ZAKŁAD ARCHITEKTURY KOMPUTERÓW

ORGANIZACJA I ARCHITEKTURA KOMPUTERÓW

LABORATORIUM

Sprawozdanie z laboratorium nr 4

Autor
PIOTR TOCICKI

Prowadzący
MGR INŻ. TOMASZ SERAFIN

29 maja 2020

Spis treści

LABORATORIUM 3

1 Sprawozdanie

ZADANIA

CZĘŚĆ 1 SIMD

1. Należy stworzyć program wykonujący działania na wektorach **128 bitowych** (+, -, *, /).
2. Liczby umieszczone w wektorach mogą być zmiennoprzecinkowe lub typu całkowitego.
3. Należy zmierzyć czas wykonania obliczeń dla **2048**, **4096** i **8192** liczb (dla wszystkich działań z osobna) - pomiar powtórzyć 10 razy i obliczyć średni czas.
4. Wyniki zanotować w postaci wykresów:
   - zmienność średniego czasu w zależności od liczby liczb
   - zmienność średniego czasu w zależności od typu działania dla 8192 liczb
5. Programy w języku C połączonym z asemblerem.
6. Wynikiem działania każdego z programów ma być plik tekstowy o następującej treści:

CZĘŚĆ 2 SISD

7. Należy napisać program analogiczny do tego z części 1 bez wykorzystania wektorów.
8. Wyniki pomiarów nałożyć na wykresy wykonane w części 1. Zysk z zastosowania mechanizmów SIMD wyrazić w procentach.
9. Na podstawie stworzonych wykresów, należy zanotować wnioski i przedstawić je w sprawozdaniu.

1.1 Przebieg pracy nad zadaniami z części I (SIMD)

1.1.1 Utworzenie plików z rozszerzeniem .s

Pierwszym etapem było utworzenie plików w języku Assembler - dla każdego wykonywanego działania - dodawanie, odejmowanie, mnożenie, dzielenie, które będziemy używać w programie z rozszerzeniem .c do wykonywania zadanych operacji. Utworzony został również plik `timestamp.s`, który będzie służył do pomiaru czasu.

1.1.2 Utworzenie pliku w języku C

Utworzony również został plik w języku C, w którym będzie się odbywało wywoływanie funkcji Assembler'owych.

1.1.3 Utworzenie makefile

Następnie utworzony został plik makefile, w którym będzie się odbywać kompilacja oraz czyszczenie (usuwanie pliku uruchomieniowego oraz plików z wynikami pomiarów).

1.1.4 Realizacja algorytmu - dodawanie

Zadania z laboratorium zostaną zrealizowane w taki sposób, że do każdej operacji będą napisane po dwie (dla SIMD i SISD) assemblerowe funkcje (jako osobne pliki), które potem będą wywoływane w programie w języku C zgodnie z konwencją wywołań ABI.

Aby połączyć kod Assemblera z programem w języku C należy wykonać pewne określone działania. Najpierw musimy przygotować ramkę stosu. Zaczynamy od umieszczenia na stosie poprzedniej wartości base pointer'a i przenosimy do rejestru `ebp` aktualną wartość wskaźnika stosu.

```
.global dodawanie_sse

.text
dodawanie_sse:
push %ebp
mov %esp, %ebp
```

Teraz należy utworzyć prototyp funkcji w programie w języku C, w którym będziemy wywoływać funkcję z języka Assembler.

```
void dodawanie_sse(int* tab1, int* tab2, int* wynik, int len);
```

Następnie napiszmy kod w języku Assembler, odpowiadający za pobranie parametrów funkcji i zapisanie ich do rejestrów:

```
mov 8(%ebp), %edi
mov 12(%ebp), %esi
mov 16(%ebp), %edx
mov 20(%ebp), %eax
```

Ponieważ wykorzystujemy rejestry, które są zachowywane przez wywołanego (chodzi o rejestry `edi` i `esi`), musimy najpierw umieścić je na stosie. W późniejszej fazie - po nadpisaniu owych rejestrów - trzeba będzie je spowrotem przywrócić rejestr.

```
push %edi
push %esi

mov 8(%ebp), %edi # tab1
mov 12(%ebp), %esi # tab2
mov 16(%ebp), %edx # wynik
mov 20(%ebp), %eax # rozmiar
```

Tworzymy pętlę, w której będziemy iterować po kolejnych liczbach. Rejestr `ecx` będzie pełnił rolę rejestru sterującego pętlą.

```
xor %ecx, %ecx

petla:

cmp %eax, %ecx
jl petla
```

Do realizacji algorytmu (część 1.) będziemy wykorzystywać jednostkę wektorową SSE. Pozwala ona na wykonywanie działań zmiennoprzecinkowych na 4-elementowych wektorach liczb pojedynczej precyzji.

