SPIS TREŚCI

WSTĘP

Minęło już 12 lat od wprowadzenia na rynek pierwszego procesora rodziny x86 z technologią MMX. Jak pokazał czas, było to na tyle udane rozwiązanie, że technologię przetwarzania wektorowego w procesorach rodziny x86 rozwijano w kolejnych wersjach procesorów. W 1999 roku wprowadzono rozszerzenie SSE umożliwiające wykonywanie operacji na czteroelementowych wektorach liczb typu IEEE-754 SINGLE. Następnie w procesorach Pentium 4 wprowadzono rozszerzenie SSE2 umożliwiające wykonywanie operacji na dwuelementowych wektorach liczb typu IEEE-754 DOUBLE. W kolejnych latach poszerzono listę instrukcji przetwarzania wektorowego o 13 nowych pozycji wraz z wprowadzeniem rozszerzenia SSE3 (2004 rok) oraz o 54 nowe instrukcje wraz z pojawieniem się rozszerzenia SSE4 (2007 rok).

Jak zapowiada firma Intel, rozwój technologii przetwarzania wektorowego w procesorach x86 będzie kontynuowany w dalszym ciągu. Na rok 2010 planowane jest wprowadzenie rozszerzenia AVX (ang. Advanced Vector Extensions), które doda do architektury procesorów 16 nowych, 256-bitowych rejestrów YMM. Umożliwi to wykonywanie operacji na ośmioelementowych wektorach liczb typu SINGLE i czteroelementowych wektorach liczb typu DOUBLE. Dodatkowo jednostka AVX będzie posiadała specjalne instrukcje wspierające algorytm szyfrowania AES.

Umiejętne wykorzystanie jednostek wektorowych procesorów rodziny x86 może przynieść wielokrotne skrócenie czasu realizacji programów. Fakt ten powinien zostać dostrzeżony przez programistów, którzy przy tworzeniu kodu źródłowego powinni szukać fragmentów możliwych do zoptymalizowania za pomocą instrukcji SIMD.

Podręcznik jest przeznaczony dla studentów kierunku INFORMATYKA oraz kierunków pokrewnych, a nade wszystko dla programistów chcących optymalizować programy numeryczne pod względem szybkości realizacji. Jego celem jest zaprezentowanie architektury i podstaw programowania rozszerzeń wektorowych: MMX, SSE, SSE2 oraz SSE3 wprowadzonych w procesorach rodziny x86. Struktura podręcznika została podzielona na cztery części tematyczne, będące kolejnymi rozdziałami.

W rozdziale pierwszym przedstawiono przegląd podstawowych zagadnień z zakresu równoległych systemów komputerowych. Sklasyfikowano systemy rów-

noległe oraz przedstawiono problemy obliczeń równoległych, ze szczególnym uwzględnieniem problematyki obliczeń wektorowych. W charakterze przykładów architektur wektorowych przedstawiono komputery wektorowe firmy Cray, legendarne superkomputery produkowane w latach 80. oraz współczesne wielordzeniowe procesory CBEA, stosowane między innymi w konsolach PlayStation 3.

W rozdziale drugim zostały przedstawione podstawowe wiadomości z zakresu programowania rozszerzeń SIMD. W rozdziale zamieszczono opis formatów danych, na których rozszerzenia mogą wykonywać operacje oraz zbiór dostępnych instrukcji, pogrupowanych tematycznie.

Rozdział trzeci zawiera opis niekomercyjnych narzędzi informatycznych do edycji oraz translacji programów w asemblerze. Autorzy proponują zastosowanie translatora NASM oraz edytora SciTe for U3.

W rozdziale czwartym zawarto zasady programowania jednostek SIMD procesorów rodziny x86, opierając się o przykładowe programy realizujące wybrane operacje z zastosowaniem rozszerzeń wektorowych MMX, SEE, SSE2 i SSE3. Każdy przykład zawiera krótkie wprowadzenie do realizowanego przez program algorytmu, charakterystykę wybranych fragmentów programu źródłowego oraz demonstrację wyników działania programu. Porównano także szybkości realizacji programów wykorzystujących przetwarzanie wektorowe w stosunku do programów wykonujących obliczenia skalarne. Daje to odpowiedź na pytanie, jaki przyrost szybkości można uzyskać, wykorzystując rozszerzenia MMX, SSE i SSE2.

W rozdziale piątym zebrano instrukcje MMX, SSE, SSE2, SSE3 wraz z pseudokodem wyjaśniającym zasadę działania każdej instrukcji, natomiast rozdział szósty zawiera instrukcje SIMD pogrupowane, biorąc pod uwagę rozszerzenie, w którym zostały dodane.

1. OBLICZENIA RÓWNOLEGŁE

Obliczenia równoległe są formą obliczeń numerycznych, w których wiele operacji przeprowadza się jednocześnie przez równolegle pracujące jednostki wykonawcze. Możliwość przeprowadzania obliczeń równoległych wynika z zasady, że problemy numeryczne wymagające wielkiej liczby operacji mogą być z zasady podzielone na problemy o mniejszej złożoności, które z kolei można rozwiązywać równocześnie.

Obliczenia równoległe są stosowane od wielu lat, głównie w komputerach o dużej mocy obliczeniowej. Zainteresowanie obliczeniami równoległymi wzrosło znacznie w ostatnich latach ze względu na ograniczenia wynikające ze wzrostu częstotliwości pracy mikroprocesorów. Ponieważ zużycie energii przez komputery (i w konsekwencji generacja energii cieplnej) stało się w ostatnich latach kluczowym problemem, obliczenia równoległe są obecnie zasadniczym paradygmatem w architekturze współczesnych komputerów.

Skalowanie częstotliwościowe było głównym sposobem zwiększania wydajności komputerów, poczynając od połowy lat 80. do 2004 roku. Ponieważ czas wykonywania programu jest równy liczbie instrukcji pomnożonej przez średni czas wykonywania jednej instrukcji, przy założeniu pozostałych elementów stałych, zwiększając częstotliwość, zmniejszamy średni czas wykonywania instrukcji, co powoduje zmniejszenie czasu realizacji programu. Jednak zużycie mocy przez mikroprocesor, określone zależnością: $P=C*V^2*F$, gdzie C – współczynnik proporcjonalny do liczby przełączanych tranzystorów, V – napięcie zasilania, F – częstotliwość, oznacza, że wraz ze wzrostem częstotliwości rośnie zużycie energii przez procesor. Pociąga to za sobą wzrost ilości wydzielanego ciepła, co wiąże się z problemami chłodzenia procesorów. Stanowi to istotne ograniczenie dla wzrostu częstotliwości taktowania procesorów.

Elektroniczną złożoność procesorów określa prawo Moore'a wynikające z empirycznej obserwacji rozwoju mikroprocesorów. Prawo to mówi, że liczba tranzystorów w mikroprocesorach podwaja się co okres od 18 do 24 miesięcy. Mimo ograniczeń związanych z wydzielanym ciepłem, prawo Moore'a nadal obowiązuje. Osiąga się to poprzez wzrost liczby jednostek przeznaczonych do obliczeń równoległych.

1.1. FORMY RÓWNOLEGŁOŚCI

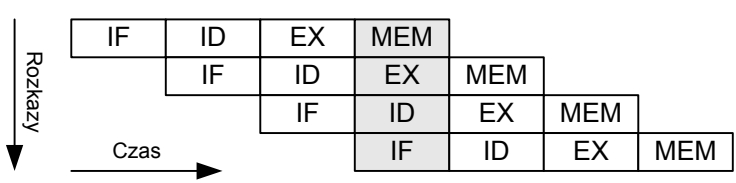
Wyróżnia się następujące formy równoległości:
– na poziomie bitów,
– na poziomie instrukcji,
– na poziomie danych,
– na poziomie zadań.

1.1.1. Równoległość na poziomie bitów

Wydłużenie słowa, na którym procesor wykonuje operacje, pociąga za sobą zmniejszenie liczby instrukcji do wykonania, dzięki czemu programy stają się krótsze. Np. operację dodawania liczb 16-bitowych procesor 8-bitowy musi zrealizować za pomocą dwóch operacji dodawania: najpierw ośmiu młodszych bitów, a następnie ośmiu starszych. Natomiast procesor operujący na słowie 16-bitowym wykonuje tę operację podczas wykonania jednego rozkazu. Mikroprocesory zwiększały długość słowa, poczynając od 4 bitów, poprzez 8, 16 i 32 bity. Mikroprocesory 32-bitowe stały się standardem w komputerach powszechnego użytku na okres około dwóch dekad. Poczynając od roku 2004, są wprowadzane do powszechnego użytku mikroprocesory 64-bitowe.

1.1.2. Równoległość na poziomie instrukcji

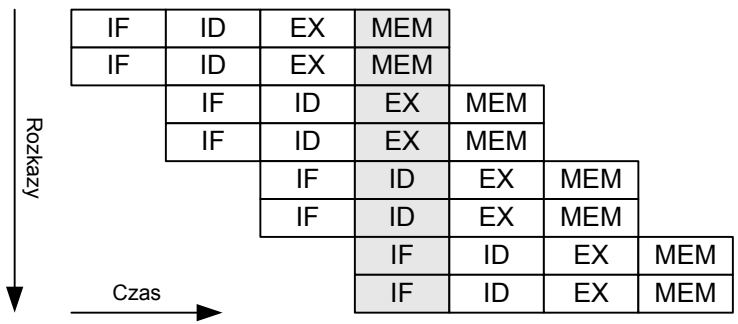
Równoległość tego rodzaju uzyskuje się poprzez potokową realizację rozkazów. Jeżeli np. w realizacji rozkazu wyróżnić cztery następujące fazy:
IF – Instruction Fetch (pobranie rozkazu),
ID – Instruction Decode (dekodowanie rozkazu),
EX – Execute (wykonanie),
MEM – Memory Access (dostęp do pamięci)
oraz istnieje możliwość równoległej pracy poszczególnych jednostek procesora realizujących wymienione fazy (rys. 1.1), to jednocześnie można w procesorze realizować cztery rozkazy (każdy rozkaz w innej fazie).



Rys. 1.1. Schemat potokowej pracy procesora

W takiej sytuacji, co prawda każdy rozkaz jest realizowany w czterech fazach (odpowiadających np. czterem taktom zegara), lecz co jeden takt zegara zostaje zrealizowany jeden rozkaz. W celu sprawnej potokowej realizacji rozkazów, mikroprocesor może zmieniać kolejność wprowadzania rozkazów do potoku, bez zmiany sensu programu. Praca potokowa zdominowała rozwój mikroprocesorów, poczynając od połowy lat 80. do połowy lat 90.

Kolejnym etapem w rozwoju mikroprocesorów była architektura superskalarna, polegająca na tym, że do procesorów wprowadzono szereg równoległych jednostek przetwarzających oraz tyle samo potoków. W ten sposób procesor w jednym takcie zegara może wykonać wiele rozkazów. Na rys. 1.2 przedstawiono schemat realizacji rozkazów przez procesor posiadający dwa potoki, a każdy rozkaz jest realizowany w czterech fazach.



Rys. 1.2. Schemat superskalarnej pracy procesora

1.1.3. Równoległość na poziomie danych

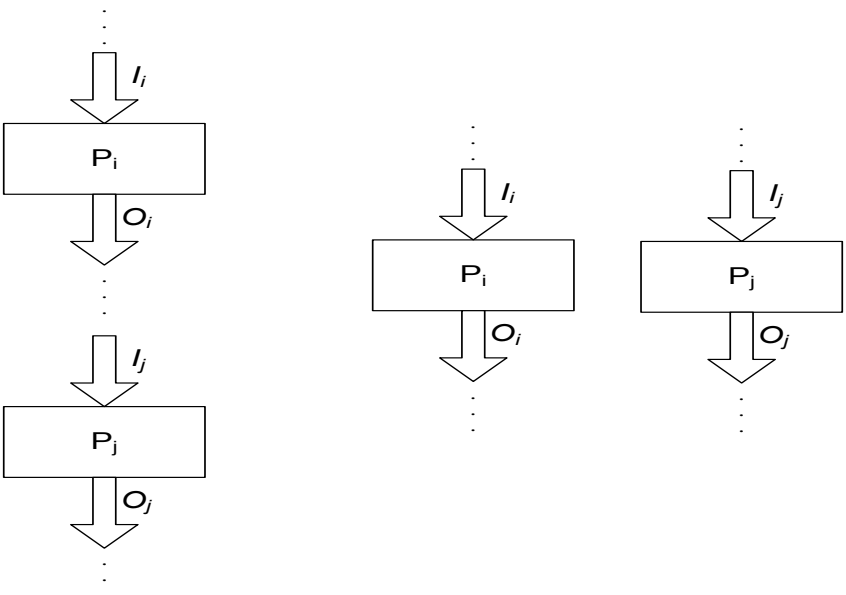
Równoległość danych ma bezpośredni związek z pętlami programowymi i skupia się na podziale przetwarzanych danych na mniejsze zbiory i ich przetwarzaniu w szeregu równoległych jednostek wykonawczych, w celu jednoczesnego wykonania operacji. Zrównoleglenie pętli często prowadzi do podobnych (niekoniecznie identycznych) sekwencji operacji lub funkcji, wykonywanych na poszczególnych podzbiorach dużej struktury danych. Wiele aplikacji naukowych oraz inżynierskich wykazuje równoległość danych.

Równoległość danych opisują warunki Bernsteina, określające czy dwa fragmenty programu mogą być wykonane równolegle. Niech P_i oraz P_j będą dwoma fragmentami programu oraz niech I_i stanowi zbiór zmiennych wejściowych, a O_i zbiór wielkości wyjściowych dla P_i oraz analogicznie dla P_j. P_i i P_j są niezależne, gdy spełniają następujące warunki (rys. 1.3):

– $I_j \cap O_i = \emptyset$,

– $I_i \cap O_j = \emptyset$,
– $O_i \cap O_j = \emptyset$.

Naruszenie pierwszego warunku wprowadza zależność przepływu, wyni-



Rys. 1.3. Sekwencyjnie i równolegle realizowane dwa fragmenty programu

kającą z faktu, że fragment programu P_j wykorzystuje wyniki wyznaczone przez fragment P_i. Warunek drugi reprezentuje antyzależność (ang. anti-dependency), gdy P_i nadpisuje zmienną używaną przez P_j. Trzeci warunek reprezentuje zależność wyjściową: gdy obydwa fragmenty programu zapisują wyniki w to samo miejsce, a ostateczny wynik jest rezultatem ostatniej operacji wykonanej przez jeden z fragmentów.

Rozpatrzmy następujące funkcje reprezentujące poszczególne typy zależności.

```
1: function f1(a, b)
2:     c := a·b
3:     d := 2·c
4: end function
```

Operacja trzecia w funkcji `f1(a,b)` musi być wykonana po operacji drugiej, gdyż wykorzystuje wynik polecenia drugiego. Narusza to pierwszy warunek i wprowadza zależność przepływu.

```
1: function f2(a, b)
```

```
2:       c := a·b
3:       d := 2·b
4:       e := a+b
5: end function
```

W funkcji `f2(a,b)` nie występuje zależność danych i wszystkie instrukcje mogą być wykonane równolegle.

Nie każde zrównoleglenie daje przyspieszenie obliczeń. Ponadto przy podziale procesu na znaczną liczbę równoległych wątków, powstaje sytuacja, gdy wiele wątków traci czas na komunikację między nimi. Powoduje to, że czas przeznaczony na komunikację między wątkami zaczyna dominować nad czasem niezbędnym do obliczeń. Określane jest to mianem spowolnienia równoległego (ang. parallel slowdown).

Jeżeli w pętli występuje zależność danych, charakteryzująca się zależnością danych wejściowych *i-tej* iteracji od danych wyjściowych poprzednich iteracji (*i-1, i-2,...*), to nie jest możliwe zrównoleglenie pętli. Np. wyznaczenie kilku pierwszych liczb ciągu Fibonacciego można zrealizować następująco:

```
1:     PREV2 := 0
2:     PREV1 := 1
3:     CUR := 1
4:     do:
5:        CUR := PREV1 + PREV2
6:        PREV2 := PREV1
7:        PREV1 := CUR
8:     while (CUR < 10)
```

Do wyznaczenia kolejnego elementu ciągu Fibonacciego niezbędne są dwa poprzednie elementy, co powoduje, że nie jest możliwe równoległe wyznaczenie elementów tego ciągu.

W systemie wieloprocesorowym realizującym jeden strumień instrukcji (SIMD), równoległość danych polega na tym, że każdy procesor wykonuje te same operacje na różnych częściach zbioru danych. W pewnych sytuacjach pojedynczy wątek przeprowadza operacje na wszystkich częściach danych. W innych, operacje wykonuje wiele wątków, lecz każdy z nich wykonuje te same operacje.

Dla przykładu, rozpatrzmy system 2-procesorowy (procesory A i B) w środowisku równoległym, wykonujący operacje na danych *d*. Załóżmy, że procesory będą wykonywać równoległe operacje na różnych częściach zbioru *d*. Jako konkretny przykład rozpatrzmy operację dodania dwóch wektorów. Dla implementacji równoległej załóżmy, że procesor A będzie wykonywał operacje na dolnej połowie wektorów, a procesor B na górnej połowie.

W poniższym przykładzie dwa procesory wykonują równoległe operacje *f* na dolnej i górnej połowie wektorów *d*.

```
if CPU = „A”
```

```
   lower_limit := 1
   upper_limit := round(d.length/2)
else if CPU = „B”
   lower_limit := round(d.length/2) + 1
   upper_limit := d.length

for i from lower_limit to upper_limit by 1
   f(d[i])
```

Przyspieszenie programu wynikłe z obliczeń równoległych jest ograniczone przez procentową część programu, która może być zrównoleglona. Dla przykładu, jeśli 90% programu może być zrównoleglone, teoretyczne maksymalne przyspieszenie wynikłe z obliczeń równoległych jest 10-krotne, niezależnie od liczby równolegle pracujących procesorów. Oznacza to, że nawet niewielka sekwencyjna część programu, która nie może być zrównoleglona, ogranicza możliwość przyspieszenia programu. Prawidłowość tę określa następująca zależność (prawo Amdahla):

$$S = \frac{1}{1-P}$$

gdzie: S – przyspieszenie programu, P – część programu, która może być zrównoleglona.

1.1.4. Równoległość na poziomie zadań

Ten typ równoległości daje możliwość jednoczesnej realizacji różnych obliczeń na tych samych lub różnych danych, w przeciwieństwie do równoległości danych, gdzie te same obliczenia są wykonywane na tych samych lub różnych zbiorach danych.

1.2. TAKSONOMIA FLYNNA

Michael J. Flynn stworzył (1966 rok) jedną z pierwszych klasyfikacji systemów do obliczeń równoległych. Flynn sklasyfikował komputery i programy, biorąc pod uwagę, czy podczas pracy wykonują jedną lub wiele instrukcji oraz czy te instrukcje przetwarzają jeden lub wiele zbiorów danych (tabela 1.1).

Systemy SISD to klasyczne komputery sekwencyjne, wykonujące w danej chwili jeden program przetwarzający jeden strumień danych. Odpowiada to powtarzającym się operacjom na poszczególnych elementach zbioru danych. Pozostałe punkty klasyfikacji dotyczą systemów równoległych. Systemy SIMD (ang. Single Instruction Multiple Data) to komputery wektorowe, realizujące rozkazy przetwa-

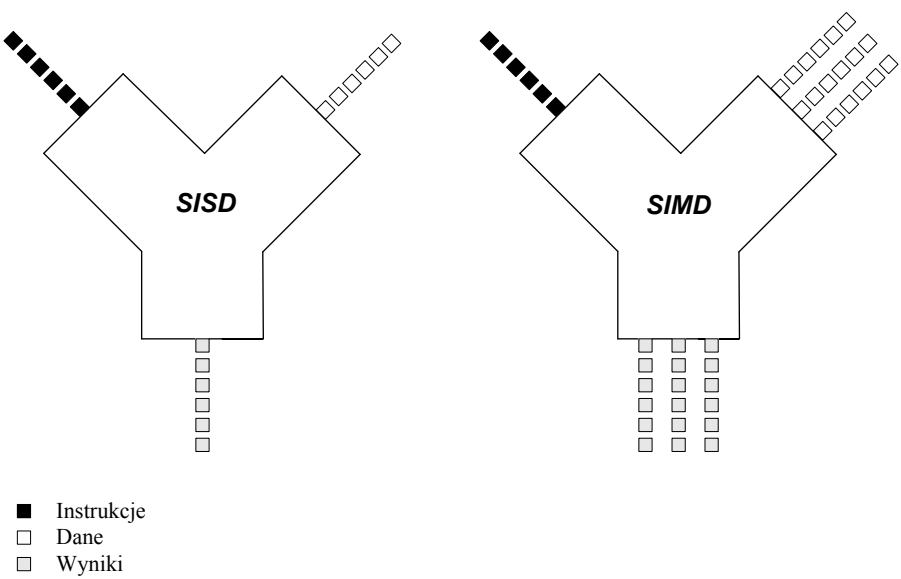
Tabela 1.1.

Taksonomia Flynna

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

rzające zbiory danych (wektor). W ten sposób zamiast realizować sekwencyjne operacje na poszczególnych elementach zbioru danych, komputer SIMD w jednym rozkazie wykonuje operację na zbiorze danych o określonej liczebności.

Systemy MIMD (ang. Multiple Instruction Multiple Data) wykonują równolegle wiele programów przetwarzających różne zbiory danych. Są to systemy wieloprocesorowe, w których każdy z procesorów realizuje inny program i przetwarza inne dane. W tej grupie można wyróżnić dwa rodzaje systemów komputerowych: wieloprocesory ze wspólną pamięcią (systemy silnie związane) oraz systemy z pamięcią rozproszoną, komunikujące się ze sobą za pośrednictwem sieci komputerowych. Systemy MISD (ang. Multiple Instruction Single Data) realizują równolegle wiele programów na tych samych danych i są współcześnie praktycznie nie stosowane. Współczesne systemy komputerowe są z zasady hybrydami powyższych kategorii, jednak klasyfikacja Flynna jest nadal powszechnie używana. Na rys. 1.4 przedstawiono logiczną organizację systemów SISD i SIMD.



Rys. 1.4. Logiczna budowa systemów SISD oraz SIMD

1.3. SYSTEMY SIMD

Główne korzyści wynikające z architektury SIMD osiągają aplikacje, w których te same operacje arytmetyczne lub logiczne, są wykonywane na dużych zbiorach danych. Ma to przede wszystkim miejsce w aplikacjach multimedialnych, gdzie na poszczególnych punktach obrazu lub próbkach dźwięku wykonuje się określone operacje. Przykładem może być zmiana jasności obrazu. Każdy piksel obrazu jest opisany trzema wartościami określającymi jasność kolorów składowych: czerwony, zielony, niebieski. Dla każdego punktu należy odczytać wartości RGB z pamięci, dodać lub odjąć określoną wartość, a wynik zapisać ponownie do pamięci.

Architektura SIMD na dwa sposoby przyspiesza ten proces.

Po pierwsze, odczyt i zapis danych do pamięci odbywa się blokowo i określona liczba danych zostaje odczytana/zapisana w jednej operacji. Po drugie, określona operacja arytmetyczna lub logiczna jest jednocześnie wykonywana na całym bloku danych, podczas wykonania jednej instrukcji.

Do podstawowych wad architektury SIMD można zaliczyć:

- nie wszystkie algorytmy mogą być wektoryzowane,
- implementacja algorytmów dla systemów SIMD wymaga udziału programisty. Większość współczesnych kompilatorów nie generuje instrukcji SIMD. Problem wektoryzacji programów jest aktualnie przedmiotem intensywnych badań naukowych.
- programowanie z zastosowaniem konkretnego zbioru instrukcji SIMD może powodować cały szereg wyzwań, np.:
  - SSE2 posiada ograniczenia związane z odpowiednim umieszczeniem w pamięci danych wektorowych; programista zaznajomiony z programowaniem procesorów rodziny x86 może być tym zaskoczony.
  - Lista instrukcji SIMD jest zależna od typu procesora; np. starsze procesory rodziny x86 nie posiadają SSE3, co powoduje, że programy powinny brać to pod uwagę. Podobnie kolejne wersje jednostek wektorowych firm INTEL i AMD nie są kompatybilne (AVX i SSE5).
  - Pierwsze wersje MMX wykorzystywały rejestry jednostki zmiennoprzecinkowej, co powodowało, że MMX nie mogło równolegle pracować z FPU.

Jednostki SIMD z krótkim wektorem (64- lub 128-bitów) zaczęto stosować w komputerach ogólnego przeznaczenia od roku 1989, kiedy to firma Digital Equipment Corporation w swoich komputerach VAX zastosowała układ Riegel, realizujący instrukcje wektorowe, a następnie w 1997 roku w procesorach Alpha (MVI – Motion Video Instructions).

Jednostki wektorowe można znaleźć w większości współczesnych procesorów. Poczynając od PowerPC (AltiVec oraz SPE) firmy IBM, PA-RISC (MAX – Multimedia Acceleration eXtensions) firmy HP, MMX, SSE, SSE2, SSE3 oraz SSSE3 firmy INTEL, 3DNow! firmy AMD, VIS firmy SPARC, MAJC firmy SUN. Konsorcjum

firm IBM, Sony, Toshiba współpracuje nad rozwojem procesorów komórkowych (ang. Cell procesors), w których jednostka SPU realizuje operacje wektorowe. Współczesne jednostki GPU (ang. Graphics Processing Unit) są jednostkami wektorowymi o długości wektora od 128- do 256-bitów.

Firmy zapowiadają, że w przyszłych procesorach będą stosować w coraz szerszym stopniu jednostki SIMD, np. AVX z wektorem 256-bitowym oraz Larrabee GPU z wektorem 512-bitowym (ang. VPU – Wide Vector Processing Unit) firmy INTEL.

Do klasycznych komputerów wektorowych zaliczamy superkomputery Cray, które w drugiej połowie lat 70. oraz w latach 80. posiadały najwyższą moc obliczeniową. W kolejnych latach bardzo wysoką mocą obliczeniową charakteryzowały się systemy MIMD, zawierające tysiące procesorów skalarnych, np. INTEL i860 XP, na skutek czego systemy SIMD na krótki czas straciły na znaczeniu. W latach 90. pojawiły się komputery osobiste, które osiągnęły moc obliczeniową wystarczającą do gier w czasie rzeczywistym. To spowodowało zapotrzebowanie na szczególny rodzaj mocy obliczeniowej związanej z przetwarzaniem grafiki. Pierwszym szeroko stosowanym układem SIMD był MMX opracowany dla procesorów rodziny x86. W następnym kroku MOTOROLA wprowadziła AltiVec do procesorów rodziny POWER. Obydwa rozwiązania były ukierunkowane na gry komputerowe w czasie rzeczywistym w celu przetwarzania grafiki i dlatego operowały na krótkich wektorach (od dwóch do czterech współrzędnych).

Współczesne superkomputery są niemal wyłącznie klastrami komputerów MIMD, w których każdy procesor implementuje układy SIMD o krótkim wektorze. Analogicznie, współczesne komputery osobiste zawierają kilka procesorów (procesory wielordzeniowe), z których każdy zawiera jednostkę SIMD operującą na krótkich wektorach.

W tabeli 1.2 przedstawiono historię wprowadzania układów SIMD w systemach mikroprocesorowych.

Przez długi czas jednostki SIMD dla procesorów rodziny x86 były praktycznie nie stosowane. Jednostki SIMD spowalniały komputery ze względu na używanie rejestrów przeznaczonych dla jednostki zmiennoprzecinkowej i co za tym idzie kosztowne przełączanie kontekstu. Jednostki SSE firmy INTEL oraz 3DNow! firmy AMD były niekompatybilne, co wywoływało dodatkowe problemy. Ponadto kompilatory języków wysokiego poziomu nie wspomagały obliczeń z użyciem SIMD (brak wektoryzacji), co wprowadzało konieczność programowania w języku asembler.

Aktualnie tak INTEL jak i AMD udostępniają zoptymalizowane biblioteki programów operujących na SIMD. Są także dostępne biblioteki open source, np. libSIMD czy SIMDx86. Firma Apple odniosła większy sukces, mimo że weszła na rynek SIMD później od pozostałych, wykorzystując AltiVec wchodzący w skład procesorów rodziny PowerPC oraz doskonałe kompilatory firm IBM, Motorola, a także GNU. Ponadto API oraz narzędzia deweloperskie (XCode) firmy Apple zostały przeniesione na procesory rodziny x86 wykorzystujące SSE2 i SSE3.

Tabela 1.2.

Chronologia jednostek SIMD w powszechnie stosowanych mikroprocesorach

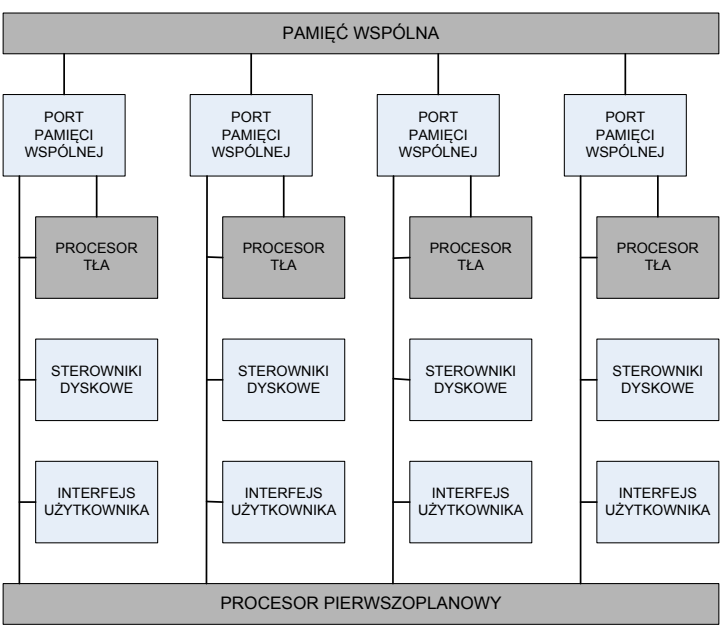
Rok	Firma	Procesor	Nazwa	Komentarz
1996	INTEL	PENTIUM	MMX	Z powodu wielu ograniczeń technicznych nie uzyskał szerokiego zastosowania
1997	MOTOROLA	PowerPC G3 PowerPC G4	AltiVec	Uzyskuje szerokie zastosowanie, szczególnie na komputerach Apple
1997	INTEL	PENTIUM 2	MMX2	Usuwa szereg ograniczeń MMX
1999	INTEL	PENTIUM 3	SSE	Szereg kolejnych usprawnień MMX. Szersze zastosowanie niż MMX, lecz znacznie mniejsze niż AltiVec
2000?	AMD	Athlon	3DNow!	Rozszerzenie MMX, bardzo podobne do SSE. Niewielkie zastosowanie
2002	INTEL	PENTIUM 4	SSE2	Sukces firmy INTEL. Bardzo szerokie zastosowanie w aplikacjach multimedialnych.
2002	IBM	PowerPC G5	AltiVec	Zbiór instrukcji pozostaje bez zmian. Zwiększenie wydajności
2003	AMD		3DNow2!	Lista instrukcji upodobniona do SSE2. Nie odnosi większego sukcesu.
2004	INTEL	PENTIUM 4	SSE3	Szeroki i elastyczny zbiór instrukcji. Poprawiona wydajność. Wykazuje większą wydajność niż AltiVec
2004	AMD	Athlon 64	SSE SSE2	Firma wdraża rozwiązania INTEL-a do swoich procesorów.
2007	INTEL	Intel Core 2 Intel Core i7	SSE4	Rozszerzenie listy instrukcji wspomagających kompresję video, wyznaczenie sumy kontrolnej CRC-32, operacje na łańcuchach znaków
2010	INTEL		AVX	16 rejestrów 256-bitowych YMM0… YMM15, z możliwością rozbudowy do 512 bitów, operacje 3-argumentowe

1.4. ARCHITEKTURA KOMPUTERÓW CRAY

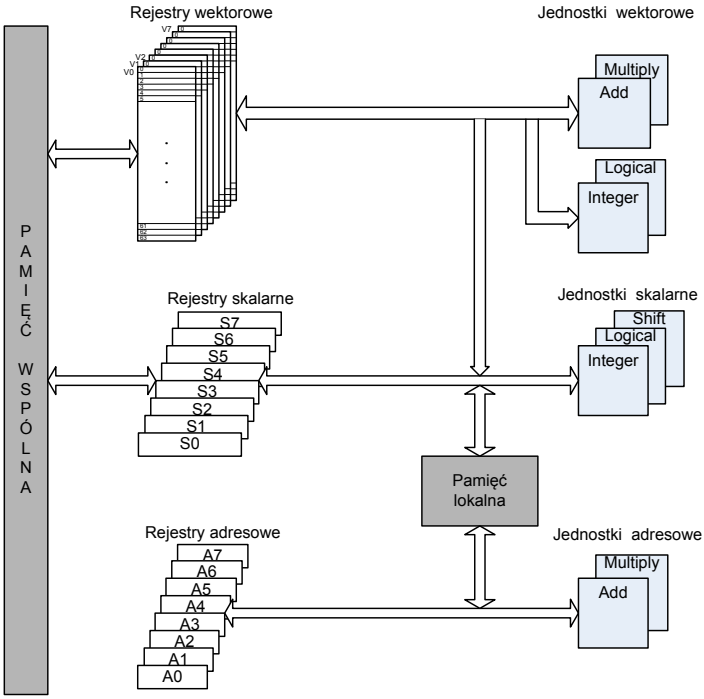
Koncepcja komputerów wektorowych została rozwinięta i wdrożona przez firmę Cray Research Inc. w drugiej połowie lat 70. oraz w latach 80., a komputery tej firmy przez szereg lat były komputerami o najwyższej mocy obliczeniowej. Architekturę tych komputerów prześledzimy na przykładzie komputera Cray-2, który w latach 1985-1989 był systemem o najwyższej mocy obliczeniowej (3.9 GFlops).

System zawierał cztery niezależne procesory wektorowe (ang. Background Procesors) taktowane z cyklem 4.1 ns, które mogły pracować niezależnie, wykonując odrębne zadania, lub wspólnie realizować jedno zadanie oraz procesor pierwszoplanowy zarządzający całym systemem w oparciu o UNIX system V opracowany przez AT&T Bell Labs. Pamięć wspólna dla wszystkich procesorów zawiera 256 M słów 64-bitowych (2GB) z równoległym dostępem przez wszystkie procesory. Wektorowa jednostka wykonawcza realizuje jednocześnie dodawanie i mnożenie 64-elementowych wektorów zmiennoprzecinkowych oraz operacje stałoprzecinkowe na wektorach 64-elementowych, wykorzystując osiem 64-elementowych rejestrów wektorowych V0…V7. Równolegle funkcjonuje jednostka skalarna wykorzystująca osiem rejestrów skalarnych S0…S7. Ogólną architekturę komputerów Cray-2 przedstawiono na rys. 1.5.

Każdy procesor tła zawiera osiem 64-elementowych rejestrów wektorowych V0…V7, w których każdy element jest słowem 64-bitowym, osiem 64-bitowych rejestrów skalarnych S0…S7, osiem 32-bitowych rejestrów adresowych A0…A7 oraz pamięć lokalną o pojemności 16 k słów 64-bitowych. Uproszczony schemat blokowy procesora wektorowego (procesora tła) przedstawiono na rys. 1.6. Procesor pierwszoplanowy (ang. Foreground Procesor) odpowiada za zarządzanie całym systemem oraz diagnostykę i realizuje oprogramowanie systemowe. Moc zasilania systemu Cray-2 wynosiła 195 kW, co wymuszało zastosowanie intensywnego chłodzenia cieczowego.



Rys. 1.5. Ogólna budowa komputera CRAY-2

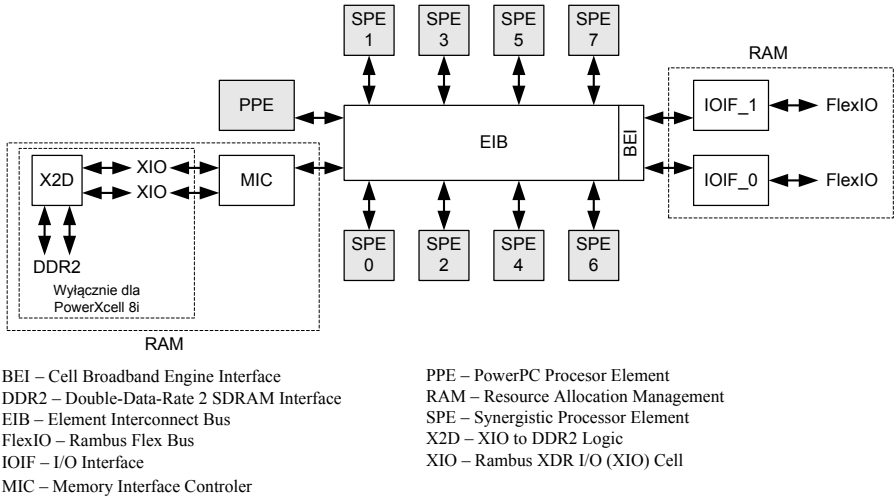


Rys. 1.6. Uproszczony schemat jednostek wykonawczych procesora tła (ang. Background Procesor) komputera CRAY-2

1.5. ARCHITEKTURA PROCESORÓW CBEA

Innym przykładem procesorów wykorzystujących jednostki wektorowe są procesory Cell/B.E oraz PowerXCell 8i określane mianem procesorów CBEA (ang. Cell Broadband Engine Architecture), opracowane przez trzy firmy: Sony, Toshiba, IBM. Architektura CBEA jest rozszerzeniem 64-bitowej architektury PowerPC. Procesory Cell/B.E zostały opracowane dla aplikacji multimedialnych (konsole gier komputerowych, telewizja HD) i operują na krótkich 4-elementowych wektorach zmiennoprzecinkowych w formacie SINGLE oraz wektorach stałoprzecinkowych. Natomiast procesory PowerXCell 8i operują na 2-elementowych wektorach zmiennoprzecinkowych w formacie DOUBLE. Obydwa procesory CBEA zawierają osiem jednostek wektorowych SPE (ang. Synergistic Processor Element) oraz jedną 64-bitową jednostkę PowerPC - PPE (ang. PowerPC Processor). Uproszczony schemat blokowy procesorów CBEA przedstawiono na rys. 1.7.

Jednostka wykonawcza PPE jest oparta na 64-bitowym procesorze PowerPC i może realizować 32- i 64-bitowe oprogramowanie systemowe oraz aplikacje. Natomiast jednostki wykonawcze SPE zostały opracowane z myślą o realizacji aplikacji



Rys. 1.7. Schemat blokowy procesorów CBEA

SIMD, w których są wykonywane masywne obliczenia wektorowe. Poszczególne SPE pracują niezależnie od siebie (realizują odrębne wątki) i korzystają z pamięci wspólnej oraz odwzorowanych w pamięci układów I/O. Dla programisty, procesor CBEA przedstawia się jako jednostka 9-procesorowa, w której najczęściej PPE realizuje wątek główny, a poszczególne SPE realizują wątki wtórne, wykonując masywne obliczenia wektorowe.

Główna różnica między jednostkami PPE i SPE polega na dostępie do pamięci. Jednostka PPE ma bezpośredni dostęp do całej pamięci wspólnej z wykorzystaniem pamięci CACHE, przesyłając dane z pamięci do rejestrów procesora i odwrotnie. Natomiast jednostki SPE mają dostęp do pamięci głównej za pośrednictwem układów DMA (ang. Direct Memory Access), które przenoszą dane i programy między pamięcią główną a pamięciami lokalnymi (ang. Local Storage) związanymi z poszczególnymi SPE. Jednostki SPE operują wyłącznie na swoich pamięciach lokalnych, nie wyposażonych w pamięci CACHE. Ta trójpoziomowa organizacja pamięci (rejestry procesora, pamięć LS, pamięć główna), z wykorzystaniem asynchronicznych przesłań DMA pomiędzy pamięcią główną a pamięciami LS, daje możliwość zrównoleglenia operacji przesłań danych i operacji obliczeniowych i jest istotnym odejściem od klasycznych architektur.

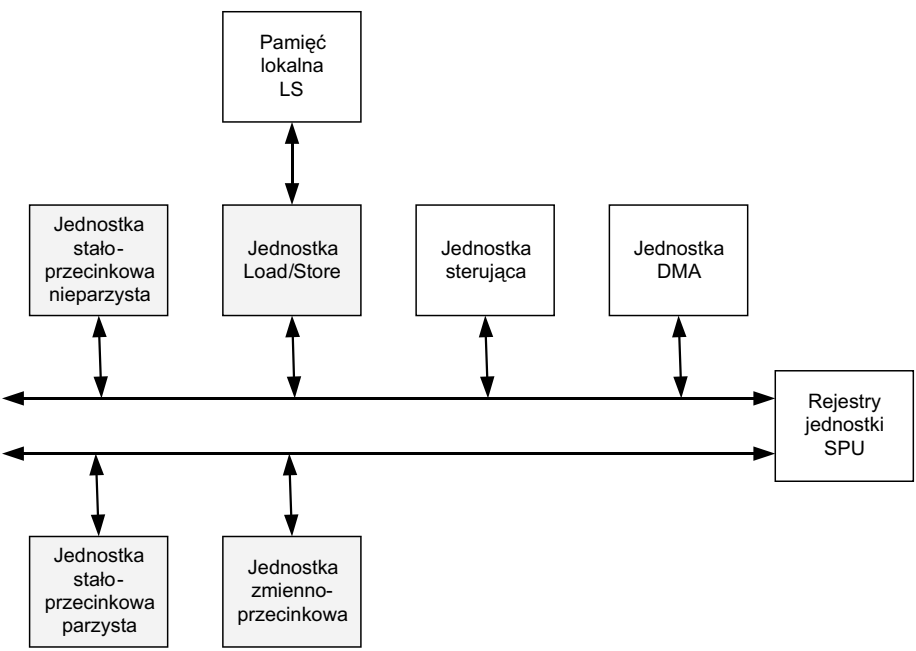
We współczesnych systemach komputerowych szybkość przesłań danych między pamięcią a procesorem stanowi podstawowe ograniczenie wydajności aplikacji (a nie wydajność procesora). W przypadku, gdy w klasycznym systemie ma miejsce chybienie (ang. Miss) podczas odczytu z pamięci CACHE, realizacja programu musi zostać wstrzymana nawet na okres kilkuset taktów zegara (w celu odczytu brakujących danych z pamięci głównej). Powoduje to istotne spowolnienie

aplikacji. W przypadku procesorów CBEA, do uruchomienia kanału DMA potrzeba kilku taktów zegara. Każdy SPE jest wyposażony w kontroler DMA zawierający 16 kanałów.

W skład jednostki SPE wchodzą dwa elementy składowe:
– SPU (ang. Synergistic Procesor Unit), 128-bitowa jednostka wykonawcza RISC przeznaczona do obliczeń skalarnych oraz SIMD,
– MFC (ang. Memory Flow Controller), układ zarządzający przesłaniami danych.

SPU jest niezależnym procesorem z własnym licznikiem rozkazów, który pobiera instrukcje oraz zapisuje i pobiera dane z pamięci lokalnej o pojemności 256 kB. Jednostka ponadto zawiera:
– 128 rejestrów 128-bitowych,
– cztery jednostki wykonawcze,
– interfejs DMA służący do transmisji danych między pamięcią główną i pamięcią lokalną oraz do komunikacji z PPE i pozostałymi SPE.

Główne jednostki wchodzące w skład SPU przedstawiono na rys. 1.8.



