]

Bity od 128/256 do MSB są zerowane.

Instrukcje testowania

VTESTP[S/D]

vtestp[s/d] xmm1, xmm2/m128

vtestp[s/d] ymm1, ymm2/m256

Wykonuje **logicznie koniunkcję (AND) na bitach znaku** liczb rzeczywistych pojedynczej / podwójnej precyzji z rejestru xmm1/ymm1 i xmm2/ymm2 lub m1128/m256, wynikiem jest ustawienie flagi ZF, jeśli wszystkie bity znaku = 0 => ZF = 1 lub ZF = 0 w przeciwnym przypadku

oraz jednocześnie

wykonuje **logicznie koniunkcję z zaprzeczeniem (AND NOT)(nie cel i źródło) na bitach znaku** liczb rzeczywistych pojedynczej / podwójnej precyzji z rejestru xmm1/ymm1 i xmm2/ymm2 lub m1128/m256, wynikiem jest ustawienie flagi CF, jeśli wszystkie bity znaku = 0 => CF=1, jeśli nie to CF=0.

Operacje konwersji

Operacje logiczne

- Całkowite na rzecywiste:

VCVTDQ2[PS/PD],VCVTSI2[SS/SD]

- Rzeczywiste na całkowite:

VCVT[PS/PD]2DQ ,VCVT[SS/SD]2SI

VCVTT[PS/PD]2DQ, VCVTT[SS/SD]2SI (z tranzakcją)

- Rzeczywiste na rzeczywiste:

VCVTSD2SS, VCVTSS2SD

VCVTPD2PS, VCVTPS2PD

Instrukcje konwersji całkowite na rzeczywiste

VCVTDQ2P[S/D]

vcvtdq2ps xmm1, xmm2/m128

vcvtdq2ps ymm1, ymm2/m256

vcvtdq2pd xmm1, xmm2/m64

vcvtdq2pd ymm1, xmm2/m128

Konwertuje dwa/cztery/osiem podwójnych słów ze znakiem z xmm2/ymm2 lub m128/m256 na dwie/cztery/osiem liczb rzeczywistych pojedynczej/podwójnej precyzji, wynik zapisuje do rejestru celu xmm1/ymm1. Konwertuje zawsze dwa/dwa, cztery/cztery, osiem/osiem.

Bity od 128/256 do MSB są zerowane.

Instrukcje konwersji całkowite na rzeczywiste

VCVTSI2S[S/D]

vcvtsi2ss xmm1, xmm2, reg/m32

vcvtsi2sd xmm1, xmm2, reg/m64

Konwertuje **pojedyncze podwójne słowo** ze znakiem z rejestru ogólnego przeznaczenia lub m32/m64 **na jedną liczbę rzeczywistą** pojedynczej/podwójnej precyzji, wynik zapisuje do xmm1/ymm1, bity [127:64/32] są przepisywane z xmm2/ymm2.

Bity od 128/256 do MSB są zerowane.

Instrukcje konwersji rzeczywiste na całkowite

VCVT[T]P[S/D]2DQ

vcvt[t]ps2dq xmm1, xmm2/m128

vcvt[t]ps2dq ymm1, ymm2/m256

Konwertuje cztery/osiem pojedynczych słów ze znakiem z xmm2/ymm2 lub m128/m256 na cztery/osiem podwójnych słów ze znakiem, wynik zapisuje w xmm1/ymm1.

vcvt[t]pd2dq xmm1, xmm2/m128

vcvt[t]pd2dq ymm1, ymm2/m256

Konwertuje dwa/cztery pojedyncze słowa ze znakiem z xmm2/ymm2 lub m128/m256 na dwa/cztery podwójne słowa ze znakiem, wynik zapisuje w xmm1/ymm1.

Zwrócona wartość jest zaokrąglana zgodnie z bitami kontrolnymi zaokrąglania w rejestrze MXCSR lub dla [T] obcięta kierunku zera.

Bity od 128/256 do MSB są zerowane.

Instrukcje konwersji rzeczywiste na całkowite

VCVT[T]S[S/D]2SI

vcvt[t]ss2si reg32, xmm1/m32

vcvt[t]ss2si reg64, xmm1/m64

Konwertuje liczbę pojedynczej precyzji z xmm1 lub pomięci m32/m64 na podwójne/poczwórne słowo ze znakiem, wynik zapisuje w rejestrze ogólnego przeznaczenia r32/r64.

vcvt[t]sd2si reg32, xmm1/m64

vcvt[t]sd2si reg64, xmm1/m64

Konwertuje liczbę podwójnej precyzji z xmm1 lub m32/m64 na podwójne/poczwórne słowo ze znakiem, wynik zapisuje w rejestrze ogólnego przeznaczenia r32/r64.

Instrukcja z T oznacza konwersję z obcięciem w kierunku zera.

Bity od 128/256 do MSB są zerowane.

Instrukcje konwersji rzeczywiste na rzeczywiste

VCVTSD2SS / VCVTSS2SD

vcvtsd2ss xmm1, xmm2, xmm3/m64

Konwertuje liczbę podwójnej precyzji z xmm3/m64 na liczbę pojedynczej precyzji, wynik umieszcza w xmm1, starsze bity xmm1 są uzupełniane z xmm2.

vcvtss2sd xmm1, xmm2, xmm3/m32

Konwertuje liczbę pojedynczej precyzji z xmm3/m32 na liczbę podwójnej precyzji, wynik umieszcza w xmm1, starsze bity xmm1 są uzupełniane z xmm2.

Bity od 128/256 do MSB są zerowane.

Instrukcje konwersji rzeczywiste na rzeczywiste

VCVTPD2PS / VCVTPS2PD

vcvtpd2ps xmm1, xmm2/m128

vcvtpd2ps xmm1, ymm2/m256

Konwertuje wektor liczb rzeczywistych podwójnej precyzji na wektor liczb rzeczywistych pojedynczej precyzji. Konwertuje dwa na dwa elementy lub cztery na cztery elementy.

vcvtps2pd xmm1, xmm2/m64

vcvtps2pd ymm1, xmm2/m128

Konwertuje wektor liczb rzeczywistych pojedynczej precyzji na wektor liczb rzeczywistych podwójnej precyzji. Konwertuje dwa na dwa elementy lub cztery na cztery elementy.

Bity od 128/256 do MSB są zerowane.