LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 251 – Abgabe zu Aufgabe A319
Sommersemester 2023

Sina Mozaffari Tabar Alireza Kamalidehghan Mostafa Nejati Hatamian

1 Einleitung

1.1 Überblick

Die Arithmetik ist eine fundamentale Disziplin der Mathematik, die sich mit den Grundoperationen wie Addition, Subtraktion, Multiplikation und Division befasst. Normalerweise sind wir es gewohnt, diese Operationen in unserem alltäglichen Leben in einem dezimalen Zahlensystem durchzuführen, bei dem die Basis 10 beträgt.
Darüber hinaus die Basen 2 und 16 sind Zahlensysteme, die in der Informatik und Mathematik häufig verwendet werden. Jedes Zahlensystem basiert auf einer bestimmten Anzahl von Symbolen, die verwendet werden, um Zahlen darzustellen. Hier ist eine Erklärung, wofür wir die Basen 2 und 16 im Allgemeinen brauchen:

Basis 2 (Binärsystem): Das Binärsystem verwendet nur zwei Symbole, normalerweise 0 und 1. Es ist das grundlegendste Zahlensystem in der digitalen Welt, da Computerinformationen auf zwei Zuständen basieren: ausgeschaltet (0) und eingeschaltet (1).

Basis 16 (Hexadezimalsystem): Das Hexadezimalsystem verwendet 16 Symbole: die Zahlen 0-9 und die Buchstaben A-F. Es bietet eine kompaktere Darstellung großer Binärzahlen und erleichtert die Lesbarkeit und Handhabung von Zahlen in der Informatik.

Es gibt eine allgemeine Formel,die eine Allgemiene Repräsentation von Zahlen in anderen Zahlensystemen in Dezimalschreibweise darstellt. Die Formel lautet:

$$A = \sum_{i=0}^{n-1} a_i * g^i \tag{1}$$

In dieser Formel steht A für die resultierende Zahl im Dezimalsystem, n für die Anzahl der Stellen unserer Zahl, g für die Basis und a_i für die i-te Ziffer der ursprünglichen Zahl.

Um ein besseres Verständnis zu bekommen gehen wir jetzt eine Beispiel ein. wir betrachten uns die Zahl 101011 in Binärsystem und mit der oben gennanten Formel wollen wir diese Zahl in Dezimalsystem umwandeln:

$$1 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3 + 0 * 2^4 + 1 * 2^5 = 1 + 2 + 8 + 32 = 43$$

1.2 Die Aufgabe

Unsere Aufgabe besteht aus zwei Teilen, einem theoretischen und einem praktischen Teil.

1.2.1 Theoretischer Teil:

In diesem Teil der Ausarbeitung untersuchen wir die Konversion von komplexen Zahlen zur Basis $(-1+i)$ und entwickeln die entsprechenden Algorithmen. Zunächst erklären wir, warum die Zahl $(3+2i)$ zur Basis $(-1+i)$ die Darstellung 1001 hat. Darauf aufbauend leiten wir die Algorithmen zur Konversion in beide Richtungen ab. Wir ziehen bei Bedarf geeignete Literaturquellen zu Rate, um unsere Argumentation zu unterstützen. Des Weiteren zeigen wir, dass komplexe Zahlen $(a+bi)$ mit den Werten $a \in \{1, 0, -1\}$ und $b \in \{1, 0, -1\}$ zur Basis $(-1+i)$ mit binären Koeffizienten dargestellt werden können.

1.2.2 Praktischer Teil:

Im praktischen Teil der Ausarbeitung implementieren wir zwei Funktionen in unserem C-Code. Die erste Funktion, `to_carthesian`, nimmt eine vom Benutzer spezifizierte Zahl (*bm1pi*) entgegen, die Bits enthält, die eine Zahl in der Darstellung zur Basis $(-1+i)$ repräsentieren. Die Funktion wandelt diese Zahl in ihre kartesische Darstellung um und liefert den Realteil *real* und den Imaginärteil *imag*. Die zweite Funktion, *to_bm1pi*, erhält eine komplexe Zahl in kartesischer Darstellung mit Realteil *real* und Imaginärteil *imag* und gibt einen Integer zurück, dessen Bits die Ziffern der Eingabezahl zur Basis $(-1+i)$ darstellen.

2 Lösungsansatz

2.1 Theoretischer Teil:

Generell gilt, dass wenn eine Basis und eine Zahl gegeben sind, die in dieser Basis erweitert werden soll, die Zahl wiederholt durch die Basis geteilt wird und die Reste notiert werden. In Bezug auf die Basis $-1+i$ wissen wir, dass der Rest bei der Erweiterung einer Gaußschen Zahl entweder 0 oder 1 ist.