Rys. 1.8. Główne jednostki SPU

Jednostka zmiennoprzecinkowa wykonuje operacje SIMD na danych całkowitoliczbowych oraz zmiennoprzecinkowych w formacie SINGLE oraz DOUBLE (IEEE 754). Dla formatu SINGLE wektor zawiera 4 elementy (128 bitów), a mak-

symalna wydajność dla SPE wynosi ok. 25 GFLOPS, co daje ok. 200 GFLOPS dla procesora CBEA. Dla formatu DOUBLE wektor zawiera 2 elementy, a maksymalna wydajność dla SPE wynosi ok. 14 GFLOPS, co daje ok. 100 GFLOPS dla procesora CBEA (w wersji PowerXcell 8i).

Można przedstawić dwa scenariusze organizacji obliczeń dla procesorów CBEA. Pierwszy z nich polega na łańcuchowym (potokowym) wykonywaniu kolejnych prostych operacji na danych przez kolejne jednostki SPE. W takim przypadku każdy SPE wykonuje wątek realizujący prostą operację na danych, przekazując kolejnemu SPE wynik. Taki scenariusz jest szczególnie przydatny dla przetwarzania danych multimedialnych. Drugi scenariusz polega na podziale danych na podzbiory, przetwarzane równolegle przez poszczególne SPE.

1.6. WEKTORYZACJA OBLICZEŃ

Przez wektoryzację rozumiemy proces konwersji programu komputerowego realizującego operacje sekwencyjne na danych skalarnych do programu wektorowego, w którym pojedyncza instrukcja może wykonywać wielokrotne operacje na parach wektorów. Poszukiwanie metod automatycznej wektoryzacji (bez udziału człowieka) to jeden z kierunków badań w informatyce. Automatyczna wektoryzacja jest wykonywana przez program, który dokonuje konwersji programu sekwencyjnego, przetwarzając ciągi operacji wykonywanych sekwencyjnie na jedną operację wykonywaną równolegle, odpowiednio do architektury procesora wektorowego.

Przykładem może być następujący program mnożenia poszczególnych elementów dwóch wektorów:

```
for (i = 0; i < 1024; i++)
 {
    C[i] = A[i]*B[i];
 }
```

Program ten może zostać przekształcony do następującej postaci wektorowej przy założeniu, że procesor realizuje operacje na wektorach o długości 4:

```
for (i = 0; i < 1024; i+=4)
 {
    C[i:i+3] = A[i:i+3]*B[i:i+3];
 }
```

Zapis C[i:i+3] oznacza tablicę czteroelementową od C[i] do C[i+3]. Jeśli jednostka wektorowa wykonuje operacje z tą samą szybkością co jednostka skalarna, to w wyniku wektoryzacji można w tym przypadku oczekiwać 4-krotnego przyspieszenia obliczeń.

Zdecydowana większość kompilatorów realizujących operacje automatycznej wektoryzacji stosuje wektoryzację na poziomie pętli. Ta technika, używana już dla

klasycznych komputerów wektorowych, próbuje odnaleźć i eksploatować równoległość SIMD na poziomie pętli. Metoda zawiera dwa główne kroki:
1. odnalezienie najgłębiej zagnieżdżonej pętli, która może być wektoryzowana,
2. transformacja pętli i generacja programu wektorowego.

W pierwszym kroku kompilator wektoryzujący poszukuje trudności, które mogą uniemożliwić wektoryzację. Głównym problemem jest występowanie zależności danych, krótszych od długości wektora.

Po stwierdzeniu, że pętla może być wektoryzowana, każda instrukcja skalarna wewnątrz pętli jest zastępowana przez odpowiednią instrukcję wektorową. Kolejne kroki transformacji powyższego programu są następujące:

– Paskowanie (ang. Stripmining)

```
for (i = 0; i < 1024; i+=4)
{
  for (ii = 0; ii < 4; ii++)
  {
     C[i+ii] = A[i+ii]*B[i+ii];
  }
}
```

– Rozłożenie pętli z użyciem tablic tymczasowych

```
for (i = 0; i < 1024; i+=4)
{
  for (ii = 0; ii < 4; ii++) tA[ii] = A[i+ii];
  for (ii = 0; ii < 4; ii++) tB[ii] = B[i+ii];
  for (ii = 0; ii < 4; ii++) tC[ii] = tA[ii]*tB[ii];
  for (ii = 0; ii < 4; ii++) C[i+ii] = tC[ii];
}
```

– Zastąpienie przez instrukcje wektorowe

```
for (i = 0; i < 1024; i+=4)
{
  vA = vec_ld( &A[i] );
  vB = vec_ld( &B[i] );
  vC = vec_mul( vA, vB );
  vec_st( vC, &C[i] );
}
```

Wektoryzację komplikują w znacznym stopniu instrukcje warunkowe występujące wewnątrz pętli. Przy obecności instrukcji warunkowych w ciele pętli, muszą być wykonane instrukcje występujące we wszystkich rozgałęzieniach, aby następnie połączyć wielokrotne wartości zmiennych. Ogólne podejście do rozwiązania tego problemu polega na wykonaniu następujących transformacji kodu: predykcja → wektoryzacja → usunięcie wektorowych predykatów → usunięcie skalarnych predykatów.

Dla przykładowego programu kolejne przekształcenia mogą przyjmować następującą postać:

```
for (i = 0; i < 1024; i++)
 {
    if (A[i] > 0)
      C[i] = B[i];
    else
      D[i] = D[i-1];
 }
```

– Predykcja

```
 for (i = 0; i < 1024; i++)
 {
    P = A[i] > 0;
    NP = !P;
    C[i] = B[i];      (P)
    D[i] = D[i-1];    (NP)
 }
```

gdzie `P` i `NP` oznacza predykat związany z określonym poleceniem (predykat jest zmienną logiczną określającą, że dana instrukcja ma zostać wykonana, gdy predykat z nią związany ma wartość TRUE).

– Wektoryzacja

```
 for (i = 0; i < 1024; i+=4)
 {
    vP  = A[i:i+3] > (0,0,0,0);
    vNP = vec_not(vP);
    C[i:i+3] = B[i:i+3];      (vP)
    (NP1,NP2,NP3,NP4) = vNP;
    D[i]   = D[i-1];          (NP1)
    D[i+1] = D[i];            (NP2)
    D[i+2] = D[i+1];          (NP3)
    D[i+3] = D[i+2];          (NP4)
 }
```

gdzie `vP` i `vNP` oznaczają predykaty wektorowe.

– Usunięcie predykatów wektorowych)

```
 for (i = 0; i < 1024; i+=4)
 {
    vP  = A[i:i+3] > (0,0,0,0);
    vNP = vec_not(vP);
    C[i:i+3] = vec_sel(C[i:i+3],B[i:i+3],vP);
    (NP1,NP2,NP3,NP4) = vNP;
    D[i]   = D[i-1];          (NP1)
```

```
    D[i+1] = D[i];            (NP2)
    D[i+2] = D[i+1];          (NP3)
    D[i+3] = D[i+2];          (NP4)
}
```

gdzie `vec_sel` jest funkcją wektorową, której wartością jest wektor o składowych równych składowym pierwszego lub drugiego argumentu funkcji, w zależności od wartości predykatu wektorowego.

– Usunięcie predykatów skalarnych

```
for (i = 0; i < 1024; i+=4)
{
    vP  = A[i:i+3] > (0,0,0,0);
    vNP = vec_not(vP);
    C[i:i+3] = vec_sel(C[i:i+3],B[i:i+3],vP);
    (NP1,NP2,NP3,NP4) = vNP;
    if (NP1) D[i]   = D[i-1];
    if (NP2) D[i+1] = D[i];
    if (NP3) D[i+2] = D[i+1];
    if (NP4) D[i+3] = D[i+2];
}
```

Konieczność wykonania instrukcji umieszczonych we wszystkich rozgałęzieniach programu jest główną przyczyną spowolnienia programów wektorowych. Spowolnienie to jest tym większe, im więcej instrukcji warunkowych w programie oraz im więcej instrukcji znajduje się w gałęziach niewykonywanych.

1.7. PODSUMOWANIE

Od szeregu lat w bardzo szybkim tempie rośnie zapotrzebowanie na moc obliczeniową komputerów. Ze względu na ograniczenia w szybkości pracy procesorów, wynikające z ilości wydzielanego ciepła, obliczenia równoległe dają możliwość zaspokojenia rosnącego zapotrzebowania. Z tego powodu obliczenia równoległe zyskują bardzo na znaczeniu.

Według taksonomii Flynna, do zasadniczych architektur równoległych zaliczamy systemy SIMD oraz MIMD. Systemy SIMD to znane od dziesiątków lat komputery wektorowe, wykonujące operacje na zbiorach danych, uporządkowanych w postaci n-elementowych wektorów. Komputery o takiej architekturze są szczególnie przydatne do przetwarzania danych numerycznych, zorganizowanych w postaci macierzy i wektorów. Takie struktury danych występują podczas różnego rodzaju obliczeń naukowych oraz technicznych.

Systemy SIMD są także niezbędne do przetwarzania danych multimedialnych w czasie rzeczywistym (grafika 3-wymiarowa, telewizja cyfrowa wysokiej roz-

dzielczości, przetwarzanie dźwięku), a także do sprawnego kodowania danych. W kolejnych rozdziałach przedstawiono problematykę architektury i programowania systemów SIMD na przykładzie jednostek MMX, SSE, SSE2, SSE3 procesorów rodziny x86.

Jeżeli prawo Moore'a będzie nadal obowiązywać, to co okres maksymalnie 2 lat będzie się podwajać liczba tranzystorów w mikroprocesorach. Z dotychczasowego rozwoju mikroprocesorów można wnioskować, że spowoduje to dalszą rozbudowę procesorów wielordzeniowych, a także jednostek wektorowych SIMD. Pociągnie to za sobą konieczność tworzenia oprogramowania równoległego tak na architektury MIMD, jak i na architektury SIMD. Niestety poziom edukacji informatycznej nie nadąża za tak szybkim rozwojem sprzętu komputerowego. Na polskim rynku wydawniczym brakuje publikacji poświęconych systemom SIMD.

Autorzy założyli, że czytelnik zna podstawową architekturę oraz podstawy programowania procesorów rodziny x86 z wykorzystaniem języka asembler. Czytelnik zainteresowany programowaniem w asemblerze ma do dyspozycji szereg książek w języku polskim, poświęconych różnym aspektom programowania [1, 4, 13, 14, 22, 27]. Niestety żadna z nich nie porusza problematyki jednostek wektorowych. Przedstawiony podręcznik może w niewielkim stopniu przyczyni się do wypełnienia tej luki.

2. ROZSZERZENIA WEKTOROWE PROCESORÓW X86

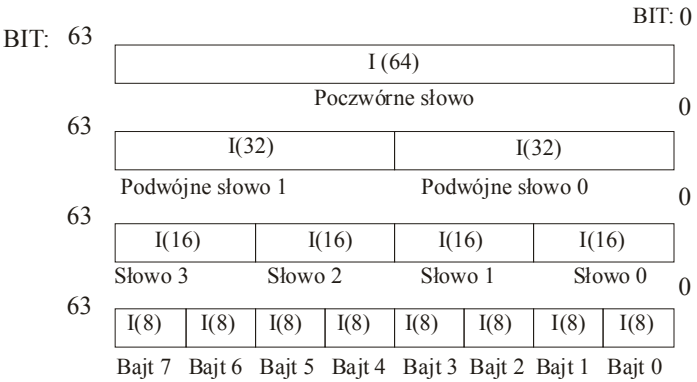
W rozdziale można wyróżnić dwie części. W pierwszej części zaprezentowano formaty danych wykorzystywane przez jednostki wektorowe. W drugiej części natomiast przedstawiono architektury, możliwości, a także listy instrukcji dla poszczególnych rozszerzeń SIMD.

2.1. STRUKTURY DANYCH

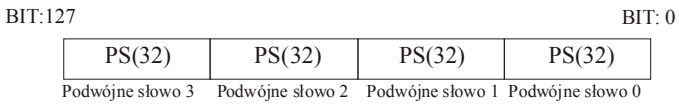
Przetwarzanie wektorowe niesie za sobą konieczność zdefiniowania nowych typów danych, stanowiących N – elementowe tablice, złożone z danych elementarnych (tzw. dane upakowane). W zależności od rodzaju rozszerzenia SIMD mamy możliwość operowania na następujących typach danych:

- Dla jednostki MMX: 64-bitowe, upakowane dane całkowite bez lub ze znakiem. W zależności od rozmiaru danej elementarnej (elementu tablicy) można wyróżnić cztery typy danych:
  - pojedyncza 64-bitowa liczba całkowita,
  - dwuelementowa tablica podwójnych słów (dwie 32-bitowe liczby całkowite),
  - czteroelementowa tablica słów (cztery 16-bitowe liczby całkowite),
  - ośmioelementowa tablica bajtów (osiem 8-bitowych liczb całkowitych).

  Poszczególne typy danych, wykorzystywane przez jednostkę MMX przedstawiono na rys. 2.1.
- Dla jednostki SSE: 128-bitowe, upakowane dane zmiennopozycyjne pojedynczej precyzji (ang. PS – Packed Single), zawierające czteroelementową tablicę 32-bitowych danych typu SINGLE (rys. 2.2).
- Dla jednostki SSE2: 128–bitowe upakowane dane zmiennopozycyjne podwójnej precyzji (ang. PD – Packed Double) oraz 128-bitowe upakowane dane całkowite bez znaku i ze znakiem. Rozszerzenie SSE2 umożliwia wykonywanie operacji na dwuelementowych wektorach typu DOUBLE. Dla liczb całkowitych, rozszerzenie SSE2 wspiera następujące cztery typy danych:
  - dwuelementowa tablica poczwórnych słów (dwie 64-bitowe liczby całkowite),
  - czteroelementowa tablica podwójnych słów (cztery 32–bitowe liczby całkowite),



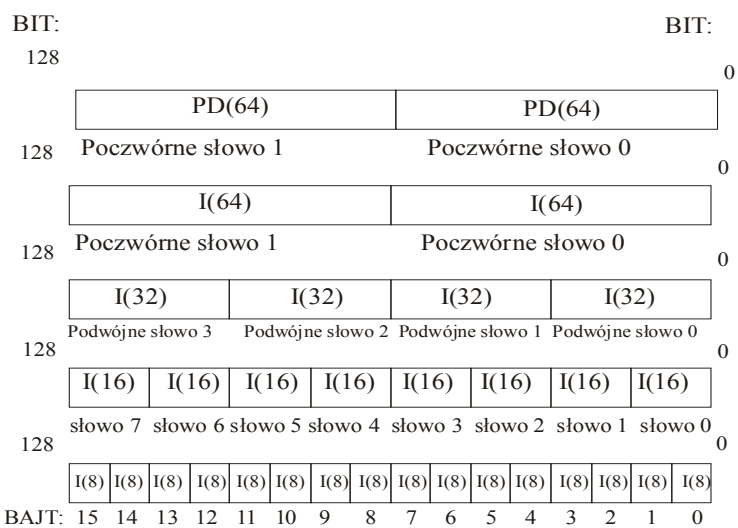
Rys. 2.1. Formaty danych wykorzystywane przez MMX



Rys. 2.2. Format danych jednostki SSE

– ośmioelementowa tablica słów (osiem 16–bitowych liczb całkowitych),
– szesnastoelementowa tablica bajtów (szesnaście 8–bitowych liczb całkowitych).

Na rys. 2.3 przedstawiono formaty danych całkowitoliczbowych używane w rozszerzeniu SSE2.



Rys. 2.3. Format danych rozszerzenia SSE2

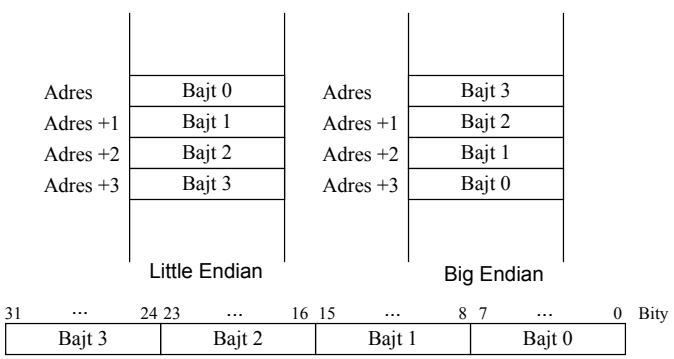
2.1.1. Dane typu całkowitoliczbowego

Liczby całkowite bez znaku, reprezentujące liczby naturalne, stanowią elementarne dane wykorzystywane w systemach komputerowych. Podstawową daną jest pojedynczy bajt, stanowiący 8-pozycyjną liczbę binarną bez znaku. Poszczególne bity liczby są numerowane, poczynając od *0* (bit najmłodszy) do *7* (bit najstarszy), a wartość dziesiętną liczby wyznacza się, sumując wagi pozycji posiadających wartość *1*, jak np.:

$$(10011101)_2 = 1*2^7+1*2^4+1*2^3+1*2^2+1*2^0 = (157)_{10}.$$

Wartości liczb bez znaku zapisanych w jednym bajcie mieszczą się w przedziale 0–255. Poza danymi typu `BYTE`, w systemach komputerowych spotykamy się z danymi złożonymi z wielu bajtów, a mianowicie:
`WORD` (słowo) – liczba 16-bitowa bez znaku, składająca się z dwóch bajtów; wartość liczby mieści się w przedziale $0–2^{16}$-1, czyli $0–65535_{10}$, lub `0000h–FFFFh`.
`DOUBLEWORD` (podwójne słowo) – liczba 32-bitowa bez znaku, składająca się z czterech bajtów; wartość liczby mieści się w przedziale $0–2^{32}$-1, lub `00000000h–FFFFFFFFh`.
`QUADWORD` (poczwórne słowo) – liczba 64-bitowa bez znaku, składająca się z ośmiu bajtów; wartość liczby mieści się w przedziale $0–2^{64}$-1.

Przy zapisie liczb wielobajtowych, w systemach komputerowych stosowane są dwie różne konwencje, związane z kolejnością poszczególnych bajtów w zapisie liczb, określane mianem Little Endian, oraz Big Endian. Kolejność określana mianem Little Endian stosowana jest przez firmę INTEL i polega na zapisie bajtu najmłodszego pod najniższym adresem, a następnie bajtów starszych (rys. 2.4). W przypadku Big Endian (stosowanym między innymi przez firmę MOTOROLA) w pierwszej kolejności zapisywany jest bajt najstarszy.



Rys. 2.4. Zapis podwójnego słowa w pamięci komputera przy zastosowaniu różnych konwencji

Do zapisu liczb całkowitych ze znakiem stosowane są w systemach komputerowych trzy kody: znak–moduł, kod spolaryzowany oraz kody uzupełnieniowe.

Kod „znak–moduł" polega na zapisie znaku liczby (najstarszy bit liczby, zgodnie z zasadą `0` – liczba dodatnia, `1` – liczba ujemna) oraz zapisie wartości bezwzględnej liczby na pozostałych bitach:

$$X = (-1)^s \sum_{i=0}^{N-2} x_i 2^i \,,$$

gdzie *s* – bit znaku liczby.

Np. 8-bitowa liczba o wartości -100_{10} ma następującą postać: $(11100100)_2$.

W przypadku kodu spolaryzowanego, kod liczby ze znakiem uzyskuje się poprzez dodanie do liczby wartości stałej, stanowiącej przesunięcie zera (ang. bias), która wynosi $B=2^{N-1}-1$:

$$X = -x_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i$$

Np. 8-bitowa liczba o wartości -100_{10} ma w tym przypadku następującą postać: $(00011011)_2$. Kody spolaryzowane są stosowane przede wszystkim do zapisu wykładnika liczb zmiennopozycyjnych.

Najczęściej stosowanymi kodami do zapisu liczb całkowitych ze znakiem są kody uzupełnieniowe. Stosowane są dwa kody uzupełnieniowe: kod uzupełnieniowy do 1 (U1) oraz kod uzupełnieniowy do 2 (U2). Aktualnie używany jest powszechnie kod uzupełnieniowy do 2 i z tego powodu ograniczymy się jedynie do omówienia tego kodu. W kodzie U2 najstarszy bit liczby oznacza znak, według zasady: 0 – liczba dodatnia, 1 – liczba ujemna. W przypadku liczby dodatniej, zapis $\{0,x_{N-2},x_{N-3},...,x_0\}$ reprezentuje wartość liczby, natomiast zapis $\{1,x_{N-2},x_{N-3},...,x_0\}$ oznacza kod uzupełnieniowy liczby ujemnej. Wartość liczby całkowitej w kodzie U2 wyznacza się z następującej zależności:

$$X = -x_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i \cdot$$

Zmiana znaku liczby zapisanej w kodzie U2 na przeciwny może polegać na zanegowaniu wszystkich bitów liczby i dodaniu 1 do pozycji najmłodszej. Użyteczność kodów uzupełnieniowych wynika z faktu, że podczas operacji arytmetycznych dodawania i odejmowania, bit znaku liczby traktowany jest tak samo jak wszystkie pozostałe bity liczby. Oznacza to, że operacje dodawania i odejmowania wykonuje się także na bitach znaku.

W tabeli 2.1 przedstawiono przedziały wartości liczb całkowitych wykorzystywanych przez jednostki SIMD.

Tabela 2.1.

Zakresy wartości liczb całkowitych

Długość bitowa:	Dla liczby bez znaku	Dla liczby ze znakiem w kodzie U2
8	Od 0 do 255	od -128 do +127
16	Od 0 do 65535	Od -32768 do +32767
32	od 0 do 4294967295	od -2147483648 do +2147483647
64	od 0 do 18446744073709551615	od -9223372036854775808 do +9223372036854775807

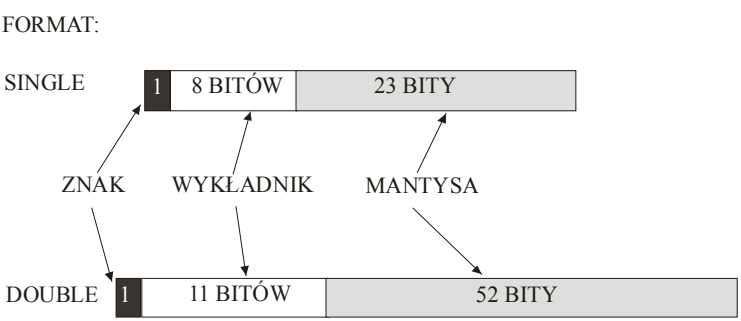
2.1.2. Dane typu zmiennopozycyjnego

Jednostki wektorowe SSE, SSE2 i SSE3 umożliwiają wykonywanie operacji na wektorach następujących liczb zmiennopozycyjnych:

– 32–bitowa liczba zmiennopozycyjna, określona standardem IEEE-754 SINGLE,
– 64–bitowa liczba zmiennopozycyjna, określona standardem IEEE-754 DOUBLE.

Reprezentacje SINGLE oraz DOUBLE składają się z trzech pól:

– bitu znaku umieszczonego na najstarszej pozycji liczby. Gdy bit znaku ma wartość 1, wartość liczby jest ujemna, a w przeciwnym przypadku - dodatnia,
– wykładnika określającego potęgę, do której należy podnieść podstawę systemu 2. W przypadku liczby pojedynczej precyzji wykładnik jest reprezentowany przez liczbę całkowitą 8–bitową zapisaną w kodzie spolaryzowanym (bias=127). Dla liczby podwójnej precyzji wykładnik zajmuje 11 bitów w kodzie spolaryzowanym (bias=1023),
– części ułamkowej mantysy, zajmującej 23 bity dla formatu SINGLE oraz 52 bity dla formatu DOUBLE. Standard IEEE-754 zakłada, że mantysa jest liczbą z przedziału <1;2), jednak dla zwiększenia efektywności formatu nie zapisuje się w sposób jawny jedynki całkowitej, stojącej przed przecinkiem i samego przecinka. Na rys. 2.5 przedstawiono formaty SINGLE oraz DOUBLE.



Rys. 2.5. Formaty liczb SINGLE i DOUBLE

Taki format reprezentacji liczb rzeczywistych pozwala na osiągnięcie następujących zakresów wartości: dla typu SINGLE od 2^{-126} do 2^{128} (od 10^{-38} do 10^{38}), natomiast dla typu DOUBLE od 2^{-1022} do 2^{1022} (od 10^{-308} do 10^{308}).

Wartość liczby zmiennopozycyjnej można wyznaczyć z następującej zależności:

$$WART = (-1)^Z \times M \times 2^W,$$

gdzie: Z – bit znaku, M – wartość mantysy, W – wartość wykładnika.

Poniżej zamieszczono przykład wyznaczenia wartości dziesiętnej liczby typu SINGLE o następującej postaci binarnej:

11000001111101000000000000000000

W pierwszej kolejności należy wyszczególnić bity znaku, wykładnika i mantysy:

Znak:	Wykładnik:	Mantysa:
1	**10000011**	**11101000000000000000000**

Bit znaku jest równy 1, w związku z czym liczba jest ujemna. Wykładnik: **10000011b = 131d** jest zapisany w kodzie spolaryzowanym. Po odjęciu wartości 127 (bias), uzyskuje się 131 – 127 = 4**.**

W kolejnym kroku wyznaczamy wartość mantysy, a następnie wartość liczby (należy przy tym pamiętać, że formaty liczb SINGLE i DOUBLE nie zawierają jedynki całkowitej mantysy):

$$1{,}11101000000000000000000 * 2^4 =$$
$$= 11110{,}10000000000000000_2 = 30.5_{10}$$

2.1.3. Wartości specjalne liczb zmiennopozycyjnych

Standard IEEE-754 definiuje pięć wartości specjalnych, które może osiągać liczba zmiennopozycyjna, w zależności od stanu jej pól:

Liczba znormalizowana. Jest to stan, w którym wykładnik liczby nie przyjmuje wartości granicznych, równych `11111111b` (nadmiar) lub `00000000b` (niedomiar), a wartość liczby mieści się w zadanych granicach. Dane przetwarzane przez jednostki SIMD powinny być liczbami znormalizowanymi.

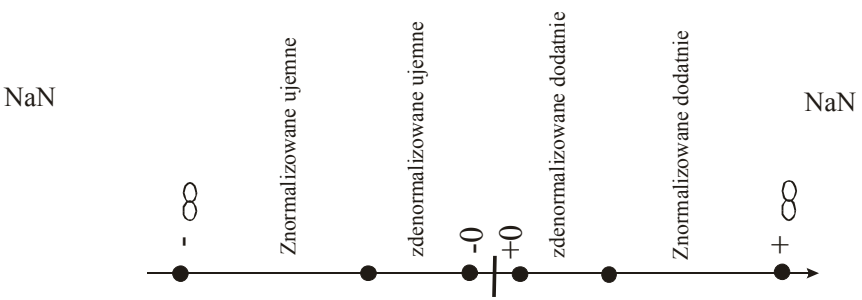
Liczba zdenormalizowana. Liczba zdenormalizowana posiada zerowy wykładnik `00000000b` oraz zerową część całkowitą mantysy. Dzięki temu, że w formacie zdenormalizowanym nie występuje domyślna część całkowita mantysy, istnieje możliwość reprezentacji liczb o mniejszych wartościach bezwzględnych niż liczby znormalizowane. Wystąpienie liczby zdenormalizowanej w typowych obliczeniach numerycznych jest zjawiskiem stosunkowo rzadkim i świadczy o niepoprawności algorytmu numerycznego.

Zero maszynowe. Zmiennopozycyjna liczba o wartości zero, w której wszystkie bity mantysy oraz wykładnika przyjmują wartość zerową. W zależności od stanu bitu znaku wyróżniane jest zero dodatnie i zero ujemne.

Nadmiar zmiennopozycyjny. Występuje wówczas, gdy wykładnik ma postać `11111111b`, przy części ułamkowej mantysy równej 0 i części całkowitej mantysy równej 1. Jest wynikiem operacji dzielenia przez zero.

NaN (Not a Number). Wszystkie inne wartości, nie opisane w poprzednich punktach, traktowane są jako nie liczby – NaN. Charakterystyczną ich cechą jest maksymalna wartość wykładnika, przy części całkowitej mantysy równej 1 i niezerowej części ułamkowej. Wartości takie zazwyczaj powstają podczas wykonywania niedozwolonej operacji (na przykład pierwiastek kwadratowy z liczby ujemnej). Wyróżnia się dwa rodzaje wartości NaN – SNaN oraz QNaN. Wartości SNaN charakteryzują się zerowym najstarszym bitem mantysy, a ich użycie jako operandu jest niedozwolone. Wartości QNaN posiadają wartość 1 na najstarszym bicie mantysy i mogą być używane w większości operacji arytmetycznych.

Na rys. 2.6 zamieszczono schemat obrazujący rozmieszczenie poszczególnych wartości specjalnych na osi liczbowej.



Rys. 2.6. Rozmieszczenie wartości specjalnych liczb rzeczywistych

Jednostki SIMD mogą wykonywać następujące operacje na wartościach specjalnych:

$$liczba + \infty = \infty \qquad liczba \times \infty = \infty$$

$$\infty - \infty = NaN \qquad \frac{\infty}{\infty} = NaN$$

$$\frac{liczba}{0} = \infty \qquad \frac{0}{0} = NaN \qquad \infty + \infty = \infty$$

2.2. JEDNOSTKI SIMD

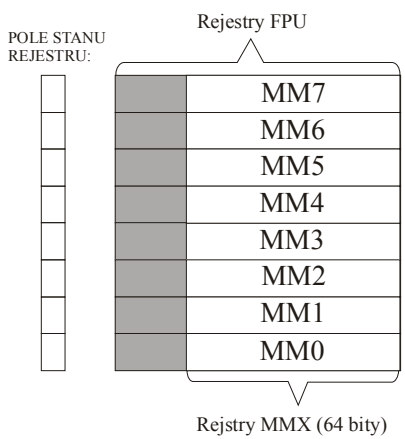
W rozdziale przedstawiono podstawowe cechy czterech rozszerzeń wektorowych, wchodzących w skład procesorów rodziny x86: MMX, SSE, SSE2 oraz SSE3. Opis poszczególnych jednostek zawiera informacje na temat budowy oraz

podstawowych możliwości. Instrukcje dostępne dla poszczególnych rozszerzeń zostały przedstawione w podrozdziale 2.3.

2.2.1. Jednostka MMX

Pełna nazwa rozszerzenia MMX brzmi MultiMedia eXtension, co oznacza rozszerzenie multimedialne. Jednostka MMX po raz pierwszy została wprowadzona w procesorach Intel Pentium MMX w celu umożliwienia szybszej obróbki danych multimedialnych, takich jak: pliki dźwiękowe, obrazy czy filmy wideo. Procesor wyposażony w technologię MMX dysponuje 57 nowymi instrukcjami, umożliwiającymi wykonywanie operacji na danych wektorowych (formaty danych używanych przez MMX przedstawiono na rys. 2.1).

Instrukcje MMX operują na ośmiu 64–bitowych rejestrach o nazwach: `MM0`, `MM1`, `MM2`, `MM3`, `MM4`, `MM5`, `MM6` i `MM7`. Pomimo tego, że przed wprowadzeniem jednostki MMX rejestry te nie były programowo dostępne, w rzeczywistości są one fizycznie ulokowane na pierwszych 64 bitach 80–bitowych rejestrów jednostki zmiennoprzecinkowej FPU. Takie umiejscowienie rejestrów MMX gwarantuje właściwe zachowanie ich stanu podczas przełączania kontekstu. Pociąga to za sobą również pewne negatywne konsekwencje, gdyż nie można jednocześnie używać jednostki FPU z jednostką MMX, gdyż jednoczesne wykonywanie operacji z użyciem obydwu jednostek powodowałoby wzajemne niszczenie danych. Budowę rejestrów MMX przedstawiono na rys. 2.7.

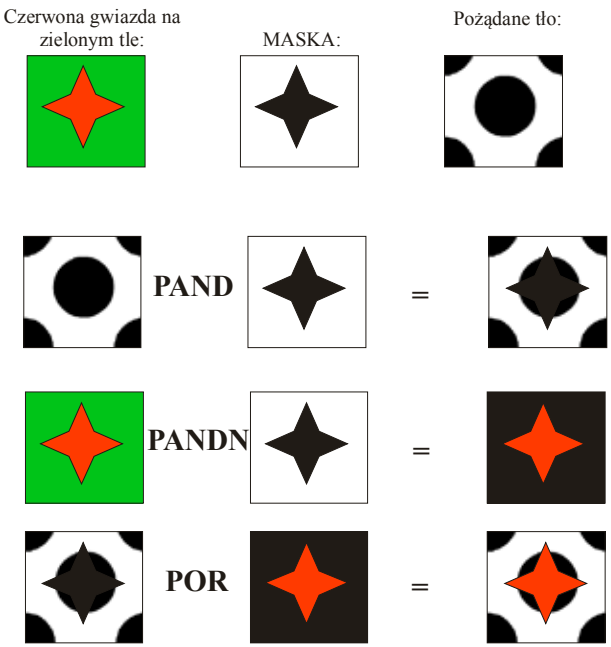


Rys. 2.7. Fizyczne położenie rejestrów MMX

Do najczęściej stosowanych operacji z użyciem rozszerzenia MMX należą:

– Zmiana jasności bitmapy, tworzenie negatywu bitmapy, binaryzacja obrazu oraz inne operacje na kolorach bitmapy.

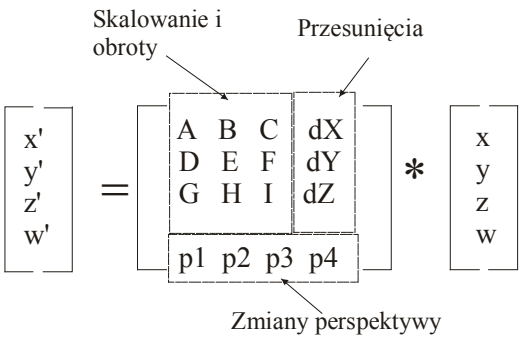
– Zamiana tła obiektu. Najbardziej charakterystycznym przykładem podmiany tła jest telewizyjna prognoza pogody, gdzie prezenter na tle mapy regionu omawia przewidywania meteorologów. W rzeczywistości jest on nagrywany na tle o jednolitym kolorze, natomiast komputerowa obróbka obrazu pozwala na zamianę koloru tła na obraz mapy regionu. W tym celu należy określić tzw. maskę bitową, będącą ciągiem danych n–bitowych, które mogą przyjmować jedną z dwóch wartości – 0 albo 2^n-1 (maksymalna wartość danej). Wartości maski będą przyjmowały maksymalną wartość dla punktów obrazu, na których znajduje się tło, natomiast minimalne - dla punktów prezentera. Następnie wykonywana jest operacja `PAND` maski z obrazem nowego tła oraz `PANDN` maski z obrazem prezentera. W ostatnim kroku, za pomocą polecenia `POR`, jest dokonywane połączenie obydwu obrazów. Poszczególne kroki schematycznie przedstawiono na rys. 2.8.



Rys. 2.8. Podmiana tła obiektu

– Wyznaczanie iloczynu skalarnego wektorów. Operacja ta jest często wykorzystywana w grafice komputerowej do wyznaczenia jasności punktów na ekranie. Wektorami biorącymi udział w operacji wyznaczenia jasności jest wektor padającego światła i normalna do powierzchni oświetlanej.
– Mnożenie macierzy przez wektor. Gry komputerowe oraz programy wykorzystujące grafikę 3D wykonują często operacje przekształceń geometrycznych,

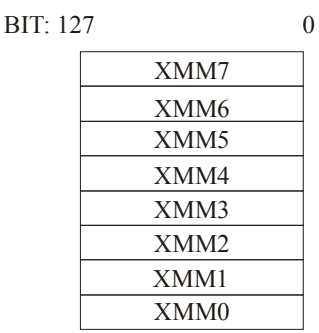
jak: przesunięcia, obroty, skalowania. Przekształcenia te wykonywane są dzięki mnożeniu współrzędnych punktu (wektora położenia) przez macierz przekształceń (rys. 2.9), gdzie: x, y, z –współrzędne punktu, x', y', z' – współrzędne po przekształceniu.



Rys. 2.9. Przekształcenia geometryczne

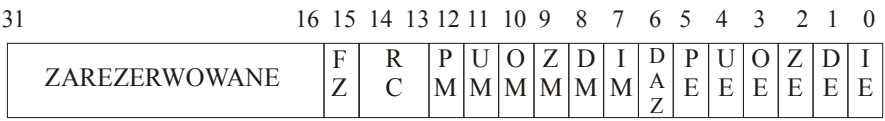
2.2.2. Jednostka SSE

Rozszerzenie SSE (ang. Streaming SIMD Extension) zostało wprowadzone po raz pierwszy w procesorach Intel Pentium III. Umożliwia ono wykonywanie równoległych operacji na czterech 32-bitowych liczbach zmiennopozycyjnych typu SINGLE. Instrukcje SSE wykorzystują 8 nowych, 128-bitowych rejestrów procesora o nazwach `XMM0` do `XMM7` (rys. 2.10).



Rys. 2.10. Rejestry jednostki SSE

Oprócz ośmiu rejestrów `XMM`, do architektury procesora dodano 32-bitowy rejestr `MXCSR`, zawierający bity stanu oraz konfiguracji rozszerzenia SSE (również dodanych później rozszerzeń SSE2 i SSE3). Budowę rejestru `MXCSR` przedstawiono na rys. 2.11.



Rys. 2.11. Budowa rejestru `MXCSR`

Najmłodsze 6 bitów rejestru `MXCSR` pełni rolę znaczników błędów (wyjątków) numerycznych. Wartość 1 na pozycji jednego z tych bitów oznacza, że podczas wykonywania operacji przez jednostkę SEE wystąpił określony błąd. Sposób obsługi zaistniałej sytuacji jest uzależniony od stanu bitów maski wyjątków zgrupowanych na bitach 7 – 12.

Flaga `FZ` (ang. Flush-To-Zero) pozwala na określenie sposobu, w jaki procesor reaguje na wystąpienie niedomiaru. W przypadku, gdy flagi `FZ=1` oraz `UM=1`, jeżeli pojawi się wyjątek niedomiaru, procesor zwróci wartość `0` przy zachowaniu znaku liczby.

Flaga `DAZ` (ang. Denormals-Are-Zeros) kontroluje sposób reakcji procesora na zdenormalizowane wartości operandów. Gdy bit ma wartość 1, następuje zaokrąglenie wartości operandu źródłowego do 0 (przy zachowaniu znaku liczby), przy czym bit `DE` pozostaje w stanie 0 i nie jest generowany wyjątek Denormal-operand exception.

Pole `RC` (bity 13 – 14) pozwala na określenie domyślnego sposobu zaokrąglania wyników operacji SSE:

- `00B` – wynik zostanie zaokrąglony do najbliższej wartości, która jest możliwa do uzyskania przy danej reprezentacji binarnej. W przypadku, gdy wynik operacji znajduje się dokładnie pośrodku dwóch możliwych do uzyskania reprezentacji binarnych, ostatecznie wybierana jest ta, której najmniej znaczący bit jest równy 0.
- `01B` – zaokrąglenie w dół (w kierunku $-\infty$),
- `10B` – zaokrąglenie w górę (w kierunku $+\infty$),
- `11B` – zaokrąglenie w kierunku zera. Oznacza to, że dodatnie wartości będą zaokrąglane w dół, natomiast ujemne w górę.

Każdorazowo, gdy wykonujący się program spowoduje wystąpienie jednego z 6 wyjątków numerycznych, procesor ustawia odpowiedni znacznik błędu, po czym w zależności od stanu odpowiedniego bitu maski wykonuje następującą czynność:

- jeżeli bit maski ma wartość 1, realizuje czynność domyślną, pozwalając na dalsze wykonywanie się programu,
- jeżeli bit maski ma wartość 0, wywołuje kod odpowiedniej procedury obsługi wyjątku numerycznego.

Poniżej przedstawiono krótką charakterystykę poszczególnych wyjątków numerycznych:

– Niedozwolona operacja (ang. Invalid Operation Exception). Wyjątek jest generowany w przypadku, gdy instrukcja zawiera niedozwolony operand. Jeśli wyjątek jest zamaskowany (ustawiona flaga IM), procesor ustawia flagę IE oraz zwraca NaN.
– Zdenormalizowany argument (ang. Denormal Operand Exception). Wyjątek sygnalizowany jest poprzez ustawienie flagi DE podczas próby wykonania operacji na zdenormalizowanym argumencie. Zamaskowanie wyjątku (ustawienie flagi DM) spowoduje, że procesor będzie kontynuował wykonywanie instrukcji zawierającej zdenormalizowany argument.
– Dzielenie przez zero (ang. Divide-By-Zero Exception). Wyjątek sygnalizowany przez ustawienie flagi ZE. Wywoływany jest każdorazowo, gdy program usiłuje podzielić niezerową liczbę przez wartość 0. Zamaskowanie błędu (ustawienie flagi ZM) spowoduje, że operacja zwróci w operandzie docelowym wartość nieskończoność.
– Nadmiar (ang. Numeric Overflow Exception). Błąd zgłaszany poprzez ustawienie flagi OE. W przypadku, gdy błąd jest zamaskowany (flaga OM), procesor, w zależności od konfiguracji trybu zaokrąglania, zwraca jedną z wartości określonych w tabeli 2.2.

Tabela 2.2

Wynik zwracany przy zamaskowanym wyjątku nadmiaru

Tryb zaokrąglania	Znak uzyskanego wyniku	Wartość zwracana
Do $-\infty$	–	$-\infty$
	+	Największa liczba dla reprezentacji
Do 0	–	Najmniejsza liczba dla reprezentacji
	+	Największa liczba dla reprezentacji
Do najbliższej liczby	–	$-\infty$
	+	$+\infty$
Do $+\infty$	–	Najmniejsza liczba dla reprezentacji
	+	$+\infty$

– Niedomiar (ang. Numeric Underflow Exception). Błąd sygnalizowany przez ustawienie flagi UE. Powstaje w sytuacji, gdy wynik operacji jest na tyle mały (bliski zeru), że nie można go zapisać w operandzie docelowym jako wartość znormalizowaną. W przypadku, gdy błąd jest zamaskowany (ustawiona flaga UM), procesor zwraca wynik w postaci zdenormalizowanej (dla wyzerowanej flagi FZ) lub 0 (dla ustawionej flagi FZ).
– Błąd precyzji wyniku (ang. Inexact-Result Exception). Wyjątek jest sygnalizowany przez ustawienie flagi PE. Występuje, gdy wynik operacji nie może zostać

precyzyjnie zapisany w formacie docelowym (na przykład wynik operacji dzielenia 1 przez 3 nie może zostać precyzyjnie określony przez binarną reprezentację zmiennopozycyjną). Błąd ten powstaje często podczas wykonywania programów obliczeniowych. Zamaskowanie wyjątku (ustawienie bitu `PM`) spowoduje, że procesor zapisze do operandu docelowego zaokrągloną wartość wyniku.

W przeciwieństwie do jednostki MMX, rozszerzenie SSE umożliwia wykonywanie operacji na liczbach zmiennopozycyjnych, co w przypadku obliczeń graficznych (takich jak wyznaczanie oświetlenia czy translacja geometryczna w przestrzeni 3D) zwiększa szybkość obliczeń i precyzję wyników. SSE w znaczący sposób wspomaga również procesy obróbki wideo, rozpoznawania mowy czy obliczenia numeryczne na blokach danych zmiennopozycyjnych.

