Grundlagenpraktikum Rechnerarchitektur
Lehrstuhl für Rechnerarchitektur & Parallele Systeme Technische Universität München

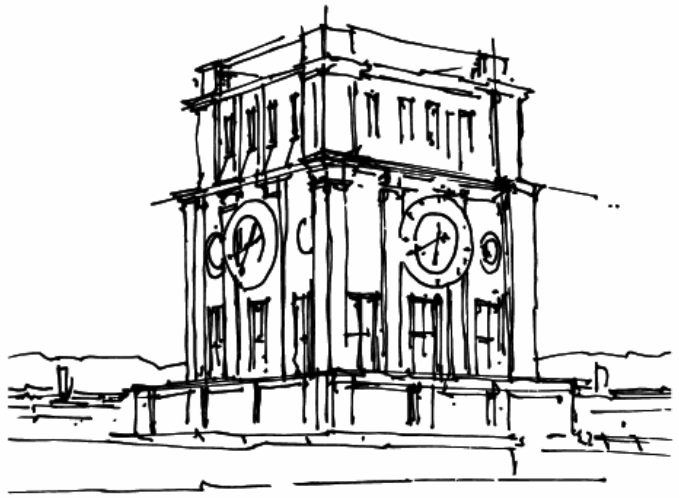
Vortrag des Abschlussprojekts

07.08.2025

Floating-Point Unit & FMA

Ahmed Amine Loukil
Nourallah Gargouri
Saifeddine Guenanna

Gruppe T104



Gliederung

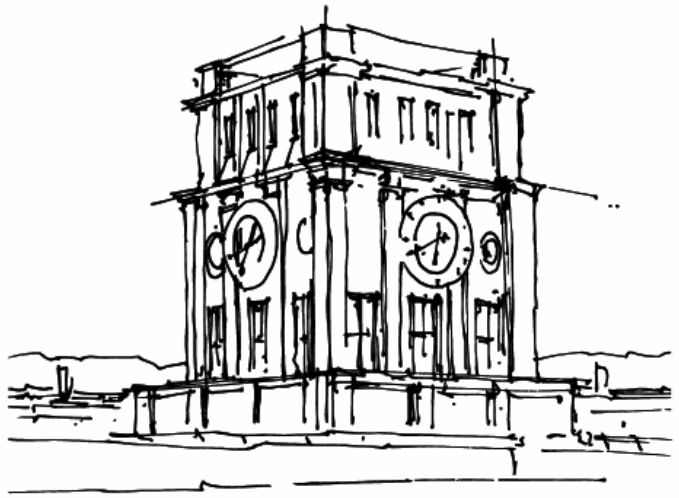
I- Überblick & Problemstellung

II- Lösungsansatz

III- Korrektheit & Testing

IV- Evaluation des Projekts

V- Zusammenfassung & Ausblick

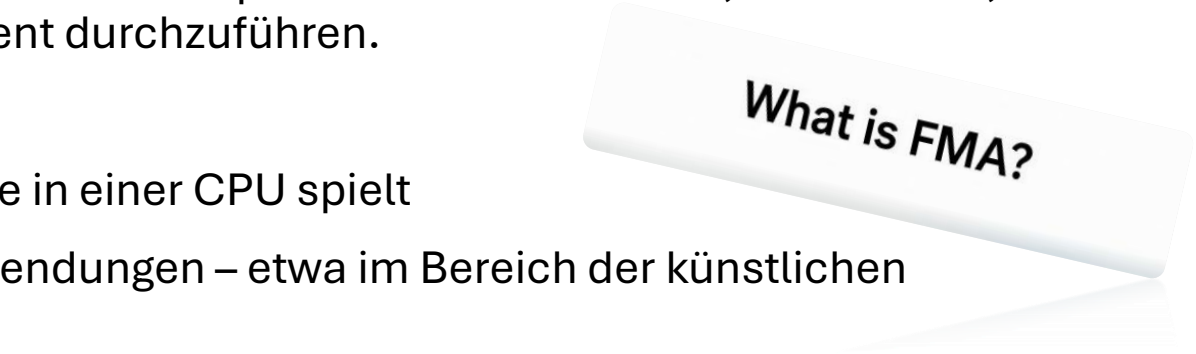


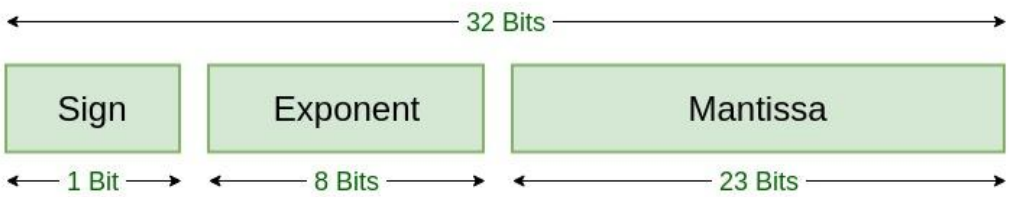
I- Überblick & Problemstellung

Im Rahmen dieses Projekts haben wir eine Floating-Point Unit (FPU) mit Unterstützung für Fused-Multiply-Add (FMA) entwickelt. Die FPU ist in der Lage, grundlegende Fließkommaoperationen wie Addition, Subtraktion, Multiplikation sowie die kombinierte FMA-Operation effizient durchzuführen.

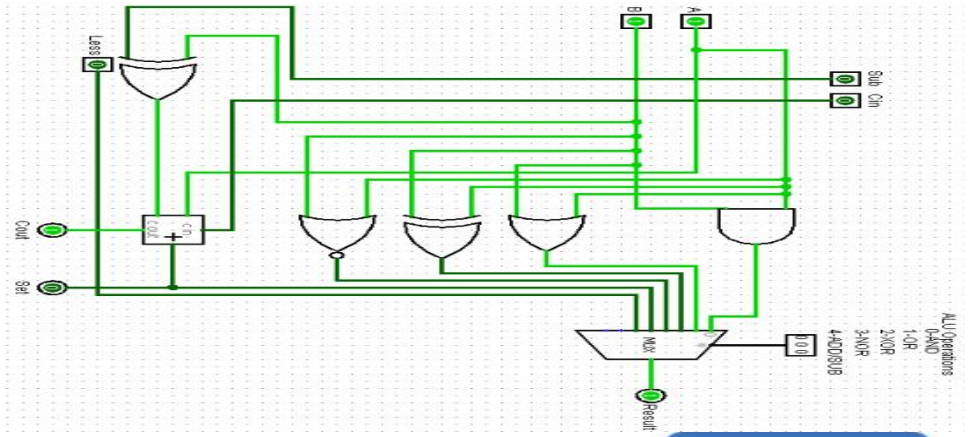
In dieser Präsentation erläutern wir:

- Was genau eine Floating-Point Unit ist und welche Rolle sie in einer CPU spielt
- Warum insbesondere FMA-Operationen in modernen Anwendungen – etwa im Bereich der künstlichen Intelligenz – von zentraler Bedeutung sind,
- Und schließlich präsentieren wir Auszüge aus unserem Code sowie einen Überblick über die modulare Architektur unserer Implementierung, testing und unsere eigene Bewertung.





Single Precision
IEEE 754 Floating-Point Standard



$$a + (b \times c) \longrightarrow \text{FMA}$$

Was ist eine Floating-Point Unit ?

- Eine Floating-Point Unit ist ein spezieller Teil der CPU, der für Berechnungen mit Fließkommazahlen zuständig ist. In unserem Projekt implementieren wir eine eigene FPU in SystemC, die Operationen wie Addition, Subtraktion, Multiplikation sowie FMA unterstützt – also genau die Rechenarten, die bei KI-Anwendungen und wissenschaftlichen Berechnungen besonders wichtig sind.

Was ist IEEE 754 ?

- IEEE 754 ist der internationale Standard zur Darstellung und Berechnung von Fließkommazahlen in Computern. Er definiert, wie Zahlen mit Vorzeichen, Exponent und Mantisse in 32 (oder 64 Bit) gespeichert werden.

Warum IEEE 754 ?

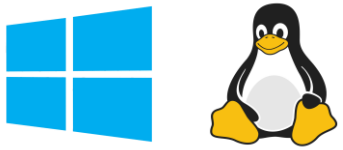
- Weil er einheitlich, präzise und plattformübergreifend ist. Man muss sich nicht selbst um Skalierung, Rundung oder Sonderfälle wie NaN oder +∞ kümmern – alles ist standardisiert.