Nehmen wir eine Zahl $n+im$ und teilen sie durch die gewählte Basis:

$$\frac{n+im}{-1+i} = \frac{(n+im)(-1-i)}{2} = \frac{m-n}{2} - i\frac{n+m}{2}$$

Wenn sowohl n als auch m gerade oder beide ungerade sind, dann sind sowohl $m-n$ als auch $m+n$ gerade. Das Ergebnis der Division ist eine Zahl und der Rest ist 0.

Diese Formulierung berücksichtigt, dass der Zähler und der Nenner mit $-1-i$ (dem konjugierten Komplexen der Basis) multipliziert wurden und dass der Rest 0 ist.

$$\frac{m-n}{2} - i\frac{n+m}{2}$$

Wenn nur eines von n oder m gerade ist (das andere ungerade), kann man die folgende Version des Quotienten betrachten:

$$\frac{n+im}{-1+i} = \frac{m-n+1}{2} - i\frac{n+m-1}{2} - \frac{1}{2}(1+i)$$

In diesem Fall ist das Ergebnis der Division die folgende Gaußsche Zahl:

$$\frac{m-n+1}{2} - i\frac{n+m-1}{2}$$

und der Rest ist 1. Tatsächlich gilt:

$$(-1+i)(\frac{m-n+1}{2} - i\frac{n+m-1}{2}) = n+im-1$$

Man kann diese Vorgehensweise mit den resultierenden Zahlen wiederholen, bis die ursprüngliche Zahl aufgrund der wiederholten Divisionen durch 2 verschwindet und die Sequenz der Reste erhalten bleibt...

Lassen Sie uns den Algorithmus anhand der angegebenen Zahl $3+2i$ veranschaulichen:
Wenn

$$(m-n) - i(n+m) = (2-3) - i(2+3) = -1-5i$$

(siehe das zweite Spaltenpaar in der folgenden Tabelle) durch 2 teilbar wäre, würden sich die Werte im dritten Spaltenpaar unverändert wiederholen und der Rest wäre null (siehe Spalte *REMAINDER*). Da $3+2i$ jedoch nicht durch zwei teilbar ist, erfolgt eine Modifikation gemäß der Formel :

$$(m-n+1) - i(n+m-1) = 0-4i$$

und der Rest ist 1. Das erste Spaltenpaar zeigt auf das Ergebnis der ersten Division. (Die Division durch 2 wird jetzt durchgeführt.) Anschließend wird der oben beschriebene Schritt wiederholt, bis der entsprechende Wert im dritten Spaltenpaar zu $0+i0$ geworden ist.

		(DIVISION BY -1+i) * 2		**MODIFIED**		
REAL	**IMAG**	**REAL**	**IMAG**	**REAL**	**IMAG**	**REMAINDER**
3	2	-1	-5	0	-4	1
0	-2	-2	2	-2	2	0
-1	1	2	0	2	0	0
1	0	-1	-1	0	0	1

Endlich haben wir es geschafft! Jetzt brauchen wir nur die Restspalte (Remainder) einfach von unten nach oben nacheinander verfassen und erhalten wir am Ende unsere Zahl in der Basis $(-1+i)$.

$$(3+2i)_{10} = (1001)_{-1+i}$$

Um die Korrektheit unseres Ergebnis zu überprüfen, können wir die oben gennanten Formel(**??**) zum Einsatz bringen, um unsere Zahl in Basis $(-1+i)$ wieder in Basis 10 zu wechseln:

$$\begin{aligned}(1001)_{-1+i} &= 1*(-1+i)^0 + 0*(-1+i)^1 + 0*(-1+i)^2 + 1*(-1+i)^3 \\ &= 1+0+0+(2+2i) \\ &= (3+2i)_{10}\end{aligned}$$

Für den zweiten Teil können wir nochmal die Formel (**??**) in Gebrauch nehmen, um es zu beweisen, dass die angeforderten Komplexe Zahlen zur Basis $(-1+i)$ mit binären Koeffizienten darstellbar sind.

i) Falls a = -1, b = -1:

$$-1-i = (-2i)+(-1+i)+0 = 1*(-1+i)^2+1*(-1+i)^1+0*(-1+i)^0 = (110)_{-1+i}$$

ii) Falls a = -1, b = 0:

$$\begin{aligned}-1+0i &= (-4)+(2+2i)+(-2i)+0+1 \\ &= 1*(-1+i)^4+1*(-1+i)^3+1*(-1+i)^2+0*(-1+i)^1+1*(-1+i)^0 \\ &= (11101)_{-1+i}\end{aligned}$$

iii) Falls a = -1, b = 1:

$$-1+i = (-1+i)+0 = 1*(-1+i)^1+0*(-1+i)^0 = (10)_{-1+i}$$

iv) Falls a = 0, b = -1:

$$0+(-i) = (-2i)+(-1+i)+1 = 1*(-1+i)^2+1*(-1+i)^1+1*(-1+i)^0 = (111)_{-1+i}$$

v) Falls a = 0, b = 0:

$$0+0i = 0 = 0*(-1+i)^0 = (0)_{-1+i}$$

vi) Falls a = 0, b = 1:

$$0+i = (-1+i)+1 = 1*(-1+i)^1+1*(-1+i)^0 = (11)_{-1+i}$$

vii) Falls a = 1, b = -1:

$$\begin{aligned}1-i &= (4-4i)+(-4)+(2+2i)+0+(-1+i)+0 \\ &= 1*(-1+i)^5+1*(-1+i)^4+1*(-1+i)^3+0*(-1+i)^2+1*(-1+i)^1+0*(-1+i)^0 \\ &= (111010)_{-1+i}\end{aligned}$$

viii) Falls a = 1, b = 0:

$$1 + 0i = 1 = 1 * (-1 + i)^0 = (1)_{-1+i}$$

ix) Falls a = 1, b = 1:

$$1+i = (2+2i)+(-2i)+(-1+i)+0 = 1*(-1+i)^3+1*(-1+i)^2+1*(-1+i)^1+0*(-1+i)^0 = (1110)_{-1+i}$$

2.2 Praktischer Teil:

2.2.1 to_carthesian Methode (1.Variante)

Zuerst ermitteln wir mithilfe einer Maske und einer Schleife die reine Länge des Binärwerts *bm1pi* ohne vorgehende Nullen. Der Gründ dafür ist, dass unsere Zahlen(Folgen von 0 und 1), ab einer bestimmten Stelle nach Links nur Nullen enthalten könnten und aus Effizienz Gründen wollen wir nicht ohne Grund jedes Mal die Ganze Zahl bis Ende durchiterieren.

Nun die intuitivste Lösung wäre es, unsere Formel (**??**) zu nutzen, aber um dies zu tun mussten wir für jedes einzelne Bit in *bm1pi* , die 1 ist, $(-1+i)^n$ gemäß seiner Position in der Zahl berechnen und anschließend alle miteinander addieren, was wegen der stetigen Potenzierung, die Effizienz unseres Codes stark verringert, unerwünscht war. Daher haben wir uns dazu entschlossen, die $(-1+i)^n$ für eine begrenzte Anzahl von n zu berechnen und ein Muster zu finden, damit wir weder Mathe Bibliothek noch Potenz nutzen müssen.

Gemäß unseres gefundenen Musters gibt es eine Relation zwischen jede acht Ziffer unserer Zahl deswegen haben wir unsere Zahl in Abschnitten der Länge 8 geteilt und in jeder Iteration überprüfen wir in jedem Abschnitt ob die Bits der Zahl gesetzt sind und führen wir gemäß des Musters die Berechnungen durch.

2.2.2 to_carthesian Methode (2.Variante)

In dieser Variante verwenden wir eine Matrixmultiplikation, um die Konvertierung durchzuführen. Es gibt zwei vordefinierte Vektoren *real_row* und *imag_row*, die die Einträge von Matrix darstellen. Jeder Vektor enthält 8 Werte vom Typ *int16_t* , die die Koeffizienten für die Berechnung von Real- bzw. Imaginärteilen der komplexen Zahl repräsentieren. Der Algorithmus verwendet eine Schleife, um die Bits des Eingabewerts *bm1pi* zu extrahieren und in einer 2D-Array-Struktur *bits* zu speichern. Diese 2D-Array repräsentiert die Ziffern von *bm1pi* in einer 8x16-Matrix, wobei jede Spalte ein Byte von *bm1pi* speichert.

In der Hauptberechnungsschleife werden die Bytes in jeder Spalte der *bits* Matrix verwendet, um die Matrixmultiplikation mit den Vektoren *real_row* und *imag_row* durchzuführen. Dies wird erreicht, indem die SIMD-Instruktionen *_mm_madd_epi16* (Multiply-Add) verwendet werden, um die Multiplikationen und Additionen in einem Schritt durchzuführen.

Die Ergebnisse der Multiplikationen werden in den temporären Variablen *temp_real* und *temp_imag* gespeichert. Dann werden die Ergebnisse in horizontaler Richtung summiert, indem die SIMD-Instruktion *_mm_hadd_epi32* (Horizontal Add) wiederholt aufgerufen wird.