2.2.3. Jednostka SSE2

Jednostka SSE2 została wprowadzona przez firmę Intel wraz z procesorami Pentium IV. Rozszerzenie nie wprowadza do architektury procesora dodatkowych rejestrów, umożliwia natomiast wykonywanie operacji na nowych typach danych. Zbiór instrukcji SSE2, w zależności od ich przeznaczenia, można podzielić na 4 grupy:

– Instrukcje umożliwiające wykonywanie operacji na skalarnych lub upakowanych danych zmiennopozycyjnych podwójnej precyzji,
– Instrukcje wykonujące operacje na danych typu całkowitoliczbowego,
– Instrukcje wprowadzone wcześniej dla MMX, wykorzystujące 128-bitowe rejestry `XMM`,
– Instrukcje wykonujące bezpośredni zapis zawartości rejestrów do pamięci operacyjnej (z pominięciem pamięci podręcznej cache).

Rozszerzenie SSE2 znajduje zastosowanie między innymi w takich dziedzinach jak kodowanie/dekodowanie wideo, rozpoznawanie mowy, aplikacje obliczeniowe i naukowe, zaawansowana grafika 3D.

2.2.4. Jednostka SSE3

Instrukcje SSE3 zostały wprowadzone przez firmę Intel wraz z pierwszym procesorem zawierającym rdzeń `Prescott`. Rozszerzenie dodaje do listy instrukcji procesora 13 nowych pozycji, nie wprowadzając dodatkowych rejestrów. Nowe instrukcje umożliwiają wykonywanie następujących operacji:

– przetwarzanie asymetryczne – jednoczesne dodawanie i odejmowanie elementów wektora,
– działania horyzontalne – wykonywanie operacji na polach wewnętrznych wektora,
– konwersja liczb,

– synchronizacja wątków.

Rozszerzenie SSE3 znajduje swoje zastosowanie w takich dziedzinach jak: grafika komputerowa (operacje horyzontalne), kodowanie wideo, arytmetyka zespolona czy synchronizacja międzyprocesowa.

2.3. INSTRUKCJE SIMD

Rozkazy SIMD zostały zgrupowane tematycznie – autorzy postanowili nie prezentować osobnych zbiorów instrukcji dla każdego rozszerzenia. Pełną listę instrukcji wektorowych MMX, SSE, SSE2 i SSE3 procesorów rodziny x86 zamieszczono w rozdz. 5 oraz 6.

2.3.1. Transfer danych

Zadaniem instrukcji przedstawionych w tym rozdziale jest przeprowadzanie wymiany danych pomiędzy rejestrami jednostki `MMX` czy `XMM`, a pamięcią lub rejestrami ogólnego przeznaczenia (tym mianem określane są rejestry procesora `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` i `ESP`). Przy użyciu instrukcji transferu danych możliwa jest również wewnętrzna wymiana danych pomiędzy rejestrami `MMX` oraz `XMM`. Nazwa każdej instrukcji transferu danych składa się z rdzenia `MOV`, do którego dodawany jest przedrostek lub przyrostek precyzujący konkretną operację. Poniżej scharakteryzowano przyrostki i przedrostki instrukcji `MOV`:

– `D` (ang. Doubleword) – określa, że transfer dotyczy podwójnego słowa,
– `Q` (ang. Quadword) – poczwórnego słowa (8 bajtów),
– `A` (ang. Aligned) – określa, że dana w pamięci jest wyrównana do granicy 16 bajtów (4 najmłodsze bity adresu są równe zero),
– `U` (ang. Unaligned) – odczyt/zapis nastąpi do zmiennej, której adres w pamięci nie musi być wyrównany do granicy 16 bajtów,
– `L` (ang. Low) – określa dolną część rejestru `XMM` (młodsze 64 bity),
– `H` (ang. High) – wskazuje na górną część rejestru `XMM` (starsze 64 bity),
– `LH` (ang. Low-to-High) – wskazuje, że młodsza część operandu źródłowego zostanie skopiowana do starszej części operandu docelowego,
– `HL` (ang. High-to-Low) – skopiowana zostanie starsza część operandu źródłowego do młodszej części operandu docelowego,
– `2` (ang. Two) – nastąpi wymiana danych pomiędzy rejestrami `MMX` a `XMM`,
– `NT` (ang. Not-Temporal) – transfer będzie wykonany z pominięciem pamięci podręcznej cache,
– `NTI` (Not-Temporal hint) – zapis wartości z rejestru ogólnego przeznaczenia do pamięci z pominięciem cache,

– MSK (ang. Mask) – transfer dotyczy wyłącznie najbardziej znaczącego bitu każdej ze spakowanych danych,
– MASK (ang. Mask) – transfer dotyczy wyłącznie określonych bitów,
– PS (ang. Packed Single-precision floating-point) – transfer dotyczy spakowanej danej typu SINGLE,
– SS (ang. Scalar Single-precision floating-point) – transfer dotyczy pojedynczej danej typu SINGLE,
– PD (ang. Packed Double-precision floating-point) – transfer dotyczy spakowanej danej typu DOUBLE,
– SD (ang. Scalar Double-precision floating-point) – transfer dotyczy pojedynczej danej typu DOUBLE,
– DUP (ang. Duplicate) – dana zostanie zapisana zarówno do młodszej, jak i starszej części rejestru.

Instrukcja transferu danych przyjmuje następującą postać:

```
X_MOV_Y OPERAND1, OPERAND2
```

gdzie: X_ – przedrostek instrukcji, _Y – przyrostek instrukcji, OPERAND1 – operand docelowy, OPERAND2 – operand źródłowy.

Np. polecenie załadowania rejestru XMM3 wartością zmiennej TABLICA, która zawiera 4 spakowane wartości typu SINGLE, przyjmuje następującą postać (składnia NASM-a):

```
MOVUPS XMM3, [TABLICA]
```

W przypadku, gdy adres zmiennej TABLICA jest wyrównany do granicy 16 bajtów można również użyć instrukcji:

```
MOVAPS XMM3, [TABLICA]
```

Oprócz ładowania rejestrów XMM danymi upakowanymi, istnieje możliwość transferu pojedynczych wartości typu SINGLE (do 32 młodszych bitów rejestru XMM) lub DOUBLE (do 64 młodszych bitów rejestru XMM):

```
MOVSS XMM2, [ZMIENNA]=    ;Dla single
MOVSD XMM2, [ZMIENNA2]=   ;Dla double.
```

Programista może również wskazać część rejestru XMM, do/z której ma nastąpić transfer. Np polecenie:

```
MOVHLPS  XMM1, XMM2
```

skopiuje bity 64 – 127 operandu źródłowego na bity 0 – 63 operandu docelowego, a polecenia:

```
MOVHPD XMM1, [ZM]
MOVLPD XMM2, [ZM]
```

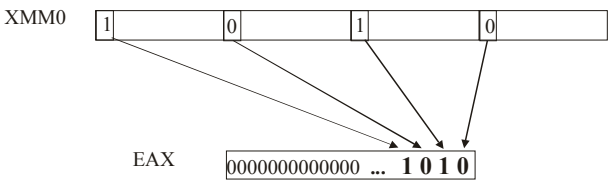
wypełnią starszą połowę rejestru XMM1 oraz młodszą połowę rejestru XMM2 wartością typu DOUBLE ze zmiennej ZM.

Oprócz standardowych operacji transferu danych, możliwe jest również wybiórcze kopiowanie wybranych bitów. Instrukcje MOVMSKPS oraz MOVMSKPD kopiują bity znaku danych spakowanych znajdujących się w operandzie źródłowym,

po czym skopiowane bity zostają umieszczone na młodszych bitach operandu docelowego (rejestru ogólnego przeznaczenia). Np. wywołanie instrukcji:

```
MOVMSKPS EAX, XMM0
```

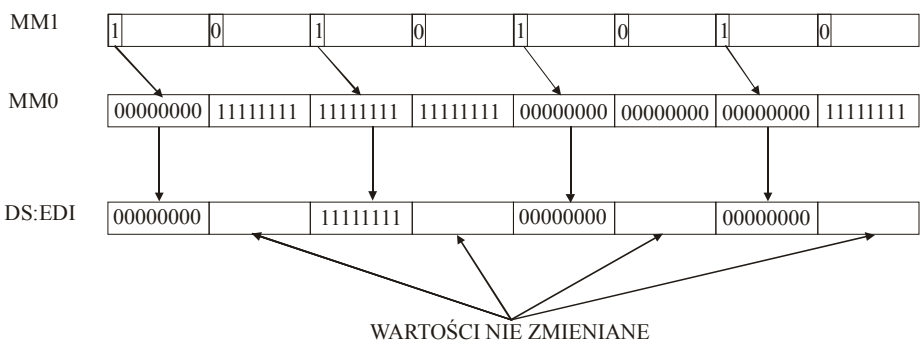
w przypadku gdy rejestr `XMM0` zawiera w kolejności: dodatnią, ujemną, dodatnią i ujemną liczbę typu SINGLE, przyniesie rezultat przedstawiony na rys. 2.12.



Rys. 2.12. Instrukcja MOVMSKPS

Instrukcja `MASKMOVQ` zachowuje kolejne bajty pierwszego operandu pod adres pamięci określony przez rejestry `DS:EDI` (rejestr `DS` zawiera segment, a `EDI` OFFSET w segmencie), w zależności od wartości drugiego operandu. W przypadku, gdy najbardziej znaczący bit kolejnego bajtu operandu maski (operand 2) jest ustawiony, następuje kopiowanie korespondującego bajtu operandu źródłowego (operand 1). W przeciwnym przypadku bajt nie zostanie skopiowany. Zarówno pierwszy jak i drugi operand musi być rejestrem `MMX` (rys. 2.13). Analogiczne działanie, ale dla rejestrów `XMM` wykonuje instrukcja `MASKMOVDQU`.

Instrukcje przesłań danych dla jednostek SIMD zostały zamieszczone w rozdz. 5, w tabeli 5.1.

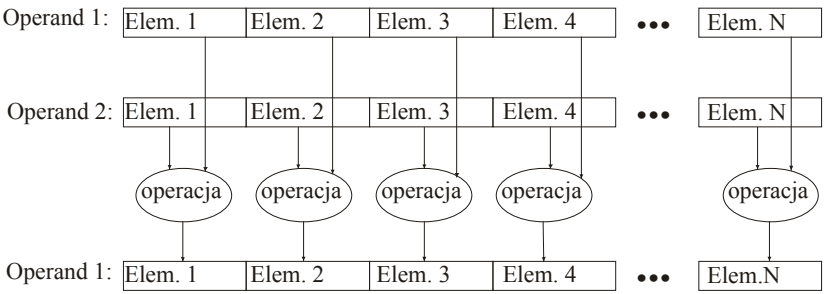


Rys. 2.13. Rezultat wykonania instrukcji `MASKMOVQ MM0, MM1`

2.3.2. Operacje arytmetyczne

Rozszerzenia SIMD dostarczają następujących instrukcji wykonujących operacje na wektorach: dodawanie, odejmowanie, mnożenie, dzielenie, wyznaczanie

odwrotności, pierwiastka kwadratowego, elementów największych i najmniejszych oraz średniej. Standardowy schemat operacji wykonywanych przez jednostki wektorowe polega na wykonaniu zadanej operacji (sprecyzowanej instrukcją) na każdym elemencie pierwszego operandu z korespondującym elementem operandu drugiego, po czym wynik jest zapisywany do operandu pierwszego (rys. 2.14).



Rys. 2.14. Schemat wykonywania operacji arytmetycznych

Podobnie jak instrukcje przesłań danych, instrukcje arytmetyczne posiadają nazwy składające się z 3 członów. W przypadku, gdy instrukcja rozpoczyna się od przedrostka `P`, to operacja dotyczy danych całkowitoliczbowych (początkowo przedrostek był charakterystyczny dla instrukcji MMX). Przedrostek `H` oznacza natomiast, że operacja będzie wykonana na elementach wewnętrznych wektora (tzw. operacje horyzontalne, dodane wraz z rozszerzeniem SSE3). Przyrostki charakteryzują typ danych (tabela 2.3) oraz wykorzystywaną arytmetykę.

Tabela 2.3.

Przyrostki określające typ danych

Przyrostek	Typ danej
`B`	Całkowita, jednobajtowa (ang. Byte integer)
`W`	Całkowita, dwubajtowa (ang. Word integer)
`D`	Całkowita, czterobajtowa (ang. Doubleword integer)
`Q`	Całkowita, ośmiobajtowa (ang. Quadword integer)
`SS`	Skalarna, zmiennopozycyjna pojedynczej precyzji (ang. Scalar Single-precision floating-point)
`SD`	Skalarna, zmiennopozycyjna podwójnej precyzji (ang. Scalar Double-precision floating-point)
`PS`	Upakowana, zmiennoprzecinkowa pojedynczej precyzji (ang. Packed Single-precision floating-point)
`PD`	Upakowana, zmiennoprzecinkowa podwójnej precyzji (ang. Packed Double-precision floating-point)

Jednostki SIMD umożliwiają wykonywanie operacji matematycznych na zmiennych całkowitoliczbowych w jednej z trzech arytmetyk: arytmetyka modularna (ang. wrap-around), z nasyceniem z uwzględnieniem znaku liczby (ang. signed saturation) oraz z nasyceniem bez uwzględniania znaku liczby (ang. unsigned saturation). Każda z nich definiuje odmienny sposób reakcji na przepełnienie:

– Operacje przeprowadzane w arytmetyce modularnej (zawijanie) w przypadku wystąpienia nadmiaru zwrócą młodsze bity wyniku, ignorując przeniesienie z najstarszego bitu. Np dodając 1 do wartości 255 dla zmiennej 8-bitowej, zostanie zwrócona wartość 0.
– Instrukcje korzystające z arytmetyki z nasyceniem z uwzględnieniem znaku liczby posiadają przyrostek `S+typ` danej. W przypadku wystąpienia dodatniego przepełnienia, instrukcje zwrócą największą liczbę możliwą do zapisania w danej reprezentacji. Ujemny nadmiar wygeneruje wynik w postaci najmniejszej wartości dla danej reprezentacji.
– Instrukcje wykorzystujące arytmetykę z nasyceniem bez uwzględniania znaku liczby posiadają przyrostek `US+typ` danej. Dla dodatniego przepełnienia zwrócą największą liczbę z zakresu, natomiast dla ujemnego przepełnienia – wartość 0.

Instrukcje dodawania sumują ze sobą odpowiadające elementy operandów według schematu przedstawionego na rys. 2.14. Rdzeń instrukcji stanowi `ADD`, do którego należy dodać przedrostek lub przyrostek precyzujący instrukcję. Przykładowe wywołania rozkazu dodawania mogą wyglądać następująco (składnia NASM):

```
PADDB MM1, MM2              ;Dodawanie dwóch wektorów bajtów w arytmetyce modularnej,
PADDW MM3, [ZMIENNA]        ;Dodawanie wektorów słów w arytmetyce modularnej,
PADDSB MM1, MM3             ;Dodawanie wektorów bajtów w arytmetyce modularnej,
ADDSS XMM1, [ZMIENNA2]      ;Dodawanie dwóch skalarnych liczb typu SINGLE,
ADDSD XMM2, XMM3            ;Dodawanie dwóch skalarnych liczb typu DOUBLE,
ADDPS XMM1, XMM4            ;Dodawanie wektorów liczb typu SINGLE,
ADDPD XMM3, [ZMIENNA3] ;Dodawanie wektorów liczb typu DOUBLE.
```

Instrukcje odejmowania wyznaczają różnicę odpowiadających elementów operandów (rys. 2.14). Rdzeń nazwy instrukcji stanowi `SUB`. Oto przykładowe wywołania instrukcji odejmowania (składnia NASM):

```
PSUBUSB MM1, [ZMIENNA1]  ;Odejmowanie wektorów bajtów w arytmetyce unsigned saturation,
SUBPS   XMM1, XMM7       ;Odejmowanie wektorów liczb typu SINGLE,
SUBSD   XMM0, [ZMIENNA2] ;Odejmowanie skalarnych liczb typu DOUBLE.
```

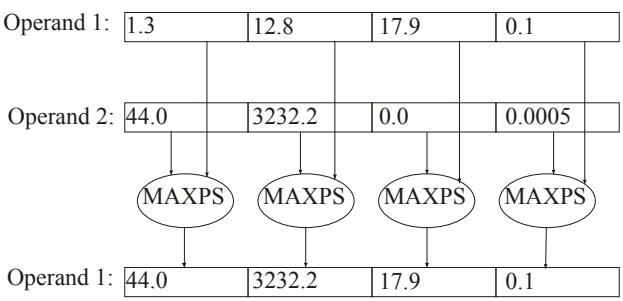
Instrukcje mnożenia wyznaczają iloczyn odpowiadających elementów operandów. Rdzeń nazwy instrukcji stanowi `MUL`. Oto przykładowe wywołania instrukcji mnożenia (składnia NASM):

```
PMULLW MM0, MM1    ;Mnożenie wektorów liczb całkowitych 16-bitowych
                    i umieszczenie wektora złożonego z 16 młodszych bitów
                    wyników w rejestrze MM0,
PMULHW MM0, MM1    ;Mnożenie wektorów liczb całkowitych 16-bitowych
                    i umieszczenie wektora złożonego z 16 starszych bitów
                    wyników w rejestrze MM0,
MULPS XMM1, [ZM1] ;Mnożenie wektorów liczb typu SINGLE,
MULSD XMM3, XMM0  ;Mnożenie skalarnych liczb typu DOUBLE.
```

Instrukcje dzielenia wyznaczają iloraz odpowiadających elementów operandów. Rdzeń nazwy instrukcji stanowi `DIV`. Oto przykładowe instrukcje dzielenia (składnia NASM):

```
afs mp=uj j NI =uj j O   ;Dzielenie wektorów liczb typu SINGLE,
afs pp=uj j NI =uj j P   ;Dzielenie skalarnych liczb typu SINGLE.
```

Instrukcje wyznaczające wartości maksymalne lub minimalne z dwóch wektorów wykonują operację porównania odpowiadających liczb, po czym zwracają wektor elementów największych lub najmniejszych (rys. 2.15). Rdzeń nazwy instrukcji wyznaczającej największe elementy wektorów to `MAX`, natomiast instrukcji wyznaczającej najmniejsze elementy wektorów - `MIN`.

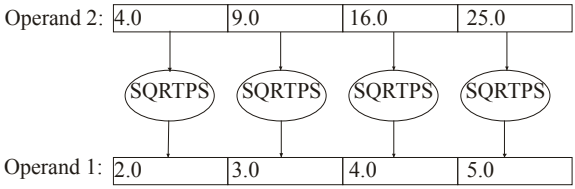


Rys. 2.15. Schemat działania instrukcji `MAXPS`

Niektóre instrukcje SIMD wykonują operacje wyłącznie na jednym operandzie (rys. 2.16). Do tej grupy należą instrukcje wyznaczające:

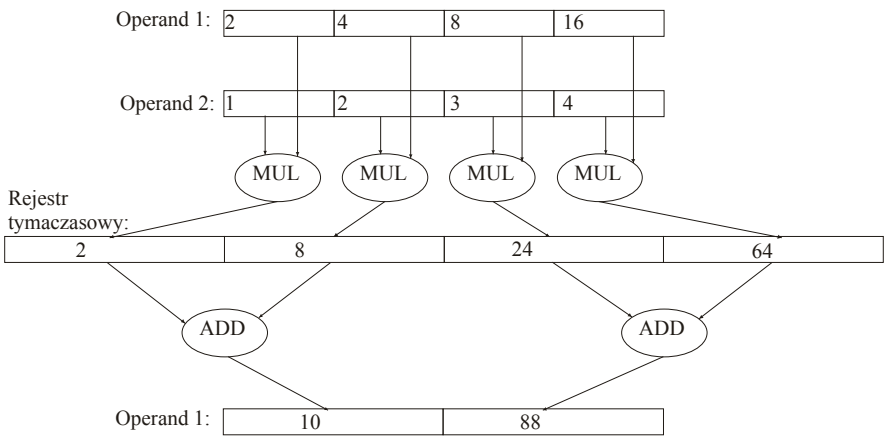
- Odwrotność wartości elementów wektora (rdzeń nazwy instrukcji `RCP`),
- Pierwiastek kwadratowy elementów wektora (rdzeń nazwy instrukcji `SQRT`),
- Odwrotność pierwiastka kwadratowego dla elementów wektora (rdzeń nazwy instrukcji `RSQRT`).

Instrukcja `PMADDWD` wykonuje operację mnożenia odpowiadających elementów (16-bitowych liczb całkowitych ze znakiem) operandu 1 i 2, po czym



Rys. 2.16. Schemat działania instrukcji `SQRTPS`

umieszcza wynik w rejestrze tymczasowym. W kolejnym kroku instrukcja sumuje parami uzyskane wcześniej iloczyny, a wynik zapisuje do operandu 1 (rys. 2.17). Podobne operacje wykonuje instrukcja `PMADDUBSW`, z tą różnicą, że operandy przechowują: 8-bitowe liczby bez znaku (w operandzie 1) oraz 8-bitowe liczby ze znakiem (w operandzie 2*).*

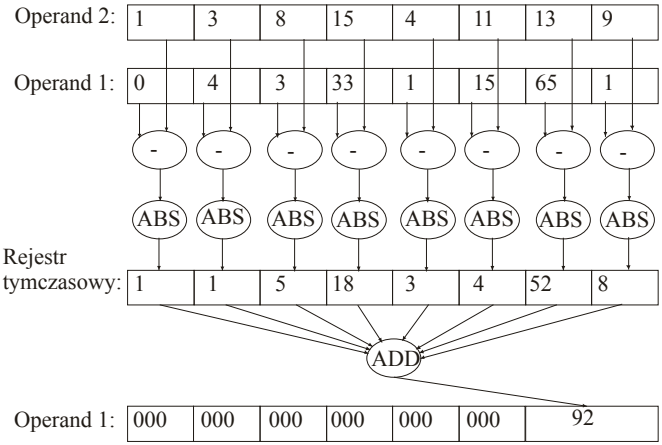


Rys. 2.17. Schemat działania instrukcji `PMADDWD`

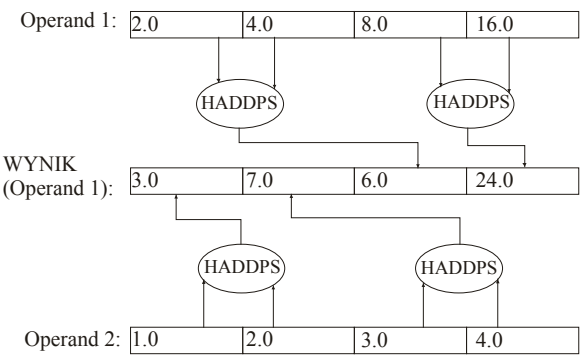
Instrukcja `PSADBW` w pierwszej kolejności wyznacza różnicę dwóch wektorów, a wynik zapisuje w rejestrze tymczasowym. Następnie wyznacza wartości bezwzględne wszystkich elementów rejestru tymczasowego. W kolejnym kroku wartości bezwzględne są sumowane, a ostateczny wynik zapisywany na 16 najmłodszych bitach operandu docelowego (rys. 2.18).

Rozszerzenie SSE3 do listy instrukcji procesora dodaje instrukcje umożliwiające wykonywanie operacji horyzontalnych oraz przetwarzania asymetrycznego. Obliczenia horyzontalne polegają na wykonywaniu operacji z użyciem elementów wewnętrznych wektora. Instrukcje SSE3 pozwalają na wyznaczenie sumy i różnicy horyzontalnej (rys. 2.19).

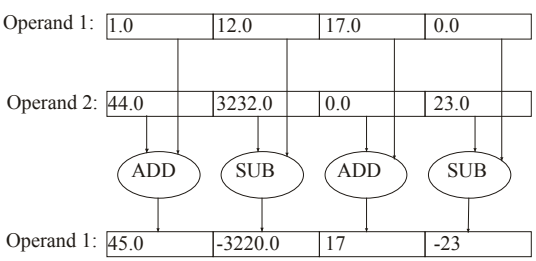
Przetwarzanie asymetryczne umożliwia wykonywanie różnych operacji na odpowiadających elementach dwóch wektorów. Rozszerzenie SSE3 dostarcza



Rys. 2.18. Schemat działania instrukcji `PSADBW`



Rys. 2.19. Schemat działania instrukcji `HADDPS`



Rys. 2.20. Schemat działania instrukcji `HADDPS`

dwóch instrukcji pozwalających na jednoczesne wykonywanie operacji dodawania i odejmowania upakowanych danych typu SINGLE i DOUBLE (rys. 2.20).

Instrukcje arytmetyczne dla jednostek SIMD zostały zebrane w rozdz. 5 (tabela 5.2).

2.3.3. Porównania

Instrukcje porównań danych wektorowych sprawdzają, czy zachodzi określona relacja (sprecyzowana instrukcją) pomiędzy operandem 1 a operandem 2. Wynik porównania wektorów zostaje zapisany do operandu pierwszego w postaci maski bitowej. Ponieważ porównywane są operandy wektorowe, rejestr `EFLAGS` (ustawiany przez instrukcję `CMP`) nie jest w tym przypadku stosowany. Jeżeli relacja jest spełniona dla danej pary wektorów, zostają ustawione odpowiednie bity operandu 1. Gdy relacja nie jest spełniona, odpowiednie bity operandu 1 są zerowane.

Rdzeniem nazwy instrukcji porównań SIMD jest `CMP`. W przypadku, gdy porównywanie będzie dotyczyło wektorów liczb całkowitych, nazwę należy poprzedzić przedrostkiem `P`. Część przyrostkowa nazwy składa się z określenia relacji, jaka będzie sprawdzana (tabela 2.4) oraz typu elementów wektora (tabela 2.3). Typ relacji w nazwie instrukcji może zostać pominięty, lecz należy w takim przypadku określić kod relacji trzecim argumentem instrukcji (tabela 2.4).

Tabela 2.4.

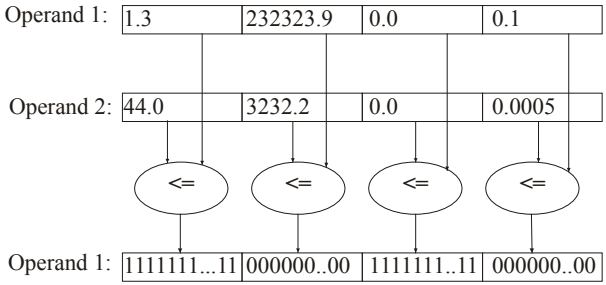
Kody relacji porównań

Typ relacji	Znaczenie	Kod relacji
`EQ`	Równy	0
`NE`	Różny	4
`LT`	Mniejszy	1
`NLT`	Nie mniejszy niż	5
`LE`	Mniejszy lub równy	2
`NLE`	Nie mniejszy i nie równy	6
`ORD`	Relacja prawdziwa gdy żaden z elementów nie jest NaN	7
`UNORD`	Relacja prawdziwa gdy przynajmniej jeden element jest NaN	3
`GT`	Większy (tylko dla rozkazów operujących na liczbach całkowitych)	–

Przykładowe wywołania instrukcji `CMP`:

```
CMPEQPD xmm1, xmm2        ;Sprawdzenie równości elementów obydwu wektorów,
CMPPD xmm1, xmm2, 0       ;Analogicznie jak powyżej (tym razem z trzecim parametrem),
CMPLEPS xmm1, [ZMIENNA]   ;Sprawdzenie czy elementy pierwszego wektora są mniejsze bądź równe elementom drugiego,
PCMPGTB mm1, mm5          ;Sprawdzenie czy elementy pierwszego wektora są większe od elementów drugiego.
```

Sposób realizacji instrukcji porównującej elementy wektorów (na przykładzie instrukcji `CMPLEPS`) zamieszczono na rys. 2.21.



Rys. 2.21. Schemat działania polecenia `CMPLEPS`

Poza instrukcjami porównującymi elementy wektorów, jednostki SIMD dostarczają kilku odmian rozkazu `COMIS..`, do porównywania danych skalarnych. Instrukcja ta zapisuje wynik porównywania na odpowiednich bitach rejestru `EFLAGS` (tabela 2.5).

Tabela 2.5.

Stan bitów `EFLAGS` w zależności od wyniku porównania

Wynik porównywania	Stan bitów rejestru `EFLAGS`		
	`ZF`	`PF`	`CF`
Wykryto NAN	1	1	1
Większy	0	0	0
Mniejszy	0	0	1
Równy	1	0	0

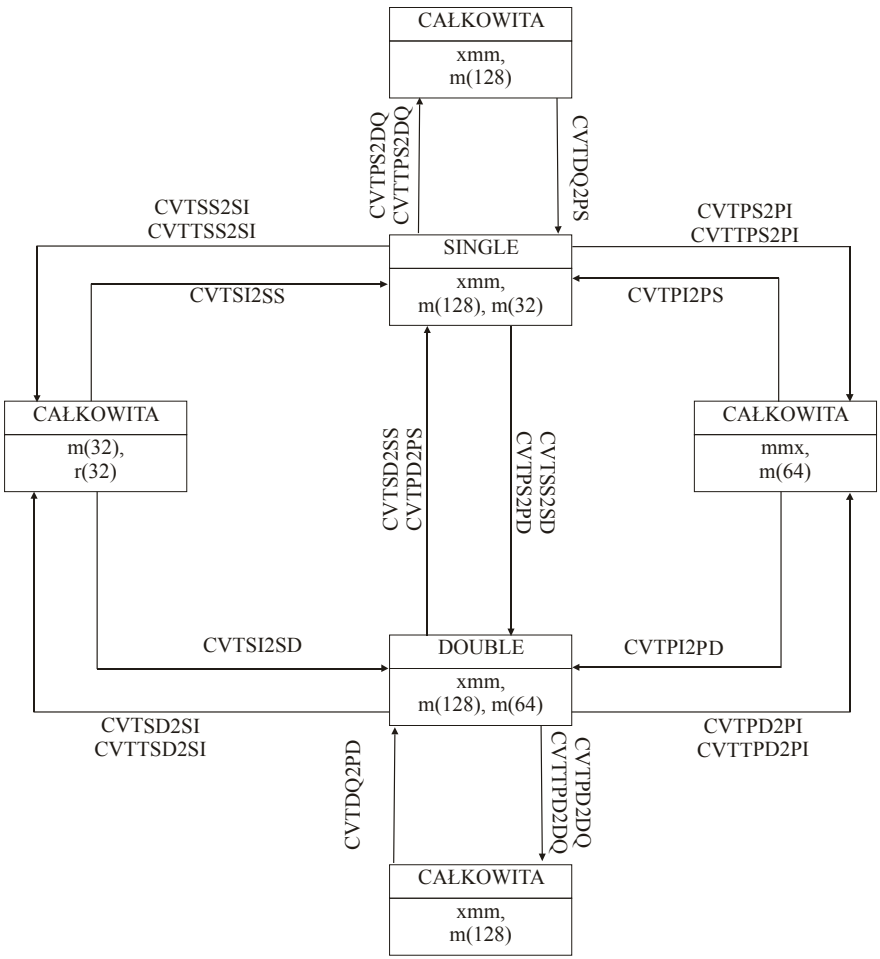
W rozdz. 5 (tabela 5.3) zebrano instrukcje porównań rozszerzeń MMX – SSE3.

2.3.4. Konwersje

Instrukcje konwersji są odpowiedzialne za zmianę typu danych zawartych w operandzie źródłowym (drugi operand). W przypadku, gdy operacja konwersji będzie wiązała się z utratą dokładności, wartość docelowa zostaje zaokrąglona zgodnie ze stanem pola `RC` rejestru `MXCSR` (rys. 2.11). Pole `RC` nie jest brane pod uwagę, gdy zostanie wywołana konwersja obcinająca wartość (opcja możliwa przy konwersji liczb zmiennopozycyjnych na całkowite).

Nazwa instrukcji konwersji składa się z czterech części: rdzenia `CVT` (`CVTT` dla konwersji obcinającej wartość), skrótu typu operandu źródłowego, cyfry „2” oraz skrótu typu operandu docelowego. Skróty typów źródłowych i docelowych dla liczb zmiennoprzecinkowych przybierają postać przedstawioną w tabeli 2.3,

natomiast dla typów całkowitych są następujące: `SI` (skalarna, 32-bitowa), `PI` (dwie 32-bitowe, spakowane), `DQ` (cztery 32-bitowe, spakowane). Schemat możliwych do przeprowadzenia konwersji zamieszczono na rys. 2.22.



Rys. 2.22. Schemat możliwych konwersji między typami

Przykładowe wywołania rozkazów konwersji typów:

```
` sqanOma=ñããNI=xòãáÉåå~z=;Konwersja dwóch 32-bitowych, spakowanych
                           liczb całkowitych na dwie spakowane liczby typu
                           DOUBLE,
` sqmaOan=ñããPI=ñããQ=      ;Konwersja w przeciwną stronę niż powyżej,
` sqmaOmf=ñããNI=ñããQ       ;Konwersja dwóch spakowanych liczb DOUBLE
                           na dwie 32-bitowe, spakowane liczby całkowite.
```

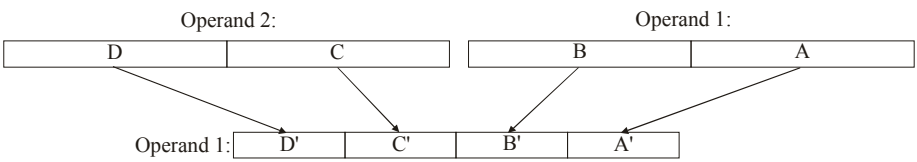
```
` s qmaOmp=ñããNl=xòãz=          ;Konwersja dwóch spakowanych liczb DOUBLE
                                  na dwie spakowane liczby SINGLE.
```

Rozszerzenie SSE3 wprowadza instrukcję konwersji operującą na wartości umieszczonej na szczycie stosu jednostki zmiennoprzecinkowej procesora (FPU). Instrukcja `FISTTP` przeprowadza konwersję liczby zmiennoprzecinkowej na liczbę całkowitą 16-, 32-, lub 64-bitową, w zależności od operandu z jakim instrukcja zostanie wywołana (m(16), m(32), m(64)).

W rozdz. 5 (tabela 5.4) zamieszczono listę instrukcji konwersji danych dla rozszerzeń MMX, SSE, SSE2, SSE3.

2.3.5. Instrukcje kompresji, dekompresji i mieszania danych

Instrukcje kompresji danych mają za zadanie zapisać dwa wektory liczb całkowitych (operand 1 i 2) do operandu docelowego (operand 1), używając konwersji elementów na typy mniej pojemne (rys. 2.23). Rdzeń nazwy instrukcji kompresji danych stanowi wyraz `PACK`, do którego dodawany jest przyrostek precyzujący format danych źródłowych. Instrukcja umożliwia kompresję dwóch wektorów słów na wektor bajtów (`WB`) lub dwóch wektorów podwójnych słów na wektor słów (`DW`). W przypadku, gdy wektor wynikowy ma składać się z elementów ze znakiem, do rdzenia nazwy instrukcji należy dopisać litery„`SS`". Gdy wektor wynikowy ma zawierać liczby naturalne, do rdzenia nazwy instrukcji dodaje się litery „`US`".

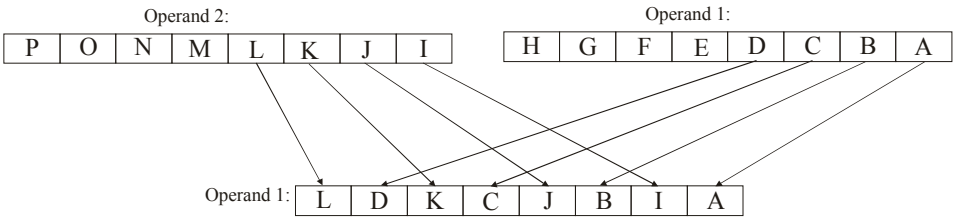


Rys. 2.23. Schemat działania polecenia `PACK`

Instrukcje dekompresji zapisują naprzemiennie wybrane elementy wektora docelowego (operand 1) i wektora źródłowego (operand 2) do operandu 1 (rys. 2.24). Rdzeniem nazwy instrukcji dekompresji danych jest `UNPCK`. W przypadku, gdy operacje będą wykonywane na danych całkowitych, do rdzenia nazwy instrukcji dodaje się przedrostek `P`. Przyrostek nazwy rozkazu składa się z dwóch części:

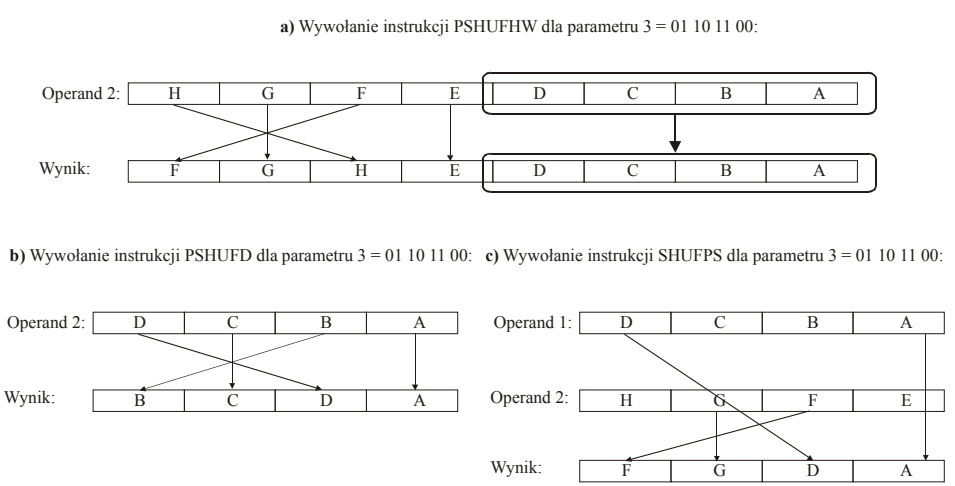
– Część pierwsza precyzuje elementy, które będą zapisywane do operandu docelowego. Dla „`L`" będą to elementy z młodszej połowy wektorów źródłowych, dla „`H`" elementy ze starszej połowy wektorów źródłowych.
– Część druga określa typ elementów składowych. Możliwe wartości dla liczb całkowitych to: „`BW`" (wektory źródłowe stanowią upakowane bajty), „`WD`" (upakowane słowa), „`DQ`" (upakowane podwójne słowa), „`QDQ`" (upakowane

poczwórne słowa). W przypadku wartości zmiennopozycyjnych: „PS” (wektory upakowanych liczb typu SINGLE), „PD” (wektory upakowanych liczb typu DOUBLE).



Rys. 2.24. Schemat działania instrukcji PUNPCKLBW

Instrukcje mieszania danych całkowitych (PSHUFW, PSHUFD) zapisują na kolejne pozycje operandu docelowego (operand 1) wybrane elementy wektora źródłowego. Prócz operandu docelowego i źródłowego, wywołanie instrukcji wymaga podania trzeciego parametru w postaci stałej, określającej numer elementu (w wektorze źródłowym) do skopiowania dla określonej pozycji w wektorze wynikowym. Do każdego pola operandu docelowego przyporządkowane są kolejne 2 bity stałej, które określają jeden z 4 elementów wektora źródłowego (rys. 2.25). Dla przykładu, gdy stała w zapisie binarnym ma postać „11 10 01 00”, na pierwszej pozycji wektora wynikowego znajdzie się element 1 wektora źródłowego, na drugiej pozycji element 2, na trzeciej element 3, a na czwartej element 4 (instrukcja nie zmieni kolejności danych). Aby odwrócić kolejność wektora źródłowego, należy podać wartość stałej równą „00 01 10 11”.



Rys. 2.25. Schemat działania instrukcji mieszających dane

W przypadku, gdy wektor źródłowy ma więcej pozycji niż 4, do rdzenia nazwy instrukcji mieszającej PSHUF należy dodać literę precyzującą, w której połowie operandu źródłowego będzie następowała zmiana kolejności składowych („L" – dla młodszej połowy, „H" – dla starszej połowy). Druga połowa operandu źródłowego zostanie skopiowana w niezmienionym szyku.

W nieco inny sposób są realizowane instrukcje mieszania danych zmiennoprzecinkowych (SHUFPS oraz SHUFPD). Instrukcja SHUFPS kopiuje do młodszej połowy operandu 1 wybrane jego pozycje (określone bitami 0 – 3 stałej, podanej jako trzeci parametr), natomiast do starszej połowy wybrane elementy operandu źródłowego (określone bitami 4 – 7 trzeciego parametru).

Polecenie SHUFPD kopiuje jeden z dwóch elementów operandu docelowego na jego młodszą połowę (numer elementu określony bitem nr 0 trzeciego parametru instrukcji) oraz jeden z dwóch elementów operandu źródłowego na połowę starszą (numer elementu określony bitem nr 1 trzeciego parametru instrukcji).

Oto przykładowe wywołania instrukcji przedstawionych w niniejszym rozdziale:

```
PACKSSWB MM1, [ZMIENNA] ;Spakowanie czterech 16-bitowych liczb całkowitych z MM1 i 4 liczb 16-bitowych ze zmiennej
                          do formatu 8 liczb 8-bitowych ze znakiem.
PUNPCKLDQ MM2, MM3      ;Zapis na bitach 32..63 rejestru MM2 bitów 0..31
                          rejestru MM3. Młodsze 32 bity rejestry MM2
                          pozostają bez zmian.
SHUFPD XMM1, XMM2, 01   ;Zapis bitów 64 – 127 na miejsce bitów 0 – 63
                          rejestru XMM1, zapis bitów 0 – 63 z XMM2 na
                          miejsce bitów 64 – 127 rejestru XMM1.
```

Lista instrukcji kompresji, dekompresji oraz mieszania danych została zebrana w rozdz. 5 (tabela 5.5).

2.3.6. Instrukcje logiczne

Instrukcje logiczne umożliwiają wykonywanie operacji boole'owskich na odpowiadających bitach dwóch operandów. Rozszerzenia SIMD dostarczają czterech grup instrukcji: OR, XOR, AND, ANDN, operujących na danych wektorowych o typie określonym przyrostkami i przedrostkami. Przedrostek „P" określa dane całkowitoliczbowe, przyrostek „PS" wektor liczb typu SINGLE, przyrostek „PD" wektor liczb typu DOUBLE.

Instrukcje POR, ORPS oraz ORPD realizują sumę logiczną dwóch operandów, instrukcje PXOR, XORPS, XORPD - różnicę symetryczną, instrukcje PAND, ANDPS, ANDPD - iloczyn logiczny, a instrukcje PANDN, ANDNPS, ANDNPD - iloczyn logiczny zanegowanego operandu 1 z operandem 2.

