ERA Praktikum SS 2017

Gleitkomma-Arithmetik

Realisierung von $sin\ x$

1. Einleitung

Im Rahmen des ERA-Praktikums an der TU München sollen zwei Projekte in einem Team von drei Studierenden bearbeitet werden.

Dies ist die Spezifikation zum Projekt 1 “Realisierung von $sin\ \ x$ in Assembler”.

2. Aufgabenkurzbeschreibung

Ziel der Aufgabe ist es die Sinusfunktion $float\ \ sin(float\ \ x)$ in x86-Assembler mit nasm Syntax umzusetzen. Es werden verschiedene Implementationsweisen berücksichtigt: einerseits die Reihenentwicklung, die sich durch die Taylor-Mac Laurin’sche Formel an die Sinusfunktion annähert, andererseits die Interpolation durch eine Lookup Table, wobei ausgewählte Werte als Stützstellen in einer Tabelle gespeichert werden (die Auswahl derer ist vom Funktionsverlauf abhängig).

Hierfür stehen ausschließlich die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division sowie die Negation als Multiplikation mit -1 als FPU-Befehle `FADD, FSUB, FMUL, FDIV` zur Verfügung.

Beide Ansätze erfordern einen Kompromiss zwischen Laufzeit bzw. Speicher und den resultierenden Abweichungen in der Berechnung.

In der Arbeitsphase emulieren wir den Assemblercode mithilfe von JASMIN.

Zur Überprüfung und Durchführung der Leistungsmessung soll ein C-Rahmenprogramm zur Verfügung gestellt werden. Zur Kompilierung des Codes durch den GNU C - Compilers wird eine Makefile erstellt.

3. Lösungsansätze

Zur Berechnung der Sinusfunktion wurden zwei Möglichkeiten erarbeitet, die im Folgenden näher betrachtet werden sollen.

Der erste Lösungsweg verwendet die Reihenentwicklung nach Taylor-Mac Laurin, während der zweite Ansatz auf eine Lookup Table zurückgreift.

Bei beiden Ansätzen wird der Wert an der Stelle 0 im Hauptspeicher übergeben und das Ergebnis im FPU-Register ST0 zurückgegeben.

Da die Sinusfunktion periodisch ist, müssen lediglich die Werte zwischen 0 und 2π berechnet werden. Hierzu wird π als Konstante im Speicher abgelegt. Alle Werte über 2π werden über die Modulo-Funktion auf besagtes Intervall abgebildet.

2.1 Lösungsweg A

Eine genaue Möglichkeit zur Annäherung der Sinusfunktion bietet die Taylor-Reihe.

Diese geht auf die Arbeiten von Brook Taylor und Colin Maclaurin im 18. Jahrhundert zurück.

Die Taylor-Mac Laurin’sche Formel zur Annäherung von Sinus ist definiert als:

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^{(n)}(0)}{n!}x^n + R_n(x)$$

wobei $R_n(x) = \frac{x^{n+1}}{(n+1)!} f^{(n+1)}(\theta x)$ mit $0 < \theta < 1.$

Da Genauigkeit aber auch Effizienz von der Länge der Reihe bzw. Anzahl der Summanden abhängen muss zunächst eine obere Grenze ($:= m$) festgelegt werden. Jedoch kann diese obere Grenze erst optimal gewählt werden wenn Testwerte zum Vergleich vorliegen. Daher wird m erst in der Testphase endgültig bestimmt.

Registerbelegung:

`EAX` := Laufvariable n

`EBX` := obere Grenze m

`ST0` := Rechenregister / Endergebnis

`ST1` := Aktuelles Zwischensumme der Teilergebnisse

`ST2` := $R_n(x)$

`ST3` := Zwischenspeicherung von π

`ST4` bis `ST7` := Rechenregister

2.1 Lösungsweg B

In der im Hauptspeicher abgelegten Lookup Table werden Zwischenwerte im Intervall von 0 bis π gespeichert. Beliebige Werte können durch Interpolation von benachbarten Zwischenwerten berechnet werden.

Da die Funktion linearer verläuft, je näher sie der x-Achse kommt, müssen an den Hoch- und Tiefpunkten kleinere Zwischenschritte gewählt werden, um Interpolationsfehler zu minimieren.

Um Speicherplatz zu sparen und somit eine höhere Dichte an Zwischenwerten zu erreichen zu können, ist eine Möglichkeit, die Sinusfunktion nur im Bereich von 0 bis $\frac{\pi}{2}$ zu speichern und die fehlenden Werte durch Spiegelung (Negation und Addition) zu ergänzen.

Zur Interpolation ziehen wir zwei Ansätze in Betracht, auf der einen Seite die lineare Interpolation und auf der anderen die Annäherung über das Newton-Verfahren.

