Abgabe von Bruno Stendal, Martin Baer, Lukas Gewinner und Christian Schäfer
4. Aufgabenblatt zum Kurs

TI2: Rechnerarchitektur

von Bernadette Keßler

bis Freitag, den .2022, 10:15 Uhr.

Zahlendarstellung und Rechnen

Beantworten Sie folgende Fragen:

1. Bei Fließkommazahlen ist die Position des Kommas nicht wie bei Festkommazahlen fixiert, das heißt, dass mein keine Kompromisse zwischen Wertebereich und Genauigkeit eingehen muss. Auf diese Weise können sowohl sehr große Zahlen, die jedoch nur wenige Nachkommsastellen besitzen, als auch sehr genaue Zahlen, die viele Nachkommastellen jedoch wenige Vorkommastellen besitzen, gespeichert werden.

2. Eine Fließkommazahl besteht aus drei Abschnitten, die Mantisse, Charakteristik und S heißen. Dabei steht S für das Vorzeichenbit. Die Fließkommazahl wird nach der Exponentialschreibweise mit $z = Mantisse \cdot Charakteristik$ dargestellt. Die Charakteristik besteht intern aus der Basis 2 mit einem Exponenten, welchem je nach Größe Bits reserviert werden.

3. Der Vorteil dem Exponenten mehr Bits zuzuteilen ist, dass der Wertebereich der darstellbaren Zahlen größer wird und der Vorteil der Mantisse mehr Bits zuzuteilen ist, dass man genauere Zahlen darstellen kann.

4. Ein Überlauf entsteht, wenn man eine größere Zahl als die größte darstellbare Zahl darstellen will. Das heißt, es sind mehr Bits gefordert, als für den Exponenten zugeteilt wurden. Beim Unterlauf ist es genau andersherum, wenn man eine kleinere Zahl als die kleinste darstellbare Zahl darstellen will. Anders gesagt entsteht eine Lücke zwischen der kleinsten darstellbaren positiven und größten darstellbaren negativen Zahl rund um die null.

5. Bei IEEE-754 werden bestimmte Standards für die Darstellung von Fließkommazahlen gesetzt. Beispielsweise werden bei 32 Bits die Bits in das „Most significant bit“ für das Vorzeichen, 8 Bits für die Charakteristik/Exponenten und 23 Bits für die Mantisse aufgeteilt. Negative Exponenten werden mittels Offset=127 ermöglicht. Des Weiteren wird die Normalisierung verwendet und durch das hidden Bit etwas Speicher eingespart, aber dann eine extra Darstellung der „0“ gefordert. Overflows über maxreal und minreal werden als +- unendlich dargestellt und eine unzulässige Darstellung NaN (not a number).

6. Betragsmäßig größte bzw. kleinste darstellbare Zahl im IEEE-754 32bit Standard:

Größte darstellbare Zahl: $(2 - 2^{-23}) \cdot 2^{127} \approx 3,410^{38}$

Kleinste darstellbare Zahl

normalisiert: $2^{-126} \approx 1,210^{-38}$

denormalisiert: $2^{-23} \cdot 2^{-126} \approx 1,410^{-45}$

RA-Übung 4

IEEE P 754: $-592.183940 + 0.91213$

Umrechnung in Binär wurde auf letztem Zettel gemacht, hier nicht mehr ausführl.

$592.183940 \hat{=} 1001010000.0010111100010110 1...$
$\cdot 2^{+9}$, dann noch 23 Bit significand

$1.001010000000101111000101|101...$
(1 ist implizit; 23; guard; round → aufrunden)

also $1. \;\ldots\;\ldots\;\ldots\;\ldots\; 1\underline{10}$

characteristic: müssen mit 2^9 multiplizieren, zusätzlich 127 Offset, also 136

$136 = 10001000_2$, Zahl ist negativ, also sign = 1

Es ergibt sich:

1 | 10001000 | 00101000000010111100011 0

S | Char. | signif.

$0.91213 \hat{=} 0.111010011000000101011010|000001...$ ($\cdot 2^1$) — abrunden

character: 2^{-1}, Offset 127 → 126 $\hat{=}$ 01111110, sign = 0

↳ 0 | 01111110 | 11010011000000101011010

Addition: (bzw Subtraktion) 2^9, 2^{-1} → $9-(-1)=10$ → Rechts-Shift vom 0,91...