Operacje logiczne są często wykorzystywane. Przykładowo, aby wyznaczyć wartość bezwzględną liczb zmiennopozycyjnych typu SINGLE, czy DOUBLE należy wyzerować bity znaku:

```
MASKA_SINGLE DQ 7FFFFFFF7FFFFFFFh,7FFFFFFF7FFFFFFFh
MASKA_DOUBLE DQ 7FFFFFFFFFFFFFFFh,7FFFFFFFFFFFFFFFh
...
MOVUPS XMM7, [MASKA_SINGLE]
MOVUPD XMM6, [MASKA_DOUBLE]
^kamp=uj j M =uj j T= ;Wyznaczenie wartości bezwzględnej elementów wektora
                      zawartego w XMM0,
^kama=uj j NI =uj j S= ;Wyznaczenie wartości bezwzględnej elementów wektora
                      zawartego w XMM1.
```

Operacje różnicy symetrycznej mogą być wykorzystywane do zerowania zawartości rejestrów XMM czy MMX. Zerowanie rejestrów tym sposobem jest bardziej efektywne od zapisu do rejestru wartości 0:

```
mul o==j j NI =j j N      ;Wyzerowanie rejestru MM1,
ul omp=uj j OI =uj j O    ;Wyzerowanie rejestru XMM2,
ul oma=uj j M =uj j M     ;Wyzerowanie rejestru XMM0.
```

Operacja sumy logicznej może posłużyć do ustawiania znaku elementów upakowanych:

```
MASKA_ZNAKU DQ 8000000080000000h,8000000080000000h
...
MOVUPS XMM1, [MASKA_ZNAKU]
l omp=uj j M =uj j N=     ;Ustawienie znaku czterech liczb typu SINGLE.
```

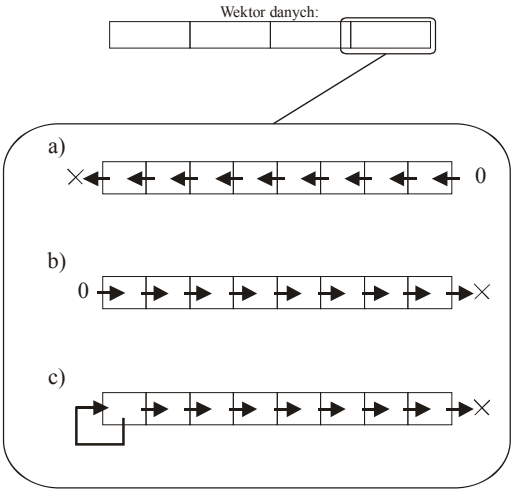
Lista instrukcji logicznych rozszerzeń wektorowych została przedstawiona w rozdz. 5 (tabela 5.6).

2.3.7. Przesunięcia bitowe

Rozszerzenia SIMD dostarczają instrukcji pozwalających na jednoczesne przesuwanie elementów wektorów. W przypadku przesunięć bitowych w lewo, N najstarszych bitów operandu jest traconych bezpowrotnie, natomiast na N najmłodszych bitach zapisywane są zera (przesunięcia logiczne w lewo). Przesunięcia bitowe w prawo charakteryzują się tym, że N najmłodszych bitów jest traconych, natomiast na N najstarszych pozycjach zapisywane są zera – dla przesunięć logicznych, lub powielany jest najstarszy bit (bit znaku) dla przesunięć arytmetycznych.

Schemat realizacji instrukcji przesunięć bitowych, dla jednego elementu wektora został zaprezentowany na rys. 2.26.

Instrukcje PSLLDQ, PSLLQ, PSLLD, PSLLW wykonują przesunięcia logiczne w lewo dla elementów o rozmiarze odpowiednio: 128 bitów (rejestr XMM), 64 bity



Rys. 2.26. Przesunięcia bitowe: a) logiczne w lewo, b) logiczne w prawo, c) arytmetyczne w prawo

(rejestr MMX lub dwa elementy wektora w XMM), 32 bity (dwa elementy w MMX lub cztery w XMM) i 16 bitów (4 elementy w MMX lub 8 w XMM). Instrukcje PSRLDQ, PSRLQ, PSRLD, PSRLW przesuwają operand w prawo, natomiast instrukcje PSRAW i PSRAD wykonują przesunięcia arytmetyczne w prawo.

Operacje przesunięć logicznych i arytmetycznych mogą zostać wykorzystane do mnożenia lub dzielenia całkowitoliczbowego elementów wektora przez potęgę dwójki. Wykonanie operacji przesunięcia bitowego w prawo o *n* pozycji jest równoważne z podzieleniem operandu przez 2^n. Przesunięcie elementu w lewo o *n* pozycji jest równoważne mnożeniu elementu przez 2^n:

```
PSLLW MM1, 1            ;Przesunięcie logiczne 4 elementów MM1 o jeden bit
                          w lewo (mnożenie przez 2),
PSLLDQ XMM0, 3          ;Przesunięcie logiczne operandu 1 o 24 bity w lewo
                          (3*8bitów),
PSRAD XMM3, 3           ;Przesunięcie arytmetyczne 4 elementów XMM3 o
                          3 bity w prawo (dzielenie przez 8 liczb ze ze zna-
                          kiem),
PSRLW MM1, MM2          ;Przesunięcie logiczne czterech elementów MM1 o
                          liczbę bitów określoną w rejestrze MM2,
PSRLQ XMM4, [ZMIENNA] ;Przesunięcie logiczne dwóch elementów XMM4 o
                          liczbę bitów określoną przez wartość zmiennej.
```

Lista instrukcji przesunięć bitowych została zamieszczona w rozdz. 5 (tabela 5.7). Przesunięcia bitowe w lewo o *n* pozycji zostały oznaczone symbolem: „<<n", natomiast przesunięcia w prawo: „>>n".

2.3.8. Pozostałe instrukcje

Instrukcje rozszerzeń wektorowych, których nie opisano w dotychczasowej części podrozdziału zostały zgrupowane w rozdz. 5 (tabela 5.8). Odpowiedzialne są one za wykonywanie takich operacji jak:
– inicjacja pracy jednostki MMX (`EMMS`),
– przesyłanie słowa pomiędzy rejestrami `MMX`, `XMM` a rejestrami ogólnego przeznaczania (`PEXTRW`, `PINSRW`),
– zachowanie i odtwarzanie stanu jednostek obliczeniowych FPU, MMX, SSE (`FXSAVE`, `FXRSTOR`),
– zachowanie i odtwarzanie stanu rejestru `MXCSR` z pamięci (`STMXCSR`, `LDMXCSR`),
– zarządzanie pamięcią podręczną cache,
– synchronizacja międzyprocesowa.

2.4. PODSUMOWANIE

W rozdziale zawarto podstawowe informacje niezbędne do prawidłowego programowania jednostek SIMD procesorów rodziny x86. Przedstawiono podstawowe struktury danych, na których operują jednostki SIMD: wektory liczb całkowitych bez znaku i ze znakiem oraz wektory liczb zmiennopozycyjnych w standardzie IEEE-754 – typu SINGLE oraz DOUBLE.

Opisano architekturę jednostek SIMD w procesorach x86: MMX, SSE, SSE2 oraz SSE3, wraz z rejestrami roboczymi, których może używać programista w swoich programach.

Szczególnie ważne dla programisty są informacje o instrukcjach, których może używać. Instrukcje procesora dla jednostek wektorowych uporządkowano zgodnie z pełnioną funkcją, niezależnie od typu rozszerzenia. Autorzy uważają, że ten sposób przedstawienia instrukcji pozwala na sprawne programowanie jednostek wektorowych.

Rozdział został oparty głównie na dokumentacji firmy INTEL: [5-12]. Czytelnika zainteresowanego bardziej szczegółowym poznaniem przedstawionych zagadnień odsyłamy do następujących pozycji: [15, 17, 19, 20, 21, 24, 25].

3. EDYCJA I TRANSLACJA PROGRAMÓW

Dla programisty w języku Asembler są istotne narzędzia, których będzie używał podczas tworzenia programów. Do podstawowych narzędzi należy zaliczyć programy tłumaczące programy źródłowe, konsolidatory tworzące pliki wykonywalne dla konkretnych systemów operacyjnych oraz edytory tekstu pozwalające na sprawną edycję tekstów. W miejsce tradycyjnych narzędzi komercyjnych firm Borland oraz Microsoft, autorzy postanowili zaproponować czytelnikowi narzędzia niekomercyjne. Są to bardzo efektywne i szeroko wykorzystywane w środowisku programistów niskopoziomowych programy: NASM oraz SciTE for U3.

3.1. NETWIDE ASSEMBLER

Netwide Assembler (określany skrótem NASM) jest translatorem języka Asembler, udostępnianym na nieodpłatnej licencji LGPL. NASM.EXE jest programem, który dokonuje tłumaczenia oraz konsolidacji programu źródłowego do formatu COM, OBJ lub BIN. Translator posiada swoje wersje dla wielu systemów operacyjnych (LINUX, Windows, DOS). Translacja programów do formatu wykonywalnego COM przy użyciu NASM-a odbywa się następująco:

```
NASM – F BIN KOD_ZRODLOWY.ASM –O PROGRAM.COM
```

gdzie:

`KOD_ZRODLOWY.ASM` – nazwa pliku z kodem źródłowym programu,
`PROGRAM.COM` – docelowa nazwa pliku z programem wykonywalnym.

3.1.1. Struktura programu

Szkielet programu źródłowego dla formatu wykonywalnego typu COM ma następującą postać:

```
1: xl od=NMMez=
2:==j l s=` uI=` p
3:==j l s=apI=` u
4:==j l s=^eI=V
5:==j l s=auI=qbhpq
6:==f kq=ONe
```

7:==j l s =^uI =M
8:==f kq=NSe
9:==j l s ==^uI Q` MMe
10:==f kq==ONe
11: qbhpq=a_=¢ml apq^t l t v=pwhf bi bq=mol do^j r =k^pj =A≤

Program wypisuje na ekranie monitora napis „PODSTAWOWY SZKIELET PROGRAMU NASM", po czym oczekuje na naciśnięcie dowolnego przycisku klawiatury przed swoim zakończeniem.

Pliki wykonywalne typu COM zawierają jeden wspólny segment, przeznaczony dla programu (kodu) oraz danych, o maksymalnym rozmiarze 64 kB. W związku z tym, w prezentowanym programie, bezpośrednio za ostatnią instrukcją programu, następuje deklaracja zmiennej TEKST (linia 11) będącej tablicą znaków ASCII. Translator, tworząc plik wykonywalny COM, dokonuje przesunięcia adresów o wartość 100h (na początku programu znajduje się dyrektywa ORG 100H). W liniach 2 – 10 zawarte są instrukcje programu. W liniach 2 – 3 do rejestru segmentowego DS zostaje przepisana zawartość rejestru segmentowego CS, tak aby segment danych był umieszczony w tym samym miejscu co segment programu. Wypisanie znaków zawartych w zmiennej TEKST następuje w liniach 4 – 6. Do tego celu używana jest funkcja 9 przerwania 21h, która wymaga podania w rejestrze DX adresu zmiennej TEKST. Znaki są wypisywane kolejno aż do napotkania znaku „A".

W liniach 7 – 8 następuje wstrzymanie wykonywania programu aż do naciśnięcia dowolnego przycisku klawiatury. Odbywa się to poprzez wywołanie funkcji 0 przerwania 16h. Linie 9 – 10 kończą działanie programu poprzez wywołanie funkcji 4Ch przerwania 21h.

3.1.2. Zmienne

Deklaracja zmiennej w składni NASM-a składa się z trzech członów. W pierwszej kolejności należy podać nazwę zmiennej, a następnie jej typ oraz wartość:

```
NAZWA_ZMIENNEJ      TYP   WARTOŚĆ
```

Nazwa zmiennej może być dowolnym ciągiem liter, cyfr oraz znaków podkreślenia, przy czym pierwszym znakiem nazwy nie może być cyfra. Typ określa rozmiar pamięci zajmowanej przez zmienną. Netwide Assembler umożliwia deklarowanie zmiennych o rozmiarze 8 bitów (dyrektywa DB), 16 bitów (dyrektywa DW), 32 bity (dyrektywa DD), 64 bity (dyrektywa DQ) oraz 80 bitów (dyrektywa DT). Ostatni człon deklaracji umożliwia zainicjowanie wartości zmiennej. W przypadku wartości całkowitoliczbowych możliwy jest zapis w formacie dziesiętnym (format domyślny), heksadecymalnym (znak „h" na końcu liczby) lub binarnym (znak „b" na końcu liczby). Dla liczb zmiennopozycyjnych inicjacja odbywa się poprzez podanie liczby w postaci części całkowitej i ułamkowej, oddzielonych kropką.

Przykładowe deklaracje zmiennych mogą przyjmować następującą postać:

```
ZMIENNA_1 DB  00010001b=  ;Zmienna bajtowa o wartości 17,=
LICZNIK   DW  500h=       ;Zmienna dwubajtowa o wartości 1280,
WEKTOR    DD  200, 300=   ;Wektor dwóch zmiennych 32-bitowych,
SINGLE_   DD  1.234=      ;32-bitowa liczba typu SINGLE,
DOUBLE_   DQ  1.34324252= ;64-bitowa liczba typu DOUBLE.
```

Odczyt oraz zapis wartości zmiennej jest realizowany z zastosowaniem instrukcji `MOV`, umieszczając nazwę zmiennej w nawiasach kwadratowych. W celu odczytania do rejestru `AX` wartości zmiennej `X` należy użyć instrukcji:

```
MOV AX, [X]
```

Pobranie OFFSET-u zmiennej (adresu zmiennej względem początku segmentu danych) wykonywane jest poprzez podanie nazwy zmiennej:

j l s =a f I =u

3.1.3. Tablice

Tablice są złożonymi strukturami danych przechowującymi elementy tego samego typu. Deklaracja tablicy w składni NASM ma następującą postać:

```
NAZWA_TABLICY TIMES WIELKOŚĆ   TYP   WARTOŚĆ
```

Pierwszym elementem deklaracji jest nazwa tablicy. Następnie występuje dyrektywa `TIMES`, informująca translator o liczbie elementów tablicy (parametr `WIELKOŚĆ`). Wszystkie elementy tablicy posiadają rozmiar określony parametrem qvmoraz wartość zainicjowaną parametrem `WARTOŚĆ`.

Przykładowe deklaracje tablic mogą przyjmować następującą postać:

```
T1 TIMES 10 DB 0              ;Deklaracja 10-elementowej tablicy bajtów,
T2 TIMES 1000 DD 0FFFFFFFFh=  ;Deklaracja 1000-elementowej tablicy da-
                               nych 32-bitowych,
T3 TIMES 10*10 DW 12          ;Deklaracja 100 elementowej tablicy słów.
```

Odwołanie do konkretnego elementu tablicy następuje poprzez podanie w nawiasie kwadratowym nazwy tablicy oraz przesunięcia elementu względem jej początku. Np. chcąc odwołać się do 100 elementu tablicy `X` złożonej z 32-bitowych wartości, należy użyć następujących instrukcji:

```
MOV [X+100*4], ECX            ;Zapis setnego elementu tablicy X,
MOV EDX, [X+100*4]            ;Odczyt setnego elementu tablicy X.
```

Należy zwrócić uwagę, że indeks 100 został przeskalowany przez 4, ponieważ każdy element tablicy zajmuje 4 bajty.

3.1.4. Procedury

Procedura (podprogram) jest wydzieloną częścią programu, stworzoną w celu wykonywania określonych działań. W NASM-ie każda procedura rozpoczyna się

etykietą stanowiącą jej nazwę, natomiast kończy - instrukcją `RET`. W programie głównym wywołanie procedury następuje poprzez zamieszczenie instrukcji `CALL` z parametrem będącym nazwą procedury.

Przykład procedury wypisującej na ekranie monitora tekst o adresie względnym zawartym w rejestrze DX, można przedstawić następująco:

```
£
` ^i i =t vmf pw=     ;Wywołanie procedury w programie głównym.
£
                      ;Deklaracja procedury wypisującej tekst
t vmf pwW=            ;Etykieta określająca początek procedury.
==j l s=^eI=V=
==f kq=ONe=
obq=                  ;Instrukcja powrotu do programu głównego.
```

Wewnątrz procedury można stosować tak zwane etykiety lokalne, które charakteryzują się tym, że ich nazwa jest poprzedzona kropką. Nazwa etykiety lokalnej, w przeciwieństwie do nazwy etykiety globalnej, może zostać powtórzona wewnątrz innej procedury. Np.:

```
£
` ^i i =mol ` N
` ^i i =mol ` O
KKK
mol ` NW
==j l s=` uI=RM
==Kmbqi ^W            ;Etykieta lokalna procedury PROC1
====KKK
==i l l m=mbqi ^
obq

mol ` OW
==j l s=` uI=NRMM
==Kmbqi ^W=           ;Etykieta lokalna procedury PROC2
  £
==i l l m=mbqi ^
obq
```

3.1.5. Makropolecenia

Makropolecenia umożliwiają umieszczenie wcześniej zdefiniowanego ciągu instrukcji w miejscu wywołania makropolecenia. Deklaracja makropolecenia w NASM-ie jest następująca:

```
%MACRO NAZWA_MAKRA LICZBA_PARAM
```

```
        INSTRUKCJE
%ENDMACRO
```

Pomiędzy dyrektywami `%MACRO` i `%ENDMACRO` należy zawrzeć wszystkie instrukcje wchodzące w skład makropolecenia. Bezpośrednio za dyrektywą `%MACRO` należy podać nazwę makropolecenia oraz liczbę wykorzystywanych parametrów. Wewnątrz makropolecenia odwołanie do kolejnych parametrów odbywa się poprzez podanie ich indeksu poprzedzonego znakiem „%". Przykładowe makropolecenie, wykonujące dodawanie dwóch rejestrów, może przybrać następującą postać:

```
KKK
al a^g=^uI =_u      ;Wywołanie makropolecenia
KKK
Bj ^` ol =al a^g=O  ;Deklaracja makropolecenia DODAJ z dwoma parametrami
==^aa=BNI =BO       ;Dodanie zawartości rejestrów
Bbkaj ^` ol         ;Koniec makropolecenia
```

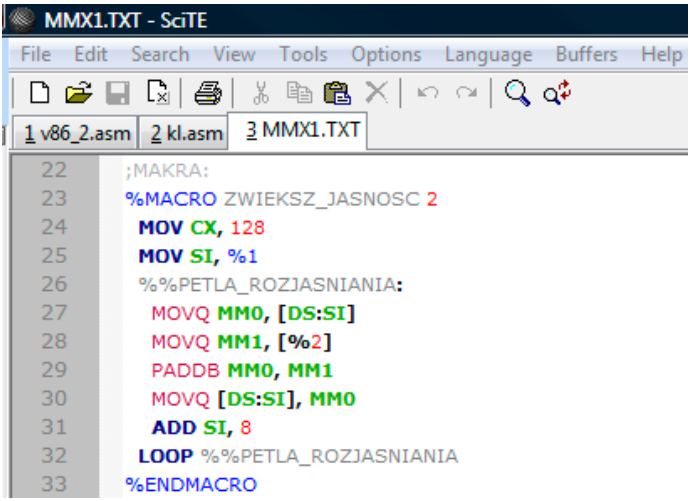
3.2. SciTE for U3

SciTE for U3 jest darmowym, mobilnym edytorem kodu źródłowego najpopularniejszych języków programowania (platforma U3 umożliwia uruchamianie programów zainstalowanych w pamięci flash, bez konieczności ich instalacji na konkretnym komputerze). Wśród rozpoznawanej składni znajdują się instrukcje asemblera, dzięki czemu wyświetlany kod dzielony jest kolorystycznie na instrukcje, nazwy rejestrów, dyrektywy, komentarze, co w znacznym stopniu ułatwia edycję. SciTE for U3 numeruje kolejne wiersze kodu źródłowego, co w przypadku błędu kompilacji pozwala w szybki sposób dotrzeć do linii zawierającej błąd. Edytor umożliwia również jednoczesną pracę z wieloma plikami, których zawartość wyświetlana jest w oddzielnych kartach. Przykładowy ekran edycji programu napisanego w składni NASM został zamieszczony na rys. 3.1.

Oprócz wyżej wymienionych funkcji, dużą zaletą edytora jest rozbudowany system poszukiwania i zastępowania określonego ciągu znaków oraz podpowiadanie składni w oparciu o pliki konfiguracyjne.

3.3. PODSUMOWANIE

Autorzy proponują czytelnikowi wykorzystanie do programowania w asemblerze narzędzi niekomercyjnych: translatora i konsolidatora NASM oraz edytora SciTE for U3. Programy te są cieszą się bardzo wysoką opinią wśród programistów i są bardzo szeroko stosowane [26]. NASM daje możliwość tworzenia pro-



Rys. 3.1. Zrzut ekranu z edycji pliku MMX1.TXT

gramów w asemblerze dla różnych systemów operacyjnych (DOS, WINDOWS, LINUX).

4. PROGRAMOWANIE JEDNOSTEK SIMD

W rozdziale przedstawiono zasady programowania jednostek SIMD procesorów rodziny x86, w oparciu o przykładowe programy realizujące wybrane operacje z zastosowaniem rozszerzeń wektorowych MMX, SEE, SSE2 i SSE3. Każdy przykład zawiera krótkie wprowadzenie do realizowanego przez program algorytmu, charakterystykę wybranych fragmentów programu źródłowego oraz demonstrację wyników działania programu. Na koniec przeprowadzono analizę szybkości obliczeń wektorowych.

4.1.WYKRYWANIE OBECNOŚCI JEDNOSTEK SIMD

Nowsze typy procesorów rodziny x86 udostępniają mechanizmy, które były niedostępne w starszych wersjach. Przykładowo, program wykorzystujący wszystkie możliwości procesora Pentium IV, nie będzie mógł być wykonywany na komputerze z procesorem 80386. Podczas tworzenia oprogramowania, które powinno być poprawnie wykonywane zarówno na najnowszych jak i na starszych komputerach, należy opracować kilka wariantów programu, przystosowanych dla konkretnych architektur. Wykonywany program powinien być w stanie, bez ingerencji użytkownika, określić właściwy wariant kodu który należy wywołać na konkretnym komputerze.

Przykładowo, aplikacja obliczeniowa wykonująca dodawanie dwóch macierzy liczb rzeczywistych, uruchomiona na komputerze z procesorem Pentium IV, powinna wykorzystywać zestaw dostępnych instrukcji SSE2, w celu zwiększenia szybkości działania. Natomiast ta sama aplikacja uruchomiona na komputerze z procesorem Pentium, może wyłącznie wykorzystywać instrukcje zmiennoprzecinkowej jednostki arytmetycznej (FPU).

Poczynając od procesorów Pentium, do listy rozkazów procesorów rodziny x86 dodano instrukcję `CPUID`, umożliwiającą identyfikację możliwości jednostki centralnej. Wywołanie instrukcji `CPUID` w przypadku, gdy rejestr `EAX` zawiera wartość 1, zwraca w rejestrach `EAX`, `EBX`, `ECX` oraz `EDX` wartości pozwalające na jednoznaczne określenie dostępnych funkcji procesora. Obecność rozszerzeń wektorowych jest określona przez bity rejestru `EDX` oraz `ECX`. Rozszerzenie MMX jest dostępne w przypadku, gdy bit nr 23 rejestru `EDX` jest w stanie 1, obecność roz-

szerzenia SSE określana jest bitem nr 25, rozszerzenie SSE2 bitem nr 26. Informacja na temat dostępności SSE3 jest zapisana na najmłodszym bicie rejestru ECX.

Program `PROG1.ASM` wykorzystuje instrukcję `CPUID` do określenia dostępnych rozszerzeń wektorowych. Poniżej został zamieszczony fragment programu (kompletny program znajduje się w pliku `PROG1.ASM`), wywołującego instrukcję `CPUID` oraz testującego stan poszczególnych bitów rejestrów `EDX` oraz `ECX`.

```
1:   jls=b^uI=Ne
2:   `mrfa==
;Testowanie obecności jednostki MMX
3:   qbpq=bauI=UMMMMMe=          ;23 bit rejestru EDX
4:   gw=kfbl_b`k^|jju
5:   ==jls=auI=quq|jju
6:   ==jls=^eI=Ve
7:   ==fkq=ONe
8:   kfbl_b`k^|jjuW
;Test obecności jednostki SEE
9:   qbpq=bauI=OMMMMMMe=         ;25 bit rejestru EDX
10:  gw=kfbl_b`k^|ppb
11:  ==jls=auI=quq|ppb
12:  ==jls=^eI=Ve
13:  ==fkq=ONe
14:  kfbl_b`k^|ppbW==
;Testowanie obecności jednostki SSE2
15:  qbpq=bauI=QMMMMMMe==        ;26 bit rejestru EDX
16:  gw=kfbl_b`k^|ppbO
17:  ==jls=auI=quq|ppbO
18:  ==jls=^eI=Ve
19:  ==fkq=ONe
20:  kfbl_b`k^|ppbOW==
;Testowanie obecności jednostki SSE3
21:  qbpq=b`uI=Ne=               ;Najmłodszy bit rejestru ECX
22:  gw=kfbl_b`k^|ppbP
23:  ==jls=auI=quq|ppbP
24:  ==jls=^eI=Ve
25:  ==fkq=ONe
26:  kfbl_b`k^|ppbPW
```

W liniach 1 – 2 następuje wywołanie instrukcji `CPUID`. Obecność rozszerzenia MMX jest sprawdzana w liniach 3 – 8, gdzie w pierwszej kolejności jest testowany bit nr 23 rejestru `EDX` (linia 3). W przypadku, gdy bit ten jest w stanie 1, zostaje wyzerowana flaga `Z` (zero) w rejestrze `EFLAGS`, w związku z czym skok

warunkowy w linii 4 nie zostanie wykonany (skok przy ustawionej fladze `Z`). Kolejno w liniach 5 – 7, na monitor ekranowy zostaje wyprowadzony tekst o obecności jednostki MMX.

Analogiczne czynności są podejmowane w celu określenia obecności rozszerzenia SSE (linie 9 – 14), SSE2 (linie 15 – 20) oraz SSE3 (linie 21 – 26).

Wykonanie programu na komputerze z procesorem Core 2 Duo E6750 przyniosło efekty przedstawione na rysunku 4.1 (wszystkie badane rozszerzenia SIMD są dostępne).

```
C:\nasm>program1
TEST DOSTEPNOSCI JEDNOSTEK SIMD:
DOSTEPNE INSTRUKCJE MMX.
DOSTEPNE INSTRUKCJE SSE.
DOSTEPNE INSTRUKCJE SSE2.
DOSTEPNE INSTRUKCJE SSE3.
```

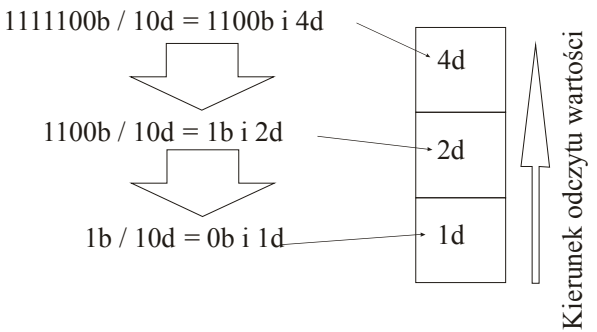
Rys. 4.1. Test dostępności rozszerzeń wektorowych dla procesora Core 2 Duo

4.2. WYPROWADZANIE LICZB CAŁKOWITYCH I ZMIENNOPOZYCYJNYCH

W rozdziale 1.1 przedstawiono formaty liczb całkowitych i zmiennopozycyjnych, którymi posługują się jednostki SIMD procesorów x86. W programach bardzo często mamy do czynienia z koniecznością wyprowadzenia na ekran wartości liczb całkowitych oraz zmiennopozycyjnych w postaci dziesiętnej. Nim przejdziemy do programowania jednostek wektorowych, przedstawiono algorytmy zamiany danych binarnych na reprezentacje dziesiętne oraz ich przykładową implementację (wszystkie prezentowane makropolecenia zostały zamieszczone w pliku `LICZBY.ASM`).

Zamiana całkowitej liczby binarnej bez znaku na liczbę dziesiętną wymaga wielokrotnego wykonania operacji dzielenia przez `10d`. Reszty powstające w wyniku dzielenia są kolejnymi cyframi dziesiętnymi liczby. Cyfry te należy zapamiętywać w tablicy (lub na stosie). Proces wyznaczania reszt kończy się w momencie, gdy wynik dzielenia jest równy 0. Aby wyprowadzić liczbę na ekran, należy wyświetlić wszystkie otrzymane reszty z dzielenia w odwrotnym porządku (po uprzedniej zamianie ich na znaki ASCII). Przykładowa zamiana liczby binarnej 1111100 na dziesiętną została przedstawiona na rys. 4.2.

Wyprowadzenie na ekran dziesiętnej wartości liczby binarnej ze znakiem (kod U2) wymaga nieco bardziej rozbudowanego algorytmu. W przypadku, gdy najstarszy bit liczby jest równy 0 (liczba dodatnia) sposób postępowania jest analogiczny do prezentowanego wcześniej. Jeżeli natomiast najstarszy bit liczby jest w stanie

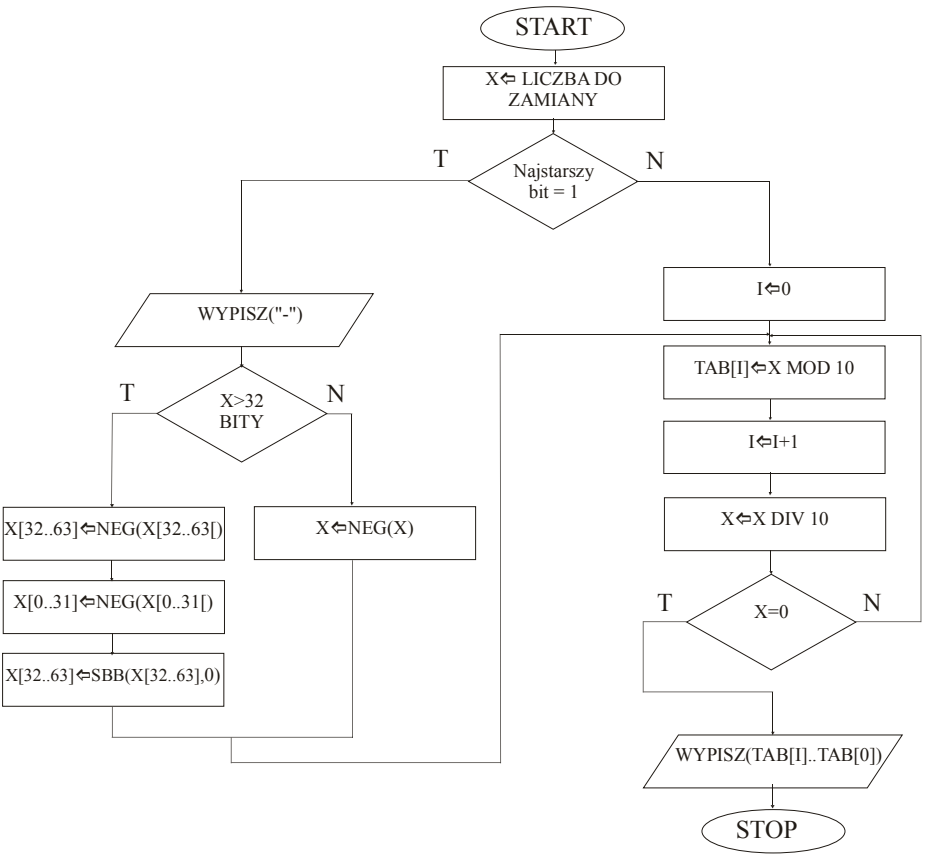


Rys. 4.2. Zamiana liczby binarnej na dziesiętną

1, należy zapamiętać znak liczby, po czym wyznaczyć jej negację (instrukcja `NEG`). Dla liczb mieszczących się w rejestrze 32-bitowym, dalsze czynności polegają na wykonywaniu wielokrotnego dzielenia przez `10d`. Gdy liczba przekracza rozmiar 32-bitów (rozszerzenia SIMD umożliwiają wykonywanie operacji na 64-bitowych liczbach całkowitych), negację rozpoczyna się od najstarszego podwójnego słowa liczby. W następnej kolejności neguje się młodszą część liczby, po czym należy przeprowadzić operację odejmowania zera z pożyczką (instrukcja `SBB`) na najstarszym podwójnym słowie liczby (w przypadku gdy druga negacja zgłosiła pożyczkę, należy ją odjąć od starszej części liczby). Schemat blokowy obrazujący kolejne czynności podejmowane podczas zamiany liczby w kodzie U2 na postać dziesiętną został przedstawiony na rys. 4.3.

Algorytm przedstawiony na rys. 4.3 jest realizowany przez makropolecenie `PRINT_INT`,= zawarte w pliku `LICZBY.ASM`. Makropolecenie wykorzystuje 4 parametry – adres zmiennej zawierającej liczbę binarną, rozmiar liczby (jednostką są 32-bity), obszar pamięci, do którego będą zapisywane kolejne reszty z dzielenia oraz adres zmiennej tymczasowej (wykorzystywanej podczas wykonywania operacji). Treść makropolecenia jest następująca:

```
1: Bj ^` ol =mofkq| fkq=Q= ;Parametry: 1 - adres liczby, 2 - rozmiar liczby wyrażony jako wielokrotność 4B, 3 - adres zmiennej tekstowej, 4 - zmienna tymczasowa
2: j l s=pfI=BN
3: j l s=afI=BQ
;Kopiowanie liczby do zmiennej tymczasowej
4: Bobm=BO
5: ==j l s=b^uI=xapWpfz=
6: ==j l s=xapWafzI=b^u
7: ==^aa=pfI=Q=
8: ==^aa=afI=Q
9: Bbkaobm
```



Rys. 4.3. Zamiana liczby binarnej w kodzie U2 na liczbę dziesiętną

```
;Badanie znaku
10: MOV SI, %4
11: MOV EAX, [DS:SI+%2*4-4]
12: AND EAX, 80000000H
13: MOV BYTE [ZNAK], ' '
14: CMP EAX, 0
15: JE %%LICZBA_BEZ_ZNAKU
16: ==NEG_INT %4, %2
17: ==MOV BYTE [ZNAK], '-'
18: %%LICZBA_BEZ_ZNAKU:
19: WYPISZ_INT %4, %2, %3
20: MOV SI, %3
21: MOV DI, %3
22: DEC BX
```

```
23: ADD DI, BX
24: MOV AX, BX
25: XOR DX, DX
26: MOV CX, 2
27: DIV CX
28: MOV CX, AX
29: %%ZMIANA_KOLEJNOSCI_ZNAKOW:
30: ==MOV AL, [DS:SI]
31: ==XCHG AL, [DS:DI]
32: ==MOV [DS:SI], AL
33: ==INC SI
34: ==DEC DI
35: LOOP %%ZMIANA_KOLEJNOSCI_ZNAKOW
36: %ENDMACRO
```

W liniach 2–9 następuje skopiowanie wartości zmiennej do obszaru pamięci określonego przez parametr czwarty. Następnie w liniach 10–14 zostaje określony znak zmiennej. W przypadku, gdy liczba jest ujemna, w liniach 16–17 następuje jej negacja poprzez wywołanie makropolecenia `NEG_INT` oraz zapamiętanie znaku w zmiennej `ZNAK`. W kolejnym kroku wywołane zostaje makropolecenie `WYPISZ_INT`, które wykonuje operację wielokrotnego dzielenia liczby przez `10d`. Uzyskane reszty z dzielenia zostają zamienione na znaki ASCII, po czym następuje ich zapisanie w kolejnych pozycjach zmiennej tablicowej (określonej trzecim parametrem). Makropolecenie `WYPISZ_INT` zwraca w rejestrze `BX` liczbę pozycji dziesiętnych liczby. W liniach 29–35 następuje odwrócenie kolejności zapamiętanych znaków tak, aby późniejsza operacja wyprowadzenia zmiennej tablicowej na ekran wyświetliła cyfry we właściwym porządku.

Wyprowadzanie na ekran liczb zmiennopozycyjnych jest realizowane przez makropolecenie `WYPISZ_DEC`, wykorzystujące 5 parametrów:

– adres zmiennej zawierającej liczbę zmiennopozycyjną,
– adres zmiennej pomocniczej,
– adres tablicy, do której zostanie zapisanych 18 cyfr dziesiętnych w kodzie BCD części całkowitej liczby,
– adres tablicy, w której zostanie zapamiętane 18 cyfr dziesiętnych w kodzie BCD części ułamkowej liczby,
– dyrektywa określająca format liczby zmiennopozycyjnej (dla liczby typu SINGLE `DWORD`, dla liczby typu DOUBLE - `QWORD`).

Konwersja liczby zmiennopozycyjnej na spakowany format BCD, odbywa się z wykorzystaniem możliwości jednostki FPU. Wykonywane jest to przy użyciu instrukcji `FBSTP`, która zamienia liczbę w formacie IEEE 754 SINGLE lub DOUBLE, znajdującą się na szczycie stosu FPU, na wartość 80-bitowej reprezentacji BCD. Kolejne czterobitowe liczby w uzyskanym formacie (za wyjątkiem najstarszego

bajtu, który określa znak liczby) odpowiadają kolejnym cyfrom dziesiętnym konwertowanej liczby. Przykładowo, konwersja liczby 987234.0 przyniesie następujący rezultat:
wartości dziesiętne:

```
          0    0   ...   9    8    7    2   3    4
```

wartości binarne w upakowanym kodzie BCD:

```
        0000 0000 ... 1001 1000 0111 0010 0011 0100
```

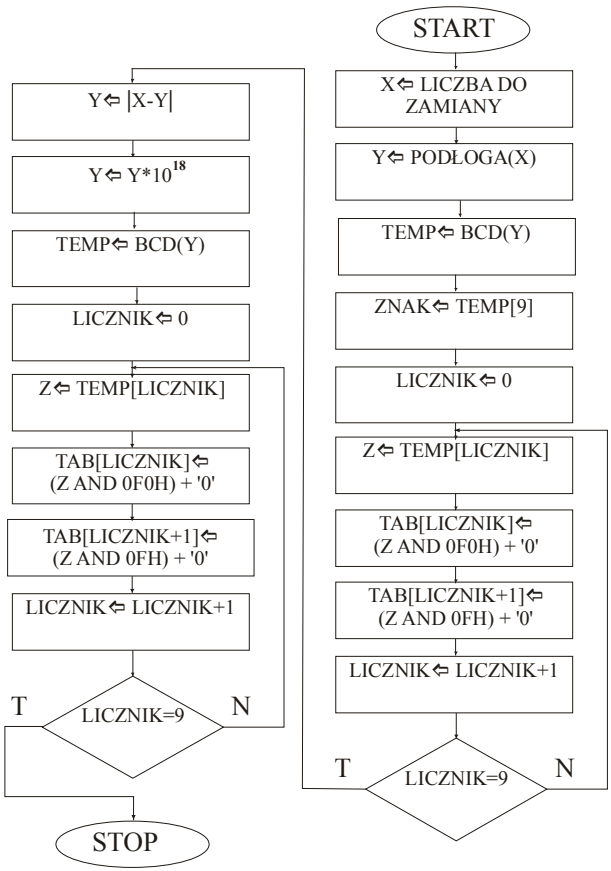
W pierwszej kolejności makropolecenie wyznacza 18 cyfr części całkowitej, a następnie 18 cyfr części ułamkowej. Zamiana cyfr dziesiętnych w kodzie BCD (wskazywanych parą rejestrów `DS:SI`) na znaki ASCII jest realizowana w następującej pętli:

```
1: %REP 9
2: ==MOV AL, [DS:SI]
3: ==AND AL, 0F0H
4: ==SHR AL, 4
5: ==ADD AL, '0'
6: ==MOV [DS:DI], AL
7: ==INC DI
8: ==MOV AL, [DS:SI]
9: ==AND AL, 0FH
10: ==ADD AL, '0'
11: ==MOV [DS:DI], AL
12: ==INC DI
13: ==DEC SI
14: %ENDREP
```

W liniach 2–4 oraz 8–9 uzyskiwane są kolejne pozycje BCD, reprezentujące cyfry dziesiętne (iloczyn logiczny zeruje odpowiednio 4 młodsze oraz 4 starsze bity). W liniach 5 oraz 10 do uzyskanej cyfry zostaje dodany kod ASCII cyfry „0", dzięki czemu w rejestrze AL otrzymuje się kod ASCII odpowiadający danej cyfrze. W liniach 6 oraz 12 ma miejsce zapis kolejnych pozycji dziesiętnych liczby do zmiennej tablicowej, która posłuży do wyprowadzania na ekran wartości liczby. Schemat blokowy ilustrujący kolejne kroki wykonywane przez makropolecenie zamieszczono na rys. 4.4.

4.3. JEDNOSTKA MMX I PODSTAWOWE OPERACJE NA BITMAPIE

Rozdział poświęcono zasadom programowania jednostki MMX, na przykładzie programów realizujących wybrane operacje na 24-bitowych plikach graficznych typu BMP.



Rys. 4.4. Schemat blokowy algorytmu zamiany liczb zmiennopozycyjnych na ciąg znaków

4.3.1. Pliki BMP

Pliki BMP (ang. BitMap Files) są najpopularniejszym formatem do przechowywania grafiki w formie bezstratnej. Pliki typu BMP składają się z dwóch części – nagłówka oraz danych obrazu w postaci mapy bitowej. Pierwsze 54 bajty pliku zajmuje nagłówek (ang. bitmap file header), który zawiera podstawowe informacje o przechowywanej grafice. Format danych przechowywanych w nagłówku pliku BMP przedstawiono w tabeli 4.1.

Bezpośrednio za nagłówkiem, w plikach BMP o 24-bitowej głębi kolorów (ang. TrueColor), występują dane charakteryzujące kolory poszczególnych pikseli mapy bitowej. Każdy piksel obrazu reprezentowany jest przez 3 bajty określające jasność kolorów podstawowych z palety RGB (ang. Red-Green-Blue). Pierwszy bajt określa intensywność koloru niebieskiego, drugi zielonego, natomiast trzeci czerwonego (kolory z palety RGB przechowywane są więc w odwróconym porządku – BGR).

Tabela 4.1.

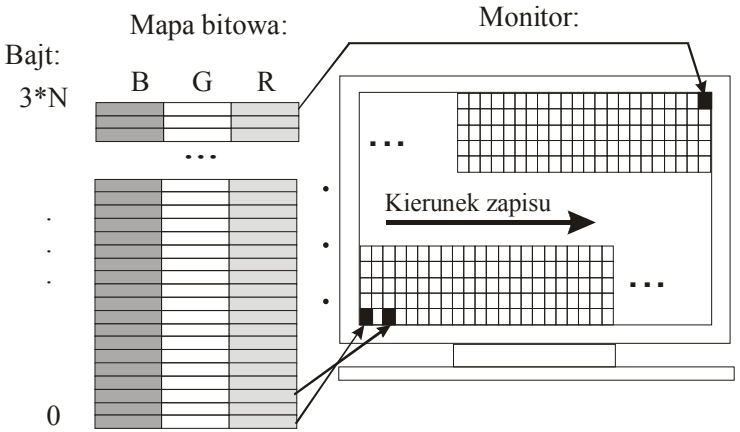
Format pierwszych 54 bajtów pliku BMP

Przesunięcie względem początku pliku	Opis znaczenia pola	Rozmiar w bajtach
0	Typ pliku – dwa bajty o wartości ASCII 'BM'	2
2	Rozmiar pliku wyrażony w bajtach	4
6	Pole zarezerwowane o wartości 0	2
8	Pole zarezerwowane o wartości 0	2
10	Przesunięcie właściwych danych (bitmapy) względem początku pliku – standardowo 54	4
14	Liczba pozostałych bajtów nagłówka (od tej pozycji do końca nagłówka) – przy standardowej wielkości nagłówka 40	4
18	Szerokość bitmapy wyrażona w pikselach	4
22	Wysokość bitmapy wyrażona w pikselach	4
26	Liczba warstw kolorów – standardowo 1	2
28	Liczba bitów wykorzystywana na reprezentację piksela (w przypadku omawianych programów zawsze 24)	2
30	Identyfikator opcjonalnego algorytmu kompresji (0 – bez kompresji, 1 – kompresja RLE8, 2 – kompresja RLE4)	4
34	Rozmiar bitmapy	4
38	Pozioma liczba pikseli na metr (rozdzielczość pozioma)	4
42	Pionowa liczba pikseli na metr (rozdzielczość pionowa)	4
46	Liczba wykorzystywanych kolorów	4
50	Liczba istotnych kolorów w palecie (dla wartości 0 wszystkie kolory są użyte)	1
51	Pole określa, czy ma nastąpić rotacja palety kolorów (gdy 0 nie wystąpi rotacja)	1
52	Pole zarezerwowane o wartości 0	2

Odpowiednia kombinacja intensywności barw podstawowych pozwala na uzyskanie jednego z 16,7 miliona kolorów.