Jednostka SSE wykorzystuje swoje rejestry. Jest ich osiem, są to rejestry 128-bitowe i mają odpowiednio oznaczenia od `xmm0` do `xmm7`.

Następnie za pomocą rozkazu movdqu przenosimy liczby do rejestru xmm0 i xmm1. Będą w nich teraz po 4 liczby, zapisane jedna za drugą. Możemy teraz wykonać dodawanie za pomocą rozkazu paddd, która spowoduje to, że 4 liczby zostaną dodane jednocześnie.

```
petla:
movdqu (%edi, %ecx, 4), %xmm0
movdqu (%esi, %ecx, 4), %xmm1

paddd %xmm0, %xmm1

cmp %eax, %ecx
jl petla
```

Musimy jeszcze przenieść wynik do pamięci i dokonać inkrementacji rejestru ecx odpowiedzialengo w naszym programie za sterowanie pętlą. Pełna pętla wygląda następująco:

```
petla:
movdqu (%edi, %ecx, 4), %xmm0
movdqu (%esi, %ecx, 4), %xmm1

paddd %xmm0, %xmm1

movdqu %xmm1, (%edx, %ecx, 4)

add $4, %ecx
cmp %eax, %ecx
jl petla
```

1.1.5 Realizacja algorytmu - odejmowanie

Postępujemy identycznie jak w przypadku algorytmu dodawania, z jedną oczywistą zmianą, zamiast dodawania - paddd, wykonujemy odejmowanie, czyli używamy rozkazu psubd.

1.1.6 Realizacja algorytmu - mnożenie

Postępujemy identycznie jak w przypadku algorytmu dodawania, z jedną oczywistą zmianą, zamiast dodawania - paddd, wykonujemy mnożenie, czyli używamy rozkazu pmulld.

1.1.7 Realizacja algorytmu - dzielenie

Dzielenie musimy wykonać nieco inaczej, ponieważ nie dysponujemy rozkazem do wektorowego dzielenia liczb całkowitych. Dlatego za pomocą rozkazu `cvtdq2ps` dokonujemy konwersji z `int` na `float` oraz dzielenie wykonujemy zmiennoprzecinkowo - rozkaz `divps`. Zmienione jest także przekazywanie wyniku do pamięci, muimy to teraz zrobić za pomocą rozkazu `movups`. Oto zawartość pętli:

```
petla:
movdqu (%edi, %ecx, 4), %xmm0
movdqu (%esi, %ecx, 4), %xmm1

cvtdq2ps %xmm0, %xmm2
cvtdq2ps %xmm1, %xmm3

divps %xmm2, %xmm3

movups %xmm3, (%edx, %ecx, 4)

add $4, %ecx

cmp %eax, %ecx
jl petla
```

1.1.8 Pomiar czasu

Do obliczania czasu wykorzystywany jest **timestamp.s**, który za pomocą rozkazu `rdtsc` podaje bieżącą wartość liczby cykli procesora i zapisuje do rejestrów EDX:EAX:

```
.global timestamp

.text
timestamp:
push %ebx

xor %eax, %eax
cpuid
rdtsc
```

```
pop %ebx
ret
```

1.1.9 Wywołanie funkcji w języku C

Należało poprawnie zdefiniować prototypy funkcji:

```
void dodawanie_sse(int* tab1, int* tab2, int* wynik, int len);
void odejmowanie_sse(int* tab1, int* tab2, int* wynik, int len);
void mnozenie_sse(int* tab1, int* tab2, int* wynik, int len);
void dzielenie_sse(int* tab1, int* tab2, float* wynik, int len);

unsigned long long timestamp();
```

Przykładowe wywołanie funkcji wraz z pomiarem czasu:

```
start = timestamp();
dodawanie_sse(tab1, tab2, wynik, rozmiar);
stop = timestamp();
cycles_sse = stop - start;
time_sse = (double)cycles_sse/(double)cycles_sec;
```

Ostateczny wynik czasowy - `time_sse` - jest dzielony przez `cycles_sec`, ponieważ chcemy uzyskać wynik w sekundach. Zmienna `cycles_sec` przechowuje ile cykli wykonuje procesor w jednej sekundzie (również jest to mierzone za pomocą timestamp).

