AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody Obliczeniowe w Nauce i Technice

Arytmetyka Komputerowa

Zadanie 1

Maciej Kmąk
Informatyka WI AGH, II rok

1 Treść Zadania

Wyznaczyć doświadczalnie parametry reprezentacji liczb zmiennoprzecinkowych (`float, double, long double`) i porównać uzyskane wartości dla różnych architektur, systemów operacyjnych i kompilatorów. Sprawdzić, czy reprezentacje są zgodne ze standardem IEEE.

Parametry do wyznaczenia:

- liczba bajtów używana do przechowywania zmiennej danego typu,
- liczba bitów na mantysę,
- liczba bitów na cechę (wliczając znak),
- "maszynowe epsilon najmniejsza liczba, dla której 1.0+e > 1.0,
- występowanie i sposób reprezentacji wartości specjalnych (±0, ±Inf, NaN).

2 Dane techniczne

Obliczenia przeprowadzono na różnych platformach sprzętowych i programowych, obejmujących różne architektury, systemy operacyjne oraz kompilatory. W poniższej sekcji przedstawiono specyfikację wykorzystanych środowisk obliczeniowych.

2.1 Komputer Osobisty

Część eksperymentów obliczeniowych przeprowadzono na tym samym komputerze, jednak z wykorzystaniem dwóch różnych systemów operacyjnych:

- Windows 11 Pro 24H2
- Fedora Linux 41 (Workstation Edition)

Specyfikacja sprzętowa komputera pozostawała niezmienna i obejmowała:

- Architektura: x86_64
- Procesor: 12th Gen Intel(R) Core(TM) i5-1235U

Na komputerze testowano różne wersje kompilatorów. W przypadku systemu Fedora wykorzystano dwa różne kompilatory: GCC 8.5.0 oraz Clang 19.1.7, co pozwoliło na porównanie wpływu kompilatora na wyniki obliczeń:

- Windows 11 Pro 24H2
  - GCC: gcc (Rev3, Built by MSYS2 project) 13.2.0
- Fedora Linux 41 (Workstation Edition)
  - GCC: gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-23)
  - Clang: clang version 19.1.7 (Fedora 19.1.7-3.fc41)

Pozwoliło to na analizę wpływu systemu operacyjnego oraz kompilatora na wyniki obliczeń przy zachowaniu tej samej architektury (a nawet identycznego procesora).

2.2 Bastion

Dodatkowe testy przeprowadzono na zdalnym serwerze obliczeniowym, dostępnym poprzez połączenie SSH. Serwer ten wykorzystuje infrastrukturę Wydziału Informatyki AGH. Specyfikacja sprzętowa serwera zdalnego:

- System Operacyjny: AlmaLinux 8.10 (Cerulean Leopard)
- Architektura: x86_64
- Procesor: Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz

Na tej platformie użyto dostępnego kompilatora GCC:

- GCC: gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-23)

Warto zaznaczyć, że AlmaLinux jest dystrybucją wywodzącą się z systemu Fedora, co zapewnia pewien stopień kompatybilności z wcześniejszymi testami przeprowadzonymi na komputerze osobistym.

2.3 Telefon

Dodatkowym środowiskiem testowym był smartfon Xiaomi 11 Lite 5G NE, wyposażony w układ o odmiennej architekturze niż pozostałe platformy. Było to jedyne testowane urządzenie oparte na architekturze ARM, co umożliwiło analizę wydajności kodu w odmiennym środowisku sprzętowym. Specyfikacja urządzenia:

- System Operacyjny: Android 14
- Architektura: ARM
- Procesor: Qualcomm Snapdragon 778G 5G

Do kompilacji na urządzeniu mobilnym wykorzystano kompilator:

- Clang

Testy przeprowadzone na tym urządzeniu pozwoliły na analizę kodu na architekturze ARM oraz ocenę potencjalnych różnic w zachowaniu programu w porównaniu do platform x86_64.