Wer benutzt heute FMA, FPU und IEEE 754?

- **FPU :**
- **Moderne Prozessoren** (Intel, AMD, Apple M-Chips, ARM, RISC-V): Jede CPU besitzt eine FPU für Fließkommazahlen.
- **Microcontroller mit FPU**: z. B. ARM Cortex-M4, -M7 (in Industrie, Sensorik, Robotik).
- **Laptops, Smartphones, Server, Spielekonsolen**: FPU ist Standard in jeder Hardwareplattform.

- **FMA (Fused Multiply-Add) :**
- **Grafikkarten** (NVIDIA, AMD): Für Shader, Raytracing, Bildverarbeitung.
- **KI-Frameworks** (TensorFlow, PyTorch): Verwenden FMA intern bei Matrixoperationen.
- **Compiler**: z. B. GCC/Clang aktivieren automatisch FMA-Befehle, wenn die Hardware es unterstützt.

- **IEEE 754 Standard**
- **Alle gängigen Programmiersprachen**: C, C++, Java, Python, Fortran, Rust, usw.
- **Betriebssysteme & Rechenbibliotheken**: Linux, Windows, MATLAB, NumPy, usw.
- **CPUs & GPUs weltweit**: Nutzen IEEE 754 als verbindlichen Standard zur Fließkommazahlendarstellung.



Und jetzt die Frage, Warum ist FMA besonders für die Berechnung von neuronalen Netzen wichtig?

- In neuronalen Netzen werden große Mengen an Matrixmultiplikationen durchgeführt – insbesondere während der Vorwärts- und Rückwärtsausbreitung (Forward- & Backpropagation). Dabei treten ständig Berechnungen der Form a + (b × c) auf.
- Die FMA-Operation führt diese Berechnung in nur einem Rechenschritt und mit nur einer Rundung durch. Das bietet zwei große Vorteile:

Höhere numerische Genauigkeit :
Reduziert Rundungsfehler, was besonders bei tiefen Netzen und langen Trainings wichtig ist.

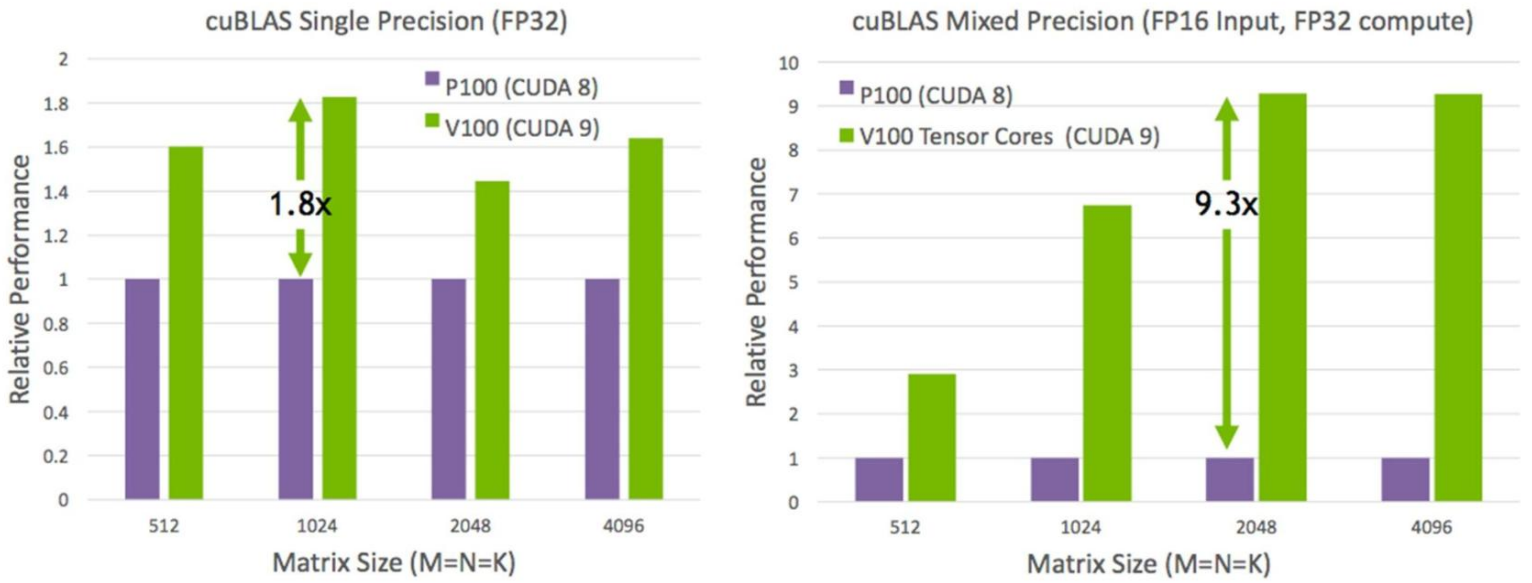
Mehr Performance :
Spart Taktzyklen, da Multiplikation und Addition nicht separat durchgeführt werden müssen.

- Diese Vorteile machen FMA zu einem wichtigen Bestandteil moderner KI-Hardware wie GPUs, TPUs oder spezialisierter KI-Beschleuniger.

Beispiel: NVIDIA Tensor Cores oder Apple Neural Engine nutzen FMA massiv für Deep Learning.

NVIDIA integrierte FMA-Operationen erstmals hardwareseitig im Rahmen der Tensor Cores, die mit der Volta-Architektur im Jahr 2017 eingeführt wurden.

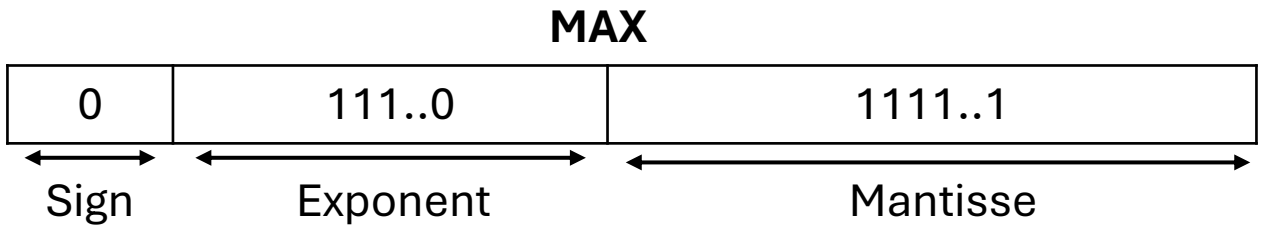
Die Tesla V100 GPU war das erste Produkt mit Tensor Cores, die Fused Multiply-Add (FMA)-Operationen in gemischter Genauigkeit (Mixed Precision) ausführten – speziell optimiert für Matrixmultiplikationen in neuronalen Netzen.



- Die Rechenleistung stieg durch die Einführung von FMA in der V100 im Vergleich zur P100 ohne FMA um ein Vielfaches – insbesondere beim KI-Training und bei Inferenzaufgaben.

Was sind die Minimal- und Maximalwerte mit verschiedenen --size-exponent und --size-mantissa?

Um Max zu bestimmen , setzen wir das Sign auf 0, Alle bits des Exponents auf 1 außer das kleinste (Sonst Inf/Nan) und alle Bits der Mantissa auf 1.



