Praktikum – Rechnerarchitektur
Assembler – Projekt
Spezifikation

G53: Adrian Kögl

Daniel Osipishin

Marc Sinner

Inhaltsverzeichnis

Aufgabenverteilung

Adrian Kögl	Projektleiter
Daniel Osipishin	Verantwortlicher Dokumentation
Marc Sinner	Verantwortlicher Vortrag

Abgabetermine:

Spezifikation	13.05.2018
Implementierung	17.06.2018
Ausarbeitung	08.07.2018
Vortrag	23.07 - 03.08.2018

Generell: Fertigstellung bis zum Mittwoch vor Abgabe fertig haben, um bei Rückfragen Betreuer bei der Sprechstunde zu fragen

Einzelaufgaben

Adrian Kögl	Deadline	Daniel Osipishin	Deadline	Marc Sinner	Deadline
Implementierung der Funktionen pow(x, n) und fac(n) (10 Std.)	30.05.18	Implementierung der Funktionen f(x) und fDer(x) (10 Std.)	30.05.18	Implementierung des Newton-Verfahrens (10 Std.)	30.05.18
Zusammenstellen des kompletten Assembler-Programms (10 Std.)	06.06.18	Zusammenstellen des kompletten Assembler-Programms (10 Std.)	06.06.18	Zusammenstellen des kompletten Assembler-Programms (10 Std.)	06.06.18
Optimierung des Assembler-Programms (10.Std)	13.06.08	Optimierung des Assembler-Programms (10.Std)	13.06.08	Implementierung des C-Testing-Programms (10. Std)	13.06.08

Aufgabenbeschreibung

Es soll die Funktion $y = \cosh^{-1}(x)$ in Assembler realisiert werden. Sowohl der Input x, als auch der Output y sollen vom Datentyp float (32 Bit) sein. Dabei soll die Nutzung der Befehle des Floating-Point Units nur auf Addition, Subtraktion, Multiplikation, Division, Negation und Speicherverwaltung und Kontrollbefehle eingeschränkt sein.

Außerdem soll ein C-Rahmenprogramm zum Testen der Assembler-Funktion und deren Laufzeit implementiert werden. Als Vergleich zur Bewertung von Laufzeit und Genauigkeit der implementierten Assembler-Funktion soll die $\cosh^{-1}(x)$ – Funktion aus der C-Bibliothek verwendet werden.

Ist – Analyse

Die benötigten Befehle von Assembler und FPU stehen zur Verfügung.

Platzhalter für Compiler + Rahmenprogramm

Der gewählte Lösungsansatz wurde durch ein Java-Programm mit den gegebenen Einschränkungen (nur +, -, * und /) simuliert und als funktionsfähig bewiesen.

Die Konstanten zur optimalen Berechnung wurden im Java-Programm getestet.

Implementierung in Assembler wurde noch nicht angefangen.

Soll – Analyse

Die Assembler-Funktion muss mit den oben genannten Einschränkungen implementiert werden.

Dabei soll die Nutzung von Realisierungs- und Laufzeitsaufwendigen Funktionen wie $a^x; x \in R$ oder $\ln x$ vermieden werden.

Das C-Rahmenprogramm zum Testen der Assembler-Funktion muss implementiert werden.

Erfolgskriterium

Der Projekterfolg ist objektiv messbar anhand von:

- Laufzeit der Funktion
- Ergebnisgenauigkeit
- Speicherverbrauch

Idealerweise soll die Funktion die Genauigkeit der float-Mantisse komplett ausnutzen (23 Bit).

Da die C-Funktion Zugang zu allen FPU-Befehlen Zugang hat, benutzt sie wahrscheinlich folgende Formel zur Berechnung (mithilfe der FPU-Befehle FYL2X / FSQRT für Berechnung von Logarithmus und Wurzel):

$$\cosh^{-1}(x) = \ln(x + \sqrt{x^2 - 1})$$

Diese Berechnung wird wahrscheinlich 10 bis 100 Mal schneller als unsere Funktion sein.

Lösungsansätze

#1: Annäherung mittels Newton-Verfahren

Es gilt:

$$y = \cosh^{-1}(x) \rightarrow \cosh(y) - x = 0 \; f\ddot{u}r \; y \geq 0$$

Dadurch lässt sich y als Nullstelle der Gleichung $\cosh(y) - x = 0 \; f\ddot{u}r \; y \geq 0$ berechnen. x ist dabei beliebig aber konstant. Die Nullstelle kann mittels Newton-Verfahren ermittelt werden:

1) Wähle ein y_0, was möglichst nah an der Nullstelle liegt (funktioniert aber mit beliebigem y_0). Ein passendes y_0 kann einer vorzeitig angefertigten Lookup-Table entnommen werden.
2) Benutze die folgende Formel, bis die erwünschte Genauigkeit erreicht ist

$$y_{n+1} = y_n + \frac{f(y_n)}{f'(y_n)}$$

Dabei sind $f(y_n) = \cosh(y_n) - x$ und $f'(y_n) = \sinh(y_n)$. x ist der Input und dadurch zur Laufzeit konstant. cosh und sinh lassen sich mittels Taylor-Reihen ausrechnen (bis zur erwünschten Genauigkeit):

$$\cosh(y) = 1 + \frac{y^2}{2!} + \frac{y^4}{4!} + \frac{y^6}{6!} + \cdots + \frac{y^{2n}}{(2n)!} + \cdots$$

$$\sinh(y) = y + \frac{y^3}{3!} + \frac{y^5}{5!} + \frac{y^7}{7!} + \cdots + \frac{y^{2n+1}}{(2n+1)!} + \cdots$$