3 Przebieg Doświadczenia

W celu przeprowadzenia niniejszego doświadczenia opracowano i uruchomiono program w języku C, którego głównym zadaniem jest doświadczalne wyznaczenie podstawowych parametrów liczb zmiennoprzecinkowych (float, double, long double).
Program został następnie uruchomiony na omówionych wyżej urządzeniach testowych, co pozwoliło porównać otrzymane wyniki dla różnych architektur sprzętowych i środowisk programistycznych. W dalszej części rozdziału przedstawiono sposób działania programu.

3.1 Liczba bajtów używana do przechowywania zmiennej danego typu

Tworzymy tablicę dwuelementową (np. `float f[2];`), a następnie dokonujemy rzutowania wskaźników na typ `char*`. Różnica `(char*)(&f[1]) - (char*)(&f[0])` oznacza w bajtach, o ile przesunięto adres tablicy przy przejściu od elementu `f[0]` do `f[1]`. Dzięki temu można uzyskać liczbę bajtów zajmowanych przez jeden element tablicy danego typu.

Kod:

```
void print_size_of_format() {

    float f[2];
    double d[2];
    long double ld[2];

    printf("Rozmiar float %td bajtów\n", (char*)(&f[1]) - (char*)(&f[0]));
    printf("Rozmiar double %td bajtów\n", (char*)(&d[1]) - (char*)(&d[0]));
    printf("Rozmiar long double %td bajtów\n", (char*)(&ld[1]) - (char*)(&ld[0])
    );
}
```

3.2 Liczba bitów na mantysę

W celu wyznaczenia liczby bitów mantysy (części ułamkowej) zaczynamy od wartości `x = 1.0` (odpowiednio `1.0f` dla `float`, `1.0` dla `double`, `1.0L` dla `long double`) i w pętli dzielimy `x` przez 2 (`x /= 2.0`). Sprawdzamy, kiedy wyrażenie `1.0 + x` przestaje być większe od 1.0.
Jeżeli w pewnym momencie `1.0 + x == 1.0`, oznacza to, że `x` jest już tak małe, iż jego dodanie nie powoduje zmiany wartości w reprezentacji zmiennoprzecinkowej – mówimy wówczas, że `x` "znika" w mantysie. Liczbę dokonanych podziałów traktujemy jako przybliżenie liczby bitów mantysy.

Kod:

```
int count_mantissa_bits_float() {
    float x = 1.0f;
    int bits = 0;

    while ((1.0f + x) != 1.0f) {
        x /= 2.0f;
        bits++;
    }
    return bits;
}

int count_mantissa_bits_double() {
    double x = 1.0;
    int bits = 0;
```

```
15
16      while ((1.0 + x) != 1.0) {
17          x /= 2.0;
18          bits++;
19      }
20      return bits;
21  }
22
23  int count_mantissa_bits_long_double() {
24      long double x = 1.0L;
25      int bits = 0;
26
27      while ((1.0L + x) != 1.0L) {
28          x /= 2.0L;
29          bits++;
30      }
31      return bits;
32  }
33
34  void print_mantissa_bits() {
35      printf("Liczba bitow mantysy (float): %d\n", count_mantissa_bits_float());
36      printf("Liczba bitow mantysy (double): %d\n", count_mantissa_bits_double());
37      printf("Liczba bitow mantysy (long double): %d\n",
        count_mantissa_bits_long_double());
38  }
```