Piksele mapy bitowej (reprezentowane przez 3-bajtowe wartości) są umieszczane na ekranie w liniach poziomych, począwszy od lewej do prawej strony, a kolejne linie obrazu układane są w wiersze od dołu do góry. Na rysunku 4.5 przedstawiono sposób wyświetlania poszczególnych pikseli mapy bitowej (N oznacza liczbę pikseli w mapie bitowej).

Ponieważ realizacja podstawowych operacji na mapie bitowej wymaga przeprowadzenia szeregu operacji plikowych, w programach źródłowych analizowanych



Rys. 4.5. Wyświetlanie bitmapy na ekranie monitora

w niniejszym rozdziale zostały zdefiniowane makropolecenia: `UTWORZ_PLIK`, `OTWORZ_PLIK`, `ZAMKNIJ_PLIK`, `POLOZENIE_WSKAZNIKA`, `ODCZYT_DO_BUFORA` oraz `ZAPISZ_DO_PLIKU`.

Makropolecenie `UTWORZ_PLIK` ma za zadanie utworzyć nowy plik w bieżącym folderze. Wykorzystuje jeden parametr w postaci adresu ciągu znaków określających nazwę tworzonego pliku. Wewnątrz makropolecenia następuje wywołanie funkcji `3Ch` przerwania `21h`, wykorzystującej następujące parametry:

– Rejestr `CL` zawiera atrybuty nowo tworzonego pliku. Zerując wszystkie bity rejestru `CL` otrzymany zostanie format pliku zwykłego.
– Para rejestrów `DS:DX` wskazuje na ciąg znaków określający ścieżkę dostępu do pliku. Wskazywany ciąg znaków musi być zakończony bajtem o wartości `0`.

Zanim zostaną podjęte jakiekolwiek operacje na istniejącym pliku, w pierwszej kolejności należy go otworzyć. Do tego celu zostało przygotowane makropolecenie `OTWORZ_PLIK`. Makropolecenie wykorzystuje dwa parametry, w postaci adresu ciągu znaków określających ścieżkę dostępu do pliku oraz adresu zmiennej do której zostanie zapisany uchwyt pliku (uchwyt jest identyfikatorem pliku wykorzystywanym w programie). Do otwarcia pliku makropolecenie wykorzystuje funkcję `3Dh` przerwania `21h`, która wymaga podania dwóch parametrów:

– Para rejestrów `DS:DX` zawiera adres ciągu znaków określającego ścieżkę dostępu do pliku (ciąg znaków musi być zakończony bajtem o wartości `0`).
– Rejestr `AL` powinien zawierać prawa dostępu do pliku.

W przypadku prawidłowej realizacji funkcji, rejestr `AX` będzie zawierał uchwyt do pliku.

Po wykonaniu wszystkich zamierzonych operacji na pliku, należy go zamknąć. Do tego celu zostało przygotowane makropolecenie `ZAMKNIJ_PLIK`, które wykorzystuje jeden parametr w postaci uchwytu do zamykanego pliku. Makropolecenie

wywołuje funkcję 3Eh przerwania 21h, która wymaga podania w rejestrze BX uchwytu do zamykanego pliku.

Podstawową operacją, jaką można wykonać na pliku, jest zapisanie do niego pewnej porcji danych. System DOS udostępnia do tego celu funkcję 40h przerwania 21h. Przed wywołaniem funkcji należy przygotować następujące jej parametry:

– Rejestr BX zawiera uchwyt pliku, do którego będzie odbywała się operacja zapisu.
– Para rejestrów DS:DX zawiera adres bufora, w którym znajdują się dane przeznaczone do zapisania na dysk.
– Rejestr CX określa liczbę bajtów do zapisania.

Jeżeli operacja zakończy się powodzeniem, rejestr AX będzie zawierał liczbę faktycznie zapisanych bajtów. Makropolecenie ZAPISZ_DO_PLIKU wywołuje funkcję 40h przerwania 21h w celu zapisania danych do pliku i wykorzystuje trzy parametry: uchwyt do pliku, liczba bajtów do zapisu oraz adresu bufora, skąd nastąpi pobieranie danych do zapisu.

Makropolecenie ODCZYT_DO_BUFORA wykonuje operację odczytu danych z pliku do bufora i wykorzystuje trzy parametry: uchwyt do pliku, liczba bajtów do odczytu oraz adres bufora. Makropolecenie wykorzystuje funkcję 3Fh przerwania 21h. Sposób jej wywołania jest analogiczny do funkcji 40h:

– Rejestr BX zawiera uchwyt do pliku, z którego będzie następował odczyt.
– Para rejestrów DS:DX zawiera adres bufora, do którego zostaną zapisane odczytywane dane.
– Rejestr CX zawiera liczbę bajtów do odczytu.

W przypadku bezbłędnego wykonania funkcji, zostanie wyzerowana flaga CF, natomiast rejestr AX będzie zawierał liczbę faktycznie przeczytanych bajtów.

Makropolecenie POLOZENIE_WSKAZNIKA aktualizuje wskaźnik określający miejsce, począwszy od którego zostanie wykonana następna operacja odczytu/zapisu w pliku i wykorzystuje trzy parametry: uchwyt do pliku, 16-bitowa wartość określająca starsze słowo nowego położenia wskaźnika, 16-bitowa wartość określająca młodsze słowo nowego położenia wskaźnika. Makropolecenie wykorzystuje funkcję 42h przerwania 21h. Funkcja ta daje możliwość swobodnego dostępu do pliku. Jej działanie bazuje na zmianie położenia wskaźnika pozycji pliku, czyli zmiennej systemowej przechowującej aktualne przesunięcie w pliku (wartość tej zmiennej stanowi miejsce wykonania kolejnej operacji na pliku).

Przed wywołaniem funkcji 42h należy podać następujące parametry:

– Rejestr BX zawiera uchwyt do pliku.
– Rejestr AL określa miejsce w pliku, od którego będzie obliczane przesunięcie. Wartość 0 wskazuje na to, że przesunięcie będzie liczone od początku pliku, wartość 1 określa przesunięcie od bieżącej pozycji, natomiast wartość 2 wskazuje na przesunięcie od końca pliku.

– Para rejestrów CX:DX zawiera informację o liczbie bajtów, o które zostanie zmieniona aktualna pozycja w pliku.

4.3.2. Rozjaśnianie obrazu przy zastosowaniu arytmetyki modularnej (ang. wrap-around)

Zwiększanie jasności wyświetlanego obrazu odbywa się poprzez rozjaśnienie wszystkich pikseli. W przypadku obrazu zachowanego w pliku typu BMP, operacja rozjaśniania polega na dodaniu określonej wartości do każdego bajtu przechowywanej mapy bitowej – w ten sposób zwiększa się intensywność poszczególnych składowych RGB wszystkich pikseli. Rozjaśnianie wymaga więc wykonania 3*N operacji dodawania, gdzie N określa liczbę pikseli wchodzących w skład obrazu.

Wykorzystując możliwości jednostki MMX, liczbę operacji dodawania można zmniejszyć ośmiokrotnie – 64 bitowe rejestry jednostki wektorowej umożliwiają wykonywanie jednoczesnej operacji na 8 bajtach. Polecenie PADDB wykonuje operację dodawania dwóch 8-bajtowych wektorów z wykorzystaniem arytmetyki modularnej (ang. wrap-around). Wykorzystanie tej arytmetyki spowoduje, że jeżeli uzyskana w wyniku dodawania wartość jest większa od 255 (maksymalna wartość możliwa do zapisania na 8 bitach), to zostanie „przycięta" do postaci 8-bitowej poprzez obcięcie bitu najstarszego.

Program MMX1.ASM realizuje operację rozjaśniania mapy bitowej z wykorzystaniem jednostki wektorowej MMX (w arytmetyce modularnej). W pierwszej kolejności program otwiera oryginalny plik BMP oraz tworzy nowy plik, do którego zostanie zapisany obraz rozjaśniony. Do nowego pliku zostaje następnie skopiowany nagłówek BMP pliku oryginalnego:

```
ODCZYT_DO_BUFORA H_PLIK_STARY, 54, BUFOR
ZAPISZ_DO_PLIKU H_PLIK_NOWY, 54,BUFOR
```

Operacja ta jest możliwa ze względu na fakt, że mapy bitowe przechowywane przez obydwa pliki będą posiadały identyczny rozmiar. W kolejnym kroku wykonywana jest pętla, w której następuje rozjaśnienie kolejnych pikseli oryginalnego pliku oraz ich zapis do nowego pliku:

```
1: PETLA_MODYFIKACJI_PLIKU:
2:   ODCZYT_DO_BUFORA H_PLIK_STARY, 1024, BUFOR
3:   PUSH AX
4:   ZWIEKSZ_JASNOSC BUFOR, PRZYROST_JASNOSCI
5:   ZAPISZ_DO_PLIKU H_PLIK_NOWY, AX, BUFOR
6:   POP AX
7:   CMP AX, 1024
8:   JNE KONIEC_ZAPISU_DO_PLIKU
9:   MOV AX, [L_WSKAZNIK_STARY]
10:   MOV DX, [H_WSKAZNIK_STARY]
```

```
11:   ADD AX, 1024
12:   MOV [L_WSKAZNIK_STARY], AX
13:   ADC DX, 0
14:   MOV [H_WSKAZNIK_STARY], DX
15:   MOV AX, [L_WSKAZNIK_NOWY]
16:   MOV DX, [H_WSKAZNIK_NOWY]
17:   ADD AX, 1024
18:   MOV [L_WSKAZNIK_NOWY], AX
19:   ADC DX, 0
20:   MOV [H_WSKAZNIK_NOWY], DX
21:   POLOZENIE_WSKAZNIKA H_PLIK_STARY, H_WSKAZNIK_STARY,
      L_WSKAZNIK_STARY
22:   POLOZENIE_WSKAZNIKA H_PLIK_NOWY, H_WSKAZNIK_NOWY,
      L_WSKAZNIK_NOWY
23:   JMP PETLA_MODYFIKACJI_PLIKU
24:   KONIEC_ZAPISU_DO_PLIKU:
```

W linii 2 następuje odczyt do bufora 1024 bajtów z oryginalnego pliku (liczba faktycznie przeczytanych bajtów jest zwracana w rejestrze `AX`). W linii 4 zostaje wywołane makropolecenie, które zwiększy wartość wszystkich bajtów znajdujących się w buforze, po czym w linii 5 następuje ich zapis do nowego pliku. Gdy zostanie przeczytana mniejsza liczba bajtów od 1024, następuje zakończenie pętli (linie 6–7), ponieważ została odczytana ostatnia porcja bajtów. Przed przejściem do kolejnego przebiegu pętli, w liniach 9–22 następuje modyfikacja o wartość 1024 wskaźników położenia w plikach (tak aby kolejna operacja odczytu/zapisu została przesunięta o 1024 bajty).

Makropolecenie `ZWIEKSZ_JASNOSC` wykorzystuje dwa parametry. Pierwszym parametrem (`BUFOR`F jest adres bufora, w którym będą wykonywane operacje na pikselach. Drugi parametr (`PRZYROST_JASNOSCI`) stanowi adres 8-elementowej tablicy bajtów, przechowującej wartości, które będą dodawane do elementów bufora. Deklaracja tablicy `PRZYROST_JASNOSCI` jest następująca:

```
PRZYROST_JASNOSCI TIMES 8 DB 100
```

Każdy element tablicy został zainicjowany wartością `100d`. Kod źródłowy makropolecenia `ZWIEKSZ_JASNOSC` jest następujący.

```
1: %MACRO ZWIEKSZ_JASNOSC 2
2:   MOV CX, 128
3:   MOV SI, %1
4:   %%PETLA_ROZJASNIANIA:
5:     MOVQ MM0, [DS:SI]
6:     MOVQ MM1, [%2]
7:     PADDB MM0, MM1
8:     MOVQ [DS:SI], MM0
```

```
9:      ADD SI, 8
10:   LOOP %%PETLA_ROZJASNIANIA
11: %ENDMACRO
```

W pierwszej kolejności licznik pętli (rejestr ` u) zostaje załadowany wartością 128 (linia 2), po czym w linii 3 do rejestru indeksowego pf zostaje zapisany adres początku bufora. Wewnątrz pętli, do ośmiu kolejnych bajtów bufora, zostają dodane wartości zapisane w tablicy `PRZYROST_JASNOSCI`, przy użyciu instrukcji `PADDB`. Efekt działania programu `MMX1.ASM` został przedstawiony na rysunku 4.6.

Rys. 4.6a. Obraz oryginalny

Rys. 4.6b. Obraz rozjaśniony z zastosowaniem arytmetyki modularnej

Porównanie rysunków 4.6a oraz 4.6b pozwala stwierdzić, że jaśniejsze obszary obrazu oryginalnego zostały zaciemnione. Jest to związane z faktem, że do

zwiększania jasności pikseli została użyta instrukcja PADDB, która wykorzystuje arytmetykę modularną (przekroczenie wartości 255 powoduje pozostawienie 8 mniej znaczących bitów).

4.3.3. Rozjaśnianie przy zastosowaniu arytmetyki z nasyceniem (saturation)

Program MMX1.ASM posiadał jedną zasadniczą wadę – rozjaśnianie obrazu powodowało zaciemnienie najjaśniejszych jego części. Zjawisko to nie miałoby miejsca, gdyby zamiast arytmetyki modularnej zastosować arytmetykę z nasyceniem. Arytmetyka ta (ang. saturation arithmetic) definiuje inny sposób postępowania w przypadku przekroczenia zakresu liczb binarnych. Wynik operacji, który jest większy od największej możliwej do zapisu liczby dla danego formatu, zostaje zapamiętany jako wartość maksymalna (dla formatu 8-bitowego bez znaku będzie to 255). Wynik mniejszy od najmniejszej możliwej do zapisu liczby dla danego formatu, zostaje zapamiętywany jako wartość minimalna (dla formatu 8-bitowego bez znaku będzie to 0). Instrukcje MMX wykorzystujące arytmetykę z nasyceniem posiadają nazwy kończące się znakami: SB (operacje na bajtach ze znakiem), SW (operacje na słowach ze znakiem), USB (operacje na bajtach bez znaku), USW (operacje na słowach bez znaku).
Program MMX2.ASM wykonuje operację rozjaśnienia mapy bitowej z wykorzystaniem arytmetyki z nasyceniem. W miejsce instrukcji m^aa_ zastosowanej w programie MMX1.ASM, użyto instrukcję PADDUSB, wykonującą dodawanie przy zastosowaniu arytmetyki z nasyceniem. Efekt działania programu MMX2.ASM został zaprezentowany na rys. 4.7.

4.3.4. Tworzenie negatywu mapy bitowej

Kolejnym przykładem zastosowania jednostki wektorowej MMX, wykorzystywanym w grafice komputerowej, jest wyznaczanie negatywu mapy bitowej. Tworzenie negatywu na podstawie obrazu oryginalnego jest wykonywane poprzez wyznaczenie dopełnień kolorów dla wszystkich jego pikseli. W przypadku pliku BMP o 24-bitowej głębi kolorów, odbywa się to poprzez odjęcie od wartości 255 kolejnych bajtów mapy bitowej. Wykorzystanie jednostki MMX umożliwia ośmiokrotne zmniejszenie liczby instrukcji odejmowania.

Struktura programu MMX3.ASM jest analogiczna do MMX1.ASM, a w miejsce operacji dodawania określonych wartości do kolejnych bajtów opisujących poszczególne piksele mapy bitowej, zastosowano makropolecenie NEGATYW. Treść makropolecenia NEGATYW jest następująca.

```
1: %MACRO NEGATYW 2
2:   MOV CX, 128
3:   MOV SI, %1
```

Rys.4.7a. Obraz oryginalny

Rys. 4.7b. Obraz rozjaśniony przy zastosowaniu arytmetyki z nasyceniem

```
4:  %%PETLA_TWORZENIA_NEGATYWU:
5:    MOVQ MM1, [DS:SI]
6:    MOVQ MM0, [%2]
7:    PSUBB MM0, MM1
8:    MOVQ [DS:SI], MM0
9:    ADD SI, 8
10:  LOOP %%PETLA_TWORZENIA_NEGATYWU
11:%ENDMACRO
```

Makropolecenie wykorzystuje dwa parametry: adres bufora oraz adres tablicy zawierającej 8 bajtów o wartościach równych 255:

```
ODJEMNA TIMES 8 DB 255
```

W liniach 4–10 znajduje się pętla, w której od wektora ośmiu bajtów o wartościach równych 255 (w rejestrze j j M) odejmowane są kolejne bajty z bufora (ładowane do rejestru j j N). Odejmowanie jest wykonywane przy użyciu instrukcji npr __ (linia 7), po czym wynik zapisywany jest na bieżącej pozycji w buforze (linia 8). Efekt działania programu `MMX3.ASM` został przedstawiony na rys. 4.8.

Rys. 4.8. Negatyw wyznaczony za pomocą programu `MMX3.ASM`

4.3.5. Binaryzacja mapy bitowej

Proces, w wyniku którego kolorowy, bądź składający się z wielu poziomów szarości, obraz źródłowy zostaje przekształcony do postaci czarno-białej, nazywany jest binaryzacją. Binaryzację obrazu przeprowadza się w oparciu o tzw. próg binaryzacji. Próg binaryzacji jest to wartość graniczna, na podstawie której będzie podejmowana decyzja, czy dany piksel (reprezentowany przez składowe RGB) w obrazie wynikowym wygasić czy też nadać mu kolor biały.

Wyróżnia się dwie podstawowe odmiany binaryzacji – binaryzację z dolnym progiem oraz binaryzację z górnym progiem. Binaryzacja z dolnym progiem wygasza wszystkie piksele o wartości sumy (lub średniej) składowych RGB mniejszej lub równej od progu binaryzacji. Binaryzacja z górnym progiem wygasza piksele o wartości sumy (lub średniej) składowych RGB większej lub równej od progu binaryzacji. Poniższe zależności prezentują sposób wyznaczania koloru piksela w bitmapie wynikowej dla obu przypadków:

$$Pix'(x,y) = \begin{cases} C; RGB(x,y) \le a \\ B; RGB(x,y) > a \end{cases} \qquad Pix'(x,y) = \begin{cases} C; RGB(x,y) \ge a \\ B; RGB(x,y) < a \end{cases}$$

Binaryzacja z dolnym progiem — Binaryzacja z górnym progiem

gdzie: Pix'(x,y) – nowa wartość koloru dla piksela o współrzędnych x i y,

C – kolor czarny,
B – kolor biały,
RGB(x,y) – średnia lub suma składowych RGB piksela,
a – próg binaryzacji.

Program `MMX4.ASM` demonstruje sposób wykorzystania jednostki MMX do przeprowadzenie binaryzacji z dolnym progiem. Dane odczytywane z oryginalnego pliku BMP poddawane są obróbce przez makropolecenie `BINARYZACJA`. Makropolecenie wykorzystuje dwa parametry: adres bufora oraz ośmioelementowa tablica bajtów zawierająca elementy równe progowi binaryzacji.

```
PROG_BINARYZACJI TIMES 8 DB 254
```

Treść makropolecenia jest następująca.

```
1: %MACRO BINARYZACJA 2 ;PARAMETRY: 1- BUFOR, 2- PRÓG BINARYZACJI
2:   MOV CX, 43
3:   MOV SI, %1
4:   MOV DI, PIKSEL
5:   MOV AX, DS
6:   MOV ES, AX
7:   CLD
8:   %%PETLA_BINARYZACJI:
9:     PUSH CX
10:     PUSH SI
11:     MOV DI, PIKSEL
12:     CALL P
13:     PADDUSB MM0, MM1
14:     PADDUSB MM0, MM2
15:     MOVQ MM1, [%2]
16:     PSUBUSB MM0, MM1
17:     MOV DWORD[PIKSEL], 0
18:     MOV DWORD[PIKSEL+4], 0
19:     MOVQ MM1, [PIKSEL]
20:     PCMPEQB MM0, MM1
21:     MOVQ [PIKSEL], MM0
22:     POP SI
23:     MOV DI, SI
24:     PUSH SI
25:     MOV SI, PIKSEL
26:     MOV CX, 8
27:     %%PETLA4:
28:       LODSB
```

```
29:       NOT AL
30:       STOSB
31:       STOSB
32:       STOSB
33:     LOOP %%PETLA4
34:     POP SI
35:     ADD SI, 24
36:     POP CX
37:   LOOP %%PETLA_BINARYZACJI
38: %ENDMACRO
```

W pętli zawartej w liniach 8–37 są wyznaczane kolory pikseli w obrazie wynikowym. Procedura `P`, której wywołanie znajduje się w linii 12, zapisuje do rejestru `MM0` składowe niebieskie, do rejestru `MM1` składowe zielone, natomiast do rejestru `MM2` składowe czerwone ośmiu kolejnych pikseli znajdujących się w buforze. W ten sposób po powrocie z procedury rejestry `MM0`–`MM2` będą zawierały wartości przedstawione na rys. 4.9.

MM0:	B(i)	B(i+1)	B(i+2)	B(i+3)	B(i+4)	B(i+5)	B(i+6)	B(i+7)
MM1:	G(i)	G(i+1)	G(i+2)	G(i+3)	G(i+4)	G(i+5)	G(i+6)	G(i+7)
MM2:	R(i)	R(i+1)	R(i+2)	R(i+3)	R(i+4)	R(i+5)	R(i+6)	R(i+7)

Rys. 4.9. Wartości zawarte w rejestrach `MM0`–`MM2` po powrocie z procedury `P`

Po powrocie z procedury, w liniach 13–14 następuje zsumowanie zawartości rejestrów `MM0`, `MM1` i `MM2` przy zastosowaniu arytmetyki z nasyceniem. W rejestrze `MM0` zostanie wyznaczona suma składowych RGB dla każdego piksela (rys. 4.10).

MM0:	RGB(i)	RGB(i+1)	RGB(i+2)	RGB(i+3)	RGB(i+4)	RGB(i+5)	RGB(i+6)	RGB(i+7)

Rys. 4.10. Zawartość rejestru `MM0` po wykonaniu dodawania

W kolejnym kroku, do rejestru `MM1` zostaje zapisana tablica `PROG_BINARYZACJI`, po czym w linii 16 następuje odjęcie od rejestru `MM0` wartości znajdujących się w rejestrze `MM1`. W linii 20 następuje porównanie otrzymanego wyniku z wartościami zerowymi. Wynik tego porównywania, dla każdego z elementów wektora, zostaje zapisany do rejestru `MM0` w następujący sposób: `0FFh` dla spełnionego warunku (składowe piksela mniejsze od progu binaryzacji) oraz `00h` dla warunku fałszywego (składowe piksela większe od progu binaryzacji). Należy w tym mo-

mencie zwrócić uwagę, że zanegowane wartości rejestru MM0 będą stanowiły nowe składowe RGB piksela (00h – kolor czarny, 0FFh – kolor biały). Zawartość rejestru MM0 zostaje zapisana do tablicy PIKSEL (linia 21), po czym w pętli znajdującej się w liniach 27–33 kolejne pozycje tablicy, po uprzednim zanegowaniu (linia 29), są kopiowane w miejsce starych składowych RGB piksela (linie 30–32). Efekt działania programu został przedstawiony na rys. 4.11.

Rys. 4.11. Efekt działania programu MMX4.ASM

Program MMX5.ASM przeprowadza analogiczne działania do programu MMX4.ASM, z tym że wykonuje binaryzację z górnym progiem. Jedyna różnica między programami występuje w linii 29 makropolecenia BINARYZACJA – w przypadku programu MMX5.ASM linia ta została ujęta w komentarz.

4.4. JEDNOSTKA SSE I OBLICZENIA ZMIENNOPOZYCYJNE

Podrozdział poświęcono zasadom programowania rozszerzenia SSE, na przykładzie programów realizujących obliczenia zmiennopozycyjne. Operacje na liczbach IEEE 754 SINGLE są wykorzystywane do utworzenia obrazu w skali szarości, obliczenia całki metodą prostokątów oraz wygenerowania fraktala Mandelbrota.

4.4.1. Konwersja obrazu kolorowego do skali szarości

Plik BMP o 24-bitowej głębi kolorów jest w stanie reprezentować 256 odcieni szarości. Każdy piksel w obrazie charakteryzuje się tym, że posiada identyczne wartości wszystkich swoich składowych RGB. Tworzenie pliku BMP zawierającego

grafikę w odcieniach szarości na podstawie grafiki kolorowej wymaga uśrednienia wartości składowych RGB piksela źródłowego:

$$R'(x,y) = \frac{R(x,y) + G(x,y) + B(x,y)}{3}$$
$$G'(x,y) = \frac{R(x,y) + G(x,y) + B(x,y)}{3}$$
$$B'(x,y) = \frac{R(x,y) + G(x,y) + B(x,y)}{3}$$

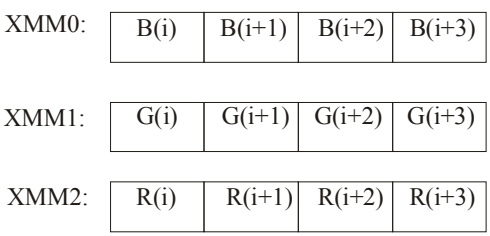
gdzie:
R(x,y), G(x,y), B(x,y) – oryginalne wartości składowych RGB piksela,
R'(x,y), G'(x,y), B'(x,y) – nowe wartości składowych RGB piksela.

Program `SSE11.ASM` wykonuje konwersję obrazu kolorowego do skali szarości przy zastosowaniu rozszerzenia SSE. Kolejno wczytywane porcje obrazu oryginalnego poddawane są obróbce poprzez wywołanie makropolecenia `KONWERTUJ`, którego kod źródłowy jest następujący.

```
1: %MACRO KONWERTUJ 1 ;Parametr: adres bufora
2:   PUSH AX
3:   MOV CX, 86
4:   MOV SI, %1
5:   MOV DI, PIKSEL
6:   MOV AX, DS
7:   MOV ES, AX
8:   CLD
9:   %%TWORZENIE_SZAROSCI:
10:     PUSH CX
11:     PUSH SI
12:     MOV DI, PIKSEL
13:     CALL P
14:     PADDD XMM0, XMM1
15:     PADDD XMM0, XMM2
16:     CVTDQ2PS XMM0, XMM0
17:     MOVUPS XMM1, [DZIELNIK]
18:     DIVPS XMM0, XMM1
19:     CVTPS2DQ XMM1, XMM0
20:     MOVDQU [PIKSEL], XMM1
21:     POP SI
22:     MOV DI, SI
23:     PUSH SI
24:     MOV SI, PIKSEL
```

```
25:     MOV CX, 4
26:     %%PETLA4:
27:       LODSD
28:       STOSB
29:       STOSB
30:       STOSB
31:     LOOP %%PETLA4
32:     POP SI
33:     ADD SI, 12
34:     POP CX
35:   LOOP %%TWORZENIE_SZAROSCI
36:   POP AX
37: %ENDMACRO
```

Makropolecenie wykorzystuje jeden parametr w postaci adresu bufora. W pętli znajdującej się w liniach 9–35 następuje zamiana kolorów pikseli na skalę szarości. Procedura `P`, której wywołanie znajduje się w 13 linii, zapisuje do rejestru `XMM0` składowe niebieskie, do rejestru `XMM1` składowe zielone, natomiast do rejestru `XMM2` składowe czerwone czterech kolejnych pikseli znajdujących się w buforze. Należy przy tym podkreślić, że poszczególne składowe są konwertowane do postaci 32-bitowej – po powrocie z procedury rejestry `XMM0–XMM2` posiadają wartości przedstawione na rys. 4.12.



Rys. 4.12. Wartość rejestrów `XMM0–XMM2` po powrocie z procedury P

W kolejnym kroku następuje zsumowanie zawartości rejestrów (linie 14–15), po czym w linii 16 uzyskany wynik jest konwertowany do wektora liczb zmiennopozycyjnych pojedynczej precyzji. W linii 17 do rejestru `XMM1` zostaje załadowana tablica `DZIELNIK`, która zostanie wykorzystana jako dzielnik przy wyznaczaniu średniej. Deklaracja tablicy `DZIELNIK` w programie ma następującą postać:

```
DZIELNIK      TIMES 4 DD 3.0
```

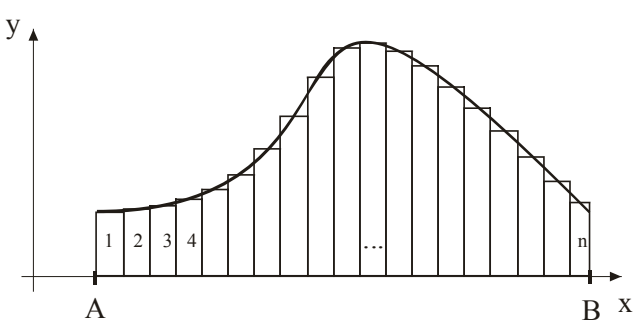
Wyznaczenie średniej wartości składowych RGB dla 4 pikseli wykonywane jest w linii 18, po czym następuje konwersja wyniku do wektora 32-bitowych liczb całkowitych. W pętli, znajdującej się w liniach 26–31, następuje zamiana oryginalnych

wartości składowych RGB pikseli znajdujących się w buforze na obliczone średnie. Efekt działania programu `SSE11.ASM` został przedstawiony na rys. 4.13.

Rys. 4.13. Efekt działania programu `SSE11.ASM`

4.4.2. Wyznaczenie całki metodą prostokątów

Jednostki wektorowe procesorów x86, oprócz zastosowań typowo multimedialnych, mogą również zostać wykorzystane do realizacji obliczeń numerycznych. Jako prosty przykład, w rozdziale zademonstrowano wykorzystanie rozszerzenia SSE do obliczenia wartości całki oznaczonej funkcji, przy zastosowaniu metody prostokątów. Metoda prostokątów bazuje na przybliżaniu wartości pola pod funkcją, poprzez zsumowanie pól prostokątów o wysokości równej wartości funkcji (rys. 4.14). Przedział całkowania jest dzielony na N odcinków o jednakowej długości, które będą stanowiły podstawy prostokątów. Wysokość prostokątów będzie natomiast odpowiadała wartości funkcji całkowanej w punkcie leżącym na środku podstawy prostokąta.



Rys. 4.14. Całkowanie metodą prostokątów

Długość podstawy prostokąta jest następująca:

$$Podst = \frac{B - A}{N},$$

natomiast wysokość prostokąta jest określona zależnością:

$$h_i = f(i * Podst - \frac{1}{2} Podst)$$

Przybliżona wartość całki oznaczonej jest następująca:

$$\int_A^B f(x) \approx Podst * \sum_{i=1}^{N} f(i * Podst - \frac{Podst}{2})$$

Program `SSE12.ASM` wyznacza całkę oznaczoną funkcji x^3-x^2+15 w przedziale <0, 2>. Została w tym celu przygotowana procedura `LICZ_CALKE`.

```
1: LICZ_CALKE:
2:   MOVUPS XMM0, [POCZATKOWY_STAN_REJESTRU]
3:   MOVUPS XMM7, [ZERO]
4:   .PETLA_LICZENIA_CALKI:
5:     MOVUPS XMM1, [PODSTAWA]
6:     MOVUPS XMM2, [PIETNASCIE]
7:     MOVUPS XMM3, [KONIEC_PRZEDZIALU]
;X^3
8:     MOVUPS XMM5, XMM0
9:     MULPS XMM5, XMM0
10:     MULPS XMM5, XMM0
;X^2
11:     MOVUPS XMM6, XMM0
12:     MULPS XMM6, XMM0
;X^3-X^2
13:     SUBPS XMM5, XMM6
;x^3-X^2+15
14:     ADDPS XMM5, XMM2
;Wartość funkcji * długość przedziału
15:     MULPS XMM5, XMM1
;Zapamiętanie wyniku w zmiennej
16:     MOVUPS [POCZATKOWY_STAN_REJESTRU], XMM5
;Przywrócenie
17:     MOVUPS XMM1, [POCZATKOWY_STAN_REJESTRU]
18:     MOVUPS XMM2, [POCZATKOWY_STAN_REJESTRU+4]
19:     MOVUPS XMM3, [POCZATKOWY_STAN_REJESTRU+8]
20:     MOVUPS XMM4, [POCZATKOWY_STAN_REJESTRU+12]
```

```
;Suma otrzymanych pól
21:     ADDPS XMM1, XMM2
22:     ADDPS XMM1, XMM3
23:     ADDPS XMM1, XMM4
;Dodanie kolejnego wyniku
24:     ADDPS XMM7, XMM1
;Przygotowanie danych do następnego kroku
25:     MOVUPS XMM1, [KROK]
26:     ADDPS XMM0, XMM1
;Sprawdzenie czy już nie należy kończyć
27:     MOVUPS XMM1, [KONIEC_PRZEDZIALU]
28:     CMPLTPS XMM1, XMM0
29:     MOVUPS [POCZATKOWY_STAN_REJESTRU], XMM1
30:     SPRAWDZ_CZY_ZERA
31:   JC .KONIEC_LICZNIENIA_CALKI
32:   JMP .PETLA_LICZENIA_CALKI
33:   .KONIEC_LICZNIENIA_CALKI:
34:   MOVUPS [POCZATKOWY_STAN_REJESTRU], XMM7
35:   EMMS
36: RET
```

W każdym przebiegu pętli, znajdującej się w liniach 4–32, zostają wyznaczane 4 kolejne pola prostokątów, natomiast wynik ostateczny zostaje zapisany do rejestru `XMM7`. Rejestr `XMM0` przechowuje wartości współrzędnych x, odpowiadające punktom leżącym na środku podstawy poszczególnych prostokątów (dla tych współrzędnych będzie wyznaczana wartość funkcji). Cztery początkowe wartości x znajdują się w tablicy `POCZATKOWY_STAN_REJESTRU`. Deklaracja tablicy `POCZATKOWY_STAN_REJESTRU` jest następująca:

```
POCZATKOWY_STAN_REJESTRU DD 0.0005, 0.0015, 0.0025, 0.0035,
0.0, 0.0, 0.0
```

Rejestr `XMM1` w linii 5 ładowany jest wartością tablicy `PODSTAWA`, która określa długość podstawy prostokątów. Deklaracja tablicy `PODSTAWA` w kodzie programu jest następująca:

```
PODSTAWA      TIMES 4 DD 0.001
```

Dla współrzędnych x znajdujących się na środku podstaw czterech kolejnych prostokątów, w liniach 8–15 są wyznaczane wartości funkcji x^3-x^2+15 (w liniach 8–10 jest wyznaczana wartość x^3, w liniach 11 – 12 - x^2, w linii 13 - x^3-x^2, natomiast w linii 14 zostaje dodana wartość 15.). Pole prostokątów jest wyznaczane w linii 15, po czym uzyskany wynik zostaje zachowywany w tablicy `POCZATKOWY_STAN_REJESTRU`.

W kolejnym kroku jest wyznaczana suma pól 4 prostokątów. W tym celu (linie 17 – 20) do rejestrów `XMM1–XMM4` zostają zapisane wartości przedstawione na rys. 4.15.

XMM1:	POLE A	POLE B	POLE C	POLE D
XMM2:	POLE B	POLE C	POLE D	0
XMM3:	POLE C	POLE D	0	0
XMM4:	POLE D	0	0	0

Rys. 4.15. Wartości zapisywane do rejestrów `XMM1–XMM4`

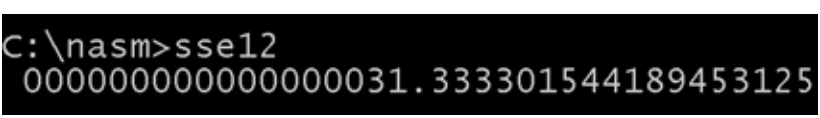
Zsumowanie rejestrów `XMM1–XMM4` (linie 21–23) spowoduje, że najstarsze podwójne słowo rejestru `XMM1` będzie zawierało sumę pól 4 prostokątów. Następnie wynik ten jest dodawany do rejestru XMM7 (rejestr ten przechowuje wyznaczoną dotychczas sumę).

W linii 26 następuje przygotowanie wartości rejestru `XMM0` do kolejnego przebiegu pętli. W tym celu, do rejestru XMM0 zostaje dodana zawartość tablicy `KROK`, której deklaracja jest następująca:

```
KROK    TIMES 4 DD 0.004
```

Operacja ta spowoduje, że nowe wartości rejestru `XMM0` będą wyznaczały środki podstaw kolejnych czterech prostokątów. Przed przejściem do następnego przebiegu pętli zostaje sprawdzony warunek, czy nie został osiągnięty koniec przedziału całkowania (linie 27–31). W przypadku przekroczenia przedziału całkowania następuje zakończenie pętli i zapisanie wyniku zawartego w rejestrze XMM7 do zmiennej `POCZATKOWY_STAN_REJESTRU`.

Po powrocie z procedury obliczającej wartość całki oznaczonej, na ekranie zostaje wyświetlony wynik przy użyciu makropolecenia t vmfpw| ab` (makropolecenie opisano w rozdziale 3.2). Efekt działania programu `SSE12.ASM` został zamieszczony na rys. 4.16.



Rys. 4.16. Wartość całki oznaczonej funkcji $f(x) = x^3-x^2+15$ obliczona przez program `SSE12.ASM`

4.4.3. Fraktal Mandelbrota

„Pod postacią zbioru Mandelbrota przyroda (a może matematyka?) daje nam wizualny odpowiednik tego, co w muzyce można by nazwać „tematem przewodnim i jego wariacjami": wszędzie powtarzają się te same kształty, ale za każdym razem powtórzenie jest trochę inne."

Benoit Mandelbrot

Pojęcie fraktala nie posiada ścisłej definicji matematycznej. Mianem tym określa się obiekty o nietrywialnej strukturze, które cechują się samopodobieństwem (każdy ich fragment przypomina wyglądem całość fraktala). Fraktal Mandelbrota jest najbardziej znanym obiektem tego typu. Niektórzy matematycy uważają, że jest to najbardziej skomplikowana forma, jaką kiedykolwiek zobaczył człowiek. W 1979 roku fraktal został odkryty przez francuskiego matematyka (polskiego pochodzenia) Benoita Mandelbrota, który wygenerował go na ekranie monitora. Zbiór Mandelbrota tworzą liczby zespolone `c`, dla których następujący ciąg rekurencyjny nie dąży do nieskończoności:

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

gdzie: c, z_{n+1}, z_n – liczby zespolone.

Przy programowym generowaniu fraktala Mandelbrota należy wykonać znaczną liczbę iteracji (im więcej, tym większa dokładność) do wyznaczania kolejnych wyrazów z_n. Punkt c uważa się za należący do zbioru Mandelbrota w przypadku, gdy dla ostatniej iteracji spełniona jest nierówność: $|z_n|<2$ (warunek zbieżności ciągu rekurencyjnego).

Po zapisaniu zmiennych zespolonych z_n oraz c w postaci sumy składowych rzeczywistej i urojonej, powyższa zależność rekurencyjna przybiera następującą postać:

$$a_{n+1} + b_{n+1}i = a_n^{\,2} - b_n^{\,2} + a_p + \left(2a_n b_n + b_p\right) i$$

gdzie: $z_{n+1} = a_{n+1} + b_{n+1}i$
$z_n^{\,2} = a_n^{\,2} - b_n^{\,2} + 2a_n b_n i$,
$c = a_p + b_p i$,
i – jednostka urojona.

Dana liczba $c = a_p + b_p i$ należy do zbioru Mandelbrota wtedy, gdy dla ostatniej iteracji zostanie spełniona nierówność:

$$\sqrt{a_{n+1}^{\,2} + b_{n+1}^{\,2}} < 2,$$

gdzie: $a_{n+1} = a_n^{\,2} - b_n^{\,2} + a_p$
$b_{n+1} = 2a_n b_n + b_p$

Program `SSE13.ASM` generuje plik BMP z fraktalem Mandelbrota. W tym celu w pierwszej kolejności zostaje utworzony plik `SSE13.BMP` oraz zostaje określana wartość pól jego nagłówka (pierwsze 54 bajty). Następnie, w pętli wywoływane jest makropolecenie `MANDELBROT`, a rezultaty jego działania

zapisywane są do pliku (każde wywołanie makropolecenia tworzy jedną linię obrazu).

Makropolecenie `MANDELBROT` wykorzystuje 5 parametrów w postaci:
- Adres bufora (`BUFOR`),
- Numer aktualnie generowanej linii obrazu (`LINIA_NR`),
- Rozdzielczość pozioma obrazu (`3072`),
- Rozdzielczość pionowa obrazu (`2048`),
- Liczba iteracji przy wyznaczaniu z_n (`25`).

Wewnątrz makropolecenia następuje wyznaczenie wartości kolejnego elementu ciągu z_n dla punktów znajdujących się w przedziale <-2,2 ; 0,8> na osi odciętych, oraz <-1 ; 1> na osi rzędnych. Ponieważ plik BMP zawiera kolejne piksele ekranu o określonej rozdzielczości, należy wyznaczyć współczynniki skali dla osi odciętych oraz osi rzędnych poprzez podzielenie długości przedziałów przez rozdzielczość poziomą (3072) i pionową (2048). Współczynniki skali w postaci dwóch wektorów zostały zapisane do rejestrów `XMM0` oraz `XMM1` według schematu przedstawionego na rys. 4.17.

XMM0:	SKALA X	SKALA X	SKALA X	SKALA X
XMM1:	SKALA Y	SKALA Y	SKALA Y	SKALA Y

Rys. 4.17. Współczynniki skalowania współrzędnych punktu ekranowego

W rejestrze `XMM2` są zapisane przeskalowane współrzędne *x*, natomiast w rejestrze `XMM3` - przeskalowane współrzędne *y* dla czterech kolejnych punktów obrazu(rys. 4.18).

XMM2:	X(i)	X(i+1)	X(i+2)	X(i+3)
XMM3:	Y(i)	Y(i+1)	Y(i+2)	Y(i+3)

Rys. 4.18. Współrzędne ***x*** i ***y*** punktów $\boldsymbol{z_n}$

Wyznaczanie wyrazu z_n dla tych punktów wykonywane jest w następującej pętli.

```
1: MOV CX, %5
2: %%OBLICZENIE:
3:   CMP CX, %5
4:   JNE %%KOLEJNE_WYKONANIE
5:   MOVUPS XMM4, XMM2
6:   MOVUPS XMM6, XMM3
```

```
7: %%KOLEJNE_WYKONANIE:
;A1= A^2-B^2+X
8:   MOVUPS XMM5, XMM4
9:   MULPS XMM5, XMM5
10:   MOVUPS XMM7, XMM6
11:   MULPS XMM7, XMM7
12:   SUBPS XMM5, XMM7
13:   ADDPS XMM5, XMM2
;B1=2*A*B+Y
14:   MOVUPS XMM7, [DWA]
15:   MULPS XMM7, XMM4
16:   MULPS XMM7, XMM6
17:   ADDPS XMM7, XMM3
;Nowe wartości A i B
18:   MOVUPS XMM4, XMM5
19:   MOVUPS XMM6, XMM7
20: LOOP %%OBLICZENIE
```

W linii 1 ma miejsce załadowanie licznika pętli (rejestru `CX`) parametrem makropolecenia określającym liczbę iteracji przy wyznaczaniu z_n. Podczas pierwszego przebiegu pętli (warunek sprawdzany w linii 3), do rejestrów `XMM4` oraz `XMM6` zostaje zapisana wartość przeskalowanych współrzędnych *x* oraz *y* czterech kolejnych punktów. Następnie w liniach 8–13 obliczane są wartości współczynników a_{n+1}, natomiast w liniach 14–17 wartości współczynników b_{n+1} dla tych punktów. Przed przejściem do kolejnego przebiegu pętli, wartości a_{n+1} zostają zachowane do rejestru `XMM4`, natomiast wartości b_{n+1} do rejestru `XMM6` – w kolejnej iteracji będą one stanowiły współczynniki a_n oraz b_n.