StoredExp = $\sum_{k=1}^{e-1} 2^k = \frac{2\cdot\left(2^{e-1}-1\right)}{2-1} = 2^e - 2$

Mantissa = $\sum_{k=1}^{m} \frac{1}{2^k} = 1 - \frac{1}{2^m} = 1 - 2^{-m}$

ActualExp = StoredExp – bias = $2^e - 2 - (2^{e-1} - 1) = 2^{e-1} - 1$ **(=bias)**

1.Mantissa = 1 + Mantissa = $2 - 2^{-m}$

Max= $1.M * 2^{E} = (2 - 2^{-m}) * 2^{2^{e-1}-1}$

Um Min zu bestimmen , setzen wir das Sign auf 1, Alle bits des Exponents auf 1 außer das kleinste (Sonst Inf/Nan) und alle Bits der Mantissa auf 1. Ah so **Min = - Max**

Beispiel: **Single-precision floating-point format mit ExpSize = 8 und ManSize= 23:**

$$\text{Max}= (2 - 2^{-23}) * 2^{127} \approx 3.4028235 * 10^{38}$$
$$\text{Min} = \text{-Max} \approx -3.4028235 * 10^{38}$$

II - Lösungsansatz

i. Unsere Komponenten und deren Verbindung

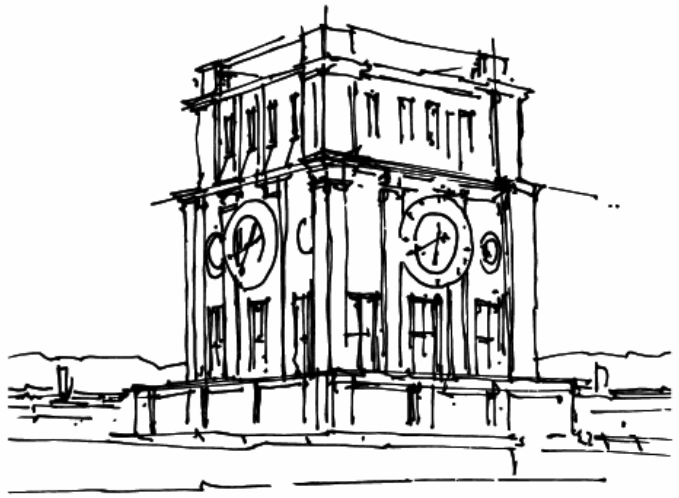
ii. Min & Max

iii. Rounding

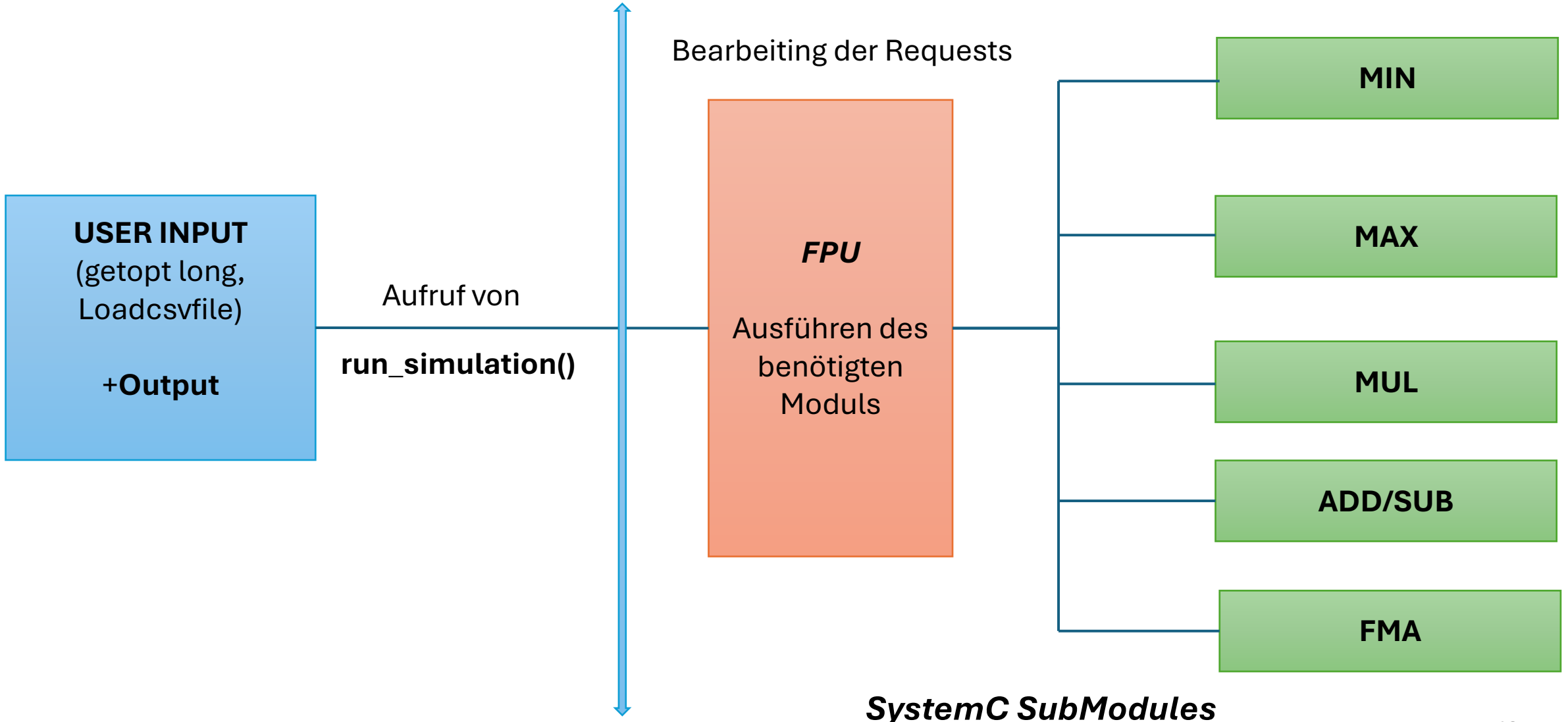
iv. Multiplikation

v. Add & Sub

vi. FMA



❑ Unsere Komponenten und deren Verbindung



Input:

Das Lesen der gegebenen Options und deren Argumenten **(über getopt long)** und das .csv Datei (mit Hilfe von **loadcsvfile()**), in dem es die Operationen und die Operanden gibt, die bearbeitet werden müssen.

Danach wird run_simulation() mit den benötigten Parametern aufgerufen.

SystemC FPU:

Das Floating Point Unit ist die **wichtigste Komponente** unserer Lösung. Sie überträgt mit Hilfe von Signalen die verschiedenen Operanden zu der entsprechenden Operation und **speichert die Ergebnisse** der Berechnungen**.**
In jedem Takt wird eine Rechenoperation ausgeführt.

Andere SytemC Modulen :

Es gibt insgesamt 6 mögliche Operationen **(add,sub,min,max,mul,fma)**. Jede Rechenoperation verfügt über ein SystemC Modul (Add und Sub verteilen das gleiche Modul) und **jedes Modul berechnet in jedem Takt** das Ergebnis der Rechenoperation.

Min & Max

- Das `Min`-Modul vergleicht zwei Gleitkommazahlen (`r1`, `r2`) und gibt die kleinere zurück. Die Logik ist analog zu `Max`, nur dass die Vergleichsrichtung umgekehrt ist.

```
int Min::compare(uint32_t a, uint32_t b) const {
    bool sign_a = a >> 31;
    bool sign_b = b >> 31;
    if (sign_a != sign_b)
        return sign_a ? 1 : -1;
    if (a == b) return 0;
    return (sign_a ? a > b : a < b) ? -1 : 1;
}
```

Drei einfache Vergleichsregeln:

- Beide positiv :
  Die Hex-Zahl mit dem kleineren Wert ist auch die kleinere Zahl.
  Kein Spezialfall nötig.
- Beide negativ :
  Der größere Hex-Wert entspricht der kleineren Zahl.
  Vergleich wird umgedreht.
- Unterschiedliche Vorzeichen :
  Die negative Zahl ist immer kleiner
  Vergleich nur über das Vorzeichen.

Warum der `Min`-Vergleich durch Hexadezimaldarstellung einfacher wird ?

Da unsere Floating-Point-Zahlen intern als 32-Bit-Hexadezimalwerte gespeichert sind (IEEE 754), wird der Vergleich im `Min`-Modul deutlich vereinfacht.