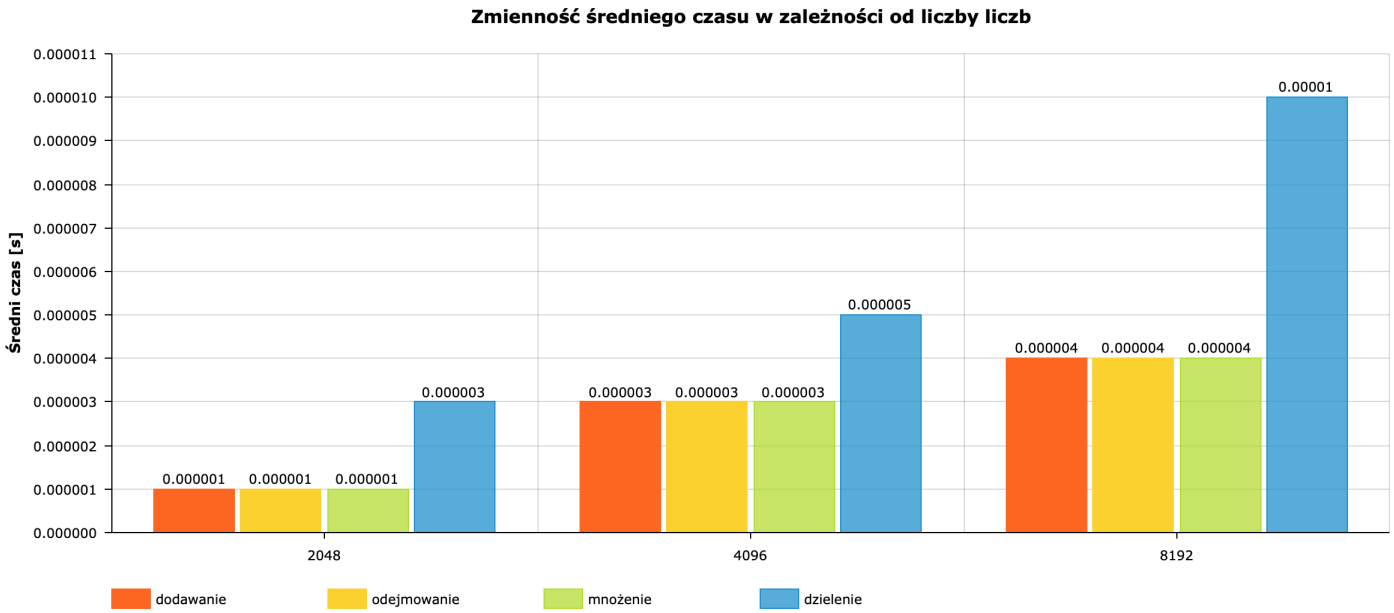
Bliźniaczo wywoływane są funkcje przeznaczone do odejmowania, mnożenia i dzielenia. Wyniki są zapisywane do pliku wyniki_simd.txt dla rozmiarów 2048, 4096, 8192.