3.3 Liczba bitów na cechę (eksponent)

Aby doświadczalnie wyznaczyć liczbę bitów w polu wykładnika (cechy), rozpoczynamy od `x = 1.0` i w pętli mnożymy `x *= 2.0`, dopóki nie osiągniemy stanu *nieskończoności* (`inf`). Zliczamy, ile razy wykonano mnożenie. Następnie liczba bitów cechy jest przybliżona przez wyrażenie

$$\lfloor \log_2(\text{liczba\_mnożeń}) \rfloor + 1.$$

Kod:

```
1   int count_exponent_bits_float() {
2       float x = 1.0f;
3       int exp = 0;
4       int max_exp = 0;
5
6       while (!isinf(x)) {
7           x *= 2.0f;
8           exp++;
9           if (exp > max_exp) max_exp = exp;
10      }
11
12      return (int)(log2(max_exp) + 1);
13  }
14
15  int count_exponent_bits_double() {
16      double x = 1.0;
17      int exp = 0;
18      int max_exp = 0;
19
20      while (!isinf(x)) {
21          x *= 2.0;
22          exp++;
23          if (exp > max_exp) max_exp = exp;
```

```
    }

    return (int)(log2(max_exp) + 1);
}

int count_exponent_bits_long_double() {
    long double x = 1.0L;
    int exp = 0;
    int max_exp = 0;

    while (!isinf(x)) {
        x *= 2.0L;
        exp++;
        if (exp > max_exp) max_exp = exp;
    }

    return (int)(log2(max_exp) + 1);
}

void print_exponent_bits() {
    printf("Liczba bitow cechy (float): %d\n", count_exponent_bits_float());
    printf("Liczba bitow cechy (double): %d\n", count_exponent_bits_double());
    printf("Liczba bitow cechy (long double): %d\n",
   count_exponent_bits_long_double());
}
```

3.4 Maszynowe epsilon

Maszynowe epsilon to najmniejsza liczba (względem 1.0), którą jesteśmy w stanie dodać do jedynki tak, by otrzymać wartość od niej różną. W pętli startujemy od `eps = 1.0` i sprawdzamy, czy

$$1.0 + \frac{\texttt{eps}}{2} > 1.0.$$

Jeśli równość jest wciąż większa, dzielimy `eps` przez 2. W momencie gdy dodanie `eps`/2 nie zmienia już wartości `1.0`, znaleziono przybliżone maszynowe epsilon.

Kod:

```
float compute_float_epsilon() {
    float eps = 1.0f;
    while (1.0f + eps/2.0f > 1.0f) {
        eps /= 2.0f;
    }
    return eps;
}

double compute_double_epsilon() {
    double eps = 1.0;
    while (1.0 + eps/2.0 > 1.0) {
        eps /= 2.0;
    }
    return eps;
}

long double compute_long_double_epsilon(void) {
    long double eps = 1.0L;
    while (1.0L + eps/2.0L > 1.0L) {
        eps /= 2.0L;
    }
```

```
    return eps;
}

void print_eps() {
    printf("Maszynowe epsilon dla float: %g\n", compute_float_epsilon());
    printf("Maszynowe epsilon dla double: %g\n", compute_double_epsilon());
    long double ld_eps = compute_long_double_epsilon();
    printf("Maszynowe epsilon dla long double: %Lg\n", ld_eps);
}
```

3.5 Wartości specjalne: ± 0, $\pm\infty$, `NaN`

Standard IEEE 754 definiuje specjalne kody bitowe reprezentujące zera z różnymi znakami, nieskończoności (dodatnią i ujemną) oraz wartości `NaN` (Not a Number). W języku C możemy je otrzymać poprzez:

- dzielenie liczby dodatniej/ujemnej przez zero, aby otrzymać `+Inf` / `-Inf`,
- obliczenie pierwiastka z liczby ujemnej (`sqrtf(-1.0f)`) w celu wytworzenia `NaN`.

Funkcja `isnan(x)` zwraca `1`, jeśli `x` to `NaN`; w przeciwnym przypadku `0`.