Der beschribene Algorithmus kann durch die folgenden Matrixsgleichung erklärt werden:

$$\begin{pmatrix} 1 & 0 \\ -1 & 1 \\ 0 & -2 \\ 2 & 2 \\ -4 & 0 \\ 4 & -4 \\ 0 & 8 \\ -8 & -8 \end{pmatrix}^T \times \begin{pmatrix} a_0 & a_8 & \cdots & a_{120} \\ a_1 & a_9 & \cdots & a_{121} \\ a_2 & a_{10} & \cdots & a_{122} \\ a_3 & a_{11} & \cdots & a_{123} \\ a_4 & a_{12} & \cdots & a_{124} \\ a_5 & a_{13} & \cdots & a_{125} \\ a_6 & a_{14} & \cdots & a_{126} \\ a_7 & a_{15} & \cdots & a_{127} \end{pmatrix} \times \begin{pmatrix} 2^4 \\ 2^8 \\ 2^{12} \\ 2^{16} \\ 2^{20} \\ 2^{24} \\ 2^{28} \\ 2^{32} \\ 2^{36} \\ 2^{40} \\ 2^{44} \\ 2^{48} \\ 2^{52} \\ 2^{56} \\ 2^{60} \\ 2^{64} \end{pmatrix} = \begin{pmatrix} \mathit{Real} \\ \mathit{Imag} \end{pmatrix}$$

Schließlich werden die Ergebnisse aus den temporären Variablen extrahiert und in die Ausgabevariablen *real* und *imag* akkumuliert. Dabei werden die Werte um eine bestimmte Anzahl von Bits verschoben und addiert, um den Wert der komplexen Zahl zu aktualisieren.

Insgesamt führt der Code eine effiziente SIMD-basierte Matrixmultiplikation durch, um die Umwandlung einer speziellen komplexen Zahlendarstellung in kartesische Koordinaten durchzuführen.

2.2.3 to_bm1pi Methode (1.Variante)

3 Korrektheit

Wir haben umfangreiche Tests für die Funktion *to_carthesian* durchgeführt und freuen uns, bekannt zu geben, dass die Implementierung erfolgreich funktioniert. Die Funktion konvertiert eine gegebene Zahl im BM1PI-Format in die entsprechenden kartesischen Koordinaten.

Wir haben eine Vielzahl von Testfällen abgedeckt, darunter positive und negative Zahlen, verschiedene Kombinationen von Bits und verschiedene Längen von BM1PI-Zahlen. In jedem Testfall wurde die Funktion aufgerufen, und die erhaltenen kartesischen Koordinaten wurden mit den erwarteten Werten verglichen.

Bei allen Tests hat die Implementierung den korrekten Ergebnissen geliefert. Die berechneten kartesischen Koordinaten stimmten genau mit den erwarteten Werten überein. Dies bestätigt, dass der Code korrekt implementiert worden ist.

Darüber hinaus wurden auch Spezialfälle wie die Konvertierung von Null oder sehr großen BM1PI-Zahlen getestet. Auch hier hat die Implementierung konsistente und korrekte Ergebnisse geliefert.

Die erfolgreichen Tests bestätigen, dass die Funktion *to_carthesian* zuverlässig arbeitet und die vorgegebenen Anforderungen erfüllt.

Außerdem kann man auch mit mathematischen Formeln, die schon in Einleitung und Theoretischem Teil vom Lösungsansatz erwähnt wurden(Formel **??**), kann man die Korrektheit unserer Implementierung nochmal beweisen.

Hier sind einige Beispielen und Testfällen mit den erwarteten Ergebnissen(laut der Berechnungen mit mathematischen Formeln) und tatsächlich gelieferten Ergebnissen von der jeweiligen Methode:

3.1 to_carthesian Methode

i) Eingabe:0 Ausgabe: $0 + 0i$ Erwartet: $0 + 0i$

ii) Eingabe: $\underbrace{111\cdots1}_{\text{128 times}}$ Ausgabe: $-7378697629483820646 - 3689348814741910323i$
Erwartet: $-7378697629483820646 - 3689348814741910323$

iii) Eingabe: 00011101001 Ausgabe: $-1 - 2i$ Erwartet: $-1 - 2i$

iv) Eingabe: $\underbrace{1000\cdots0}_{\text{127 times}}$ Ausgabe: $-9223372036854775808 - 9223372036854775808i$
Erwartet: $-9223372036854775808 - 9223372036854775808i$

3.2 to_bm1pi Methode

i) Eingabe: 0,0 Ausgabe: 0 Erwartet: 0

ii) Eingabe: 3,2 Ausgabe: 1001 Erwartet: 1001

iii) Eingabe: 123456789,123456789
Ausgabe: $11101000011001101000111000001110111011101110000011100110111 0$
Erwartet: $11101000011001101000111000001110111011101110000011100110111 0$

iv) Eingabe: -123456789,-123456789
Ausgabe: $10001110100010000110000001100110011001100000011011100110$
Erwartet: $10001110100010000110000001100110011001100000011011100110$

4 Performanzanalyse

5 Zusammenfassung und Ausblick

6 Quellenverzeichnis

Ausarbeitung