1.1.10 Kompilacja i uruchomienie programu

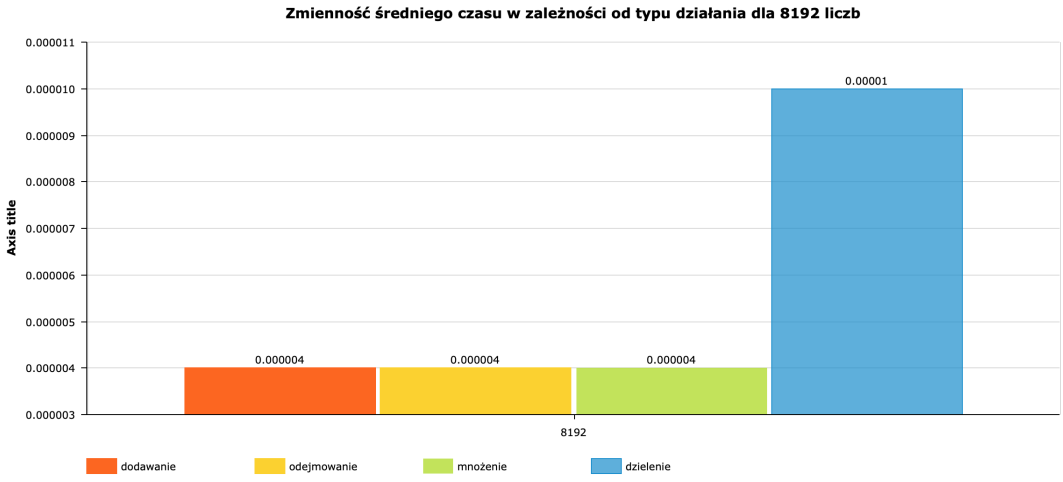
Program kompilujemy za pomocą polecenia `make`. Natomiast za uruchomienie skompilowanego programu odpowiada wpisanie w konsoli:

```
./lab4
```

1.1.11 Wykresy



Rysunek 1: Wykres dla SIMD



Rysunek 2: Wykres 2 dla SIMD

1.2 Przebieg prac nad zadaniami z części II (SISD)

1.2.1 Utworzenie plików z rozszerzeniem .s

Ponownie pierwszym etapem jest utworzenie plików w języku Assembler - dla każdego wykonywanego działania - dodawanie, odejmowanie, mnożenie, dzielenie, które będziemy używać w programie z rozszerzeniem .c do wykonywania zadanych operacji.

1.2.2 Program w języku C

Funkcje będą wywoływane we wcześniej utworzonym programie w języku C.

1.2.3 Zaktualizowanie makefile

Następnie należy pamiętać, aby do utworzonego wcześniej Makefile'a, dodać nowo utworzone pliki.

1.2.4 Realizacja algorytmu - dodawanie

Realizacja operacji dodawania, w porównaniu do tej z części I, została zmodyfikowana. Nie używamy już rejestrów `xmm`. Pojedyncza liczba z pamięci, zapisywana jest do nieużywanego wcześniej rejestru `ebx`, następnie liczba za pomocą rozkazu `add` dodajemy liczbę z drugiego wektora do `ebx` oraz przenosimy wynik z rejestru `ebx` do pamięci i dokonujemy inkrementacji o 1 (wcześniej było o 4, bo mieliśmy 4 liczby). Pętla wygląda następująco:

```
petla2:
mov (%edi, %ecx, 4), %ebx
add (%esi, %ecx, 4), %ebx
mov %ebx, (%edx, %ecx, 4)

add $1, %ecx

cmp %eax, %ecx
jl petla2
```

1.2.5 Realizacja algorytmu - odejmowanie

Postępujemy identycznie jak w przypadku algorytmu dodawania dla SISD, z jedną oczywistą zmianą, zamiast dodawania - `add`, wykonujemy odejmowanie, czyli używamy rozkazu `sub`.

1.2.6 Realizacja algorytmu - dzielenie

W przypadku dzielenia pętla wygląda następująco:

```
petla2:

fildl (%esi, %ecx, 4)
fidivl (%edi, %ecx, 4)

fstps (%edx, %ecx, 4)

add $1, %ecx

cmp %eax, %ecx
jl petla2
```

Za pomocą rozkazu `fildl` ładujemy na stos liczbę z pamięci, a następnie wykonujemu dzielenie za pomocą `fidivl`. Kolejnym krokiem jest skopiowanie szczytu stosu do miejsca w pamięci, gdzie przechowujemy wynik.

1.2.7 Realizacja algorytmu - mnożenie

Postępujemy identycznie jak w przypadku algorytmu dzielenia dla SISD, z jedną oczywistą zmianą, zamiast dzielenia - `fidiv`, wykonujemy mnożenie, czyli używamy rozkazu `fimul`.