Behandlung von NaN und Inf-Zahlen ist straight-forward ->

```
if (is_inf(val1) && is_inf(val2)) {
    if (val1 == val2) {
        ro.write(val1);
    } else {
        nan.write(true);
        ro.write(get_nan());
    }
    return;
}
if (is_inf(val1)) {
    ro.write(val2);
    return;
}
if (is_inf(val2)) {
    ro.write(val1);
    return;
}
```

```
if (is_nan(val1) && is_nan(val2)) {
    nan.write(true);
    ro.write(get_nan());
    return;
}
if (is_nan(val1)) {
    ro.write(val2);
    return;
}
if (is_nan(val2)) {
    ro.write(val1);
    return;
}
```

Rounding

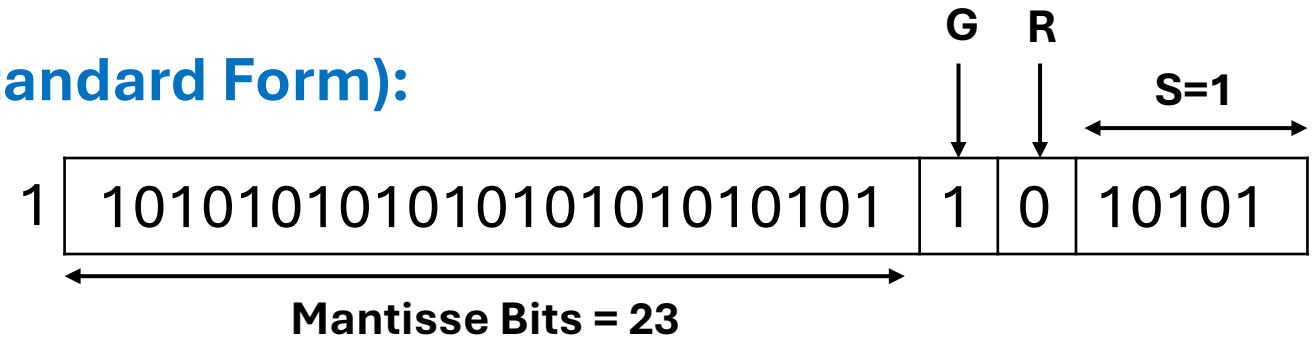
- In der Gleitkommaarithmetik nach IEEE-754 (einschließlich single-precision) ist eine Rundung erforderlich, wenn das Ergebnis einer Operation mit den verfügbaren Bits nicht genau dargestellt werden kann.

- Bei Operationen wie Addition, Subtraktion oder Multiplikation im Gleitkommaformat können Zwischenergebnisse mehr Bits ergeben, als gespeichert werden können.

Definitionen:

- **Guard Bit(G):** Das Bit direkt nach dem $MantissaSize. Bit der Mantisse. (nach 23. Bit in Standard).
- **Round Bit(R):** Das nächste Bit nach dem Guard Bit.
- **Sticky Bit(S):** Das ODER aller verbleibenden Bits nach dem Rundungsbit. Es zeigt an, ob nach dem R-Bit noch weitere "1"-Bits vorhanden sind.

Beispiel: (in Standard Form):



Nach dem Abschneiden der benötigten Mantissenbits wird durch das GRS entschieden, ob eine Erhöhung um 1 notwendig ist, um zu runden

Mode	Rounding Mode	Bedingung für die Runde (Erhöhung um 1)	Beispiel GRS
0	Round to nearest, ties to even	G=1 & (R=1 || S=1)	111,101,110
1	Round to nearest, ties **away** from 0	G=1 (unabhängig von R oder S)	100
2	Round toward zero	Never	
3	Round toward +INF	Sign=0 & G=1	101 (mit sign=0)
4	Round toward -INF	Sign=1 & G=1	110 (mit sign=1)

Multiplikation

Idee:

$$a*b = (1.M1 * 2^{e1}) * (1.M2 * 2^{e2}) = (1.M1 * 1.M2) * 2^{e1+e2}$$

Aus dieser Formel können wir schließen, dass die Multiplikation von 2 Floating Point Zahlen 5 Hauptschritte erfordert:

1- Das Exponent des Ergebnisses temporär **auf e1+e2 setzen**

2- **Multiplikation der Mantissen** nach dem Hinzufügen der Leading 1

3- Das Exponent anpassen anhand dessen + Rounding falls nötig

4- Das Sign bestimmen (einfach **sign(a) XOR sign(b)**)

5- Alles zusammensetzen und das Ergebnis finden

Algorithm reference :Multiplying Floating Point Numbers - GeeksforGeeks

Die Schritte lassen sich am besten anhand eines Beispiels näher erklären!

Um das Beispiel so einfach und klar wie möglich zu machen, betrachten wir einen Exponenten der Größe 4 und eine Mantisse der Größe 3 :

1- Das Exponent des Ergebnisses temporär auf e1+e2 setzen

Bias = 8-1= 7 , e1= storedExp1 – Bias = 7-7=0, e2= storedExp2 – Bias = 9-7=2

=> Exp = 2+0=2

2- Multiplikation der Mantissen nach dem Hinzufügen der Leading 1

1.M1 * 1.M2 = 1.111* 1.011= 10.100101

3- Das Exponent anpassen anhand dessen + Rounding falls nötig

Wir müssen 10.100101 auf 1,0100101 normalisieren (shift nach links um die Mantisse des Ergebnisses zu bestimmen ah so auf dem Form 1.M zu umwandeln) und den Exponenten mit 1 erhöhen

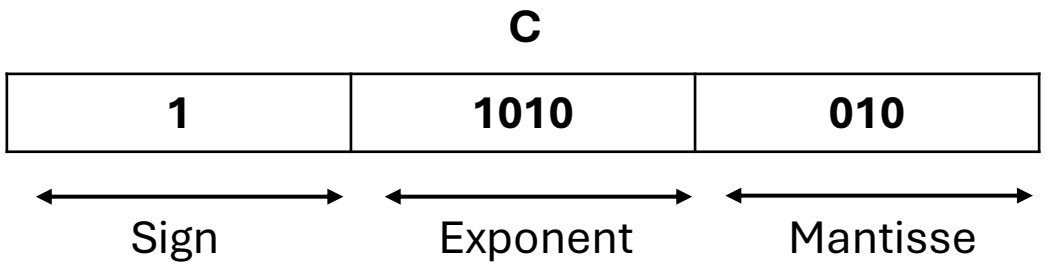
=> M= 0100101 (Rounding nötig weil Size Man = 3) => Nach Rounding (Das Ergebnis wird auf die nächstgelegene Zahl gerundet – Default Rounding) M= 010

=> ActExp = 2+1=3 => StoredExp = ActExp + bias = 3+7=10 = 1010

4- Das Sign bestimmen (einfach sign(a) XOR sign(b))

=> S = 0 XOR 1= 1

5- Alles zusammensetzen und das Ergebnis finden



Schwierigkeiten bei der Implementierung:

- **Die Normalisierung des Produkts** war herausfordernd, da sie eine genaue Bitverschiebung und exponentielle
- Korrektur erfordert.

- **Das Rundungsverhalten gemäß IEEE-754** war komplex, insbesondere die Behandlung von Guard-,Round- und Sticky-Bits.

- **Überlauf und Unterlauf** korrekt zu erkennen **zu behandeln** erforderte zusätzliche Kontrolllogik.

AddSub

Idee:

$$1.M1 \cdot 2^a + 1.M2 \cdot 2^b = 2^{\max(a,b)} \cdot \left(M_{\max} \pm \frac{M_{\min}}{2^{|a-b|}} \right)$$

Wir fokussieren jetzt auf addition, weil subtraktion eine Art äquivalent ist zu addition.

Die Hauptschritte sind:

1. Shift der kleineren mantissa , bis exponent das gleiche ist, G/S ensprechedn zu setzen
2. Je nach Vorzeichengleichung , entweder addieren wir die mantissen , oder subtrahieren wir
3. Normalizierung falls notwendig
4. runden

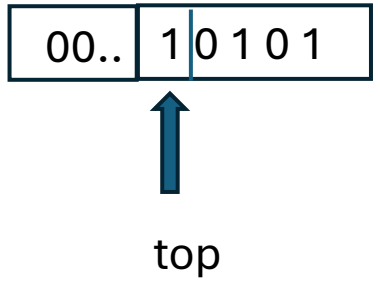
1. Shift der kleineren mantissa , bis exponent das gleiche ist, G/S entsprechend zu setzen

Beispiel:

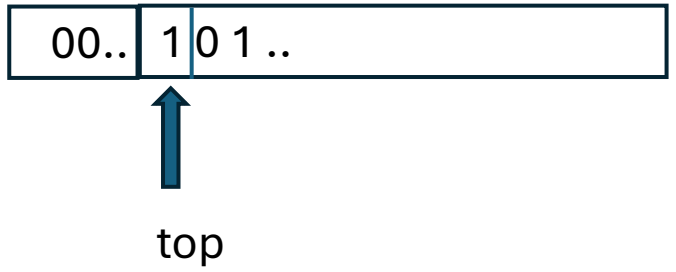
e1= 1

e2= 3

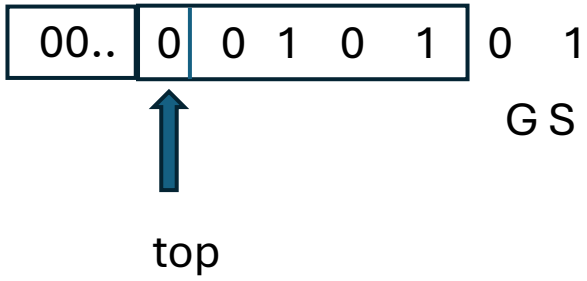
1.m1

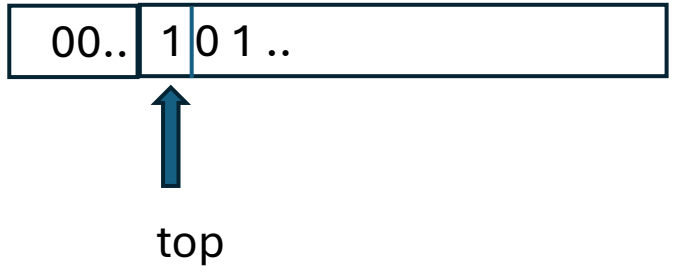


1.m2



Kleinere mantissa , weil geshiftet

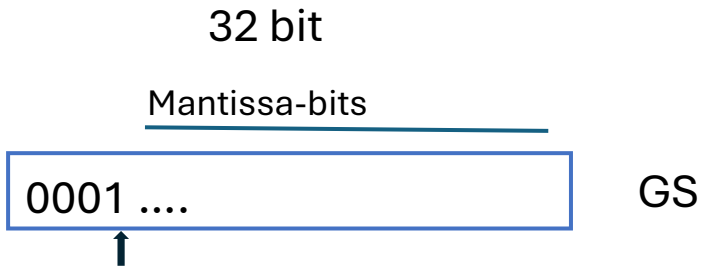




2. Je nach Vorzeichengleichung , entweder addieren wir die mantissen , oder subtrahieren wir

Fall 1: Vorzeichen ist gleich

Einfach zwischenergebnisse addieren :

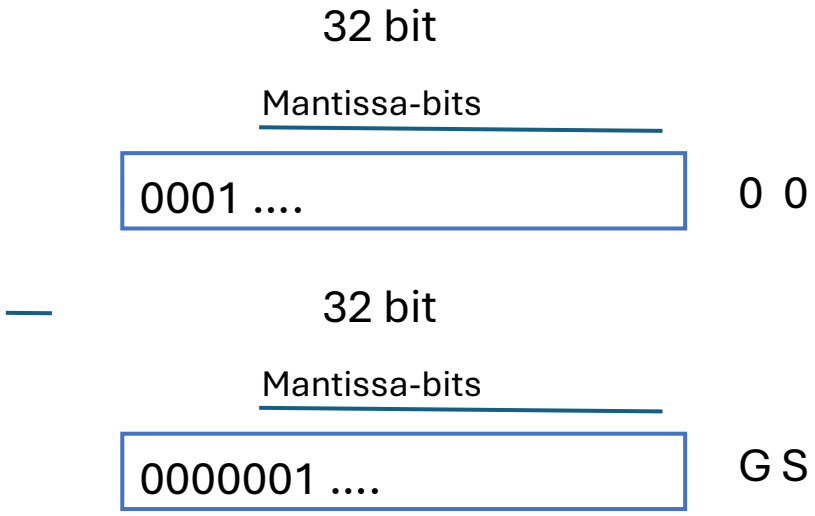


Bemerkung:
Hier sind 32 bits ausreichend da 1.m maximal 31 bits hat (mantissa-bits=0 is verboten) , und 31 bits + 31 bits ergeben maximal 32 bits

Achtung! Nach addition zweier zahlen kann eine zahl ergeben ,wo index(leading 1)=top +1.Indem Fall einfach rechts shift der mantissa und final-exponent++ (achten auf Overflow)

Fall 2 : Vorzeichen ist ungleich

Hier muss die größere Mantisse minus die kleinere Mantisse gebildet werden, da sonst ein Wrap-around auftreten könnte, was der Natur der Fließkommazahlen nicht entspricht.



Achtung! Nach subtraktion zwei zahlen kann eine zahl ergeben ,wo index(leading 1)<top .(vielleicht gibt's keine leading 1 , dann 0 zurueckgeben.)

Kernfrage: Nach einer Subtraktion – wenn man runden muss – nimmt man dann die G- und S-Bits einfach so, wie sie sind, und rundet wie bei einer gewöhnlichen Addition zweier Zahlen mit gleichem Vorzeichen?

Antwort: Nein!

Intuition:

Bei gleichem Vorzeichen (Addition):

Die G- und S-Bits stellen eine **natürliche Fortsetzung** der Mantisse des Ergebnisses dar, da man einfach zwei Mantissen addiert.

Beispiel:

Angenommen, wir können nur eine Kommazahl exakt darstellen, und das Ergebnis der Mantissenaddition ist 4,7, mit G = 1 und S = 1.

Dann erkennt man sofort:

Das ist wie +4,7+0,0GS

Die Fortsetzung ist > 0,5 → also **aufrunden!**

Das Ergebnis ist äquivalent zu etwa 4,77, was auch für negative Zahlen mit gleichem Vorzeichen gilt.

Aber bei Subtraktion (unterschiedlichen Vorzeichen):

Hier gilt das **nicht**!

Beispiel:

Ergebnis ist 4,7, mit G = 1 und S = 1 →

Das ist **nicht** wie +4,7 + 0,0GS,

sondern entspricht + 4,7 - 0,0GS! (unterschiedliche Vorzeichen)

Für negative Zahlen ist es ähnlich:

Beispiel: -4,7 + 0,0GS entspricht - 4,7 + etwas Positives.

Lösung (bei Subtraktion):

Man kann **1 Einheit vom Bit links von G „ausleihen“** und so umschreiben:

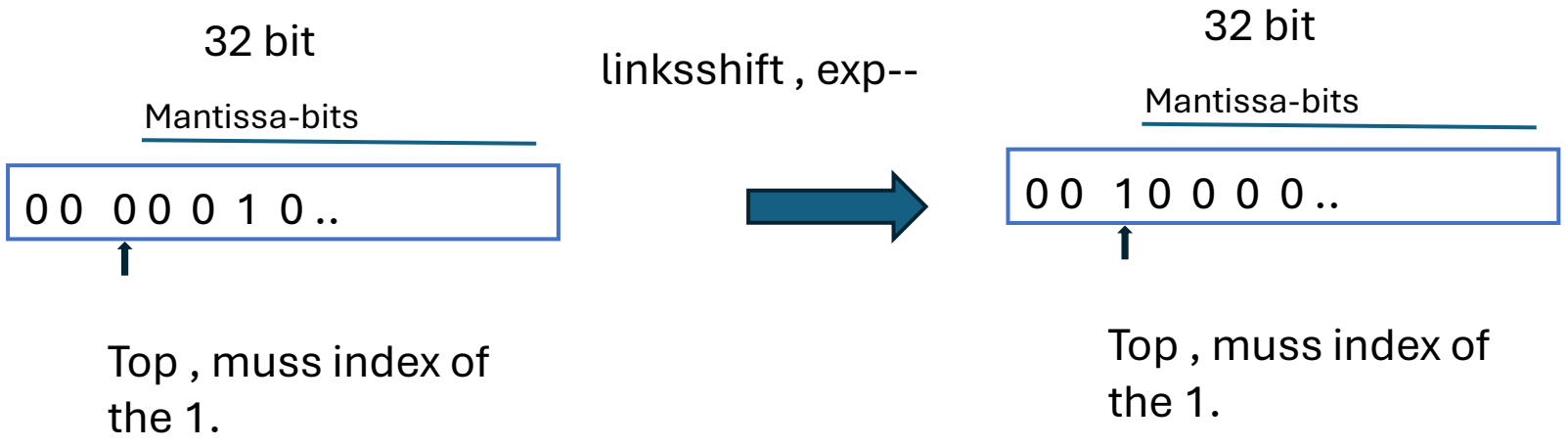
$4{,}7-0{,}0GS=4{,}6+0{,}1-0{,}0GS=4{,}6+0{,}G'S'$

Das funktioniert auch für negative Zahlen:

$-4{,}7+0{,}0GS=-4{,}6-0{,}1+0{,}0GS=-4{,}6-0{,}G'S'$

4. Normalizierung falls notwendig

Falls addition zweier Zahlen unterschiedlicher Zahlen stattfindet , und gerundet , kann weiterhin der leading 1 nicht an bit index mantissa-bits.