Po wykonaniu ostatniej iteracji zostają obliczone moduły uzyskanych z_n oraz sprawdzony warunek przynależności do zbioru Mandelbrota:

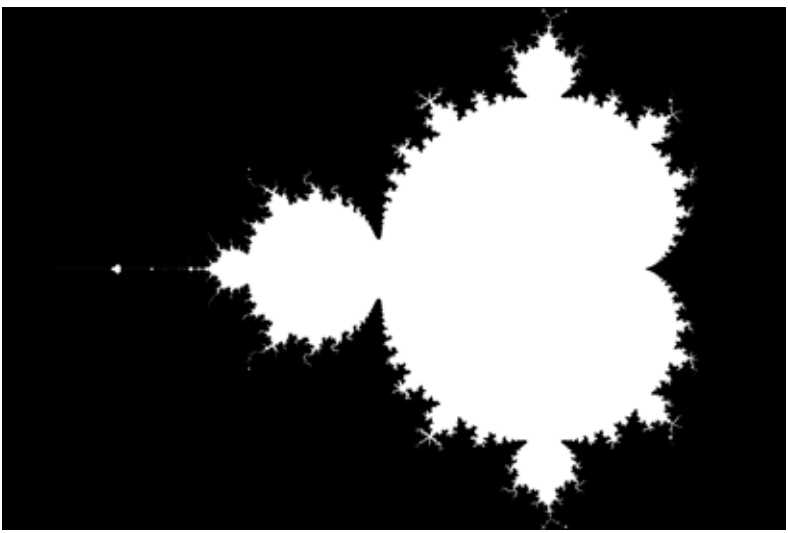
```
;Wyznaczenie R=PIERWIASTEK(A^2+B^2)
1: MULPS XMM4, XMM4
2: MULPS XMM6, XMM6
3: ADDPS XMM4, XMM6
4: SQRTPS XMM4, XMM4
5: MOVUPS XMM5, [PD]
6: CMPLTPS XMM5, XMM4
```

W linii nr 1 wyznaczane są wartości a^2_{n+1} dla czterech kolejnych punktów, w linii 2 wartości b^2_{n+1}, a w linii 3 następuje ich dodanie. Natomiast w linii 4 są obliczone pierwiastki kwadratowe z uzyskanej wcześniej sumy. Określenie przynależności danego punktu do zbioru Mandelbrota odbywa się w linii 6, poprzez porównanie rejestru `XMM4` ze zmienną `PD` zawierającą cztery wartości 2.0:

```
PD    TIMES 4 DD 2.0
```

W przypadku, gdy dany punkt należy do zbioru Mandelbrota, pod odpowiedni adres bufora zostaje zapisana wartość `0FFFFFFFFh` (3 składowe RGB o wartości 255). Gdy punkt nie należy do zbioru Mandelbrota, odpowiedni piksel zostaje wygaszony.

Bitmapa wygenerowana przez program `SSE13.ASM` jest przedstawiona na rys. 4.19.



Rys. 4.19. Fraktal Mandelbrota wygenerowany przez program `SSE13.ASM`

4.5. ROZSZERZENIE SSE2 I OPERACJE NA ZMIENNYCH TYPU DOUBLE

Rozdział poświęcono zasadom programowania rozszerzenia SSE2, na przykładzie programów realizujących obliczenia zmiennopozycyjne na dwuelementowych wektorach liczb typu DOUBLE. Pierwsze dwa programy – obliczanie całki metodą prostokątów oraz rysowanie fraktala Mandelbrota są zmodyfikowaną wersją programów prezentowanych w podrozdziale 4.4. Wykorzystanie liczb zmiennoprzecinkowych podwójnej precyzji zapewni jednak większą dokładność wyników. Dodatkowo w punkcie 4.5.3 zademonstrowano program generujący fraktal Julii.

4.5.1. Wyznaczenie całki metodą prostokątów

Algorytm wyznaczania wartości całki oznaczonej przy wykorzystaniu metody prostokątów został zaprezentowany w punkcie 4.4.2. Przy pomocy instrukcji rozszerzenia SSE możliwe było równoczesne wyznaczenie pól czterech kolejnych

prostokątów. Należy jednak zwrócić uwagę, że przy niektórych zastosowaniach 32-bitowa precyzja dla liczb zmiennoprzecinkowych może być niewystarczająca. Program `SSE21.ASM` przeprowadza analogiczne obliczenia, wykorzystując dwuelementowe wektory liczb typu DOUBLE. Zastosowanie tego formatu pozwoliło na zwiększenie precyzji obliczeń – tablica definiująca szerokość podstawy prostokątów zawiera tym razem wartości 0,000001:

```
PODSTAWA TIMES 2 DQ 0.000001,
```

co w porównaniu z programem `SSE12.ASM` stanowi wartość 1000 krotnie mniejszą. Należy zaznaczyć, że operacje na 64-bitowym formacie zmiennoprzecinkowym są dokładniejsze – a zatem błąd powstały przy zaokrąglaniu kolejnych wyników będzie znacznie mniejszy.

Procedura wyznaczająca wartość całki oznaczonej funkcji $f(x) = x^3-x^2+15$ w przedziale <0,2> jest następująca.

```
LICZ_CALKE:
  MOVUPD XMM0, [POCZATKOWY_STAN_REJESTRU]
  MOVUPD XMM7, [ZERO]
  .PETLA_LICZENIA_CALKI:
    MOVUPD XMM1, [PODSTAWA]
    MOVUPD XMM2, [PIETNASCIE]
    MOVUPD XMM3, [KONIEC_PRZEDZIALU]
;X^3
    MOVUPD XMM5, XMM0
    MULPD XMM5, XMM0
     MULPD XMM5, XMM0
;X^2
     MOVUPD XMM6, XMM0
     MULPD XMM6, XMM0
;X^3-X^2
     SUBPD XMM5, XMM6
;x^3-X^2+15
     ADDPD XMM5, XMM2
;Wartość funkcji * długość przedziału
     MULPD XMM5, XMM1
;Zapamiętanie wyniku w zmiennej
     MOVUPD [POCZATKOWY_STAN_REJESTRU], XMM5
;Przywrócenie
     MOVUPD XMM1, [POCZATKOWY_STAN_REJESTRU]
     MOVUPD XMM2, [POCZATKOWY_STAN_REJESTRU+8]
;Suma otrzymanych pól
     ADDPD XMM1, XMM2
;Dodanie kolejnego wyniku
```

```
20:     ADDPD XMM7, XMM1
;Przygotowanie danych do następnej iteracji
21:     MOVUPD XMM1, [KROK]
22:     ADDPD XMM0, XMM1
;Sprawdzenie czy już nie należy kończyć
23:     MOVUPD XMM1, [KONIEC_PRZEDZIALU]
24:     CMPLTPD XMM1, XMM0
25:     MOVUPD [POCZATKOWY_STAN_REJESTRU], XMM1
26:     SPRAWDZ_CZY_ZERA
27:   JC .KONIEC_LICZENIA_CALKI
28:   JMP .PETLA_LICZENIA_CALKI
29:   .KONIEC_LICZENIA_CALKI:
30: MOVUPD [POCZATKOWY_STAN_REJESTRU], XMM7
31: EMMS
32: RET
```

Treść tej procedury jest analogiczna do procedury zawartej w programie `SSE12.ASM`. Zmianie uległy wyłącznie nazwy instrukcji – instrukcje operujące na wektorach danych typu SINGLE (kończące się literami `PS`) zastąpiono instrukcjami operującymi na dwuelementowych wektorach liczb typu DOUBLE (kończące się literami `PD`).

Efekt działania programu `SSE21.ASM` przedstawiono na rys. 4.20.

Porównując wynik zamieszczony na rys.4.20 z wynikiem obliczonym przez

```
C:\nasm>sse21
 00000000000000031.33333333355803063
```

Rys. 4.20. Wartość całki oznaczonej funkcji $f(x) = x^3-x^2+15$ w przedziale <0, 2>, wyznaczona przez program `SSE21.ASM`. Wartość dokładna wynosi $31^1/_3$

program `SSE12.ASM` (rys. 4.16) należy stwierdzić, że dzięki zastosowaniu 64-bitowej reprezentacji liczb zmiennopozycyjnych dokładność obliczeń znaczenie się zwiększyła (4 miejsca po przecinku w przypadku programu `SSE12.ASM`, a 10 miejsc po przecinku w programie `SSE21.ASM`).

4.5.2. Fraktal Mandelbrota

Program `SSE22.ASM` generuje fraktal Mandelbrota przy wykorzystaniu operacji na wektorach liczb typu DOUBLE. Zastosowanie pojemniejszego formatu liczb zmiennoprzecinkowych pozwoliło na zwiększenie szczegółowości generowanej bitmapy – w programie zwiększono dwukrotnie szerokość i wysokość bitmapy (z 3072x2048 na 6144x4096 pikseli). Przy wyznaczaniu wyrazu z_n liczba iteracji została zwiększona do 35 (z 25 w programie `SSE13.ASM`).

Makropolecenie MANDELBROT, w porównaniu ze swoim odpowiednikiem z programu SSE13.ASM, wykorzystuje instrukcje operujące na dwuelementowych wektorach liczb typu DOUBLE. Pętla wyznaczająca wartości wyrazów z_n dla dwóch punktów jednocześnie ma następującą postać.

```
%%OBLICZENIE:
  CMP CX, %5
  JNE %%KOLEJNE_WYKONANIE
  MOVUPD XMM4, XMM2
  MOVUPD XMM6, XMM3
  %%KOLEJNE_WYKONANIE:
;A1= A^2-B^2+X
  MOVUPD XMM5, XMM4
  MULPD XMM5, XMM5
  MOVUPD XMM7, XMM6
   MULPD XMM7, XMM7
   SUBPD XMM5, XMM7
   ADDPD XMM5, XMM2
;B=2*A*B+Y
   MOVUPD XMM7, [DWA]
   MULPD XMM7, XMM4
   MULPD XMM7, XMM6
   ADDPD XMM7, XMM3
;Nowe wartości A i B
   MOVUPD XMM4, XMM5
   MOVUPD XMM6, XMM7
LOOP %%OBLICZENIE
```

Fragment programu określający przynależność danych punktów do zbioru Mandelbrota jest w tym przypadku następujący.

```
;OBLICZANIE R=PIERWIASTEK(A^2+B^2)
MULPD XMM4, XMM4
MULPD XMM6, XMM6
ADDPD XMM4, XMM6
SQRTPD XMM4, XMM4
MOVUPD XMM5, [PD]
CMPLTPD XMM5, XMM4
```

4.5.3. Fraktal Julii

Zbiór Julii zawdzięcza swoją nazwę francuskiemu matematykowi Gaston Julii, który prowadził badania w zakresie metod iteracyjnych dla funkcji zespolonych. Prace Julii stały się inspiracją dla Mandelbrota, który kontynuował

badania w tym zakresie, przyczyniając się między innymi do odkrycia zbioru Mandelbrota.

Fraktal Julii tworzą punkty z_n, dla których następujący ciąg rekurencyjny nie dąży do nieskończoności:

$$z_0 = a_0 + b_0 i$$
$$z_{n+1} = z_n^2 + c$$

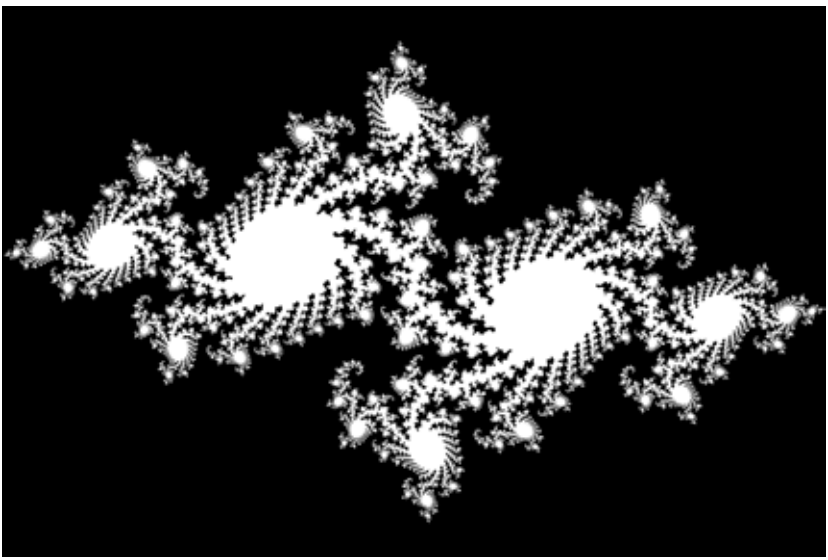
gdzie: c, z_{n+1}, z_n – liczby zespolone.

Powyższa zależność jest analogiczna jak dla fraktala Mandelbrota z tą różnicą, że tym razem współrzędne punktu nie są przypisywane do współczynnika c, lecz do z_0. Wartości współczynnika c mogą być dowolne, a ich modyfikacja przyczyni się do uzyskania innych kształtów fraktala. Przy generowaniu fraktala Julii (podobnie jak to miało miejsce w przypadku fraktala Mandelbrota) przyjmuje się ograniczoną liczbę iteracji do wyznaczania kolejnych wyrazów z_n. Punkt z_0 uważa się za należący do zbioru Julii w przypadku, gdy spełniona jest nierówność: $|z_n|<2$ dla ostatniej iteracji.

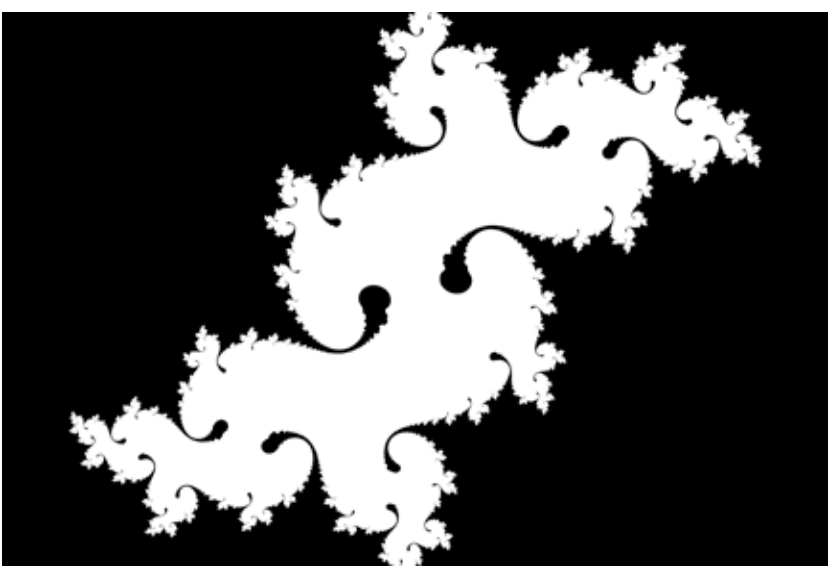
Program `SSE23.ASM` tworzy plik BMP, do którego zapisuje wygenerowany fraktal Julii. Wszystkie obliczenia wykonywane w programie wykorzystują możliwości rozszerzenia SSE2 (każda iteracja wyznacza wartości z_n dwóch kolejnych punktów jednocześnie). Struktura programu `SSE23.ASM` jest analogiczna do programu `SSE22.ASM`, jedyną różnicą jest pętla wyznaczająca kolejne wartości z_n.

```
1: %%OBLICZENIE:
2:    MOVUPD XMM0, [P_OBL]
3:    MOVUPD XMM1, [Q_OBL]
;A1= A^2-B^2+X
4:    MOVUPD XMM4, XMM2
5:    MULPD XMM4, XMM4
6:    MOVUPD XMM5, XMM3
7:    MULPD XMM5, XMM5
8:    SUBPD XMM4, XMM5
9:    ADDPD XMM4, XMM0
;B=2*A*B+Y
10:   MOVUPD XMM5, [DWA]
11:   MULPD XMM5, XMM2
12:   MULPD XMM5, XMM3
13:   ADDPD XMM5, XMM1
;Nowe wartości A i B
14:   MOVUPD XMM2, XMM4
15:   MOVUPD XMM3, XMM5
16: LOOP %%OBLICZENIE
```

Przed rozpoczęciem realizacji pętli, rejestry `XMM2` oraz `XMM3` zawierają współrzędne x i y punktów, dla których zostaną wyznaczone wyrazy z_n. W liniach 2–3 następuje określenie wartości współczynników c (przy generowaniu fraktala



Rys. 4.21. Fraktal Julii wygenerowany przez program `SSE23.ASM` dla parametru c = -0,73+0,19i



Rys. 4.22. Fraktal Julii wygenerowany przez program `SSE23.ASM` dla parametru c = 0,285+0,013i

Julii, *c* posiada wartość stałą). Następnie w liniach 4–8 są wyznaczane wartości współczynników a_{n+1}, natomiast w liniach 10–13 wartości współczynników b_{n+1}. Przed przejściem do kolejnego przebiegu pętli, wartości a_{n+1} zostają zapisane do rejestru XMM2, natomiast wartości b_{n+1} do rejestru XMM3 – w kolejnej iteracji będą one stanowiły współczynniki a_n oraz b_n.

Efekty działania programu SSE23.ASM zostały zamieszczone na rys. 4.21 oraz rys. 4.22.

4.6. ROZSZERZENIE SSE3

Rozdział poświęcono zasadom programowania rozszerzenia SSE3, ze szczególnym uwzględnieniem operacji horyzontalnych, na przykładzie wyznaczania wartości liczb ciągu Fibonacciego.

4.6.1. Wyznaczanie liczb Fibonacciego

Leonardo Fibonacci w 1202 roku w dziele Liber Abaci (Księga rachunków), w której sformułował pytanie o liczebność populacji królików po upływie roku, przy założeniu, że każda para królików co miesiąc rodzi nową parę. Do rozwiązania tego problemu należy wyznaczyć dwunastą liczbę Fibonacciego, która jest określona następującą zależnością rekurencyjną:

$$F_k = \begin{cases} 1, & k = 1,2 \\ F_{k-1} + F_{k-2,} & k \geq 3 \end{cases}$$

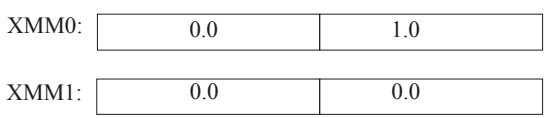
gdzie: F_k – wartość k-tej liczby Fibonacciego,
F_{k-1}, F_{k-2} – wartość liczb Fibonacciego z poprzednich dwóch iteracji,
k – numer iteracji.

Program SSE31.ASM oblicza liczby Fibonacciego o indeksie równym: 86, 50, 40, 30 oraz 5. Operowanie na tak dużych wartościach wymaga użycia liczb zmiennoprzecinkowych podwójnej precyzji. Do obliczenia wartości *k-tej* liczby Fibonacciego, w programie zostało zdefiniowane makropolecenie LICZBA_FIBON, wykorzystujące jeden parametr w postaci indeksu liczby. Treść makropolecenia LICZBA_FIBON jest następująca.

```
1: %MACRO LICZBA_FIBON 1
2:   MOV CX, %1
3:   MOVUPD XMM0, [JED]
4:   MOVUPD XMM1, [ZER]
5:   MOV AL, 0
```

```
6:    JCXZ %%PETLA_END
7:    %%FIBON:
8:      CMP AL, 0
9:      JNE %%DALEJ
10:     HADDPD XMM1, XMM0
11:     JMP %%KONIEC
12:     %%DALEJ:
13:     HADDPD XMM0, XMM1
14:     %%KONIEC:
15:     NOT AL
16:   LOOP %%FIBON
17:   %%PETLA_END:
18:   MAXPD XMM0, XMM1
19:   MOVUPD [WYNIK], XMM0
20:   MOV EAX, [WYNIK]
21:   XCHG EAX, [WYNIK+8]
22:   MOV [WYNIK], EAX
23:   MOV EAX, [WYNIK+4]
24:   XCHG EAX, [WYNIK+12]
25:   MOV [WYNIK+4], EAX
26:   MOVUPD XMM1, [WYNIK]
27:   MAXPD XMM0, XMM1
28:   MOVUPD [WYNIK], XMM0
29: %ENDMACRO
```

W linii 2 następuje zapisanie do licznika pętli (rejestr `CX`) indeksu wyznaczanej liczby Fibonacciego. Następnie zostaje określona początkowa zawartość rejestrów `XMM0` oraz `XMM1` (linie 3–4), zgodnie z rys. 4.23.



Rys. 4.23. Początkowa wartość rejestrów `XMM0`, `XMM1` przed rozpoczęciem realizacji pętli obliczającej ***k***-tą liczbę Fibonacciego

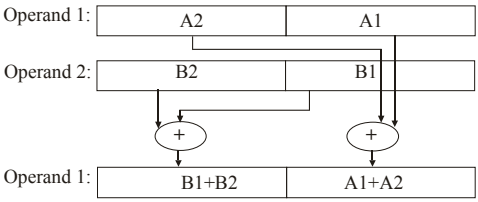
W pętli zawartej w liniach 7–16 jest wyznaczana wartość kolejnych liczb Fibonacciego poprzez naprzemienne wywoływanie instrukcji:

```
HADDPD XMM1, XMM0          (linia 10)
```

oraz

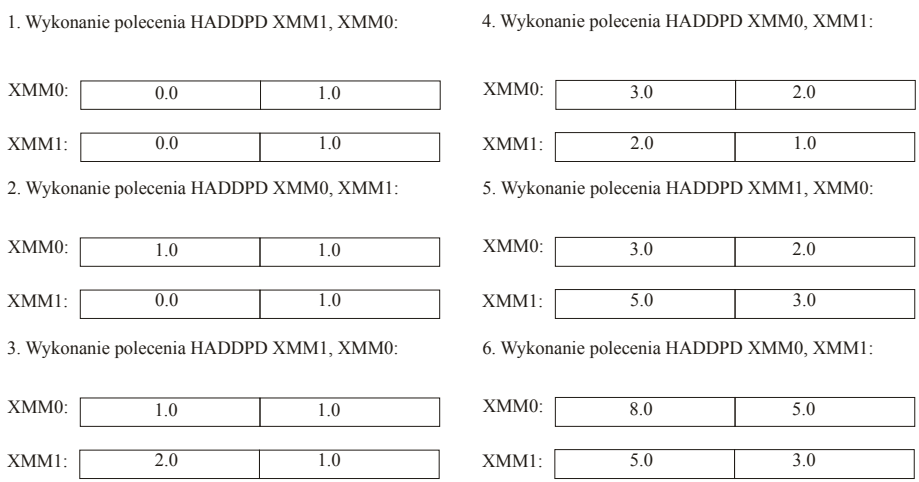
```
HADDPD XMM0, XMM1          (linia 13)
```

Instrukcja `HADDPD` wykonuje operację dodawania horyzontalnego, według schematu przedstawionego na rys. 4.24.



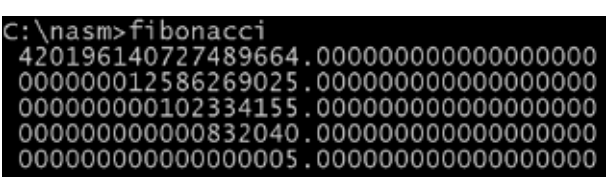
Rys. 4.24. Schemat działania instrukcji HADDPD

Stan rejestrów `XMM0` oraz `XMM1` dla sześciu pierwszych iteracji został przedstawiony na rys. 4.25.



Rys. 4.25. Stan rejestrów XMM0 i XMM1 po wykonywaniu operacji dodawania horyzontalnego

Jak można zauważyć z rys. 4.25, *k*-ta liczba Fibonacciego jest maksymalną wartością ze wszystkich elementów zawartych w rejestrach `XMM0` oraz `XMM1`. W związku z tym w liniach 18–28 następuje wyznaczenie największej wartości z obydwu rejestrów, po czym wynik zostaje zachowywany w zmiennej `WYNIK`. Efekt działania programu `SSE31.ASM` został zamieszczony na rys. 4.26.



Rys. 4.26. Wartości liczb Fibonacciego o indeksie 86, 50, 40, 30 oraz 5 wyznaczone przez program `SSE31.ASM`

4.7. REALIZACJA WYBRANYCH PRZEKSZTAŁCEŃ GRAFICZNYCH PRZY WYKORZYSTANIU JEDNOSTEK WEKTOROWYCH

Rozdział poświęcono realizacji prostych przekształceń graficznych figur geometrycznych wyświetlanych na ekranie monitora. Wszystkie opisane przekształcenia zostały zaimplementowane w programie `GRAF.ASM`. Do podstawowych przekształceń graficznych należy zaliczyć następujące operacje:
– Translację, czyli przesunięcie figury o zadaną liczbę punktów (punkt 4.7.2),
– Skalowanie, mające na celu zmianę oryginalnego rozmiaru figury (punkt 4.7.3),
– Obrót względem początku układu współrzędnych (punkt 4.7.4),
– Obrót względem własnego środka (punkt 4.7.5),
– Pochylenie figury (punkt 4.7.6).

4.7.1. Wyświetlanie figury na ekranie monitora

W celu wyświetlenia punktu na ekranie monitora należy wykorzystać funkcje BIOS'u zawarte w przerwaniu `10h`. Wyświetlanie tekstu/grafiki na ekranie monitora może odbywać się w kilku trybach. Tryb można określić dzięki wykorzystaniu funkcji `0h` przerwania `10h`, która wykorzystuje parametr w rejestrze `AL` o formacie przedstawionym w tabeli 4.2.

Tabela 4.2.

Parametr funkcji `0h` przerwania `10h`

7	6	5	4	3	2	1	0
1 – pozostawia obecny stan pamięci ekranu 0 – zeruje pamięć ekranu	Bity określają numer trybu						

W przypadku, gdy bit nr 7 jest w stanie 1, przy wywołaniu funkcji zostaje zachowany obecny stan pamięci ekranu (pamięć ekranu nie jest kasowana), natomiast w przeciwnym przypadku następuje wykasowanie zawartości ekranu (pamięć ekranu jest zerowana). Bity 0–6 określają numer trybu. W programie `GRAF.ASM`, obrazującym przekształcenia graficzne, wykorzystano tryb `12h`, który pozwala na wyświetlanie grafiki o rozdzielczości 640/480 pikseli w 16 kolorach.

Funkcja `0CH` umożliwia wyświetlenie punktu o zadanych współrzędnych i kolorze. Parametrami funkcji są rejestry:
– `BH` – określający numer strony, na której będzie wyświetlony punkt (standardowo wartość `0`),
– `AL` – zawierający kolor wyświetlanego punktu,

– CX – określający kolumnę, w której zostanie wyświetlony punkt (współrzędna x),
– DX – zawierający wiersz, w którym zostanie wyświetlony punkt (współrzędna y).

W programie GRAF.ASM współrzędne punktów wyświetlanej figury są przechowywane w dwóch tablicach:

```
X_KWADRATU TIMES 2500 DW 0
Y_KWADRATU TIMES 2500 DW 0
```

Tablica X_KWADRATU przechowuje współrzędne x kolejnych punktów figury, natomiast tablica Y_KWADRATU przechowuje współrzędne y kolejnych punktów.

Procedura INICJUJ_X_Y została stworzona w celu zapisania do poszczególnych współrzędnych obydwu tablic przykładowych wartości, tak aby uzyskana figura przedstawiała kwadrat (lewy górny punkt kwadratu znajduje się w punkcie x=200, y=300. Długość boku kwadratu wynosi 50 punktów). Do wyświetlania figury przygotowano dwie następujące procedury.

```
1: RYSUJ_PUNKT:
2: PUSHA
3:    MOV AH, 0CH
4:    MOV AL, 15
5:    MOV BH, 0
6:    INT 10H
7: POPA
8: RET
9: RYSUJ_KWADRAT
10: PUSHA
11:    MOV CX, 2500
12:    XOR SI, SI
13:    RYSOWANIE:
14:       PUSH CX
15:       MOV CX, [X_KWADRATU+SI]
16:       MOV DX, [Y_KWADRATU+SI]
17:       INC SI
18:       INC SI
19:       CALL RYSUJ_PUNKT
20:       POP CX
21:    LOOP RYSOWANIE
22: POPA
23: RET
```

Procedura RYSUJ_PUNKT wywołuje funkcję 0Ch przerwania 10h w celu wyświetlenia na ekranie monitora punktu o współrzędnych zawartych w rejestrach CX i DX (parametry procedury). Wyświetlany punkt będzie posiadał kolor biały (linia 4).

Procedura RYSUJ_KWADRAT, w pętli zawartej w liniach 13–21, wyświetla na ekranie monitora (przy użyciu procedury RYSUJ_PUNKT) kolejne punkty figury, której współrzędne przechowują tabele X_KWADRATU oraz Y_KWADRATU.

4.7.2. Przesunięcie figury geometrycznej

Przesunięcie figury jest najprostszą operacją graficzną, polegającą na przeniesieniu wszystkich punktów figury o określoną liczbę punktów na osi X oraz Y (dla grafiki dwuwymiarowej). Operację przesunięcia można zrealizować poprzez następujące przekształcenie:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} X \\ Y \end{bmatrix} + \begin{bmatrix} \delta x \\ \delta y \end{bmatrix}$$

gdzie: X, Y – współrzędne punktu w przestrzeni 2D,
X', Y' – współrzędne punktu po przekształceniu,
δd, δd – przesunięcie wzdłuż osi *Ox* i *Oy*.

Operację przesunięcia można przeprowadzić, wykorzystując jednostki wektorowe dla 8-elementowych wektorów danych 16-bitowych. Tekst procedury realizującej przesunięcie w programie GRAF.ASM jest następujący.

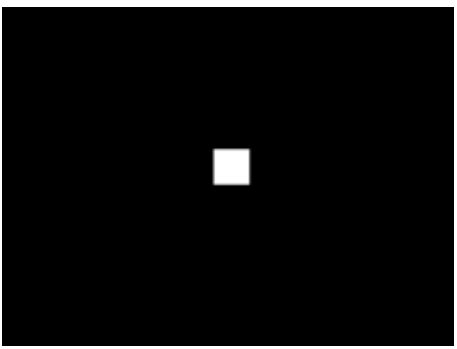
```
1: TRANSLACJA:
2: PUSHA
3:    MOVDQU XMM0, [DELTA]
4:    MOV CX, 313
5:    PETLA_TRANSLACJI:
6:       MOVDQU XMM1, [SI]
7:       PADDW XMM1, XMM0
8:       MOVDQU [SI], XMM1
9:       ADD SI, 16
10:    LOOP PETLA_TRANSLACJI
11: POPA
12: RET
```

W linii 3 następuje załadowanie do rejestru uj j Mwektora przesunięcia, o który będą zmieniane współrzędne *x* lub *y*. W liniach 5–10 zamieszczona jest pętla modyfikująca wartości współrzędnych punktów figury. W linii 6 do rejestru XMM1 zapisywane są współrzędne ośmiu kolejnych punktów z tablicy X_KWADRATU=lub Y_KWADRATU (w zależności od zawartości rejestru SI). Następnie, przy pomocy polecenia PADDW (w linii 7), następuje dodanie wektora przesunięcia, po czym w linii 8 nowe wartości *x* lub *y* zostają zapisane do odpowiedniej tablicy. Przed przejściem do kolejnego przebiegu pętli, wartość rejestru indeksowego zwiększana jest o 16, tak aby wskazywał na 8 kolejnych współrzędnych *x* lub *y*. W tabeli 4.3 przedstawiono efekty działania programu.

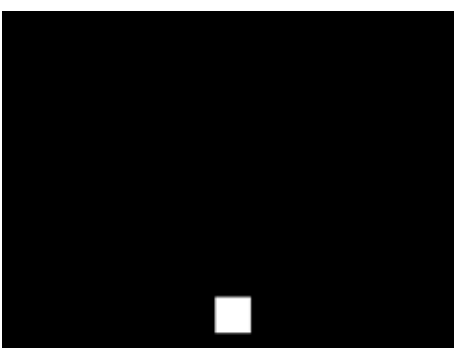
Tabela 4.3.

Efekty translacji prostokąta w programie `GRAF.ASM`

`CALL RYSUJ_KWADRAT`	`MOV SI, X_KWADRATU`
	`CALL TRANSLACJA`
	`CALL RYSUJ_KWADRAT`

`MOV SI, Y_KWADRATU`
`CALL TRANSLACJA`
`CALL RYSUJ_KWADRAT`
	`MOV SI, X_KWADRATU`
	`CALL TRANSLACJA`
	`MOV SI, Y_KWADRATU`
	`CALL TRANSLACJA`
	`CALL RYSUJ_KWADRAT`

4.7.3. Zmiana rozmiaru figury dwuwymiarowej

Zmiana rozmiaru figury płaskiej może zostać wykonana następująco:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} S_X & 0 \\ 0 & S_Y \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \end{bmatrix}$$

gdzie S_X, S_Y – współczynniki powiększenia poziomego i pionowego.

Mnożenie współrzędnych *x* i *y* poszczególnych punktów figury można w łatwy sposób zaimplementować z użyciem jednostek wektorowych. W programie `GRAF.ASM` operację skalowania figury wykonuje procedura `SKALOWANIE`, której treść jest następująca.

```
1: ph^i l t ^kf bW
2: mr pe^
3: ==j l s=` uI=ORMMLQ
4: ==ul o=pf I=pf
5: ==mbqi ^| ph^i l t ^kf ^W
6: ====` ^i i=l mbo^` gb| ph^i l t ^kf ^
7: ====^aa=pf I=U
8: ==i l l m=mbqi ^| ph^i l t ^kf ^
9: ml m^
10: obq
```

W pętli o 625 przebiegach (dzięki użyciu jednostek wektorowych czterokrotnie ograniczono liczbę operacji) wywoływana jest procedura `OPERACJE_SKALOWANIA`. Tekst procedury został zamieszczony poniżej:

```
1: l mbo^` gb| ph^i l t ^kf ^W
2: ==t v` wvp` | w^j f^kb
3: ==movwdl qr g| al | hl kt bopgf| u
4: ==j l s anr=uj j RI=xw^j f^k^z
5: ==` s qanOmp=uj j MI=uj j R
6: ==t v` wvp` | w^j f^kb
7: ==movwdl qr g| al | hl kt bopgf| v
8: ==j l s anr=uj j RI=xw^j f^k^z
9: ==` s qanOmp=uj j NI=uj j R
10: ==j l s r mp=uj j OI=xph^i ^z
11: ==j l s r mp=uj j QI=uj j M=          ;XMM4=X
12: ==j l s r mp=uj j SI=xpq^oq| uz
13: ==pr _mp=uj j QI=uj j S
14: ==j r i mp=uj j QI=uj j O=           ;X*SKALA
15: ==^aamp=uj j QI=uj j S
16: ==` s qmpOan=uj j TI=uj j Q
17: ==j l s anr=xw^j f^k^zI=uj j T
18: ==movwt ol ` | u
19: ==j l s r mp=uj j RI=uj j N=         ;XMM5=Y
20: ==j l s r mp=uj j SI=xpq^oq| vz
21: ==pr _mp=uj j RI=uj j S
22: ==j r i mp=uj j RI=uj j O=           ;Y*SKALA
23: ==^aamp=uj j RI=uj j S
24: ==` s qmpOan=uj j TI=uj j R
```

```
25:   MOVDQU [ZAMIANA], XMM7
26:   PRZYWROC_Y
27: RET
```

W celu zwiększenia czytelności programu, opracowano następujące makropolecenia.

`WYCZYSC_ZAMIANE` – makropolecenie zeruje zmienną `ZAMIANA`, w której 16-bitowe współrzędne punktów figury będą zamieniane na wartości 32-bitowe,

`PRZYGOTUJ_DO_KONWERSJI_X` – makropolecenie rozszerza cztery kolejne elementy tablicy `X_KWADRATU` (zawiera 16-bitowe współrzędne *x* punktów) do postaci 32-bitowej (przechowywane w tablicy `ZAMIANA`),

`PRZYGOTUJ_DO_KONWERSJI_Y` – makropolecenie poszerza cztery kolejne elementy tablicy `Y_KWADRATU` (zawiera 16-bitowe współrzędne *y* punktów) do postaci 32-bitowej (przechowywane w tablicy `ZAMIANA`),

`PRZYWROC_X` – makropolecenie zamienia cztery kolejne 32-bitowe wartości z tablicy `ZAMIANA` na cztery wartości 16-bitowe, przechowywane na kolejnych pozycjach tablicy `X_KWADRATU`,

`PRZYWROC_Y` – makropolecenie zamienia cztery kolejne 32-bitowe wartości z tablicy `ZAMIANA` na cztery wartości 16-bitowe przechowywane na kolejnych pozycjach tablicy `Y_KWADRATU`.

Procedura `OPERACJE_SKALOWANIA` w liniach 2–5 konwertuje cztery kolejne współrzędne *x* punktów figury na liczby typu SINGLE, przechowywane w rejestrze `XMM0`. W liniach 6–9 w ten sam sposób konwertowane są cztery kolejne współrzędne *y* punktów figury, przechowywane w rejestrze `XMM1`. W linii nr 10, do rejestru `XMM2` jest ładowany wektor skali, zawierający cztery liczby typu SINGLE, przez które będą mnożone współrzędne *x* i *y* punktów figury.

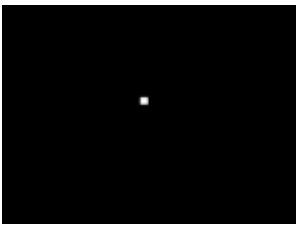
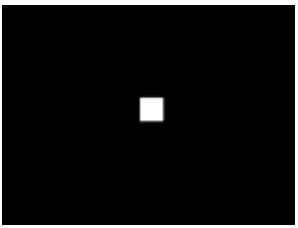
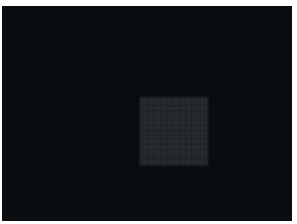
Przed wykonaniem mnożenia współrzędnych *x* punktów figury przez wektor skali, należy przeprowadzić operację translacji współrzędnych *x* na początek układu współrzędnych (tak, by pierwsza współrzędna *x* figury pokrywała się z punktem 0 na osi *Ox*). Operacja ta wykonywana jest w linii 13. W linii 14 współrzędne *x* (po procesie translacji) są mnożone przez współczynnik zmiany skali, po czym w linii 15 następuje translacja odwrotna współrzędnych *x* (dodanie wartości odjętej w linii 13, aby uzyskać właściwe umiejscowienie figury na ekranie). Następnie w liniach 16–18 nowe wartości współrzędnych *x* są konwertowane na liczby całkowite i zapisywane we właściwe miejsce tablicy `X_KWADRATU`.

W liniach 19–26 analogiczne operacje wykonywane są na współrzędnych *y* czterech kolejnych punktów należących do figury. W tabeli 4.4 przedstawiono efekty działania programu w zależności od stanu zmiennej `SKALA`.

Trzecia kolumna tabeli pokazuje, że dla współczynnika skali większego od 1 (powiększenie figury) powoduje rozproszenie pikseli figury na ekranie po jej powiększeniu. W tym przypadku pomiędzy nowo wyznaczonymi punktami znajdują się również punkty tła.

Tabela 4.4.

Skalowanie prostokąta w programie GRAF.ASM

SKALA TIMES 4 DD 0.3	SKALA TIMES 4 DD 1.0	SKALA TIMES 4 DD 3.0
		

4.7.4. Obrót figury względem początku układu współrzędnych

Obrót figury geometrycznej w dwuwymiarowym układzie współrzędnych jest możliwy do wykonania przy użyciu następującej zależności:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \end{bmatrix}$$

gdzie: α – kąt obrotu.

W programie GRAF.ASM procedura OBROT wywołuje cyklicznie (w pętli PETLA_OBROTU) procedurę OPERACJE_OBROTU w celu obliczenia nowych współrzędnych dla poszczególnych punktów figury obracanej względem początku układu współrzędnych (lewy, górny punkt ekranu). Tekst procedury OPERACJE_OBROTU jest następujący.

```
1: OPERACJE_OBROTU:
2:    WYCZYSC_ZAMIANE
3:    PRZYGOTUJ_DO_KONWERSJI_X
4:    MOVDQU XMM5, [ZAMIANA]
5:    CVTDQ2PS XMM0, XMM5
6:    WYCZYSC_ZAMIANE
7:    PRZYGOTUJ_DO_KONWERSJI_Y
8:    MOVDQU XMM5, [ZAMIANA]
9:    CVTDQ2PS XMM1, XMM5
10:    MOVUPS XMM2, [SIN]
11:    MOVUPS XMM3, [COS]
12:    MOVUPS XMM4, XMM0   ;XMM4=X
13:    MULPS XMM4, XMM3=   ;X*COS(ALFA)
14:    MOVUPS XMM5, XMM1=  ;XMM5=Y
15:    MULPS XMM5, XMM2=   ;Y*SIN(ALFA)
```

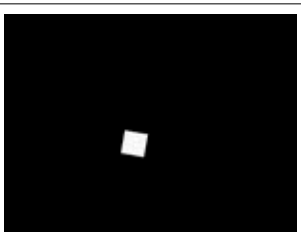
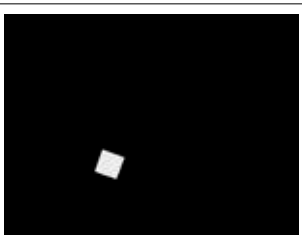
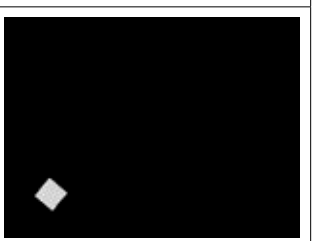
```
16:   SUBPS XMM4, XMM5    ;X*COS(ALFA)-Y*SIN(ALFA)
17:   CVTPS2DQ XMM7, XMM4
18:   MOVDQU [ZAMIANA], XMM7
19:   PRZYWROC_X
20:   MOVUPS XMM4, XMM0=  ;XMM4=X
21:   MULPS XMM4, XMM2=   ;X*SIN(ALFA)
22:   MOVUPS XMM5, XMM1=  ;XMM5=Y
23:   MULPS XMM5, XMM3=   ;Y*COS(ALFA)
24:   ADDPS XMM4, XMM5=   ;X*SIN(ALFA)+Y*COS(ALFA)
25:   CVTPS2DQ XMM7, XMM4
26:   MOVDQU [ZAMIANA], XMM7
27:   PRZYWROC_Y
28: RET
```

Po zamianie kolejnych czterech współrzędnych x i y punktów figury na liczby typu SINGLE (linie 2–10), rejestr `XMM2` zostaje załadowany czteroelementowym wektorem wcześniej obliczonych wartości sinusa kąta obrotu, natomiast rejestr `XMM3` - czteroelementowym wektorem cosinusa kąta obrotu. W liniach 12–16 zostają wyznaczone współrzędne x czterech kolejnych punktów, które po zamianie na wartości całkowite (linie 17–19) zostają zapisane w tablicy `X_KWADRATU`. Analogiczne operacje są wykonywane dla współrzędnych y (linie 20–27). Efekty działania programu dla różnych wartości kąta obrotu zostały przedstawione w tabeli 4.5.

Tabela 4.5.