1.2.8 Wywołanie funkcji w języku C

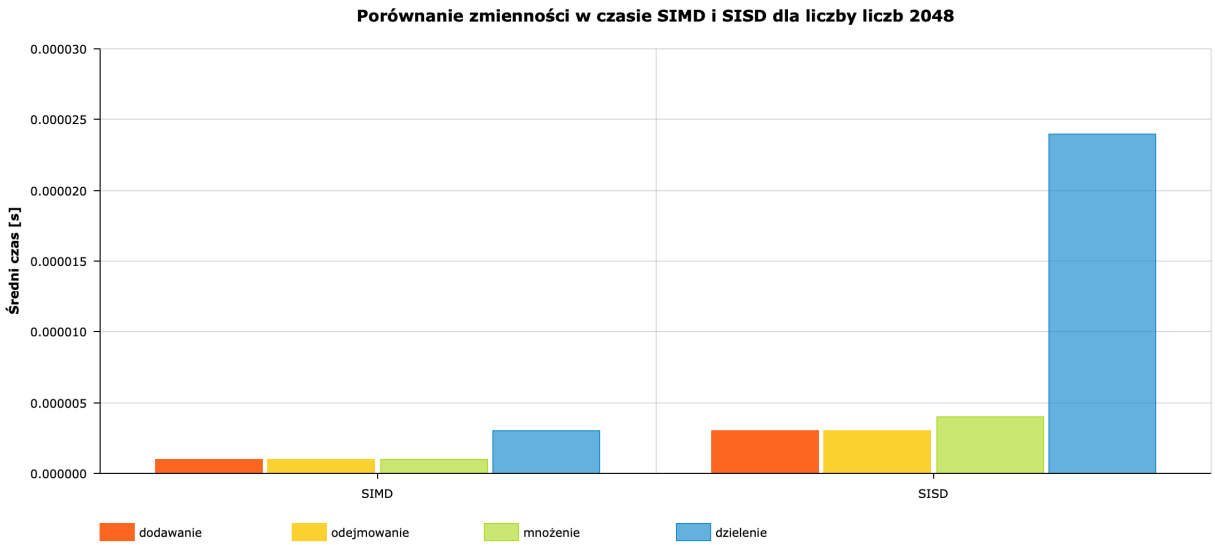
Należało poprawnie zdefiniować prototypy funkcji:

```
void dodawanie_sisd(int* tab1, int* tab2, int* wynik, int len);
void odejmowanie_sisd(int* tab1, int* tab2, int* wynik, int len);
void mnozenie_sisd(int* tab1, int* tab2, float* wynik, int len);
void dzielenie_sisd(int* tab1, int* tab2, float* wynik, int len);
```

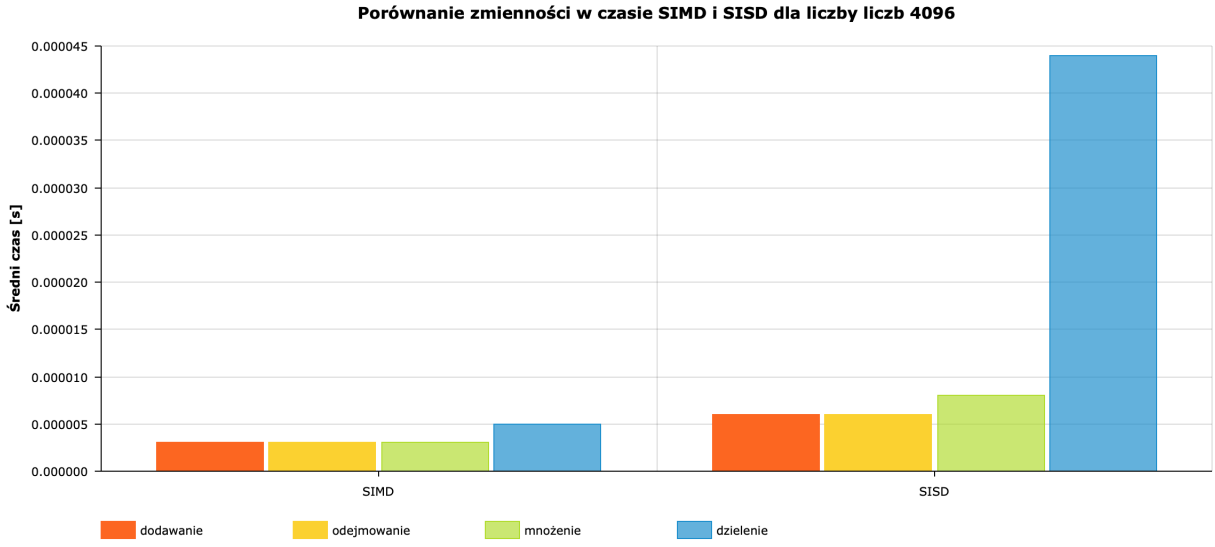
Przykładowe wywołanie funkcji wraz z pomiarem czasu:

```
start = timestamp();
odejmowanie_sisd(tab1, tab2, wynik, rozmiar);
stop = timestamp();
cycles_sse = stop - start;
time_sisd = (double)cycles_sse/(double)cycles_sec;
```