Kod:

```
void check_special_values_float() {
    float zero_p = 0.0f;
    float zero_n = -0.0f;
    float inf_p = 1.0f / zero_p;
    float inf_n = -1.0f / zero_p;
    float nan_v = sqrtf(-1.0f); // NaN

    printf("FLOAT:\n");
    printf("+0: %.1f | -0: %.1f\n", zero_p, zero_n);
    printf("+Inf: %f | -Inf: %f\n", inf_p, inf_n);
    printf("NaN: %f (czy NaN? %d)\n", nan_v, isnan(nan_v));
}

void check_special_values_double() {
    double zero_p = 0.0;
    double zero_n = -0.0;
    double inf_p = 1.0 / zero_p;
    double inf_n = -1.0 / zero_p;
    double nan_v = sqrt(-1.0); // NaN

    printf("\nDOUBLE:\n");
    printf("+0: %.1f | -0: %.1f\n", zero_p, zero_n);
    printf("+Inf: %f | -Inf: %f\n", inf_p, inf_n);
    printf("NaN: %f (czy NaN? %d)\n", nan_v, isnan(nan_v));
}

void check_special_values_long_double() {
    long double zero_p = 0.0L;
    long double zero_n = -0.0L;
    long double inf_p = 1.0L / zero_p;
    long double inf_n = -1.0L / zero_p;
    long double nan_v = sqrtl(-1.0L); // NaN

    printf("\nLONG DOUBLE:\n");
    printf("+0: %.1Lf | -0: %.1Lf\n", zero_p, zero_n);
    printf("+Inf: %Lf | -Inf: %Lf\n", inf_p, inf_n);
```

```
    printf("NaN: %Lf (czy NaN? %d)\n", nan_v, isnan(nan_v));
}

void print_special_values() {
    check_special_values_float();
    check_special_values_double();
    check_special_values_long_double();
}
```

3.6 Główna funkcja `main`

Funkcja `main` wywołuje wszystkie zaprezentowane wyżej procedury, prezentując:

1. Rozmiar zmiennych (`sizeof`) w celu kontrolnym,
2. Opisane powyżej podpunkty doświadczenia

Kod:

```
int main(void) {

    printf("Rozmiar z uzyciem sizeof:\n");
    printf("float: %llu\n", sizeof(float));
    printf("double: %llu\n", sizeof(double));
    printf("long double: %llu\n", sizeof(long double));

    printf("\nRozmiar wyznaczony doswiadczalnie:\n");
    print_size_of_format();
    printf("---------------------------------\n");

    printf("\nRozmiar mantysy wyznaczony doswiadczalnie:\n");
    print_mantissa_bits();
    printf("---------------------------------\n");

    printf("\nRozmiar cechy wyznaczony doswiadczalnie:\n");
    print_exponent_bits();
    printf("---------------------------------\n");

    printf("\nMaszynowe epsilon wyznaczone doswiadczalnie:\n");
    print_eps();
    printf("---------------------------------\n");

    print_special_values();

    return 0;
}
```

4 Opracowanie Danych

W niniejszym rozdziale przedstawiono i omówiono wyniki testów przeprowadzonych na różnych konfiguracjach sprzętowo-programowych. Dla przejrzystości, wyniki zostały podzielone na trzy grupy:

- **Systemy Operacyjne** – ten sam laptop, dwa różne systemy (Sekcja 2.1 Komputer Osobisty Systemy), a także Serwer (Sekcja 2.2)
- **Kompilatory** – ten sam system (Sekcja 2.1 Komputer Osobisty Kompilatory), dwa różne kompilatory (np. GCC i Clang),
- **Architektury** – różne urządzenia (Sekcja 2.1 i 2.3).

We wszystkich przeprowadzonych testach, niezależnie od platformy sprzętowo-programowej operator `sizeof` wskazywał rozmiary: `float` = 4 bajty, `double` = 8 bajtów oraz `long double` = 16 bajtów.

4.1 Systemy Operacyjne

Testy wykonano na tym samym laptopie, uruchamiając dwie różne platformy systemowe (Windows 11 i Fedora Linux). Dodatkowe testy przeprowadzono również na serwerze (AlmaLinux). Do kompilacji użyto dostępnej wersji GCC. Tabele: 1. i 2. przedstawiają przykładowe porównanie otrzymanych wyników.