Operacje obrotu figury względem środka układu współrzędnych w programie `GRAF.ASM`

`;10 stopni`
`COS TIMES 4 DD`
`0.984807753`
`SIN TIMES 4 DD`
`0.173648178`
	`;20 stopni`
	`COS TIMES 4 DD`
	`0.939692621`
	`SIN TIMES 4 DD`
	`0.342020143`
		`;40 stopni`
		`COS TIMES 4 DD`
		`0.766044443`
		`SIN TIMES 4 DD`
		`0.64278761`

4.7.5. Obrót figury względem własnego środka

Obrót figury geometrycznej względem własnego środka jest wykonywany poprzez:
– wykonanie w pierwszej kolejności przesunięcia figury do początku układu współrzędnych tak, aby środek figury znajdował się w punkcie (0,0).
– realizacja przekształceń opisanych w rozdziale 4.7.4.
– przesunięcie figury do pierwotnego położenia (dodanie do współrzędnych x i y punktów figury wartości odjętych w pierwszym kroku).

Podobnie jak w przypadku realizacji obrotów względem początku układu współrzędnych, tak i tym razem możliwe jest wykorzystanie jednostek wektorowych procesora. Procedura `ROTACJA_SRODKOWA` (zawarta w programie `GRAF.ASM`) wywołuje cyklicznie procedurę `ROTACJA` w celu obliczenia nowych współrzędnych przekształcanej figury. Tekst procedury `ROTACJA` jest następujący.

```
1: ROTACJA:
2:    WYCZYSC_ZAMIANE
3:    PRZYGOTUJ_DO_KONWERSJI_X
4:    MOVDQU XMM5, [ZAMIANA]
5:    CVTDQ2PS XMM0, XMM5
6:    WYCZYSC_ZAMIANE
7:    PRZYGOTUJ_DO_KONWERSJI_Y
8:    MOVDQU XMM5, [ZAMIANA]
9:    CVTDQ2PS XMM1, XMM5
10:    MOVUPS XMM7, [START_X]
11:    SUBPS XMM0, XMM7
12:    MOVUPS XMM7, [POLOWA_BOKU]
13:    SUBPS XMM0, XMM7
14:    MOVUPS XMM7, [START_Y]
15:    SUBPS XMM1, XMM7
16:    MOVUPS XMM7, [POLOWA_BOKU]
17:    SUBPS XMM1, XMM7
18:    MOVUPS XMM2, [SIN]
19:    MOVUPS XMM3, [COS]
20:    MOVUPS XMM4, XMM0=      ;XMM4=X
21:    MULPS XMM4, XMM3=       ;X*COS(ALFA)
22:    MOVUPS XMM5, XMM1=      ;XMM5=Y
23:    MULPS XMM5, XMM2=       ;Y*SIN(ALFA)
24:    SUBPS XMM4, XMM5=       ;X*COS(ALFA)-Y*SIN(ALFA)
25:    MOVUPS XMM7, [START_X]
26:    ADDPS XMM4, XMM7
27:    MOVUPS XMM7, [POLOWA_BOKU]
28:    ADDPS XMM4, XMM7
```

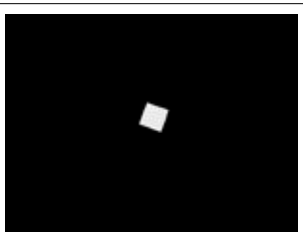
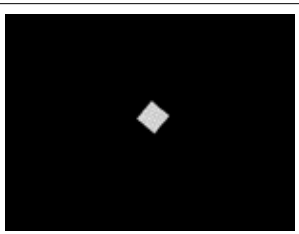
```
29:   CVTPS2DQ XMM7, XMM4
30:   MOVDQU [ZAMIANA], XMM7
31:   PRZYWROC_X
32:   MOVUPS XMM4, XMM0=      ;XMM4=X
33:   MULPS XMM4, XMM2=       ;X*SIN(ALFA)
34:   MOVUPS XMM5, XMM1=      ;XMM5=Y
35:   MULPS XMM5, XMM3=       ;Y*COS(ALFA)
36:   ADDPS XMM4, XMM5=       ;X*SIN(ALFA)+Y*COS(ALFA)
37:   MOVUPS XMM7, [START_Y]
38:   ADDPS XMM4, XMM7
39:   MOVUPS XMM7, [POLOWA_BOKU]
40:   ADDPS XMM4, XMM7
41:   CVTPS2DQ XMM7, XMM4
42:   MOVDQU [ZAMIANA], XMM7
43:   PRZYWROC_Y
44: RET
```

Efekty działania procedury w zależności od wartości wektorów SIN i COS zostały zamieszczone w tabeli 4.6.

Tabela 4.6.

Rotacja figury względem własnego środka w programie `GRAF.ASM`

;20 stopni
`COS TIMES 4 DD 0.939692621`
`SIN TIMES 4 DD 0.342020143`
	;40 stopni
	`COS TIMES 4 DD 0.766044443`
	`SIN TIMES 4 DD 0.64278761`

4.7.6. Przekształcenie pochylające

Przekształcenia pochylające pozwalają osiągnąć efekt pochylenia figury w kierunku jednej z osi współrzędnych. Pochylenie figury można osiągnąć poprzez zastosowanie następującej zależności:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \end{bmatrix}$$

gdzie: a – współczynnik pochylenia wzdłuż osi *Ox*,
b – współczynnik pochylenia wzdłuż osi *Oy*.

W programie GRAF.ASM przekształcenia pochylające są wykonywane przy użyciu procedury POCHYLENIE, wywołującej cyklicznie procedurę OPERACJE_POCHYLENIA. Tekst procedury OPERACJE_POCHYLENIA jest następujący.

```
1: OPERACJE_POCHYLENIA:
2:   WYCZYSC_ZAMIANE
3:   PRZYGOTUJ_DO_KONWERSJI_X
4:   MOVDQU XMM5, [ZAMIANA]
5:   CVTDQ2PS XMM0, XMM5
6:   WYCZYSC_ZAMIANE
7:   PRZYGOTUJ_DO_KONWERSJI_Y
8:   MOVDQU XMM5, [ZAMIANA]
9:   CVTDQ2PS XMM1, XMM5
10:   MOVUPS XMM2, [WSPOLCZYNNIK_POCHYLENIA_X]
11:   MOVUPS XMM3, [WSPOLCZYNNIK_POCHYLENIA_Y]
12:   MOVUPS XMM4, XMM0=    ;XMM4=X
13:   MOVUPS XMM5, XMM1=    ;XMM5=Y
14:   MULPS XMM5, XMM2=     ;Y*WSLPLCZYNNIK_POCHYLENIA_X
15:   ADDPS XMM4, XMM5=     ;X+Y*WSPOLCZYNNIK_POCHYLENIA_X
16:   CVTPS2DQ XMM7, XMM4
17:   MOVDQU [ZAMIANA], XMM7
18:   PRZYWROC_X
19:   MOVUPS XMM4, XMM0=    ;XMM4=X
20:   MULPS XMM4, XMM3=     ;X*WSPOLCZYNNIK_POCHYLENIA_Y
21:   MOVUPS XMM5, XMM1=    ;XMM5=Y
22:   ADDPS XMM4, XMM5=     ;X*WSPOLCZYNNIK_POCHYLENIA_Y+Y
23:   CVTPS2DQ XMM7, XMM4
24:   MOVDQU [ZAMIANA], XMM7
25:   PRZYWROC_Y
26: RET
```

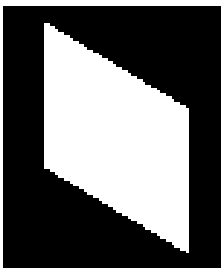
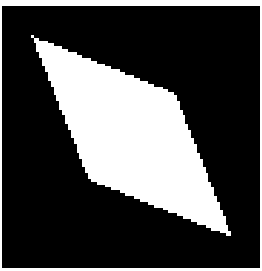
W liniach 10–11 procedury, do rejestrów XMM2 oraz XMM3 są ładowane wartości współczynników pochylenia odpowiednio wzdłuż osi *Ox* i *Oy*. Przekształcenie pochylające figurę wzdłuż osi *Ox* jest realizowane w liniach 14–15, natomiast przekształcenie pochylające figurę wzdłuż osi *Oy* – w liniach 20–22.

W tabeli 4.7 przedstawiono efekt działania programu w zależności od wartości współczynników pochylenia.

Jak wynika z tabeli 4.7, w wyniku pierwszego przekształcenia krawędzie dolne i górne figury zostały równoległe względem osi *Ox*. W przekształceniu drugim krawędzie boczne pozostały równoległe względem osi *Oy*. Ostatnie przekształcenie spowodowało zmianę kątów względem osi *Ox* i *Oy* wszystkich krawędzi.

Tabela 4.7.

Pochylenie figury w zależności od współczynnika pochylenia w programie `GRAF.ASM`

`WSPOLCZYNNIK_POCHYLENIA_X TIMES 4 DD 0.8 WSPOLCZYNNIK_POCHYLENIA_Y TIMES 4 DD 0.0`	`WSPOLCZYNNIK_POCHYLENIA_X TIMES 4 DD 0.0 WSPOLCZYNNIK_POCHYLENIA_Y TIMES 4 DD 0.6`	`WSPOLCZYNNIK_POCHYLENIA_X TIMES 4 DD 0.4 WSPOLCZYNNIK_POCHYLENIA_Y TIMES 4 DD 0.4`
		

4.8. ANALIZA WYDAJNOŚCI JEDNOSTEK SIMD

W podrozdziale poddano analizie stopień przyspieszenia realizacji programów z rozdziału 4 w wyniku zastosowania jednostek wektorowych, w porównaniu do klasycznych obliczeń skalarnych z zastosowaniem podstawowych jednostek obliczeniowych procesora. Maksymalne teoretyczne przyspieszenie obliczeń wynika z długości wektora. Np. dla obliczeń z zastosowaniem czteroelementowego wektora liczb typu SINGLE, maksymalne przyspieszenie obliczeń w porównaniu z jednostką zmiennopozycyjną (FPU- Float Point Unit) wynosi 4. Rzeczywiste przyspieszenie dla konkretnego programu może być znacznie mniejsze, gdyż znaczna część programu może być realizowana skalarnie (prawo Amdahla). Porównanie szybkości realizacji programów zrealizowano w oparciu o pomiar czasu.

4.8.1. Metoda pomiaru czasu

Pomiar czasu realizacji poszczególnych programów został zrealizowany dzięki wykorzystaniu czasomierza systemowego. Układ 8253 z cyklem około 55 ms (18,2 razy na sekundę) generuje przerwanie sprzętowe, którego procedura obsługi inkrementuje 32-bitową zmienną, znajdującą się pod adresem `40h:6Ch`. W celu wyznaczenia różnicy czasów pomiędzy uruchomieniem programu a jego zakończeniem przygotowano makropolecenia: `START_POMIARU, WYZNACZ_CZAS` oraz `WYPISZ_CZAS`.

Makropolecenie START_POMIARU zapisuje do zmiennej POMIAR_1 aktualny stan licznika systemowego spod adresu 40h:6Ch. Treść makropolecenia jest następująca.

```
1: %MACRO START_POMIARU 0
2:   PUSH AX
3:   MOV AX, ES
4:   PUSH AX
5:   PUSH DX
6:   MOV AX, 40H
7:   MOV ES, AX
8:   MOV EDX, [ES:6CH]
9:   MOV [POMIAR_1], EDX
10:   POP DX
11:   POP AX
12:   MOV ES, AX
13:   POP AX
14: %ENDMACRO
```

Makropolecenie WYZNACZ_CZAS odczytuje aktualną wartość licznika przerwań układu 8253, po czym odejmuje ją od wartości zmiennej POMIAR_1 – w ten sposób zostaje obliczony czas pomiędzy wywołaniem makropoleceń START_POMIARU oraz WYZNACZ_CZAS (jednostką jest interwał równy ok. 55ms).

```
1: %MACRO WYZNACZ_CZAS 0
2:   PUSH AX
3:   MOV AX, ES
4:   PUSH AX
5:   PUSH DX
6:   MOV AX, 40H
7:   MOV ES, AX
8:   MOV EDX, [ES:6CH]
9:   MOV EAX, [POMIAR_1]
10:   SUB EDX, EAX
11:   MOV [CZAS_WYKONANIA], EDX
12:   POP DX
13:   POP AX
14:   MOV ES, AX
15:   POP AX
16: %ENDMACRO
```

Zadaniem makropolecenia WYPISZ_CZAS jest wyprowadzenie na ekran (w formacie dziesiętnym) czasu, jaki upłynął pomiędzy wywołaniem makropoleceń START_POMIARU oraz WYZNACZ_CZAS. W tym celu wartość zmiennej CZAS_WYKONANIA zostaje pomnożona jest przez 55 – uzyskana wartość określa

czas w milisekundach. Następnie przy użycia makropolecenia `PRINT_INT` (makropolecenie opisane w podrozdziale 4.2) uzyskany wynik jest wyświetlony na ekranie monitora.

4.8.2. Prezentacja wyników

Rezultaty prezentowane w niniejszym rozdziale umożliwiają porównanie szybkości wektorowych metod realizacji algorytmów z metodami skalarnymi. W tym celu zostały opracowane programy realizujące przedstawione zagadnienia metodami skalarnymi:

- `bMMX1.ASM` – program rozjaśniający mapę bitową (bez kontroli przekroczenia zakresu liczb 8-bitowych), odpowiednik programu `MMX1.ASM`.
- `bMMX2.ASM` – program rozjaśniający mapę bitową (nie dopuszczający do przekroczenia zakresu liczb 8-bitowych), odpowiednik programu `MMX2.ASM`.
- `bMMX3.ASM` – program tworzący negatyw mapy bitowej, odpowiednik programu `MMX3.ASM`.
- `bMMX4.ASM` – program generujący obraz czarno-biały z kolorowej mapy bitowej. Odpowiednik programu `MMX4.ASM`.
- `bSSE11.ASM` – program tworzący obraz w skali szarości na podstawie kolorowej mapy bitowej. Odpowiednik programu `SSE11.ASM`.
- `bSSE12.ASM` – program wyznaczający wartość całki oznaczonej metodą prostokątów wykorzystujący jednostkę zmiennoprzecinkową FPU do wykonywania operacji na liczbach typu SINGLE. Odpowiednik programu `SSE12.ASM`.
- `bSSE13.ASM` – program generujący fraktal Mandelbrota, wykorzystujący jednostkę zmiennoprzecinkową FPU do wykonywania operacji na liczbach typu SINGLE. Odpowiednik programu `SSE13.ASM`.
- `bSSE21.ASM` – program wyznaczający wartość całki oznaczonej metodą prostokątów, wykorzystujący jednostkę zmiennoprzecinkową FPU do wykonywania operacji na liczbach typu DOUBLE. Odpowiednik programu `SSE21.ASM`.
- `bSSE22.ASM` – program generujący fraktal Mandelbrota, wykorzystujący jednostkę zmiennoprzecinkową FPU do wykonywania operacji na liczbach typu DOUBLE. Odpowiednik programu `SSE22.ASM`.
- `bSSE23.ASM` – program generujący fraktal Julii, wykorzystujący jednostkę zmiennoprzecinkową FPU do wykonywania operacji na liczbach typu DOUBLE. Odpowiednik programu `SSE23.ASM`.

Podczas wyznaczania czasu działania poszczególnych programów poczyniono pewne założenia:

- Ze względu na zbyt krótki czas realizacji programów `MMX1.ASM`–`MMX4.ASM` w porównaniu z częstotliwością przerwań układu 8253 założono, że makropolecenie wykonujące obliczenia na mapie bitowej w każdym przebiegu pętli modyfikującej plik, zostanie wywołane 5000 razy. Zmiany te zostały wprowa-

dzone zarówno w programach wykorzystujących obliczenia na jednostkach wektorowych (`MMX1.ASM–MMX4.ASM`) jak i w programach wykorzystujących obliczenia skalarne (`bMMX1.ASM–bMMX4.ASM`).
– W programach `SSE1.ASM` oraz `bSSE1.ASM`, makropolecenie wyznaczające skalę szarości w każdym przebiegu pętli modyfikującej plik zostaje wywołane 1000 razy (ze względu na zbyt krótki czas wykonania się programów oryginalnych).
– W programach `SSE12.ASM`, `bSSE12.ASM` oraz `SSE21.ASM`, `bSSE21.ASM` przedział całkowania został zmieniony z < 0 ; 2 > do < 0 ; 500> w celu wydłużenia obliczeń.
– W programach generujących fraktale Mandelbrota i Julii wykonywanych jest 15 iteracji do wyznaczenia ostatecznej wartości wyrazu z_n.

W tabeli 4.8 porównano czasy realizacji programów dla obliczeń wektorowych i skalarnych.

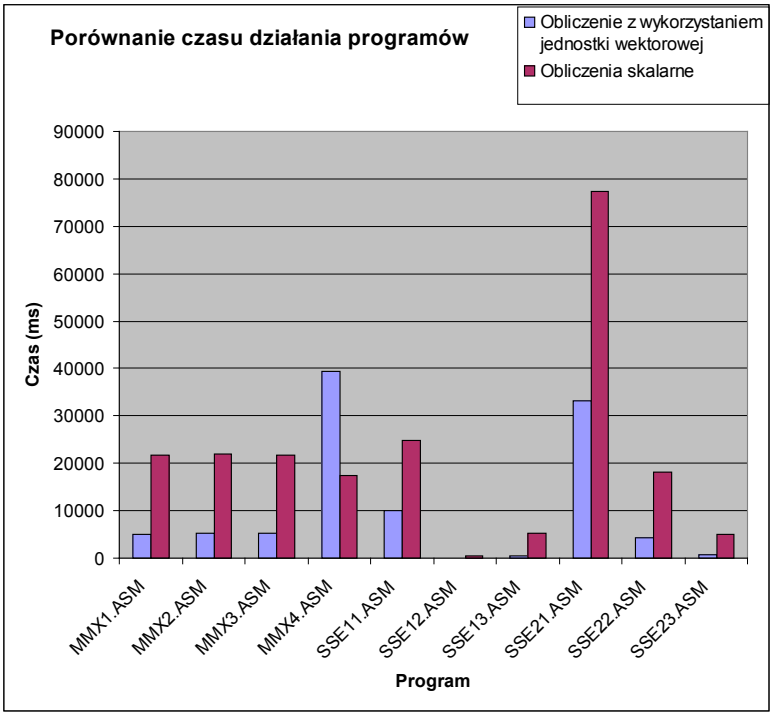
Tabela 4.8.

Porównanie czasu realizacji programów dla obliczeń wektorowych i skalarnych

Program wykorzystujący jednostkę wektorową:	Czas w milisekundach	Program wykonujący analogiczny algorytm na wartościach skalarnych:	Czas w milisekundach
MMX1.ASM	5050	bMMX1.ASM	21835
MMX2.ASM	5270	bMMX2.ASM	21890
MMX3.ASM	5160	bMMX3.ASM	21725
MMX4.ASM	39500	bMMX4.ASM	17500
SSE11.ASM	9790	bSSE11.ASM	24750
SSE12.ASM	55	bSSE12.ASM	495
SSE13.ASM	550	bSSE13.ASM	5335
SSE21.ASM	33220	bSSE21.ASM	77440
SSE22.ASM	4290	bSSE22.ASM	18095
SSE23.ASM	770	bSSE23.ASM	5060

Na rys. 4.27 przedstawiono czasy realizacji programów wektorowych i skalarnych w postaci graficznej.

Z przedstawionego na rys. 4.27 wykresu wynika, że programy wykorzystujące jednostki wektorowe były realizowane szybciej (z wyjątkiem programu `MMX4.ASM`) od programów wykonujących obliczenia skalarne. Uzyskany przyrost szybkości wynosił od 230% w przypadku programu wyznaczającego całkę oznaczoną metodą prostokątów (program `SSE21.ASM`) do 970% w przypadku programu generującego fraktal Mandelbrota (program `SSE13.ASM`). Tak duży przyrost



Rys. 4.27. Wykres przedstawiający porównanie czasu działania programów wykorzystujących jednostki wektorowe do analogicznych programów realizujących obliczenia skalarne

szybkości w przypadku programów generujących fraktale nie jest wyłącznie wynikiem operacji wektorowych. Dodatkowym czynnikiem wpływającym na wzrost szybkości programów wykorzystujących rejestry `XMM` było zredukowanie odwołań do pamięci operacyjnej. Programy wykonujące obliczenia zmiennoprzecinkowe z wykorzystaniem jednostki FPU, ze względu na specyfikę programowania jednostki zmiennoprzecinkowej (stosowanie odwrotnej notacji polskiej), muszą zapisywać swoje wyniki pośrednie do zmiennych (wyrazy a_{n-1} i b_{n-1} w pętli obliczającej kolejne wartości wyrazu z_n). W przypadku programów wykorzystujących rozszerzenia SIMD zapamiętywanie wyrazów pośrednich odbywało się w rejestrach `XMM`, dzięki możliwości swobodnego dostępu do każdego rejestru.

Słaby wynik programu wykonującego binaryzację obrazu (program `MMX4.ASM`) w porównaniu z jego skalarnym odpowiednikiem, jest spowodowany koniecznością wykonania dużo większej liczby odwołań do pamięci operacyjnej. W każdym przebiegu pętli wyznaczającej docelowe wartości ośmiu kolejnych pikseli, w programie `MMX4.ASM` wywoływana jest procedura `P` przygotowująca dane według schematu przedstawionego na rys. 4.9. (co wymaga wielokrotnych operacji odczytu i zapisu do pamięci). Dodatkowo efekt działania instrukcji `PCMPEQB`

(sprawdzającej czy suma składowych RGB danego piksela jest większa od progu binaryzacji) jest zapisywany do zmiennej `PIKSEL`, skąd następnie porcjami jest odczytywany i po przetworzeniu zapisywany do właściwej pozycji bufora.

4.9. PODSUMOWANIE

W rozdziale zawarto podstawowe informacje dotyczące programowania jednostek SIMD procesorów x86. Na wielu przykładach przedstawiono zasady programowania jednostek wektorowych, poczynając od MMX, przez SSE, SSE2 do SSE3. Pełne wersje programów są przedstawione na stronie Wydziału Elektrotechniki, Automatyki i Informatyki Politechniki Opolskiej: http://we.po.opole.pl. Czytelnika zainteresowanego bardziej szczegółowym poznaniem przedstawionych zagadnień odsyłamy do następujących pozycji: [3, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25].

5. ZBIÓR INSTRUKCJI WEKTOROWYCH

5.1. ZESTAW INSTRUKCJI TRANSFERU DANYCH

W tabeli przyjęto następujące oznaczenia:
A – pierwszy operand wywoływanej instrukcji,
B – drugi operand wywoływanej instrukcji,
← – przepływ danych,
[x | z] – alternatywne możliwości użycia x albo z,
[x..z] – bity od x do z,
mmx – jeden z rejestrów mmx,
xmm – jeden z rejestrów xmm,
m(x) – zmienna x-bitowa,
r – rejestr ogólnego przeznaczenia,
if – pod warunkiem.

Tabela 5.1.

Zestaw instrukcji transferu danych dla rozszerzeń MMX, SSE, SSE2, SSE3

Składnia instrukcji	Operacja
MOVD [mmx | xmm], [r | m(32)] MOVD [r | m(32)], [mmx | xmm]	A[0..31] ← B A← B[0..31]
MOVQ [mmx | xmm], [r | m(64)] MOVQ [r | m(64)],[mmx | xmm]	A [0..63]← B A← B[0..63]
MOVAPS xmm, [xmm | m(128)] MOVAPS m(128), xmm	A ← B, gdzie m(128) wyrównany do granicy 16 bajtów
MOVHLPS xmm, xmm	A[0..63] ← B[64..127]
MOVLHPS xmm, xmm	A[64..127] ← B[0..63]
MOVHPS xmm, m(64) MOVHPS m(64), xmm	A[64..127] ← B A ← B[64..127]
MOVLPS xmm, m(64) MOVLPS m(64), xmm	A[0..63] ← B A ← B[0..63]
MOVMSKPS r, xmm	A[0..3] ← B[31, 63, 95, 127] (rysunek 2.12)
MOVSS xmm, [xmm | m(32)] MOVSS m(32), xmm	A[0..31] ← B A ← B[0..31]
MOVUPS xmm, [xmm | m(128)] MOVUPS m(128), xmm	A ← B

MASKMOVQ mmx, mmx	[DS:EDI] ← A[0..7] if B[7]==1; [DS:EDI+1] ← A[8..15] if B[15]==1; … [DS:EDI+7] ← A[56..63] if B[63]==1; (rysunek 2.13)
MOVNTPS m(128), xmm	A ← B, bez użycia cache
MOVNTQ m(64), mmx	A ← B, bez użycia cache
MOVSD xmm, [xmm | m(64)] MOVSD m(64), xmm	A[0..63] ← B A ← B[0..63]
MOVAPD xmm, [xmm | m(128)] MOVAPD m(128), xmm	A ← B, gdzie m(128) wyrównany do granicy 16 bajtów
MOVUPD xmm, [xmm | m(128)] MOVUPD m(128), xmm	A ← B
MOVLPD xmm, m(64) MOVLPD m(64), xmm	A[0..63] ← B A ← B[0..63]
MOVHPD xmm, m(64) MOVHPD m(64), xmm	A[64..127] ← B A ← B[64..127]
MOVDQ2Q mmx, xmm	A ← B[0..63]
MOVQ2DQ xmm, mmx	A[0..63] ← B
MOVNTPD m(128), xmm	A ← B, bez użycia cache
MOVNTDQ m(128), xmm	A ← B, bez użycia cache
MOVNTI [m(32) | m(64)], r	A ← B, bez użycia cache
MASKMOVDQU xmm, xmm	[DS:EDI] ← A[0..7] if B[7]==1; [DS:EDI+1] ← A[8..15] if B[15]==1; … [DS:EDI+15] ← A[120..127] if B[127]==1;
PMOVMSKB r, mmx PMOVMSKB r, xmm	A[0..7] ← B[7,15,23,31,39,47,55,63] A[0..15]← B[7,15,23,31,39,47,55,63,71,79,87,95,103,111,119,127]
MOVDDUP xmm, [xmm | m(64)]	A[0..63] ← B[0..63]; A[64..127] ← B[0..63];
MOVSHDUP xmm, [xmm | m(128)]	A[0..31] ← B[32..63]; A[32..63] ← B[32..63]; A[64..95] ← B[96..127]; A[96..127] ← B[96..127];
MOVSLDUP xmm, [xmm | m(128)]	A[0..31] ← B[0..31]; A[32..63] ← B[0..31]; A[64..95] ← B[64..95]; A[96..127] ← B[64..95];
LDDQU xmm, m(128)	A ← B
MOVDQA xmm, [m(128) | xmm] MOVDQA m(128), xmm	A ← B, gdzie m(128) wyrównany do granicy 16 bajtów
MOVDQU xmm, [m(128) | xmm] MOVDQU m(128), xmm	A ← B

5.2. ZESTAW INSTRUKCJI ARYTMETYCZNYCH

Tabela 5.2.

Zestaw instrukcji arytmetycznych rozszerzeń MMX, SSE, SSE2 i SSE3

Składnia instrukcji	Operacja
PADDB mmx, [mmx | m(64)] PADDB xmm, [xmm | m(128)]	A←A+B
PADDW mmx, [mmx | m(64)] PADDW xmm, [xmm | m(128)]	A←A+B
PADDD mmx, [mmx | m(64)] PADDD xmm, [xmm | m(128)]	A←A+B
PADDSB mmx, [mmx | m(64)] PADDSB xmm, [xmm | m(128)]	A←A+B
PADDSW mmx, [mmx | m(64)] PADDSW xmm, [xmm | m(128)]	A←A+B
PADDUSB mmx, [mmx | m(64)] PADDUSB xmm, [xmm | m(128)]	A←A+B
PADDUSW mmx, [mmx | m(64)] PADDUSW xmm, [xmm | m(128)]	A←A+B
PSUBB mmx, [mmx | m(64)] PSUBB xmm, [xmm | m(128)]	A←A–B
PSUBW mmx, [mmx | m(64)] PSUBW xmm, [xmm | m(128)]	A←A–B
PSUBD mmx, [mmx | m(64)] PSUBD xmm, [xmm | m(128)]	A←A–B
PSUBSB mmx, [mmx | m(64)] PSUBSB xmm, [xmm | m(128)]	A←A–B
PSUBSW mmx, [mmx | m(64)] PSUBSW xmm, [xmm | m(128)]	A←A–B
PSUBUSB mmx, [mmx | m(64)] PSUBUSB xmm, [xmm | m(128)]	A←A–B
PSUBUSW mmx, [mmx | m(64)] PSUBUSW xmm, [xmm | m(128)]	A←A–B
PMULHW mmx, [mmx | m(64)]	TEMP[0..31]←A[0..15]*B[0..15]; A[0..15]←TEMP[16..31], … TEMP[0..31] ←A[48..63]*B[48..63]; A[48..63]←TEMP[16..31].
PMULHW xmm, [xmm | m(128)]	TEMP[0..31] ←A[0..15]*B[0..15]; A[0..15]←TEMP[16..31], … TEMP[0..31]←A[112..127]*B[112..127]; A[112..127]←TEMP[16..31].
PMULLW mmx, [mmx | m(64)]	TEMP[0..31]←A[0..15]*B[0..15]; A[0..15]←TEMP[0..15], … TEMP[0..31] ←A[48..63]*B[48..63]; A[48..63]←TEMP[0..15].
PMULLW xmm, [xmm | m(128)]	TEMP[0..31] ←A[0..15]*B[0..15]; A[0..15]←TEMP[0..15], … TEMP[0..31]←A[112..127]*B[112..127]; A[112..127]←TEMP[0..15].

PMADDWD mmx, [mmx | m(64)]	A[0..31] ←(A[0..15]*B[0..15]+A[16..31]*B[16..31]); A[32..63] ←(A[32..47]*B[32..47]+A[48..63]*B[48..63]).
PMADDWD xmm, [xmm | m(128)]	A[0..31] ←(A[0..15]*B[0..15]+A[16..31]*B[16..31]); … A[96..127] ←(A[96..111]*B[96..111]+A[112..127]*B[112..127]).
ADDPS xmm, [xmm | m(128)]	A ← A+B
ADDSS xmm, [xmm | m(32)]	A[0..31] ← A[0..31]+B[0..31]
SUBPS xmm, [xmm | m(128)]	A ← A–B
SUBSS xmm, [xmm | m(32)]	A[0..31] ← A[0..31] –B[0..31]
MULPS xmm, [xmm | m(128)]	A ← A*B
MULSS xmm, [xmm | m(32)]	A[0..31] ← A[0..31] *B[0..31]
DIVPS xmm, [xmm | m(128)]	A ← A/B
DIVSS xmm, [xmm | m(32)]	A[0..31] ← A[0..31] /B[0..31]
RCPPS xmm, [xmm | m(128)]	A[0..31] ←(1/B[0..31]); A[32..63] ←(1/B[32..63]); A[64..95] ←(1/B[64..95]); A[96..127] ←(1/B[96..127]).
RCPSS xmm, [xmm | m(32)]	A[0..31] ←(1/B[0..31]).
SQRTPS xmm, [xmm | m(128)]	A[0..31] ← $\sqrt{B[0..31]}$; A[32..63] ← $\sqrt{B[32..63]}$; A[64..95] ← $\sqrt{B[64..95]}$; A[96..127] ←($\sqrt{B[96..127]}$).
SQRTSS xmm, [xmm | m(32)]	A[0..31] ← $\sqrt{B[0..31]}$.
RSQRTPS xmm, [xmm | m(128)]	A[0..31] ←(1/ $\sqrt{B[0..31]}$); A[32..63] ← (1/ $\sqrt{B[32..63]}$); A[64..95] ←(1/ $\sqrt{B[64..95]}$); A[96..127] ←(1/ $\sqrt{B[96..127]}$).
RSQRTSS xmm, [xmm | m(32)]	A[0..31] ←(1/ $\sqrt{B[0..31]}$).
MAXPS xmm, [xmm | m(128)]	A[0..31] ←max(A[0..31],B[0..31]); … A[96..127] ←max(A[96..127], B[96..127]).
MAXSS xmm, [xmm | m(32)]	A[0..31] ←max(A[0..31],B[0..31]).
MINPS xmm, [xmm | m(128)]	A[0..31] ←min(A[0..31],B[0..31]); … A[96..127] ←min(A[96..127], B[96..127]).
MINSS xmm, [xmm | m(32)]	A[0..31] ←min(A[0..31],B[0..31]).
PAVGB mmx, [mmx | m(64)]	A[0..7] ←(A[0..7]+B[0..7]+1)/2; … A[56..63] ←(A[56..63]+B[56..63]+1)/2.
PAVGB xmm, [xmm | m(128)]	A[0..7] ←(A[0..7]+B[0..7]+1)/2; … A[120..127] ←(A[120..127]+B[120..127]+1)/2.
PAVGW mmx, [mmx | m(64)]	A[0..15] ←(A[0..15]+B[0..15]+1)/2; … A[48..63] ←(A[48..63]+B[48..63]+1)/2.
PAVGW xmm, [xmm | m(128)]	A[0..7] ←(A[0..7]+B[0..7]+1)/2; … A[112..127] ←(A[112..127]+B[112..127]+1)/2.

PSADBW mmx, [mmx | m(64)]	TEMP0 ←ABS(A[0..7] –B[0..7]); … TEMP7 ←ABS(A[56..63] –B[56..63]); A[0..15] ←SUM(TEMP0:TEMP7).
PSADBW xmm, [xmm | m(128)]	TEMP0←ABS(A[0..7] –B[0..7]); … TEMP15 ←ABS(A[120..127] –B[120..127]); A[0..15] ←SUM(TEMP0:TEMP7); A[64..79] ←SUM(TEMP8:TEMP15);
PMAXSW mmx, [mmx | m(64)]	A[0..15] ←max(A[0..15],B[0..15]); … A[48..63] ←max(A[48..63], B[48..63]).
PMAXSW xmm, [xmm | m(128)]	A[0..15] ←max(A[0..15],B[0..15]); … A[112..127] ←max(A[112..127], B[112..127]).
PMAXUB mmx, [mmx | m(64)]	A[0..7] ←max(A[0..7],B[0..7]); … A[56..63] ←max(A[56..63], B[56..63]).
PMAXUB xmm, [xmm | m(128)]	A[0..7] ←max(A[0..7],B[0..7]); … A[120..127] ←max(A[120..127], B[120..127]).
PMINSW mmx, [mmx | m(64)]	A[0..15] ←min(A[0..15],B[0..15]); … A[48..63] ←min(A[48..63], B[48..63]).
PMINSW xmm, [xmm | m(128)]	A[0..15] ←min(A[0..15],B[0..15]); … A[112..127] ←min(A[112..127], B[112..127]).
PMINUB mmx, [mmx | m(64)]	A[0..7] ←min(A[0..7],B[0..7]); … A[56..63] ←min(A[56..63], B[56..63]).
PMINUB xmm, [xmm | m(128)]	A[0..7] ←min(A[0..7],B[0..7]); … A[120..127] ←min(A[120..127], B[120..127]).
PMULHUW mmx, [mmx | m(64)]	TEMP[0..31]←A[0..15]*B[0..15]; A[0..15]←TEMP[16..31], … TEMP[0..31] ←A[48..63]*B[48..63]; A[48..63]←TEMP[16..31].
PMULHUW xmm, [xmm | m(128)]	TEMP[0..31] ←A[0..15]*B[0..15]; A[0..15]←TEMP[16..31], … TEMP[0..31]←A[112..127]*B[112..127]; A[112..127]←TEMP[16..31].
ADDPD xmm, [xmm | m(128)]	A ← A+B
ADDSD xmm, [xmm | m(64)]	A[0..63] ← A[0..63]+B[0..63]
SUBPD xmm, [xmm | m(128)]	A ← A–B

SUBSD xmm, [xmm | m(64)]	A[0..63] ← A[0..63] −B[0..63]
MULPD xmm, [xmm | m(128)]	A ← A*B
MULSD xmm, [xmm | m(64)]	A [0..63]← A[0..63]*B[0..63]
DIVPD xmm, [xmm | m(128)]	A ← A/B
DIVSD xmm, [xmm | m(64)]	A [0..63]← A[0..63]/B[0..63]
MAXPD xmm, [xmm | m(128)]	A[0..63] ←max(A[0..63],B[0..63]); A[64..127] ←max(A[64..127], B[64..127]).
MAXSD xmm, [xmm | m(64)]	A[0..63] ←max(A[0..63],B[0..63]).
MINPD xmm, [xmm | m(128)]	A[0..63] ←min(A[0..63],B[0..63]); A[64..127] ←min(A[64..127], B[64..127]).
MINSD xmm, [xmm | m(64)]	A[0..63] ←min(A[0..63],B[0..63]).
PADDQ mmx, [mmx | m(64)] PADDQ xmm, [xmm | m(128)]	A ← A+B
PSUBQ mmx, [mmx | m(64)] PSUBQ xmm, [xmm | m(128)]	A ← A−B
PMULUDQ mmx, [mmx | m(64)] PMULUDQ xmm, [xmm | m(128)]	A[0..63] ←A[0..31]*B[0..31]. A[0..63] ←A[0..31]*B[0..31]; A[64..127] ←A[64..95]*B[64..95].
SQRTPD xmm, [xmm | m(128)]	A[0..63] ← $\sqrt{B[0..63]}$; A[64..127] ← $\sqrt{B[64..127]}$.
SQRTSD xmm, [xmm | m(64)]	A[0..63] ← $\sqrt{B[0..63]}$
ADDSUBPD xmm, [xmm | m(128)]	A[0..63] ← A[0..63] −B[0..63]; A[64..127] ← A[64..127] +B[64..127].
ADDSUBPS xmm, [xmm | m(128)]	A[0..31] ← A[0..31] −B[0..31]; A[32..63] ← A[32..63] +B[32..63]; A[64..95] ← A[64..95] −B[64..95]; A[96..127] ← A[96..127] +B[96..127].
HADDPD xmm, [xmm | m(128)]	A[0..63] ←A[0..63]+A[64..127]; A[64..127] ←B[0..63]+B[64..127].
HADDPS xmm, [xmm | m(128)]	A[0..31] ←A[0..31]+A[32..63]; A[32..63] ←A[64..95]+A[96..127]; A[64..95] ←B[0..31]+B[32..63]; A[96..127] ←B[64..95]+B[96..127].
HSUBPD xmm, [xmm | m(128)]	A[0..63] ←A[0..63] −A[64..127]; A[64..127] ←B[0..63] −B[64..127].
HSUBPS xmm, [xmm | m(128)]	A[0..31] ←A[0..31] −A[32..63]; A[32..63] ←A[64..95] −A[96..127]; A[64..95] ←B[0..31] −B[32..63]; A[96..127] ←B[64..95] −B[96..127].

5.3. ZESTAW INSTRUKCJI PORÓWNAŃ

Tabela 5.3.

Instrukcje porównań rozszerzeń SIMD

Składnia instrukcji	Operacja
PCMPEQB mmx, [mmx | m(64)] PCMPEQB xmm, [xmm | m(128)]	A[x] ←0FFh if A[x]==B[x]; else A[x] ←0;
PCMPGTB mmx, [mmx | m(64)] PCMPGTB xmm, [xmm | m(128)]	A[x] ←0FFh if A[x]>B[x]; else A[x] ←0;
PCMPEQW mmx, [mmx | m(64)] PCMPEQW xmm, [xmm | m(128)]	A[x] ←0FFFFh if A[x]==B[x]; else A[x] ←0;
PCMPGTW mmx, [mmx | m(64)] PCMPGTW xmm, [xmm | m(128)]	A[x] ←0FFFFh if A[x]>B[x]; else A[x] ←0;
PCMPEQD mmx, [mmx | m(64)] PCMPEQD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]==B[x]; else A[x] ←0;
PCMPGTD mmx, [mmx | m(64)] PCMPGTD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]>B[x]; else A[x] ←0;
CMPEQPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]==B[x]; else A[x] ←0;
CMPLTPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]<B[x]; else A[x] ←0;
CMPLEPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]<=B[x]; else A[x] ←0;
CMPNEPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]!=B[x]; else A[x] ←0;
CMPNLTPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]>=B[x]; else A[x] ←0;
CMPNLEPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]>B[x]; else A[x] ←0;
CMPORDPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]!=NAN AND B[x]!=NAN; else A[x] ←0;
CMPUNORDPS xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFh if A[x]==NAN OR B[x]==NAN; else A[x] ←0;
CMPEQSS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]==B[0]; else A[0] ←0;
CMPLTSS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]<B[x]; else A[0] ←0;
CMPLESS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]<=B[x]; else A[0] ←0;
CMPNESS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]!=B[0]; else A[0] ←0;
CMPNLTSS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]>=B[0]; else A[0] ←0;
CMPNLESS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]>B[x]; else A[0] ←0;
CMPORDSS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]!=NAN AND B[0]!=NAN; else A[0] ←0;
CMPUNORDSS xmm, [xmm | m(32)]	A[0] ←0FFFFFFFFh if A[0]==NAN OR B[0]==NAN; else A[0] ←0;
CMPEQPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]==B[x]; else A[x] ←0;
CMPLTPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]<B[x]; else A[x] ←0;
CMPLEPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]<=B[x]; else A[x] ←0;
CMPNEPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]!=B[x]; else A[x] ←0;
CMPNLTPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]>=B[x]; else A[x] ←0;
CMPNLEPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]>B[x]; else A[x] ←0;

CMPORDPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]!=NAN AND B[x]!=NAN; else A[x] ←0;
CMPUNORDPD xmm, [xmm | m(128)]	A[x] ←0FFFFFFFFFFFFFFFFh if A[x]==NAN OR B[x]==NAN; else A[x] ←0;
CMPEQSD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]==B[0]; else A[0] ←0;
CMPLTSD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]<B[x]; else A[0] ←0;
CMPLESD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]<=B[x]; else A[0] ←0;
CMPNESD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]!=B[0]; else A[0] ←0;
CMPNLTSD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]>=B[0]; else A[0] ←0;
CMPNLESD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]>B[x]; else A[0] ←0;
CMPORDSD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]!=NAN AND B[0]!=NAN; else A[0] ←0;
CMPUNORDSD xmm, [xmm | m(64)]	A[0] ←0FFFFFFFFFFFFFFFFh if A[0]==NAN OR B[0]==NAN; else A[0] ←0;
COMISS xmm, [xmm | m(32)]	EFLAGS(ZF,PF,CF) ←111 IF A[0]==NAN OR B[0]==NAN; EFLAGS(ZF,PF,CF) ←000 IF A[0]> B[0]; EFLAGS(ZF,PF,CF) ←001 IF A[0]< B[0]; EFLAGS(ZF,PF,CF) ←100 IF A[0]==B[0];
UCOMISS xmm, [xmm | m(32)]	EFLAGS(ZF,PF,CF) ←111 IF A[0]==NAN OR B[0]==NAN; EFLAGS(ZF,PF,CF) ←000 IF A[0]> B[0]; EFLAGS(ZF,PF,CF) ←001 IF A[0]< B[0]; EFLAGS(ZF,PF,CF) ←100 IF A[0]==B[0]; Bez generowania wyjątku po natrafieniu na NAN.
CMPPD xmm, [xmm | m(128)], KOD	W zależności od kodu działanie jak jedna z instrukcji w tabeli.
CMPSD xmm, [xmm | m(64)], KOD	W zależności od kodu działanie jak jedna z instrukcji w tabeli.
CMPPS xmm, [xmm | m(128)], KOD	W zależności od kodu działanie jak jedna z instrukcji w tabeli.
CMPSS xmm, [xmm | m(32)], KOD	W zależności od kodu działanie jak jedna z instrukcji w tabeli.
COMISD xmm, [xmm | m(64)]	EFLAGS(ZF,PF,CF) ←111 IF A[0]==NAN OR B[0]==NAN; EFLAGS(ZF,PF,CF) ←000 IF A[0]> B[0]; EFLAGS(ZF,PF,CF) ←001 IF A[0]< B[0]; EFLAGS(ZF,PF,CF) ←100 IF A[0]==B[0];
UCOMISD xmm, [xmm | m(64)]	EFLAGS(ZF,PF,CF) ←111 IF A[0]==NAN OR B[0]==NAN; EFLAGS(ZF,PF,CF) ←000 IF A[0]> B[0]; EFLAGS(ZF,PF,CF) ←001 IF A[0]< B[0]; EFLAGS(ZF,PF,CF) ←100 IF A[0]==B[0]; Bez generowania wyjątku po natrafieniu na NAN.