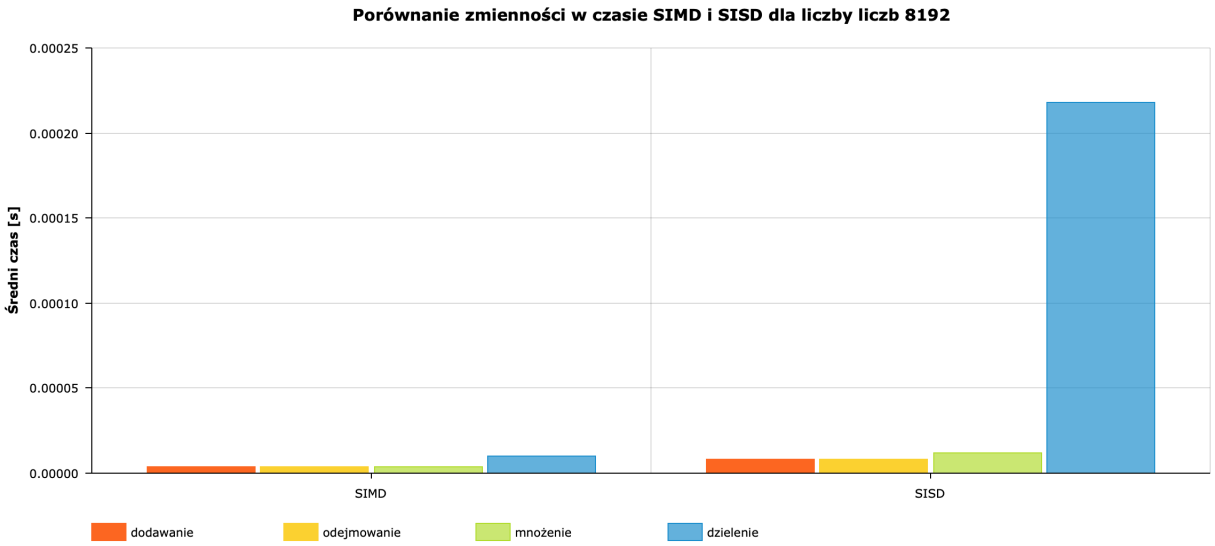
1.2.9 Wykresy



Rysunek 3: Wykres porównanie SIMD i SISD



Rysunek 4: Wykres 2 porównanie SIMD i SISD



Rysunek 5: Wykres 3 porównanie SIMD i SISD

1.3 Zysk SIMD względem SISD

1.3.1 Liczba liczb - 2048

	Operacja	SIMD	SISD	Zysk
2048	+	0.000001	0.000003	300%
	-	0.000001	0.000003	300%
	*	0.000001	0.000004	400%
	/	0.000003	0.000024	800%

1.3.2 Liczba liczb - 4096

	Operacja	SIMD	SISD	Zysk
4096	+	0.000003	0.000006	200%
	-	0.000003	0.000006	200%
	*	0.000003	0.000008	266%
	/	0.000005	0.000044	880%

1.3.3 Liczba liczb - 8192

	Operacja	SIMD	SISD	Zysk
8192	+	0.000004	0.000008	200%
	-	0.000004	0.000008	200%
	*	0.000004	0.000012	300%
	/	0.000010	0.000218	2180%

1.4 Podsumowanie i wnioski

Na wykresach widać wyraźną **korzystną** przewagę średniego czasu wykonywania SIMD względem SISD. Dzięki SIMD czas wykonywania działań jest zdecydowanie szybszy. Jest to oczywiście spowodowane faktem, że SIMD jednocześnie wykonuje operacje na 4 parach liczb, a SISD na tylko jednej.

Wszystkie programy i funkcje działają poprawnie i zwracają zarówno poprawne wyniki (zostało to również sprawdzone), jak i wiarygodne pomiary. Podczas wykonywania zadań ponownie najbardziej pomocna była dokumentacja Intela.