Tabela 1: Porównanie wyników dla Windows i Linux na tym samym laptopie oraz na serwerze Bastion

Parametr	Windows	Linux	Bastion
Rozmiar `float` (B)	4	4	4
Rozmiar `double` (B)	8	8	8
Rozmiar `long double` (B)	16	16	16
Bity mantysy (`float`)	24	24	24
Bity mantysy (`double`)	53	53	53
Bity mantysy (`long double`)	64	64	64
Bity cechy (`float`)	8	8	8
Bity cechy (`double`)	11	11	11
Bity cechy (`long double`)	15	15	15
Epsilon (`float`)	1.19209e-07	1.19209e-07	1.19209e-07
Epsilon (`double`)	2.22045e-16	2.22045e-16	2.22045e-16
Epsilon (`long double`)	XXXXe-312	1.0842e-19	1.0842e-19

Tabela 2: Występowanie wartości specjalnych w zależności od formatu (`float`, `double`, `long double`) i systemu

Typ	Wartość	Windows	Linux	Bastion
`float`	`+0`	0.0	0.0	0.0
	`-0`	-0.0	-0.0	-0.0
	`+Inf`	inf	inf	inf
	`-Inf`	-inf	-inf	-inf
	`NaN`	-nan(ind) (isnan=1)	-nan (isnan=1)	-nan (isnan=1)
`double`	`+0`	0.0	0.0	0.0
	`-0`	-0.0	-0.0	-0.0
	`+Inf`	inf	inf	inf
	`-Inf`	-inf	-inf	-inf
	`NaN`	-nan(ind) (isnan=1)	-nan (isnan=1)	-nan (isnan=1)
`long double`	`+0`	0.0	0.0	0.0
	`-0`	0.0	-0.0	-0.0
	`+Inf`	0.000000	inf	inf
	`-Inf`	0.000000	-inf	-inf
	`NaN`	0.000000 (isnan=0)	-nan (isnan=1)	-nan (isnan=1)

Przeprowadzone testy na różnych systemach operacyjnych – Windows, Fedora Linux oraz AlmaLinux (serwer Bastion) – wykazały istotne różnice w sposobie reprezentacji liczb zmiennoprzecinkowych w czasie wykonywania programu. Wyniki pokazują, że teoretyczne parametry typów float, double oraz long double powinny być identyczne na każdej platformie, ale w praktyce precyzja obliczeń oraz wartości specjalne różnią się w zależności od systemu.

Najbardziej zauważalnym problemem w systemie Windows była niestabilna reprezentacja wartości epsilon dla long double, która została przedstawiona w tabeli 1. jako XXXXe-312. Wartość ta zmieniała się przy każdym uruchomieniu programu, co sugeruje problem z deterministycznością reprezentacji liczb. Może to wynikać z ograniczeń kompilatora GCC w środowisku Microsoft Windows oraz Visual Studio Code (VSC).

Dodatkowo, system Windows wykazał nieoczekiwane wartości dla long double, gdzie wartości +Inf, -Inf oraz NaN zostały błędnie przedstawione jako 0.000000. Jest to sprzeczne ze standardem IEEE 754. W systemach Linux oraz AlmaLinux wartości te były poprawnie reprezentowane jako inf, -inf oraz NaN, co wskazuje na większą stabilność obliczeń w tych środowiskach.

Analiza wykazała, że Windows oraz jego środowiska programistyczne mogą powodować błędy w precyzyjnej reprezentacji liczb zmiennoprzecinkowych, szczególnie dla rozszerzonych typów danych takich jak long double. Może to wynikać z ograniczeń bibliotek standardowych.

4.2 Kompilatory

Poniżej znajdują się uzupełnione tabele 3. i 4. przedstawiające wyniki dla dwóch kompilatorów (GCC i Clang) uruchamianych na tym samym systemie (Fedora Linux 41). W pierwszej tabeli zaprezentowano podstawowe parametry (rozmiary typów, liczba bitów mantysy, liczba bitów cechy, epsilon), a w drugiej – występowanie wartości specjalnych w poszczególnych formatach.