5.4. ZESTAW INSTRUKCJI KONWERSJI TYPÓW

Tabela 5.4.

Lista instrukcji konwertujących dane rozszerzeń MMX, SSE, SSE2, SSE3

Składnia instrukcji	Operacja
CVTPI2PS xmm, [mmx | m(64)]	2 x single ← 2 x integer(32)
CVTPS2PI mmx, [xmm | m(64)]	2 x integer(32) ← 2 x single
CVTSI2SS xmm, [r(32) | m(32)]	single ← integer(32)
CVTSS2SI r(32), [xmm | m(32)]	integer(32) ← single
CVTTPS2PI mmx, [xmm | m(64)]	2 x integer(32) ← 2 x single zaokrąglenie z przycięciem wartości
CVTTSS2SI r(32), [xmm | m(32)]	integer(32) ← single zaokrąglenie z przycięciem wartości
CVTDQ2PD xmm, [xmm | m(64)]	2 x double ← 2 x integer(32)
CVTDQ2PS xmm, [xmm | m(128)]	4 x single ← 4 x integer(32)
CVTPD2PI mmx, [xmm | m(128)]	2 x integer(32) ← 2 x double
CVTPD2DQ xmm, [xmm | m(128)]	2 x integer(32) ← 2 x double
CVTPD2PS xmm, [xmm | m(128)]	2 x single ← 2 x double
CVTPI2PD xmm, [mmx | m(64)]	2 x double ← 2 x integer(32)
CVTPS2DQ xmm, [xmm | m(128)]	4 x integer(32) ← 4 x single
CVTPS2PD xmm, [xmm | m(64)]	2 x double ← 2 x single
CVTSD2SI r(32), [xmm | m(64)]	integer(32) ← double
CVTSD2SS xmm, [xmm | m(64)]	single ← double
CVTSI2SD xmm, [r(32) | m(32)]	double ← integer(32)
CVTSI2SS xmm, [r(32) | m(32)]	single ← integer(32)
CVTSS2SD xmm, [xmm | m(32)]	double ← single
CVTSS2SI r(32), [xmm | m(32)]	integer(32) ← single
CVTTPD2PI mmx, [xmm | m(128)]	2 x integer(32) ← 2 x double zaokrąglenie z przycięciem wartości
CVTTPD2DQ xmm, [xmm | m(128)]	2 x integer(32) ← 2 x double zaokrąglenie z przycięciem wartości
CVTTPS2DQ xmm, [xmm | m(128)]	4 x integer(32) ← 4 x single zaokrąglenie z przycięciem wartości
CVTTPS2PI mmx, [xmm | m(64)]	2 x integer(32) ← 2 x single zaokrąglenie z przycięciem wartości
CVTTSD2SI r(32), [xmm | m(64)]	integer(32) ← double zaokrąglenie z przycięciem wartości
CVTTSS2SI r(32), [xmm | m(32)]	integer(32) ← single zaokrąglenie z przycięciem wartości
FISTTP m(16) FISTTP m(32) FISTTP m(64)	integer(16) ← ST(0) zaokrąglenie z przycięciem wartości integer(32) ← ST(0) zaokrąglenie z przycięciem wartości integer(64) ← ST(0) zaokrąglenie z przycięciem wartości

5.5. ZESTAW INSTRUKCJI KOMPRESJI, DEKOMPRESJI I MIESZANIA DANYCH

Tabela 5.5.

Lista instrukcji kompresji, dekompresji i mieszania danych rozszerzeń wektorowych

Składnia instrukcji	Operacja
PACKSSDW mmx, [mmx | m(64)] PACKSSDW xmm, [xmm | m(128)]	4 x integer(16) ← (2 x integer(32) w A) i (2 x integer(32) w B); 8 x integer(16) ← (4 x integer(32) w A) i (4 x integer(32) w B);
PACKSSWB mmx, [mmx | m(64)] PACKSSWB xmm, [xmm | m(128)]	8 x integer(8) ← (4 x integer(16) w A) i (4 x integer(16) w B); 16 x integer(8) ← (8 x integer(16) w A) i (8 x integer(16) w B);
PACKUSWB mmx, [mmx | m(64)] PACKUSWB xmm, [xmm | m(128)]	8 x integer(8) ← (4 x integer(16) w A) i (4 x integer(16) w B); 16 x integer(8) ← (8 x integer(16) w A) i (8 x integer(16) w B); Elementy wektora wynikowego bez znaków.
PUNPCKHBW mmx, [mmx | m(64)] PUNPCKHBW xmm, [xmm | m(128)]	A[0]←A[4]; A[1]←B[4]; A[2] ←A[5]; A[3] ←B[5]; A[4]←A[6]; A[5]←B[6]; A[6]←A[7]; A[7]←B[7]. (elementy 8-bitowe) A[0]←A[8]; A[1]←B[8]; A[2] ←A[9]; A[3] ←B[9]; A[4]←A[10]; A[5]←B[10]; A[6]←A[11]; A[7]←B[11]; A[8]←A[12]; ...; A[14]←A[15]; A[15]←B[15]. (elementy 8-bitowe)
PUNPCKHDQ mmx, [mmx | m(64)] PUNPCKHDQ xmm, [xmm | m(128)]	A[0]←A[1]; A[1]←B[1]. (elementy 32-bitowe) A[0]←A[2]; A[1]←B[2]; A[2]←A[3]; A[3]←B[3]; (elementy 32-bitowe)
PUNPCKHWD mmx, [mmx | m(64)] PUNPCKHWD xmm, [xmm | m(128)]	A[0]←A[2]; A[1]←B[2]; A[2]←A[3]; A[3]←B[3]. (elementy 16-bitowe) A[0]←A[4]; A[1]←B[4]; A[2]←A[5]; A[3]←B[5]; A[4]←A[6]; A[5]←B[6]; A[6]←A[7]; A[7]←B[7]; (elementy 16-bitowe)
PUNPCKHQDQ xmm, [xmm | m(128)]	A[0]←A[1]; A[1]←B[1]. (elementy 64-bitowe)
PUNPCKLBW mmx, [mmx | m(64)] PUNPCKLBW xmm, [xmm | m(128)]	A[0]←A[0]; A[1]←B[0]; A[2] ←A[1]; A[3] ←B[1]; A[4]←A[2]; A[5]←B[2]; A[6]←A[3]; A[7]←B[3]. (elementy 8-bitowe) A[0]←A[0]; A[1]←B[0]; A[2] ←A[1]; A[3] ←B[1]; A[4]←A[2]; A[5]←B[2]; A[6]←A[3]; A[7]←B[3]; A[8]←A[4]; ...; A[14]←A[7]; A[15]←B[7]. (elementy 8-bitowe)
PUNPCKLDQ mmx, [mmx | m(64)] PUNPCKLDQ xmm, [xmm | m(128)]	A[0]←A[0]; A[1]←B[0]. (elementy 32-bitowe) A[0]←A[0]; A[1]←B[0]; A[2]←A[1]; A[3]←B[1]; (elementy 32-bitowe)
PUNPCKLWD mmx, [mmx | m(64)] PUNPCKLWD xmm, [xmm | m(128)]	A[0]←A[0]; A[1]←B[0]; A[2]←A[1]; A[3]←B[1]. (elementy 16-bitowe) A[0]←A[0]; A[1]←B[0]; A[2]←A[1]; A[3]←B[1]; A[4]←A[2]; A[5]←B[2]; A[6]←A[3]; A[7]←B[3]; (elementy 16-bitowe)
PUNPCKLQDQ xmm, [xmm | m(128)]	A[0]←A[0]; A[1]←B[0]. (elementy 64-bitowe)
UNPCKHPS xmm, [xmm | m(128)]	A[0]←A[2]; A[1]←B[2]; A[2]←A[3]; A[3]←B[3]. (liczby single)

UNPCKLPS xmm, [xmm | m(128)]	A[0]←A[0]; A[1]←B[0]; A[2]←A[1]; A[3]←B[1]. (liczby single)
UNPCKHPD xmm, [xmm | m(128)]	A[0]←A[1]; A[1]←B[1]. (liczby double)
UNPCKLPD xmm, [xmm | m(128)]	A[0]←A[0]; A[1]←B[0]. (liczby double)
PSHUFW mmx, [mmx | m(64)], stala	A[0]←B[stala AND 11B]; A[1]←B[(stala AND 1100B)>>2]; A[2]←B[(stala AND 110000B)>>4]; A[3]←B[(stala AND 11000000B)>>6];
SHUFPS xmm, [xmm | m(128)], stala	A[0]←A[stala AND 11B]; A[1]←A[(stala AND 1100B)>>2]; A[2]←B[(stala AND 110000B)>>4]; A[3]←B[(stala AND 11000000B)>>6];
PSHUFD xmm, [xmm | m(128)], stala	A[0]←B[stala AND 11B]; A[1]←B[(stala AND 1100B)>>2]; A[2]←B[(stala AND 110000B)>>4]; A[3]←B[(stala AND 11000000B)>>6];
PSHUFHW xmm, [xmm | m(128)], stala	A[0..63] ←B[0..63]; A[4]←B[4+(stala AND 11B)]; A[5]←B[4+(stala AND 1100B)>>2]; A[6]←B[4+(stala AND 110000B)>>4]; A[7]←B[4+(stala AND 11000000B)>>6];
PSHUFLW xmm, [xmm | m(128)], stala	A[0]←B[stala AND 11B]; A[1]←B[(stala AND 1100B)>>2]; A[2]←B[(stala AND 110000B)>>4]; A[3]←B[(stala AND 11000000B)>>6]; A[64..127] ←B[64..127]

5.6. ZESTAW INSTRUKCJI OPERACJI LOGICZNYCH

Tabela 5.6.

Lista instrukcji logicznych rozszerzeń SIMD

Składnia instrukcji	Operacja
PXOR mmx, [mmx | m(64)] PXOR xmm, [xmm | m(128)]	A ← A XOR B
POR mmx, [mmx | m(64)] POR xmm, [xmm | m(128)]	A ← A OR B
PAND mmx, [mmx | m(64)] PAND xmm, [xmm | m(128)]	A ← A AND B
PANDN mmx, [mmx | m(64)] PANDN xmm, [xmm | m(128)]	A ← (NOT A) AND B
ANDNPS xmm, [xmm | m(128)]	A ← (NOT A) AND B
ANDPS xmm, [xmm | m(128)]	A ← A AND B
ORPS xmm, [xmm | m(128)]	A ← A OR B
XORPS xmm, [xmm | m(128)]	A ← A XOR B
ANDNPD xmm, [xmm | m(128)]	A ← (NOT A) AND B
ANDPD xmm, [xmm | m(128)]	A ← A AND B
ORPD xmm, [xmm | m(128)]	A ← A OR B
XORPD xmm, [xmm | m(128)]	A ← A XOR B

5.7. ZESTAW INSTRUKCJI PRZESUNIĘĆ BITOWYCH

Tabela 5.7.

Instrukcje przesunięć bitowych

Składnia instrukcji	Operacja
PSRLW mmx, [mmx | m(64) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B); A[2] ←(A[2]>>B); A[3] ←(A[3]>>B). (operacje na słowach)
PSRLW xmm, [xmm | m(128) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B); …; A[7] ←(A[7]>>B). (operacje na słowach)
PSRLD mmx, [mmx | m(64) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B). (operacje na podwójnych słowach)
PSRLD xmm, [xmm | m(128) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B); A[2] ←(A[2]>>B); A[3] ←(A[3]>>B). (operacje na podwójnych słowach)
PSRLQ mmx, [mmx | m(64) | stala]	A[0] ←(A[0]>>B). (operacja na poczwórnym słowie)
PSRLQ xmm, [xmm | m(128) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B). (operacja na poczwórnych słowach)
PSRLDQ xmm, stala	A ←(A>>(B*8)).
PSLLW mmx, [mmx | m(64) | stala]	A[0] ←(A[0]<<B); A[1] ←(A[1] <<B); A[2] ←(A[2] <<B); A[3] ←(A[3] <<B). (operacje na słowach)
PSLLW xmm, [xmm | m(128) | stala]	A[0] ←(A[0] <<B); A[1] ←(A[1] <<B); …; A[7] ←(A[7] <<B). (operacje na słowach)
PSLLD mmx, [mmx | m(64) | stala]	A[0] ←(A[0] <<B); A[1] ←(A[1] <<B). (operacje na podwójnych słowach)
PSLLD xmm, [xmm | m(128) | stala]	A[0] ←(A[0] <<B); A[1] ←(A[1] <<B); A[2] ←(A[2] <<B); A[3] ←(A[3] <<B). (operacje na podwójnych słowach)
PSLLQ mmx, [mmx | m(64) | stala]	A[0] ←(A[0] <<B). (operacja na poczwórnym słowie)
PSLLQ xmm, [xmm | m(128) | stala]	A[0] ←(A[0] <<B); A[1] ←(A[1] <<B). (operacje na poczwórnych słowach)
PSLLDQ xmm, stala	A ←(A <<(B*8)).
PSRAW mmx, [mmx | m(64) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B); A[2] ←(A[2]>>B); A[3] ←(A[3]>>B). Przesunięcie arytmetyczne. (operacje na słowach)
PSRAW xmm, [xmm | m(128) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B); …; A[7] ←(A[7]>>B). Przesunięcie arytmetyczne. (operacje na słowach)
PSRAD mmx, [mmx | m(64) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B). Przesunięcie arytmetyczne. (operacje na podwójnych słowach)
PSRAD xmm, [xmm | m(128) | stala]	A[0] ←(A[0]>>B); A[1] ←(A[1]>>B); A[2] ←(A[2]>>B); A[3] ←(A[3]>>B). Przesunięcie arytmetyczne. (operacja na podwójnych słowach)

5.8. ZESTAW POZOSTAŁYCH INSTRUKCJI WPROWADZONYCH Z ROZSZERZENIAMI SIMD

Tabela 5.8.

Pozostałe instrukcje rozszerzeń SIMD

Składnia instrukcji	Operacja
EMMS	Inicjuje pracę jednostki MMX.
PEXTRW r(32), [mmx | xmm], stala	r[0..16] ←B[stala]. Kopiuje słowo o indeksie określonym 3 operandem z rejestru MMX lub XMM do rejestru ogólnego przeznaczenia.
PINSRW [mmx | xmm], r(32), stala	B[stala] ←r[0..16]. Kopiuje młodsze słowo rejestru ogólnego przeznaczania do rejestru MMX lub XMM na pozycję określoną trzecim operandem.
FXRSTOR m(512)	(FPU, MMX, XMM, MXCSR) ← A. Odtworzenie stanu jednostek obliczeniowych procesora.
FXSAVE m(512)	A ← (FPU, MMX, XMM, MXCSR). Zapisanie stanu jednostek: FPU, MMX, SSE.
LDMXCSR m(32)	MXCSR ← A. Odtworzenie stanu rejestru MXCSR.
STMXCSR m(32)	A ← MXCSR. Zachowanie rejestru MXCSR.
PREFETCHT0 m(8)	CACHE ← A. Zapis danej do wszystkich poziomów pamięci cache.
PREFETCHT1 m(8)	CACHE ← A. Zapis danej na 2 poziom pamięci cache i wyżej.
PREFETCHT2 m(8)	CACHE ← A. Zapis danej na 2 poziom pamięci cache i wyżej.
PREFETCHNTA m(8)	CACHE ← A. Zapis danej tak, aby procesor mógł mieć do niej szybszy dostęp, minimalizując zaśmiecenie pamięci cache.
SFENCE	Gwarantuje zapisanie do pamięci operacyjnej wszystkich danych trzymanych dotąd w buforze zapisu.
CLFLUSH m(8)	Unieważnia linie pamięci cache, w których zawartość pochodzi spod adresu liniowego określonego adresem operandu.
LFENCE	Gwarantuje odczyt z pamięci operacyjnej wszystkich danych, które dotąd nie zostały odczytane.
MFENCE	Gwarantuje odczyt i zapis wszystkich danych (połączone instrukcje SFENCE i LFENCE).
PAUSE	Instrukcja wstrzymująca chwilowo pracę procesora. Należy ją wykorzystywać w pętlach spowolniających program.
MONITOR	Instrukcja używana do synchronizacji międzyprocesowej. Jej wywołanie spowoduje postawienie monitora sprzętowego na adres określony parą rejestrów DS:EAX (rozmiar obszaru objętego monitorem jest zależny od wersji procesora). Próba zapisu do obszaru objętego monitorem spowoduje zawieszenie wykonywanie się procesu.
MWAIT	Instrukcja pozwala na przejście logicznego procesora (technologia HT) w optymalny stan podczas oczekiwania na wykonanie instrukcji zapisu pod obszar określony instrukcją MONITOR.

6. LISTY INSTRUKCJI DLA POSZCZEGÓLNYCH ROZSZERZEŃ

6.1. INSTRUKCJE MMX

Tabela 6.1.

Lista instrukcji dodana wraz z rozszerzeniem MMX

TYP INSTRUKCJI	MNEMONIK	KOD OPERACJI
TRANSFER DANYCH	MOVD	0F6E dla MMX←[R|M(32)] 0F7E dla [R|M(32)]←MMX
	MOVQ	0F6E dla MMX←M(64) 0F7E dla M(64)←MMX
INSTRUKCJE PAKUJĄCE DANE	PACKSSDW	0F6B
	PACKSSWB	0F63
	PACKUSWB	0F67
	PUNPCKHBW	0F68
	PUNPCKHDQ	0F6A
	PUNPCKHWD	0F69
	PUNPCKLBW	0F60
	PUNPCKLDQ	0F62
	PUNPCKLWD	0F61
INSTRUKCJE ARYTMETYCZNE	PADDB	0FFC
	PADDD	0FFE
	PADDSB	0FEC
	PADDSW	0FED
	PADDUSB	0FDC
	PADDUSW	0FDD
	PADDW	0FFD
	PMADDWD	0FF5
	PMULHW	0FE5
	PMULLW	0FD5
	PSUBB	0FF8
	PSUBD	0FFA
	PSUBSB	0FE8
	PSUBSW	0FE9
	PSUBUSB	0FD8
	PSUBUSW	0FD9
	PSUBW	0FF9

INSTRUKCJE PORÓWNANIA	PCMPEQB	0F74
	PCMPEQD	0F76
	PCMPEQW	0F75
	PCMPGTB	0F64
	PCMPGTD	0F66
	PCMPGTW	0F65
OPERACJE LOGICZNE	PAND	0FDB
	PANDN	0FDF
	POR	0FEB
	PXOR	0FEF
PRZESUNIĘCIA BITOWE	PSLLD	0FF2 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F72/6 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSLLQ	0FF3 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F73/6 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSLLW	0FF1 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F71/6 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSRAD	0FE2 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F72/4 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSRAW	0FE1 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F71/4 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSRLD	0FD2 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F72/2 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSRLQ	0FD3 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F73/2 gdy ilość przesuwanych bitów określona stałą natychmiastową
	PSRLW	0FD1 gdy ilość przesuwanych bitów określona przez [mmx | m(64)] 0F71/2 gdy ilość przesuwanych bitów określona stałą natychmiastową
POZOSTAŁE	EMMS	0F77

6.2. INSTRUKCJE SSE

Tabela 6.2.

Instrukcje dodane wraz z rozszerzeniem SSE

TYP INSTRUKCJI	MNEMONIK	KOD OPERACJI
TRANSFER DANYCH	MOVAPS	0F28 dla XMM←[XMM|M(128)] 0F29 dla [XMM|M(128)]←XMM
	MOVHLPS	0F12
	MOVHPS	0F16 dla XMM←M(64) 0F17 dla M(64)←XMM
	MOVLHPS	0F16
	MOVLPS	0F12 dla XMM←M(64) 0F13 dla M(64)←XMM
	MOVMSKPS	0F50
	MOVSS	F30F10 dla XMM←[XMM|M(32)] F30F11 dla [XMM|M(32)]←XMM
	MOVUPS	0F10 dla XMM←[XMM|M(128)] 0F11 dla [XMM|M(128)]←XMM
	PMOVMSKB	0FD7 dla R(32)←MMX 660FD7 dla R(32) ←XMM
	MOVNTPS	0F2B
	MOVNTQ	0FE7
	MASKMOVQ	0FF7
OPERACJE ARYTMETYCZNE	ADDPS	0F58
	ADDSS	F30F58
	DIVPS	0F5E
	DIVSS	F30F5E
	MAXPS	0F5F
	MAXSS	F30F5F
	MINPS	0F5D
	MINSS	F30F5D
	MULPS	0F59
	MULSS	F30F59
	RCPPS	0F53
	RCPSS	F30F53
	RSQRTPS	0F52
	RSQRTSS	F30F52
	SQRTPS	0F51
	SQRTSS	F30F51
	SUBPS	0F5C
	SUBSS	F30F5C
	PMAXSW	0FEE dla MMX←[MMX|M(64)] 660FEE dla XMM ←[XMM|M(128)]
	PMAXUB	0FDE dla MMX←[MMX|M(64)] 660FDE dla XMM ←[XMM|M(128)]
	PMINSW	0FEA dla MMX←[MMX|M(64)] 660FEA dla XMM ←[XMM|M(128)]
	PMINUB	0FDA dla MMX←[MMX|M(64)] 660FDA dla XMM ←[XMM|M(128)]

OPERACJE ARYTMETYCZNE	PMULHUW	0FE4 dla MMX←[MMX|M(64)] 660FE4 dla XMM ←[XMM|M(128)]
	PSADBW	0FF6 dla MMX←[MMX|M(64)] 660FF6 dla XMM ←[XMM|M(128)]
	PAVGB	0FE0 dla MMX←[MMX|M(64)] 660FE0 dla XMM ←[XMM|M(128)]
	PAVGW	0FE3 dla MMX←[MMX|M(64)] 660FE3 dla XMM ←[XMM|M(128)]
INSTRUKCJE PORÓWNANIA	CMPEQPS	0FC2
	CMPLTPS	0FC2
	CMPLEPS	0FC2
	CMPNEQPS	0FC2
	CMPNLTPS	0FC2
	CMPNLEPS	0FC2
	CMPORDPS	0FC2
	CMPUNORDPS	0FC2
	CMPEQSS	F30FC2
	CMPLTSS	F30FC2
	CMPLESS	F30FC2
	CMPNEQSS	F30FC2
	CMPNLTSS	F30FC2
	CMPNLESS	F30FC2
	CMPORDSS	F30FC2
	CMPUNORDSS	F30FC2
	COMISS	0F2F
	UCOMISS	0F2E
OPERACJE LOGICZNE	ANDNPS	0F55
	ANDPS	0F54
	ORPS	0F56
	XORPS	0F57
INSTRUKCJE KONWERSJI	CVTPI2PS	0F2A
	CVTPS2PI	0F2D
	CVTSI2SS	F30F2A
	CVTSS2SI	F30F2D
	CVTTPS2PI	0F2C
	CVTTSS2SI	F30F2C
INSTRUKCJE PAKUJĄCE I MIESZAJĄCE DANE	SHUFPS	0FC6
	UNPCKHPS	0F15
	UNPCKLPS	0F14
	PEXTRW	0FC5 dla R(32)←MMX, stala 660FC5 dla R(32) ←XMM, stala
	PINSRW	0FC4 dla MMX←[R(32)|M(16)], stala 660FC4 dla XMM ←[R(32)|M(16)], stala
	PSHUFW	0F70
POZOSTAŁE	LDMXCSR	0FAE/2
	STMXCSR	0FAE/3
	PREFETCHNTA	0F18/0
	PREFETCHT0	0F18/1
	PREFETCHT1	0F18/2
	PREFETCHT2	0F18/3
	SFENCE	0FAE/7

6.3. INSTRUKCJE SSE2

Tabela 6.3.

Instrukcje dodane wraz z rozszerzeniem SSE2

TYP INSTRUKCJI	MNEMONIK	KOD OPERACJI
INSTRUKCJE TRANSFERU DANYCH	MOVAPD	660F28 dla XMM←[XMM|M(128)] 660F29 dla [XMM|M(128)]←XMM
	MOVHPD	660F16 dla XMM←M(64) 660F17 dla M(64)←XMM
	MOVLPD	660F12 dla XMM←M(64) 660F13 dla M(64)←XMM
	MOVMSKPD	660F50
	MOVSD	F20F10 dla XMM←[XMM|M(64)] F20F11 dla [XMM|M(64)]←XMM
	MOVUPD	660F10 dla XMM←[XMM|M(128)] 660F11 dla [XMM|M(128)]←XMM
	MOVDQ2Q	F20FD6
	MOVDQA	660F6F dla XMM←[XMM|M(128)] 660F7F dla [XMM|M(128)]←XMM
	MOVDQU	F30F6F dla XMM←[XMM|M(128)] F30F7F dla [XMM|M(128)]←XMM
	MOVQ2DQ	F30FD6
	MASKMOVDQU	660FF7
	MOVNTDQ	660FE7
	MOVNTI	0FC3
	MOVNTPD	660F2B
INSTRUKCJE OPERACJI ARYTMETYCZNYCH	ADDPD	660F58
	ADDSD	F20F58
	DIVPD	660F5E
	DIVSD	F20F5E
	MAXPD	660F5F
	MAXSD	F20F5F
	MINPD	660F5D
	MINSD	F20F5D
	MULPD	660F59
	MULSD	F20F59
	SQRTPD	660F51
	SQRTSD	F20F51
	SUBPD	660F5C
	SUBSD	F20F5C
	PADDQ	0FD4 dla MMX, [MMX|M(64)] 660FD4 dla XMM|←[XMM|M(128)]
	PMULUDQ	0FF4 dla MMX, [MMX|M(64)] 660FF4 dla XMM|←[XMM|M(128)]
	PSUBQ	0FFB dla MMX, [MMX|M(64)] 660FFB dla XMM|←[XMM|M(128)]

INSTRUKCJE OPERACJI LOGICZNYCH	ANDNPD	660F55
	ANDPD	660F54
	ORPD	660F56
	XORPD	660F57
PORÓWNANIA	CMPEQPD	660FC2
	CMPLTPD	660FC2
	CMPLEPD	660FC2
	CMPNEQPD	660FC2
	CMPNLTPD	660FC2
	CMPNLEPD	660FC2
	CMPORDPD	660FC2
	CMPUNORDPD	660FC2
	CMPEQSD	F20FC2
	CMPLTSD	F20FC2
	CMPLESD	F20FC2
	CMPNEQSD	F20FC2
	CMPNLTSD	F20FC2
	CMPNLESD	F20FC2
	CMPORDSD	F20FC2
	CMPUNORDSD	F20FC2
	COMISD	660F2F
	UCOMISD	660FE
INSTRUKCJE KOMPRESJI, DEKOMPRESJI I MIESZANIA DANYCH	SHUFPD	660FC6
	UNPCKHPD	660F15
	UNPCKLPD	660F14
	PSHUFD	660F70
	PSHUFHW	F30F70
	PSHUFLW	F20F70
	PUNPCKHQDQ	660F6D
	PUNPCKLQDQ	660F6C
INSTRUKCJE KONWERSJI	CVTDQ2PD	F30FE6
	CVTPD2DQ	F20FE6
	CVTPD2PI	660F2D
	CVTPD2PS	660F5A
	CVTPI2PD	660F2A
	CVTPS2PD	0F5A
	CVTSD2SI	F20F2D
	CVTSD2SS	F20F5A
	CVTSI2SD	F20F2A
	CVTSS2SD	F30F5A
	CVTTPD2DQ	660FE6
	CVTTPD2PI	660F2C
	CVTTSD2SI	F20F2C
	CVTDQ2PS	0F5B
	CVTPS2DQ	660F5B
	CVTTPS2DQ	F30F5B

INSTRUKCJE PRZESUNIĘĆ BITOWYCH	PSLLDQ	660F73/7
	PSRLDQ	660F73/3
POZOSTAŁE INSTRUKCJE	CLFLUSH	0FAE/7
	LFENCE	0FAE/5
	MFENCE	0FAE/6
	PAUSE	F390

6.4. INSTRUKCJE SSE3

Tabela 6.4.

Instrukcje dodane wraz z rozszerzeniem SSE3

TYP INSTRUKCJI	MNEMONIK	KOD OPERACJI
INSTRUKCJE TRANSFERU DANYCH	LDDQU	F20FF0
	MOVDDUP	F20F12
	MOVSHDUP	F30F16
	MOVSLDUP	F30F12
INSTRUKCJE OPERACJI ARYTMETYCZNYCH	ADDSUBPD	660FD0
	ADDSUBPS	F20FD0
	HADDPD	660F7C
	HADDPS	F20F7C
	HSUBPD	660F7D
	HSUBPS	F20F7D
INSTRUKCJE KONWERSJI	FISTTP	DF/1 dla operandu 16-bitowego DB/1 dla operandu 32-bitowego DD/1 dla operandu 64-bitowego
POZOSTAŁE INSTRUKCJE	MONITOR	0F01C8
	MWAIT	0F01C9

SŁOWNIK WYBRANYCH POJĘĆ

3DNow! – Rozszerzenie wektorowe wprowadzone przez firmę AMD umożliwiające wykonywanie operacji wektorowych na liczbach zmiennoprzecinkowych.

AltiVec – zestaw instrukcji umożliwiający wykonywanie operacji wektorowych, zaimplementowany w procesorach PowerPC.

Architektura superskalarna (superscalar architecture) – polega na wprowadzeniu do procesora szeregu równoległych jednostek przetwarzających ze swoimi potokami.

AVX (Advanced Vector Extensions) – planowane rozszerzenie instrukcji przetwarzania wektorowego, które ma zostać wprowadzone przez firmę Intel. Rozszerzenie będzie wprowadzało do architektury procesora 16 nowych 256-bitowych rejestrów YMM. Dodatkowo rozszerzenie będzie posiadało instrukcje wspomagające szyfrowanie algorytmem AES.

Big Endian – sposób przechowywania danych w pamięci polegający na umieszczeniu najbardziej znaczącego bitu danej jako pierwszego.

BMP (Bitmap files) – najpopularniejszy format przechowywania grafiki w postaci bezstratnej.

Cache (pamięć podręczna) – szybki bufor podręczny procesora, w którym gromadzone są dane lub rozkazy ostatnio przetwarzane.

CBEA (Cell Broadband Engine Architecture) – współczesny mikroprocesor wytwarzany w technologii 90nm SOI. W skład jego struktury wchodzi jedna jednostka PPE oraz osiem jednostek wektorowych SPE.

DMA (Direct Memory Access) – układ umożliwiający bezpośredni dostęp urządzeń komputera do pamięci operacyjnej z pominięciem procesora.

Double – liczba zmiennoprzecinkowa podwójnej precyzji, której format jest określony standardem IEEE 754 (1bit znaku, 11bitów wykładnika, 52 bity mantysy).

Flops (FLoating point Operations Per Second) – liczba operacji zmiennoprzecinkowych na sekundę (jednostka wydajności komputerów).

FPU (Floating Point Unit) – jednostka procesora umożliwiająca wykonywanie operacji zmiennoprzecinkowych na danych skalarnych.

Fraktal – obiekt samopodobny o nietrywialnej strukturze.

Instrukcja horyzontalna – operacja, która jest wykonywana na elementach składowych wektora zapisanych w tym samym rejestrze.

Kod ASCII (American Standard Code for Information Interchange) – sposób zapamiętywania znaków alfabetu angielskiego, cyfr oraz znaków specjalnych pod postacią wartości liczbowych.

Kod BCD (Binary-Coded Decimal) – format zapisu liczb polegający na kodowaniu cyfr dziesiętnych przy użyciu 4 bitów.

Kod maszynowy – zbiór instrukcji możliwych bezpośrednio do wykonania przez procesor.

Kompilacja – proces tłumaczenia kodu źródłowego na kod maszynowy.

Konwersja typów – proces, w wyniku którego następuje zmiana formatu liczby z operandu źródłowego na format pożądany, zachowywany w operandzie docelowym.

Little Endian – sposób przechowywania danych w pamięci polegający na zapisie najmniej znaczącego bitu danej jako pierwszego.

MIMD (Multiple Instruction Multiple Data) – sposób przetwarzania danych typowy dla systemów wieloprocesorowych (wiele programów operuje na różnych zbiorach danych).

MIPS (Milion Instructions Per Second) – miara wydajności komputerów – określa liczbę milionów operacji na sekundę (dotyczy jednostki stałoprzecinkowej).

MFLOPS (Milion Float Operations Per Second) – miara wydajności komputerów – określa liczbę milionów operacji zmiennoprzecinkowych na sekundę.

MISD (Multiple Instruction Single Data) – sposób przetwarzania polegający na wykorzystywaniu przez wiele programów jednego zbioru danych.

MMX (Multimedia eXtensions lub Matrix Math eXtensions) – rozszerzenie wektorowe wprowadzone przez firmę Intel, umożliwiające wykonywanie operacji równoległych na liczbach całkowitych.

Najmłodszy bit (Least significant bit) – bit umieszczony na zerowej pozycji liczby binarnej, decydujący o tym, czy liczba jest parzysta czy nie.

Najstarszy bit (Most significant bit) – bit umieszczony na najwyższej pozycji liczby binarnej, nadający liczbie największą wartość (lub określający znak liczby).

NaN (Not a Number) – powstaje jako wynik operacji zmiennoprzecinkowej na niedozwolonych wartościach argumentów.

NASM (Netwide Assembler) – niekomercyjny translator języka asembler.

OFFSET – przesunięcie (adres względny) względem początku segmentu.

Operand – argument instrukcji procesora.

Overflow (przepełnienie) – sytuacja pojawia, się gdy wynik operacji jest większy od maksymalnej wartości możliwej do zapamiętania przez operand docelowy.

Pentium – nazwa grupy mikroprocesorów wytwarzanych przez firmę Intel. Mikroprocesory Pentium są następcami procesorów 80486.

Piksel – najmniejszy element obrazu wyświetlanego na ekranie.

PlayStation3 – konsola gier wideo produkowana przez firmę Sony. Konsola wyposażona jest w mikroprocesor CBEA.

PPE (Power Processor Element) – element architektury CBEA odpowiedzialny za wykonywanie kodu systemu operacyjnego oraz aplikacji. Zgodny z 64-bitową architekturą procesorów Power.

Procesory rodziny x86 – procesory firmy Intel, stosowane między innymi w komputerach PC, dla których jest zapewniona kompatybilność w dół (każdy kolejny procesor rodziny realizuje wszystkie funkcje realizowane przez procesory wcześniejsze). Rodzina procesorów została zapoczątkowana przez 16-bitowy procesor 8086 (1978 r.).

Przerwanie (interrupt) – pojawienie się przerwania powoduje zawieszenie aktualnie wykonującego się programu oraz wywołanie procedury odpowiedzialnej za obsługę pojawiającego się zdarzenia.

Przetwarzanie asymetryczne – operacja umożliwiająca wykonywanie różnych operacji arytmetycznych na poszczególnych elementach wektorów.

Rejestr flagowy – zawiera szereg bitów charakteryzujących wynik ostatniej operacji arytmetycznej (EFLAGS).

Rejestry ogólnego przeznaczenia – cechą charakterystyczną tych rejestrów jest możliwość ich użycia w dowolnych operacjach arytmetycznych i logicznych. Są to rejestry EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI oraz ich części składowe (AX, BX, CX, DX, BP, SP, SI, DI, AH, AL, BH, BL, CH, CL, DH, DL).

Rejestry segmentowe – w trybie rzeczywistym rejestry przechowują adresy bazowe segmentów, natomiast w trybie chronionym – selektory segmentów. Do rejestrów segmentowych zaliczamy rejestry: CS, DS, SS, ES, FS, GS.

RGB – model określający kolor piksela na ekranie monitora na podstawie intensywności trzech składowych: R – czerwonej, G – zielonej, B – niebieskiej.

Rozszerzenie wektorowe, jednostka wektorowa, rozszerzenie SIMD – układ wykonawczy procesora wykonujący jednoczesne operacje na kilku danych tego samego typu.

SciTE for U3 – mobilny edytor programistyczny (zamieszczony w pamięciach flush obsługujących technologię U3), wspierający między innymi składnię asemblera.

Segment – logiczna część programu, zawierająca kod, dane lub stos. Segment jest przechowywany w pamięci operacyjnej pod adresem określonym przez jeden z rejestrów segmentowych. W ramach segmentu adresowanie odbywa się poprzez określenie OFFSET-u (przesunięcia względem początku segmentu).

SIMD (Single Instruction, Multiple Data) – sposób przetwarzania polegający na tym, że pojedyncza instrukcja operuje na kilku danych jednocześnie (operacje wektorowe).

Single – liczba zmiennopozycyjna pojedynczej precyzji, sprecyzowana standardem IEEE 754 (1bit znaku, 8bitów wykładnika, 23 bity mantysy).

SISD (Single Instruction Single Data) – klasyczne, sekwencyjne przetwarzanie danych (jeden program przetwarzający jeden strumień danych).

SPE (Synergistic Processor Elements) – element architektury CBEA, umożliwiający wykonywanie operacji wektorowych. Mikroprocesor CBEA posiada 8 jednostek SPE.

SSE (Streaming SIMD Extensions) – rozszerzenie wektorowe umożliwiające wykonywanie operacji równoległych na liczbach zmiennopozycyjnych pojedynczej precyzji.

SSE2 (Streaming SIMD Extensions 2) – rozszerzenie wektorowe umożliwiające wykonywanie operacji równoległych na liczbach zmiennopozycyjnych podwójnej precyzji.

SSE3 (Streaming SIMD Extensions 3) - rozszerzenie wektorowe umożliwiające wykonywanie instrukcji horyzontalnych, przetwarzania asymetrycznego oraz synchronizacji wątków. Rozszerzenie nie wprowadza nowych typów danych.

SSE4 (Streaming SIMD Extensions 4) – zbiór instrukcji wektorowych mających na celu między innymi przyśpieszenie kompresję wideo, wyznaczanie sumy kontrolej CRC-32, czy zliczanie liczby ustawionych bitów w elemencie. Rozszerzenie nie wprowadza nowych typów danych.

SSE5 (treaming SIMD Extensions 5) – planowane rozszerzenie instrukcji przetwarzania wektorowego, które ma zostać wprowadzone przez firmę AMD. Do charakterystycznych cech rozszerzenia będzie należało wprowadzenie instrukcji wieloargumentowych oraz nowego typu danych (16-bitowej liczby zmiennopozycyjnej).

U1 – kod zapisu liczb całkowitych ze znakiem (kod uzupełnieniowy do 1). Liczby dodatnie mają postać naturalnego kodu binarnego z zerowym najstarszym bitem. Liczby ujemne natomiast mają postać zanegowanej liczby binarnej dodatniej z ustawionym najstarszym bitem.

U2 – kod zapisu liczb całkowitych ze znakiem (kod uzupełnieniowy do 2). Liczby dodatnie mają postać naturalnego kodu binarnego z zerowym najstarszym bitem. Liczby ujemne natomiast są uzupełnieniem do 2^n, gdzie n – liczba bitów reprezentacji liczby.

Układ 8253 – czasomierz systemowy, generujący co ok. 55ms przerwanie zegarowe.

Underflow (niedomiar) – powstaje w sytuacji, gdy wynik operacji zmiennopozycyjnej jest mniejszy od najmniejszej możliwej do zapamiętania wartości.

Wektoryzacja – konwersja programu realizującego przetwarzanie skalarne do programu wykorzystującego operacje wektorowe.

Wskaźnik rozkazu (Instruction pointer) – rejestr zawierający adres rozkazu do wykonania (EIP).

Wyjątek (Exception) – mechanizm obsługi sytuacji wyjątkowych (na przykład błędów powstałych przy wykonywaniu operacji arytmetycznych).

LITERATURA

[1] ABEL P.: Programowanie Asembler IBM PC, RM, Warszawa 2004
[2] Advanced Micro Devices, Software optimization guide for AMD64 processors, Advanced Micro Devices, Inc. 2005
[3] BARON B., PIĄTEK Ł.: Metody numeryczne w C++ Builder, Helion, Gliwice 2004
[4] Hyde R.: Assembler. Sztuka programowania, Helion, Gliwice 2004
[5] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture, Intel Corporation, 2007
[6] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2A: Instruction Set Reference, A-M, Intel Corporation 2007
[7] Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation 2007
[8] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference, N – Z , Intel Corporation 2007
[9] Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1, Intel Corporation 2007
[10] Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture, Intel Corporation, 1999
[11] Intel AVX: New frontiers in performance improvements and energy Efficiency, Intel Corporation, 2008 http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel%20AVX_New%20Frontiers%20in%20Performance%20Improvements%20and%20Energy%20Efficiency_WP.pdf , Pobrano: 12.03.2009
[12] MMX Technology Overwiev, Intel Corporation 1996
[13] IRVINE K.: Asembler dla procesorów Intel. Vademecum profesjonalisty, Helion, Gliwice 2003
[14] KRUK S.: Asembler w koprocesorze, Mikom, Warszawa 2003
[15] KUSIAK K.: Standard IEEE 754, http://www.ipipan.gda.pl/~stefan/Dydaktyka/SemDyplomowe/Raporty/Kusiak/2008-06-19.pdf (pobrano 27.03.2007)
[16] LEVINE J.: Programowanie plików graficznych w C/C++, Translator s. c.
[17] METZGER P.: Anatomia PC wydanie VIII, Helion, Gliwice 2003
[18] PEITGEN H.-O., JÜRGENS H., SAUPE D.: Granice chaosu Fraktale część 2, wydanie drugie, Wydawnictwo Naukowe PWN, Warszawa 2002
[19] PELEG A., WEISER U.: MMX Technology Extension to the Intel architecture, IEEE Computer Society Press, Los Alamitos 1996
[20] POGODA Z.: Arytmetyka komputerów, Instytut Elektroniki Politechniki Śląskiej, Gliwice 2003
[21] STALLINGS W.: Organizacja i architektura systemu komputerowego. Projektowanie systemu a jego wydajność. Wydanie drugie, Wydawnictwo Naukowo-Techniczne WNT, Warszawa 2000, 2003

[22] STANISŁAWSKI W.: Programowanie procesorów rodziny x86. Wydawnictwa Politechniki Opolskiej, Opole 2008
[23] SYSŁO M.: Algorytmy, Wydawnictwa Szkolne i Pedagogiczne, Warszawa, 1997
[24] TANENBAUM A.: Strukturalna organizacja systemów komputerowych. Wydanie V, Helion, Gliwice 2006
[25] The Institute of Electrical and Electronics Engineers, IEEE Standard for Binary Floating-Point Arithmetic, http://fluorescence.fjfi.cvut.cz/~adamek/nm/ieee754.pdf, (pobrano 21.03.2007)
[26] The NASM Development Team, NASM – The Netwide Assembler version 0.98.34, The NASM Development Team 2002 http://www.et.byu.edu/groups/ece425web/stable/labs/nasmmanual.pdf, (pobrano 21.01.2008)
[27] WRÓBEL E.: Asembler. Ćwiczenia praktyczne, Helion, Gliwice 2002