Bei der linearen Interpolation wird durch die simplere Berechnung eine schnellere Laufzeit erzielt, allerdings besteht ein erhöhter Speicherbedarf, um die Genauigkeit aufrecht zu erhalten.

Die Annäherung über das Newton-Verfahren kommt mit weniger Zwischenwerten aus, ist jedoch ineffizienter in der Berechnung.

Es folgt ein erster Implementationsansatz in Pseudocode:

```

// In Look-Up-Table nur Werte von 0 - Pi/2 gespeichert (--> sin(x) periodisch),
// daher wird x auf diesen Bereich abgebildet

$AH = 1              // Wenn AH auf -1 gesetzt ist, wird Ergebnis am Ende negiert
Eingabe $x           // Wert auf den Sinus angewendet werden soll
$pi = 3.14....       // Wert von Pi in 32 Bit gespeichert
$x = $x mod (2*$pi)  // x auf Bereich 0 - 2Pi abbilden (da periodische Funktion)
                     // Bereich Pi - 2Pi auf Bereich 0 - Pi abbilden,
if($x > $pi){        // (da nur Bereich 0 - (Pi / 2) in Look-Up-Table gespeichert)
    $AH = -1         // Endergebnis wird nun später negiert
    $x = $x - $pi    // x' auf Bereich 0 - Pi abbilden
}
if($x > $pi/2){      // Bereich Pi/2 - Pi auf Bereich 0 - Pi/2 abbilden (Spiegelung)
    $x = $pi - $x    // x'' auf Bereich 0 - Pi/2 abbilden
}


// Nun wird mit Binärer Suche in der Look-Up-Table nach dem passenden Interval gesucht
// In diesem Pseudocode wird vernachlässigt, dass in der Look-Up-Table Werte und zugehörige
// Funktionswerte abwechselnd in aufeinanderfolgenden Speicherzellen gespeichert sind.
// Es wird vereinfachend davon ausgegangen, dass der x-Wert dem Index entspricht und sich
// der zugehörige Funktionswert in der Speicherzelle befindet.

$EAX = Startadresse der Look-Up-Table // Untere Schranke des Intervalls
$EBX = Endadresse der Look-Up-Table // Obere Schranke des Intervalls

Start:               // Marke, zu der später gesprungen wird
$ECX = ($EBX - $ECX) / 2 + $EAX     // Pivotelement in $ECX speichern
if( $x == $ECX){     // Wenn x bereits mit Wert übereinstimmt,
    JMP Ende         // wird die Suche beendet
}
elseif($x < $ECX){   // Wenn x kleiner das Pivotelement ist,
    $EBX = $ECX      // wird obere Schranke auf Pivotelement gesetzt
}
else{                // Sonst muss x größer als Pivotelement sein,
    $EAX = $ECX      // also wird untere Schranke auf Pivotelement gesetzt
}
if($EBX - $EAX > 1){ // Solange Interval größer als 1 ist
    JMP Start        // wird immer wieder zur Startmarke gesprungen
}

Ende:                // Endmarke


// In $EBX ist nun das kleinste Element gespeichert, dass größer oder gleich x ist.
// In der Speicherzelle vor $EBX muss sich also das größte Element befinden, das
// kleiner als x ist.
// Mittels Interpolation wird nun der Funktionswert von x angenährt

$a = Inhalt($EAX)    // Inhalt von Speicherzelle ist (vereinfacht angenommen) der Funktionswert
$b = Inhalt($EBX)
$prozA = ($x - $a) / ($b - $a)   // Prozentuale Nähe von $x zu $a jetzt in $prozA
$ergebnis = $a * $prozA + (1-$prozA) * $b // Berechnetes Ergebnis nun in $ergebnis
$ergebnis = $ergebnis * $AH     // Falls x im Bereich Pi - 2Pi lag, wird das Ergebnis negiert

```

4. Bewertung der Ansätze

	Lösungsansatz A	Lösungsansatz B
Vorteile	• Hohe Genauigkeit • Einfache Implementierung • Geringer Speicherverbrauch	• Sehr effizient für ähnliche Genauigkeit wie Ansatz A
Nachteile	• Ineffizient (Hohe Laufzeit) • Aufwendige Berechnung	• Hoher Speicherverbrauch • Aufwendigere Implementierung • Leicht ungenauer als Ansatz A

5. Entscheidungswahl

Die Lookup Table bietet bei ausreichender Tabellengröße eine ähnlichen Genauigkeit und gewährleistet durch kleineren Berechnungsaufwand eine deutlich höhere Effizienz.

Speicherplatz ist heutzutage erschwinglich und stellt somit keinen limitierenden Faktor mehr dar.

Effizienz hingegen gewinnt immer mehr an Bedeutung, weshalb wir uns trotz aufwendigerer Implementierung für Lösungsansatz B entschieden haben.

Aus denselben Gründen ziehen wir innerhalb des Lösungsansatzes B die lineare Interpolation der Newton’schen Annäherung vor.