Tabela 3: Porównanie wyników dla GCC i Clang na tym samym systemie (Fedora Linux 41)

Parametr	GCC	Clang
Rozmiar `float` (B)	4	4
Rozmiar `double` (B)	8	8
Rozmiar `long double` (B)	16	16
Bity mantysy (`float`)	24	24
Bity mantysy (`double`)	53	53
Bity mantysy (`long double`)	64	64
Bity cechy (`float`)	8	8
Bity cechy (`double`)	11	11
Bity cechy (`long double`)	15	15
Epsilon (`float`)	1.19209e-07	1.19209e-07
Epsilon (`double`)	2.22045e-16	2.22045e-16
Epsilon (`long double`)	1.0842e-19	1.0842e-19

Tabela 4: Występowanie wartości specjalnych w zależności od kompilatora (GCC i Clang)

Typ	Wartość	GCC	Clang
`float`	`+0`	0.0	0.0
	`-0`	-0.0	-0.0
	`+Inf`	inf	inf
	`-Inf`	-inf	-inf
	`NaN`	-nan (isnan=1)	-nan (isnan=1)
`double`	`+0`	0.0	0.0
	`-0`	-0.0	-0.0
	`+Inf`	inf	inf
	`-Inf`	-inf	-inf
	`NaN`	-nan (isnan=1)	-nan (isnan=1)
`long double`	`+0`	0.0	0.0
	`-0`	-0.0	-0.0
	`+Inf`	inf	inf
	`-Inf`	-inf	-inf
	`NaN`	-nan (isnan=1)	-nan (isnan=1)

W powyższych tabelach widać, że niezależnie od użytego kompilatora (`GCC 8.5.0` czy `Clang 19.1.7`), uzyskano identyczne rozmiary typów, liczbę bitów mantysy i cechy oraz wartości maszynowego epsilon. W drugiej tabeli widać, że zarówno w przypadku kompilatora **GCC**, jak i **Clang**, wartości specjalne są poprawnie obsługiwane we wszystkich formatach. Funkcja `isnan()` zwraca `1` dla `NaN`, co wskazuje na zgodność z oczekiwanym zachowaniem standardu IEEE 754.

4.3 Architektury

Na końcu dokonano porównania wyników między różnymi urządzeniami: laptopem (architektura `x86_64`) a telefonem z procesorem ARM. Tabele 5. i 6. ilustrują przykładowe rozmieszczenie wyników pomiarów.

Tabela 5: Porównanie wyników na laptopie (`x86_64`) i telefonie (`ARM`)

Parametr	Laptop (x86_64)	Telefon (ARM)
Rozmiar `float` (B)	4	4
Rozmiar `double` (B)	8	8
Rozmiar `long double` (B)	16	16
Bity mantysy (`float`)	24	24
Bity mantysy (`double`)	53	53
Bity mantysy (`long double`)	64	113
Bity cechy (`float`)	8	8
Bity cechy (`double`)	11	11
Bity cechy (`long double`)	15	15
Epsilon (`float`)	1.19209e-07	1.19209e-07
Epsilon (`double`)	2.22045e-16	2.22045e-16
Epsilon (`long double`)	1.0842e-19	1.92593e-34

Tabela 6: Występowanie wartości specjalnych w zależności od architektury (`x86_64` i `ARM`)

Typ	Wartość	x86_64	ARM
`float`	`+0`	0.0	0.0
	`-0`	-0.0	-0.0
	`+Inf`	inf	inf
	`-Inf`	-inf	-inf
	`NaN`	-nan (isnan=1)	nan (isnan=1)
`double`	`+0`	0.0	0.0
	`-0`	-0.0	-0.0
	`+Inf`	inf	inf
	`-Inf`	-inf	-inf
	`NaN`	-nan (isnan=1)	nan (isnan=1)
`long double`	`+0`	0.0	0.0
	`-0`	-0.0	-0.0
	`+Inf`	inf	inf
	`-Inf`	-inf	-inf
	`NaN`	-nan (isnan=1)	nan (isnan=1)