!Achtung : waehrend exp -- , auf underflow achten

Und was macht man, wenn man subtrahieren will?

Subtraktion ist im Grunde nichts anderes als eine Addition. Man kann also denselben Algorithmus wie zuvor verwenden – nur mit geändertem Vorzeichen:

$A - B = A + (-B)$

Jetzt einfach die Addition ausführen – so einfach ist es!

FMA

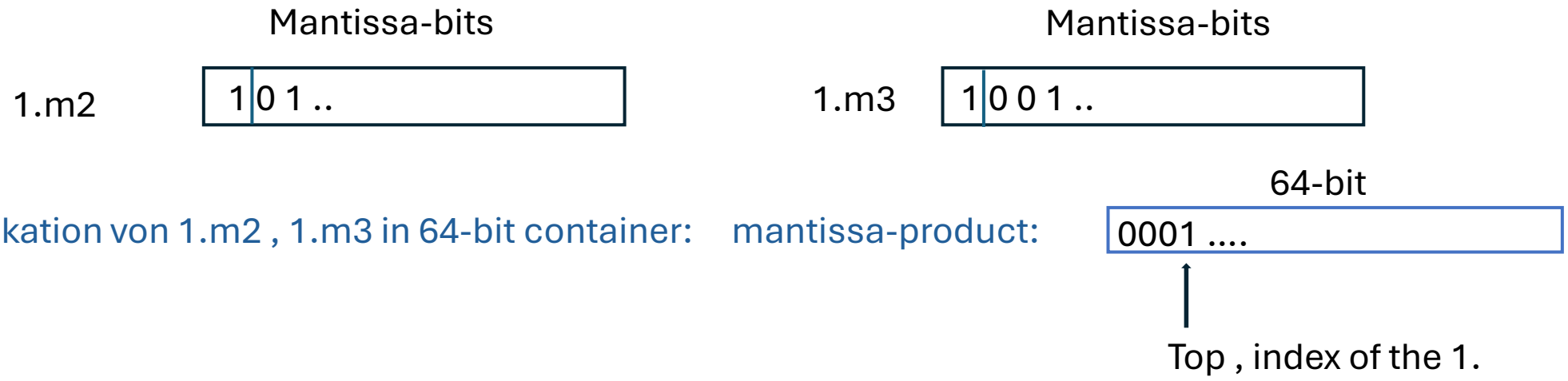
Idee:

$$a + (b \cdot c) = (1.M1 \cdot 2^{e1}) + \left((1.M2 \cdot 2^{e2}) \cdot (1.M3 \cdot 2^{e3})\right) = (1.M1 \cdot 2^{e1}) + (1.M2 \cdot 1.M3 \cdot 2^{e2+e3})$$

Die Hauptschritte sind:

1. Multiplikation von mantissen 1.m2 und 1.m3 in 64 bit-container und addierung deren Exponent
2. M1 in eine 64-bit form so zu bringen , dass index(leading 1 1.m1)= index(leading 1 1.m2*1.m3)
3. Links shift der mantissa ,bestimmen,G/S werden entsprechend gesetzt während Shift
4. Bestimmung des finalen Vorzeichens
5. Jenach Vorzeichengleichung zwischen s1 und (s2 xor s3) mantissen addieren oder subtrahieren
6. G/S bits in der 64-bit zwischenergebnis zu propagieren
7. Final sign , final exponent und 64-bit zwischenergebnis in 32 bit zusammenzusetzen
8. Anhand G/S runden

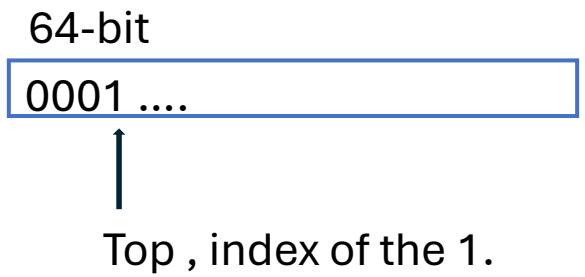
1.Multiplikation von mantissen 1.m2 und 1.m3 in 64 bit-container und addierung deren Exponent



Dann werden die echten Exponente addiert in product-exponent: product-exponent=unbiased e2+unbiased e3

2. M1 in eine 64-bit form so zu bringen , dass index(leading 1 1.m1)= index(leading 1 1.m2*1.m3)

bigM1:



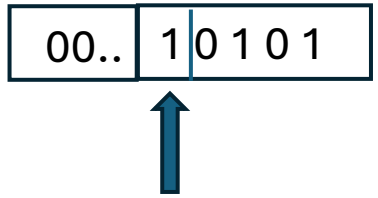
3. Links shift der mantissa ,G/S werden entsprechend gesetzt während Shift

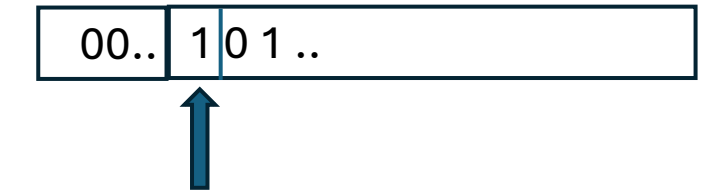
Beispiel:

Unbiased e1= 1

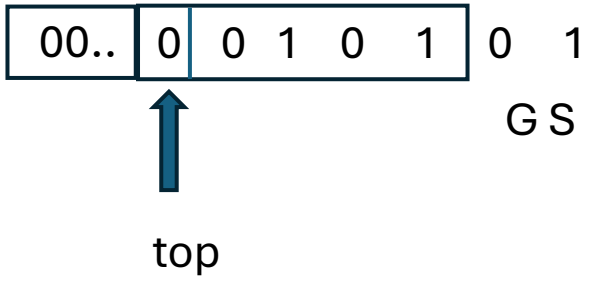
Product exponent= 3

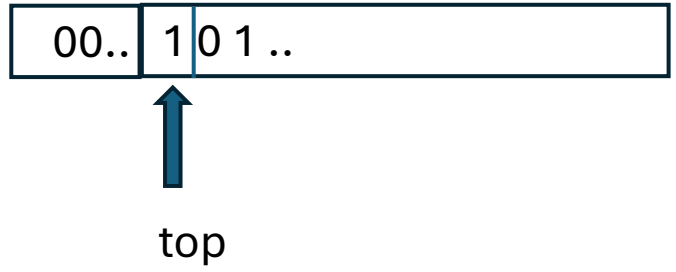
1.bigM1





Kleinere mantissa , weil geshiftet





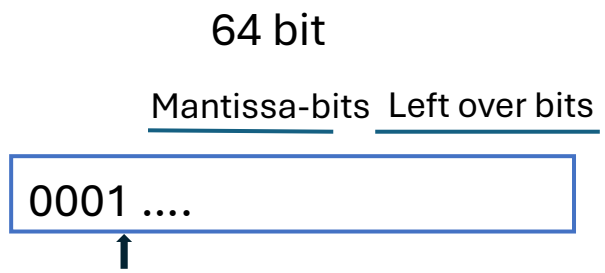
4. Bestimmung des finalen Vorzeichens:

Die logik hier ist einfach: wenn wir die größere Mantissa minus die kleinere machen , dann ist final exponent positive , an sonsten negativ . Das ist möglich weil jetzt die Exponente gleich sind . Wenn wir die beiden addieren , bleibt das exponent dasselbe (weil addieren in diesem Sinn bedeutet vorzeichen von r1 und (r2*r3) gleich

5.Je nach Vorzeichengleichung zwischen s1 und (s2 xor s3) mantissen addieren oder subtrahieren

Fall 1: Vorzeichen ist gleich

Einfach zwischenergebnisse addieren :

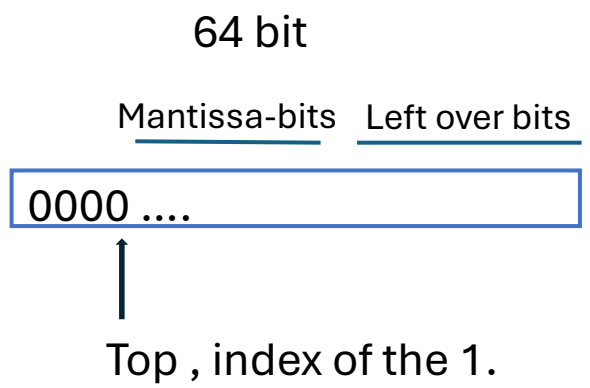


Top , index of the 1.

