Wrocław 5 czerwca 2015

Kamil Cała  
209954  
Środa 7:15 TN

Sprawozdanie z laboratorium nr 6

*Data laboratorium: 20.05.2015r  
Rok akademicki 2014/2015, Informatyka  
Prowadzący: Mgr. Aleksandra Postawka*

# Opis i cel ćwiczenia

Celem tego ćwiczenia było zapoznanie się z obsługą jednostki MMX procesorów architektury x86 poprzez wykonanie za jej pomocą złożonej operacji na obrazach, oraz porównanie wydajności algorytmów wykonywanych standardowo a na jednostce wektorowej.

Ćwiczenie składało się z zadań:

1. Napisać program (f-kcje) w C, która z bitmapy tworzy np. negatyw
2. Zmierzyć l. cykli procesora (rdtsc)
3. Napisac funkcję w Asemblerze z użyciem rozszerzeń wektorowych i porównać

# Zadanie 1 - program w C

Zgodnie z radą prowadzącej laboratoria, jako podstawy do stworzenia programu użyłem gotowego szablonu “showimage.c” dostępnego na stronie Zakładu Architektury Komputerów PWr  
<http://zak.iiar.pwr.wroc.pl/materials/architektura/laboratorium/MMX/>

Program ten, używając biblioteki SDL oraz SDL\_image otwiera plik obrazu zapisany w większości znanych formatów jak bmp, jpeg czy png i wyświetla go w nowym oknie. Następnie, program przechwytuje naciśnięcia klawiszy, wykonując konkretne czynności.

* q - wyjście z programu
* **f - nałożenie filtra**
* r - ponowne załadowanie obrazu
* s - zapisanie obrazu w pliku nowy.bmp

Najważniejsza dla nas jest funkcja nałożenia filtra. Naciśnięcie przycisku f powoduje wywołanie funkcji

void Filter(unsigned char \* buf, int width,int height,int size,char bpp)

Funkcja ta ma w domyśle modyfikować w jakiś sposób obraz przeprowadzając operację na wskazanym w argumencie obszarze pamięci char\* buf. W załączonym szablonie ciało funkcji było puste (funkcja była jedynie prototypem).

Poprzez implementację tej funkcji wykonałem więc pierwsze zadanie z tego laboratorium. Filtr zamienia więc obraz w negatyw. Dzieje się to w pętli for iterującej po wszystkich bajtach obrazu i wykonującej na każdym operację logicznej negacji.

for(i=0; i<width\*height\*bpp; i++)

{

buf[i] = (~buf[i]);

}

Dzięki użyciu szablonu ze strony wynik operacji można od razu obserwować w otwartym oknie.

# Zadanie 2 - pomiar cykli procesora

W celu pomiaru liczby cykli procesora pomiędzy rozpoczęciem wykonywania funkcji a jej zakończeniem stworzyłem nową funkcję, która zwraca aktualny stan licznika **Time Stamp Counter (TSC).**

static inline unsigned long long tick()

Time Stamp Counter to specjalny 64-bitowy rejestr obecny w procesorach x86 od czasu pierwszego modelu linii Pentium. Mierzy on liczbę cykli które wykonał procesor od momentu resetu. Dzięki użyciu asemblerowej dyrektywy **rdtsc** w bloku inline jesteśmy w stanie wczytać aktualną wartość tego rejestru do naszego programu w C.

Liczba cykli procesora jest więc zapisywana przed i po wykonaniu pętli tworzącej negatyw obrazu, a różnica pomiędzy tymi wartościami to liczba cykli procesora w czasie jej wykonywania. Nie jest to wartość dokładna, jako że program nie ma wyłączności na wykorzystanie procesora. W międzyczasie system operacyjny może przydzielić czas procesora innym procesom itd. Można jednak w ten sposób z grubsza określić rząd wartości który reprezentuje zmierzona liczba cykli. Na moim domowym komputerze były to wartości wachające się pomiędzy **30-70 milionami cykli.**

Repainting after filtered... Done.

Start filtering... Liczba cykli procesora = 44480356

Done.

# Zadnie 3 - użycie jednostki MMX

Ostatnia część laboratorium polegała na podmianie pętli tworzącej negatyw obrazu przez funkcję Assemblerową robiącą to samo, ale przy użyciu jednostki MMX.

Stworzyłem więc funkcję

char\* \_negative(char \*bitmap, unsigned int rozmiar)

Wywołanie funkcji z punktu widzenia Assemblera odbywa się w standardowy sposób opisany w ABI. Pętla wykonująca negowanie obrazu wygląda następująco:

movq %rdi, %r11 #copy bitmap's address to r11

addq %r10, %r11 #add program counter to address

movq (%r11), %mm0 #move 8bits under calculated address to %mm0

pcmpeqw %mm4, %mm4 #fill %mm4 register with 1's

pxor %mm4, %mm0 #effectively - NOT the %mm0 register

movq %mm0, (%r11) #move negated values back to memory

addq $8, %r10 #increment i

Do rejestru %mm0 za każdym obiegiem pętli wczytywane jest 8 bajtów, dlatego licznik pętli za każdym razej zwiększany jest o 8. Pętla jest przerywana kiedy licznik przekroczy całkowity rozmiar bajtów w obrazie obliczony w następujący sposób:

int pictureSize = width\*height\*bpp;

BBP, to liczba bajtów na której zakodowany jest jeden piksel. Każde kolejne 8 bajtów jest ładowane do rejestru %mm0 za pomocą standardowej komendy movq. Rejestr %mm4 jest wypełniany samymi jedynkami przy użyciu polecenia pcmpeqw. Działa on tak, że porównuje dwa podane mu rejestry ze sobą. Jeżeli są różne wypełnia rejestr docelowy samymi zerami. Jeżeli tak jak w naszym wypadku rejestry są identyczne (bo to ten sam rejestr), to rejestr docelowy wypełniany jest jedynkami. Następnie, rejestry %mm0 i %mm4 są ze sobą XOR-owane, co z racji tego że %mm4 jest wypełniony samymi jedynkami powoduje taki sam efekt jaki dałoby przeprowadzenie operacji NOT na %mm0. Przetworzony wynik zapisywany jest spowrotem w to samo miejsce pamięci.

Po zmierzeniu liczby cykli procesora taką samą metodą jak poprzednio, otrzymuję na swoim komputerze wartości rzędu **1-3 milionów cykli.**

# Wnioski

Jak widać, użycie technologii MMX do stworzenia negatywu obrazu spowodowało ponad **10 krotnie szybsze** wykonanie operacji, a więc różnica sięga całego rzędu wartości. Tak duże różnice może nie są widoczne przy edycji pojedynczego obrazu - szczególnie na procesorze który ma taktowanie powyżej 3GHz, ale przy większych obrazach, bardziej zaawansowanych filtrach czy też edycji wideo różnice będą znacznie bardziej widoczne dla użytkownika.

Z drugiej strony napisanie takiego samego algorytmu przy użyciu języków wyższego poziomu jest o wiele prostsze i zajmuje mniej czasu programisty. W praktyce więc do tak niskiego poziomu schodzi się tylko w wypadku aplikacji w których wydajność jest absolutnie krytyczna, takich jak kodeki video, oprogramowanie systemów pokładowych aut i samolotów, niektóre gry wideo itd. Warto jednak zauważyć że w miażdżącej ilości wypadków “biznesowe” spojrzenie na tą kwestię pokazuje że czas programistów jest o wiele bardziej kosztowny niż czas pracy komputera, przez co rozwój oprogramowania sukcesywnie przenosi się na coraz wyższe poziomy abstrakcji.