Die Programmstruktur würde für diesen Ansatz so aussehen:

- Newton-Verfahren:

```
public static float invCosh(float x) {
    float y = lookupY(x);
    while (isPrecise(y) == false) {
        y = y + (f(y, x) / fDerivative(y));
    }
    return y;
}
```

- $f(y_n) = \cosh(y_n) - x$:

```
private static float f(float y, float x) {
    float output = 1;
    for (int i = 0; i < taylorDepth1; ++i) {
        output += (float) Math.pow(y, 2 * i) / fac(2 * i);
    }
    return output - x;
}
```

- $f'(y_n) = \sinh(y_n)$:

```
private static float fDerivative(float y) {
    float output = y;
    for (int i = 1; i < taylorDepth2; ++i) {
        output += (float) Math.pow(y, 2 * i + 1) / fac(2 * i + 1);
    }
    return output;
}
```

Dieser Lösungsansatz verwendet nur die gegebenen Grundoperationen (+, -, *, /) sowie leicht realisierbare Potenzieren und Fakultät.

#2: Lookup-Table + Interpolation

Werte von $y = \cosh^{-1}(x)$ für bestimmte Werte von x werden vorzeitig berechnet und in einer Wertetabelle gespeichert. Wird ein x eingegeben, was in der Wertetabelle vorhanden ist, wird der dazugehörige y-Wert ausgegeben. Ansonsten werden zwei oder mehr benachbarte Werte genommen und dafür verwendet, mittels Interpolation ein y-Wert anzunähern.

Es gibt mehrere Interpolationsmethoden, die mit unterschiedlicher Genauigkeit, Komplexität und Laufzeit arbeiten. Die gängigen Methoden sind lineare, polynomielle, Logarithmische und Kosinus-Interpolation.

Die lineare Interpolation erstellt mithilfe der benachbarten Werte eine lineare Funktion, in die der eingegebene Wert eingesetzt wird. Nach Vereinfachung sieht das Vorgehen so aus:

$$y = y_1(1-\mu) + y_2\mu; \quad \mu = \frac{x - x_1}{x_2 - x_1}$$

Die polynomielle Interpolation erstellt mithilfe von mindestens 4 benachbarter Werte ein Polynom, in den der eingegebene Wert eingesetzt wird. Das Vorgehen variiert je nach dem Grad des gewünschten Polynoms.

Die Kosinus-Interpolation ähnelt der linearen Interpolation, benutzt jedoch Kosinus für einen glatteren Übergang zwischen den Funktionsbereichen. Nach Vereinfachung sieht das Vorgehen so aus:

$$y = y_1(1-\mu) + y_2\mu; \quad \mu = \frac{1 - \cos(\frac{x - x_1}{x_2 - x_1}\pi)}{2}$$

Ähnlich nähert die logarithmische Interpolation einen y-Wert an, aber mithilfe eines natürlichen Logarithmus:

$$y = y_1(1-\mu) + y_2\mu; \quad \mu = \ln(\frac{x - x_1}{x_2 - x_1})$$

Argumentation für die Entscheidung

Interpolation erweist sich als schlecht für die Aufgabenstellung geeignet. Lineare Interpolation ist entweder sehr ungenau, was nur mit einer sehr ausführlichen Lookup-Table bekämpft werden kann. Diese würde jedoch viel Speicherplatz verbrauchen.
Polynomielle Interpolation eignet sich schlecht für $\cosh^{-1}(x)$, da die Funktion an sich logarithmisch ist und dadurch sich von einem Polynom stark unterscheidet (z.B. Anstieg immer positiv). Dies würde zu Ungenauigkeiten führen. Außerdem haben die dadurch erstellten Polynome i.d.R. eine vergleichsweise hohe Laufzeit beim Berechnen von y.
Die Kosinus-Interpolation und vor allem die logarithmische Interpolation würden mir einer sehr guten Genauigkeit operieren. Die Realisierung dieser Methoden würde jedoch die Implementierung von $y =$

$\cos(x)$ oder $y = \ln(x)$ verlangen, da die Aufgabenstellung die Verwendung der dazugehörigen FPU-Befehle verbietet.
Außerdem wird Interpolation i.d.R. zur Berechnung der Werte einer Funktion mit unbekanntem bzw. sehr komplexen Term verwendet, was in dieser Aufgabe nicht der Fall ist. Somit ist es effizienter und genauer, den Wert der Funktion direkt auszurechnen.

Das Newton-Verfahren liefert sehr genaue Funktionswerte, verbraucht sehr wenig Speicherplatz und besitzt eine gute Laufzeit, die durch den flexiblen Genauigkeit/Laufzeit-Tradeoff nach Wunsch anpassbar ist. Die Realisierung dieses Lösungsansatzes verlangt keine Implementierung komplexer Funktionen und bietet viel Spielraum für sämtliche Optimierungen (z.B. das Speichern von oft verwendeten Fakultäts- und Potenzierungsergebnissen im Arbeitsspeicher).