Achtung! Nach addition zwei zahlen kann eine zahl ergeben ,wo index(leading 1)=top +1.Indem Fall einfach final-exponent++ ,top ++(achten auf Overflow),top zeigt jetzt wieder auf leading 1

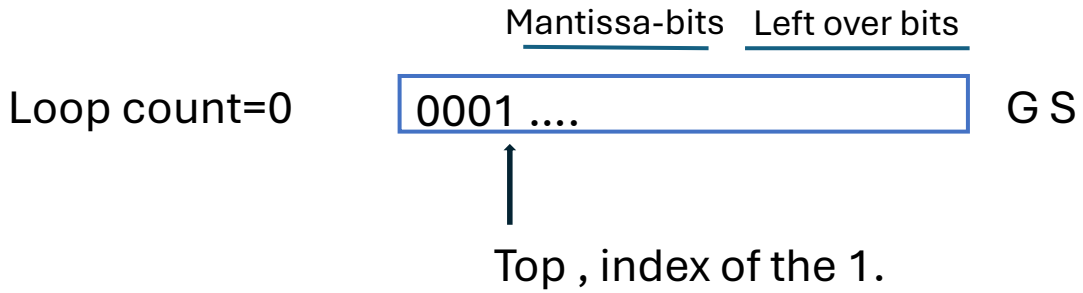
Fall 2: Vorzeichen ist ungleich

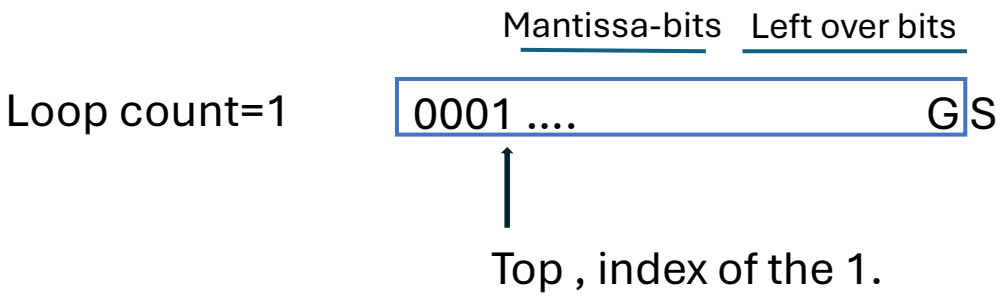
Einfach zwischenergebnisse von einander subtrahieren :

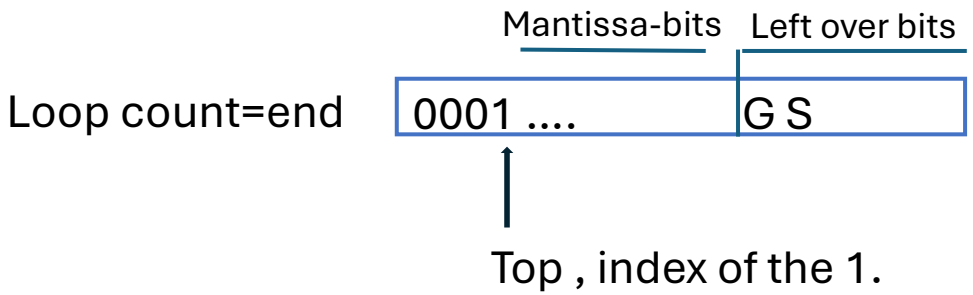


Achtung! Nach sub zwei zahlen kann eine zahl ergeben ,wo index(leading 1)<top (vielleicht gibt's nicht , dann 0 als Ergebnis) .Hier muss auch G/S anders gesetzt werden wie schon vorher gezeigt bei ungleiches Vorzeichen addition

6. G/S bits in der 64-bit zwischenergebnis zu propagieren:







7.runden :

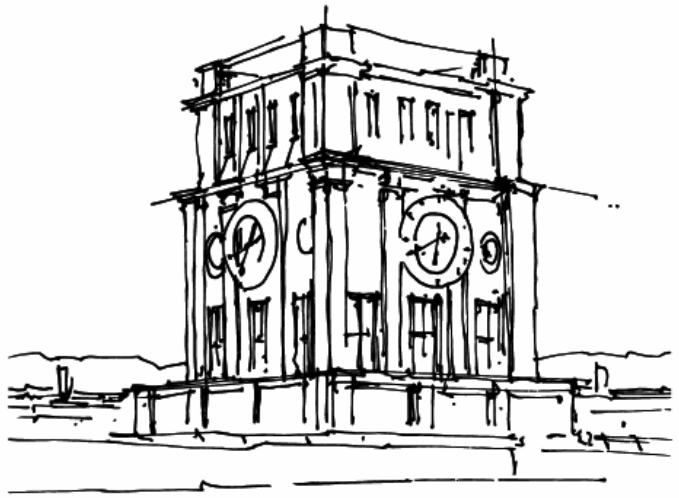
Jetzt findet die "rounding" statt.

Danach muss sichergestellt werden dass mantissa normalized ist (leading 1 an der stelle mantissa-bits) weil ungleiches Vorzeichen passieren kann .

Im ungleichen Vorzeichen Fall muss man links shiften , bis leading 1 an der gleichen Stelle ist , Final-exponent -- (auf Underflow achten).

III-Korrektheit & Testing

i. Testing des Inputs
ii. Testing Strategie & Tools
iii. Überprüfung der Ergebnisse mit unterschiedlichen Inputs



Testing der Nutzereingaben

Man muss sich überlegen, welche Patterns man akzeptieren muss und welche nicht.

das Ziel / die allgemeine Valide form :

./project [valid long options mit validen Argumente]* VALIDE eingabe.csv length <=13

Bemerkung:
Muss man nur genau diese Reihnfolge akzeptieren ?
Nein! Aber wie spaeter erklaert , kann man nur auf diesen Muster fokussieren

1)Einzige Annahme jetzt:

jetzt → Die Eingabe soll immer mit dem Names des ausführbaren Datei beginnen :

./project ……

2)Sicherstellen dass Anzahl an argumente maximal 13 sind (alle legalen short options + deren argumente + csv datei als positional argument)

jetzt →

./project -a b c … length <=13

3) Da getopt eingabe immer so sortiert , dass ./project an der erstellen Stelle ist , long/short options direkt danach , und positional args am ende , reicht es auf diesem Pattern zu fokussieren:

jetzt →

./project [long/shortoptions]* [positional arguments] * , length <=13

4) getopt muss nur long options unterstuetzen:

jetzt →

./project [long options] *[positional arguments]* , length <=13

5)muss sichergestellt werden , dass long options valid sind:

Optionen müssen in der Option-Struktur definiert sein.

Options Argumente sind valid.

Abwesenheit von manchen Options sollen sinnvoll behandelt werden.

jetzt → **./project [valid long options mit validen Argumente]* [positional arguments]*** **length <=13**

6) muss sichergestellt werden, dass positional arguments length=1 hat , und dass diese positional arguement der .csv ist

jetzt → **./project [valid long options mit validen Argumente]*eingabe.csv** **length <=13**

6) Sicherstellen dass die eingabe datei eine valid .csv ist

jetzt → **./project [valid long options mit validen Argumente]* VALIDE eingabe.csv** **length <=13**

Ziel erreicht!

Bemerkung : Falls sich `--help` irgendwo in den Eingabeargumenten befindet, kann man – nach der Prüfung der Argumentanzahl – gezielt danach suchen und gegebenenfalls eine passende Hilfe-Nachricht ausgeben.

Testing Strategie & Tools

Integration Testing :

Alle Submodule wurden vor der Integration in das Projekt getestet.