Powyższe tabele pokazują, jak różne architektury wpływają na reprezentację zmiennoprzecinkową. Wartość bajtów dla `long double` jest identyczna (16 B), ale liczba bitów mantysy w przypadku ARM wynosi aż 113, podczas gdy na x86_64 jest to 64.
Wartość maszynowego epsilon dla `long double` na ARM jest znacznie mniejsza niż na x86_64, co sugeruje inną reprezentację liczb o większej precyzji.
Jest to zgodne z faktem, że architektura ARM wykorzystuje w tym przypadku pełny 128-bitowy format zgodny ze standardem ISO/IEC/IEEE 60559:2011 (który jest rozwinięciem IEEE 754-2008 dla arytmetyki zmiennoprzecinkowej), gdzie `long double` faktycznie zajmuje pełne 16

bajtów i oferuje wyższą precyzję w porównaniu do x86_64, gdzie zazwyczaj stosuje się 80-bitową precyzję rozszerzoną z wypełnieniem do 16 bajtów lub podwójnie precyzyjny format o wielkości 16 bajtów, ale z ograniczoną mantysą.
Dzięki temu ARM dostarcza bardziej zgodną ze standardem reprezentację 128-bitową, eliminując niejednoznaczności związane z różnymi implementacjami `long double` na architekturach x86_64.

5 Podsumowanie Zagadnienia

5.1 Systemy Operacyjne

Analiza wykazała, że system operacyjny może mieć istotny wpływ na reprezentację i precyzję obliczeń zmiennoprzecinkowych. Chociaż teoretycznie standard IEEE 754 gwarantuje spójność dla formatów `float`, `double` i `long double`, praktyka pokazuje pewne rozbieżności. W szczególności **Windows** wykazuje nietypowe zachowanie dla `long double`, gdzie wartości `+Inf`, `-Inf` oraz `NaN` były błędnie przedstawiane jako `0.000000`. Ponadto, wartość epsilon dla `long double` była niestabilna i zmieniała się przy każdym uruchomieniu programu.

5.2 Kompilatory

Testy przeprowadzone na `Fedora Linux` z kompilatorami `GCC` i `Clang` wykazały, że wybór kompilatora nie miał wpływu na wyniki dla standardowych typów. Oba kompilatory zwracały identyczne wartości epsilon, liczby bitów mantysy oraz cechy. Co więcej, wartości specjalne (`±0`, `±Inf`, `NaN`) były poprawnie obsługiwane zarówno przez GCC, jak i Clang. Oznacza to, że standard IEEE 754 został zachowany niezależnie od użytego kompilatora.

5.3 Architektury

Porównanie wyników na `x86_64` i `ARM` wykazało istotne różnice w implementacji `long double`. Na architekturze x86_64 format `long double` ma 64 bity mantysy, podczas gdy na ARM wartość ta wynosi aż **113 bitów**, co sugeruje pełną 128-bitową reprezentację zmiennoprzecinkową (zgodną ze standardem IEEE 754).

6 Wnioski

Przeprowadzone badania wykazały, że pomimo teoretycznej zgodności standardów zmiennoprzecinkowych IEEE 754, w praktyce istnieją istotne różnice wynikające z systemu operacyjnego, kompilatora oraz architektury sprzętowej. Najważniejsze wnioski to:

- Windows nie implementuje poprawnie standardu IEEE 754 dla `long double`,
- Linux zapewnia spójniejsze i bardziej przewidywalne wyniki zgodne ze standardem IEEE 754.
- Wybór popularnego kompilatora (GCC vs. Clang) nie wpływa na poprawność reprezentacji.
- Architektura ARM obsługuje `long double` jako pełne 128-bitowe liczby zmiennoprzecinkowe, co zapewnia większą precyzję niż standardowe rozwiązania na x86_64.