~~implizite 1 →~~
~~0.0000000001|1101001100000|011 ← sticky gesetzt~~
~~G R S~~
~~− 1.00101000000010111100011 0 000~~
~~0110110000101101000110100 11~~

```
   1.00101000000010111100011 0 0 0
−  0.0000000001110100110000000|1 0 1   ← sticky gesetzt
   ─────────────────────────────────
   1.00100111101011100000101|0 1 1
                             G R S
```

→ abrunden, ist schon normalisiert

> SideNote: Bin der Meinung in der VL ist was falsch, ich brauche ja bei der Papier-Rechnung 1+23 Bits wegen der impliziten 1? und dann eben erst guard, round, sticky

sign = 1 (da Zahlen in Rechnung getauscht)

↳ 1 | 10001000 | 00100111101011100000101

2^9 (wie oben)

↳ $\hat{=} -1001001111{,}0101110000 0101_2 \overset{?}{=} 591{,}35968017\ldots$

3981.1729 x -2.91762

3981,1729 ≙ 111110001101,001011100010 0|0011 ... (abrunden)

$\cdot 2^{11}$

<u>character</u>: 11 + 127 = 138 , sign = 0

0 | 10001010 | 11110001101001011000100

(−)2.91762 ≙ 10,1110101011101001001001|00111 (abrunden)

$\cdot 2^{-1}$

<u>character</u>: 1 + 127 = 128 , sign = 1

1 | 10000000 | 01110101011101001001001

Sorry, aber die Multiplikation der Mantissen werde ich mal nicht aufschreiben...

Ergebnis ist 10,1101010111111000110010|011 ... ← abrunden

← um eine Stelle verschieben ⇒ in character. +1

sign = 1 xor 0 → 1   <u>character</u>:

```
  10001010
+ 10000000 ⎫
- 01111111 ⎭ = 00000001
-----------
  10001011
```

aber +1 wegen Normalisierung oben, also

10001100

1 | 10001100 | 01101010111111000110010

(−) = 140 → 2^{13}, also 10110101011111 1,1000110010

≙ −11615,548828125.

Floating Point Rechner

Implementieren Sie eine Floating Point Unit in Software. Zwei gegebene 32-bit IEEE-754 Zahlen (operand1 und operand2) sollen addiert werden können. Das Ergebnis soll ebenfalls im 32-bit IEEE-754 Floating Point Format zurückgegeben werden. Verwenden Sie keine Floating Point Register / Befehle außer MOVD.

```
;Bruno Stendal, Martin Baer, Lukas Gewinner, Christian Schaefer
            global calc_add

calc_add:
        mov rax, 0;
        movd edx, xmm0;
        shr edx, 31;
        mov ah, dl;

        movd edx, xmm0;
        shl edx, 1;
        shr edx, 24;
        mov al, dl;

        movd edx, xmm0;
        shl edx, 9;
        shr edx, 9;
        mov esi, edx;
        add esi, 8388608;

        mov rcx, 0;
        mov rdx, 0;

        movd edx, xmm1;
        shr edx, 31;
        mov ch, dl;

        movd edx, xmm1;
        shl edx, 1;
        shr edx, 24;
        mov cl, dl;

        movd edx, xmm1;
        shl edx, 9;
        shr edx, 9;
        mov edi, edx;
        add edi, 8388608;

        ;ah sign a
        ;al expo a
```

```
41          ;esi mat a
42
43          cmp ah, 1;
44          jne .matisse;
45          xor esi, 0xff;
46
47          ;ch sign b
48          ;cl expo b
49          ;edi mat b
50
51          cmp ch, 1;
52          jne .matisse;
53          xor edi, 0xff;
54
55          mov rdx, 0;
56
57      .matisse:
58          cmp al, cl;
59          jl .shiftA;
60          jg .shiftB;
61          je .noShift;
62
63      .shiftA:
64
65          mov rdx, 0;
66          mov dl, cl;
67          mov cl, 0;
68
69          mov cl, dl;
70          sub cl, al;
71
72          mov al, cl;
73          mov edx, 1;
74
75          shr esi, cl;
76
77          mov al, dl;
78          mov cl, dl;
79
80          jmp .noShift;
81
82      .shiftB:
83
84          mov rdx, 0;
85          mov dl, cl;
86          mov cl, 0;
87
```

```
88          mov cl, al;
89          sub cl, dl;
90
91          shr edi, cl;
92
93          mov cl, dl;
94          jmp .noShift;
95
96      .noShift:
97
98          mov rdx, 0;
99          mov edx, esi;
100         add edx, edi;
101
102
103         mov rbx, 0;
104         mov ebx, edx;
105         shr ebx, 24;
106
107         cmp ebx, 0;
108         jne .expoPlus;
109         je .end;
110
111     .expoPlus:
112         cmp ebx, 0;
113         jne .loop;
114         je .end;
115     .loop:
116         add al, 1;
117         shr rbx, 1;
118         jmp .expoPlus;
119
120     .end:
121         shl edx, 9;
122         shr edx, 9;
123
124         cmp ah, 1;
125         jne .endBuild
126         xor edx, 0xff;
127         cmp ah, 1;
128         je .endBuild
129
130         cmp ch, 1;
131         jne .endBuild
132         xor edx, 0xff;
133
134     .endBuild:
```

```
135
136         mov rbx, 0;
137         mov bh, ah;
138         mov bl, al;
139         shl rbx, 23;
140         add ebx, edx;
141         movd xmm0, ebx;
142         ret
```