System Testing:

Nach der Integration aller Komponenten begannen wir mit dem Testen des gesamten Projekts. Dieser Systemtest umfasste nicht nur den funktionalen Teil des Projekts, sondern auch dessen Leistung und Struktur.

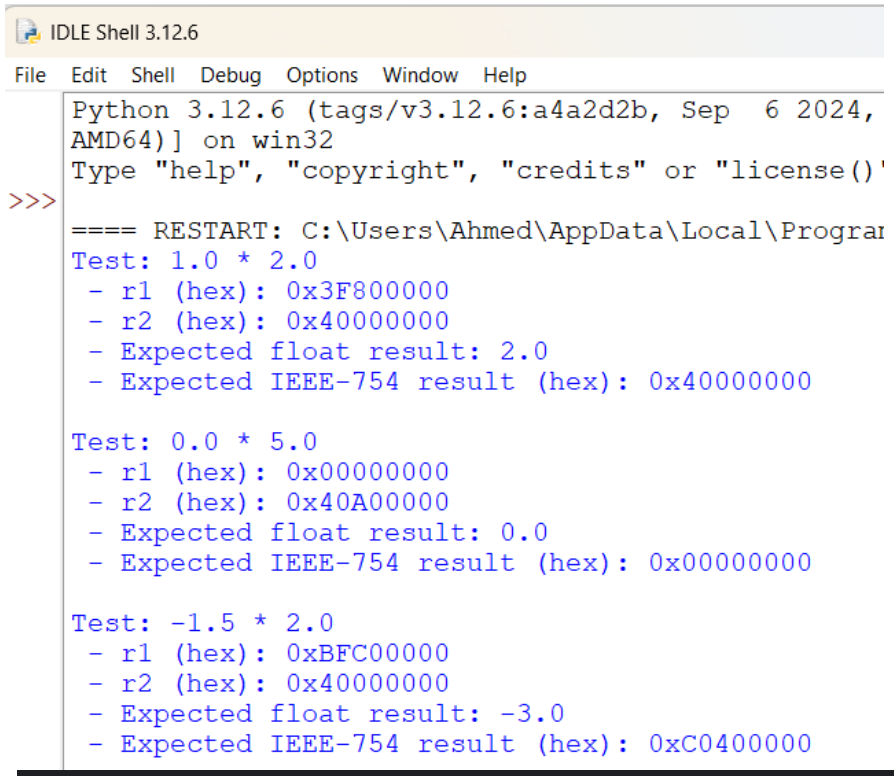
Ziel: Feststellen, ob das System die Anforderungen (funktional und nicht-funktional) erfüllt.

Verwendete Tools:

Für die Generierung der Fälle : Python, KI und verschiedene Websites (Z.B Numeral Systems Calculators),...

Für das Debugging & das Überprüfen der Ergebnisse :

Python, Tracefiles, .csv Dateien, Konsolenausgabe,...



test.csv

```
op,r1,r2,r3
8, 1.0, 1065353216
9, 0x3F800000, 1.0
10, 0x40490FD0, 0x402DF84D
13, 0x3F800000, 0x40000000
14, 0x3F800000, 0x40000000
15, 0x7fc00000, 0x3f800000, 0x40000000
15, 0x3E000000, 0x40200000, 0x3F000000
```

```c
printf("Simulation results:\n");

for (int i = 0; i < r.cycles; i++)
{
    printf("0x%08" PRIx32 " ", requests[i].ro);
}

printf("\nCycles: %u\nSigns: %u\nOverflows: %u\nUnderflows: %u\nInexact: %u\nNaNs: %u\n",
       r.cycles, r.signs, r.overflows, r.underflows, r.inexacts, r.nans);
```

```
$ ./project --size-exponent 8 --size-mantissa 23 test/test.csv
```

```
Trying to open: test/test.csv
Simulation results:
0x40000000 0x00000000 0x4108a2b3 0x3f800000 0x40000000 0x7f800001 0x3fb00000
Cycles: 7
Signs: 0
Overflows: 0
Underflows: 0
Inexact: 1
NaNs: 1
```

Überprüfung der Ergebnisse mit unterschiedlichen Inputs

- Einer der anspruchsvollsten Teile des Projekts bestand darin, alle möglichen Testfälle zu testen, einschließlich **Sonderfällen**, in denen die Eingangssignale **Inf Nan oder 0** sein können.

- Außerdem war es entscheidend, jedes Mal festzustellen, ob nach der Ausführung einer Operation **ein Überlauf oder ein Unterlauf** auftritt.

Bemerkungen:

- 0 wird durch Mantisse und Exponenten von 0 dargestellt.
- +INF und -INF werden mit einem Exponenten aus nur Einsen und einer Mantisse aus nur Nullen dargestellt.
- Nan wird mit einem Exponenten aus nur Einsen und einer beliebigen Mantisse *!*= 0 dargestellt.

Wichtigste betrachtete Testfälle:

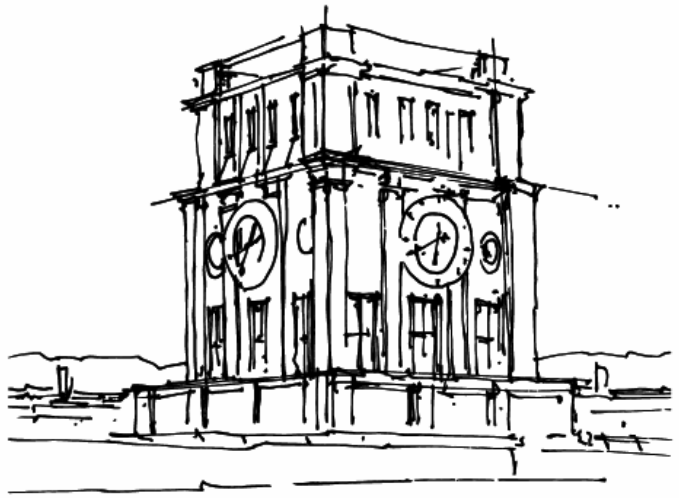
OP	Fall	R1 - R2	Ergebnis
Mul	Nan/inf special cases Nan/inf special cases Basic Big numbers , no overflow : Small numbers , no underflow: Overflow Underflow	0x7F7FFFFF 0x40000000 0x7F800000 0x00000000 0x3F800000 0x40000000 0x7F7FC99E 0x3DCCCCCD 0x00800000 0x40000000 0xFF7FFFFF 0x40000000 0x006CE3EE 0x2EDBE6FF	0x7F800000 0xFFC00000 0x40000000 0x7DCCA14B 0x01000000 0xFF800000 0x00000000
Add/Sub	Nan/inf special cases Nan/inf special cases Basic Same sign with rounding Different sign Overflow Underflow	7f800000 7f800000 7f800010 ff800000 20000002 20000006 7b000003 7b800000 3fa00000 bfa00000 7f7fffff 7e7ffffe 80000002 00000001	7f800000 7f800001 20800004 7bc00002 00000000 7f800000 00000000

OP	Fall	R1 - R2- R3	Ergebnis
FMA	Nan/inf special cases Nan/inf special cases Basic Big numbers , no overflow : Small numbers , no underflow: Overflow Underflow	7FC00000 3F800000 40000000 00000000 7F800000 3F800000 3D800000 3E800000 3E000000 47f03c40 461c4300 45f69a00 BF800000 3F802000 3F800000 7F000000 5F800000 5F000000 00800001 80800000 3f800000	7F800001 7F800000 3DC00000 4c96c271 3A800000 7F800000 00000000
Max	Gleiches Verzeichen Ungleiches Verzeichen	0x3F800000 0x40000000 0xBFC00000 0x3F800000	0x40000000 0x3F800000
Min	Gleiches Verzeichen Ungleiches Verzeichen	0x3F800000 0x40000000 0xBFC00000 0x3F800000	0x3F800000 0xBFC00000

Wir haben auch verschiedene Operationen mit großen und kleinen Zahlen, Rundungsarten und Operationen mit unterschiedlicher Mantissen- und Exponentengröße getestet.

IV-Evaluation des Projekts

i. Verbesserte Leistung von dem FPU
ii. Speichersicherheit
iii. Vergleich mit anderen Lösungen der realen Welt



Wie funktioniert genau das FPU ?

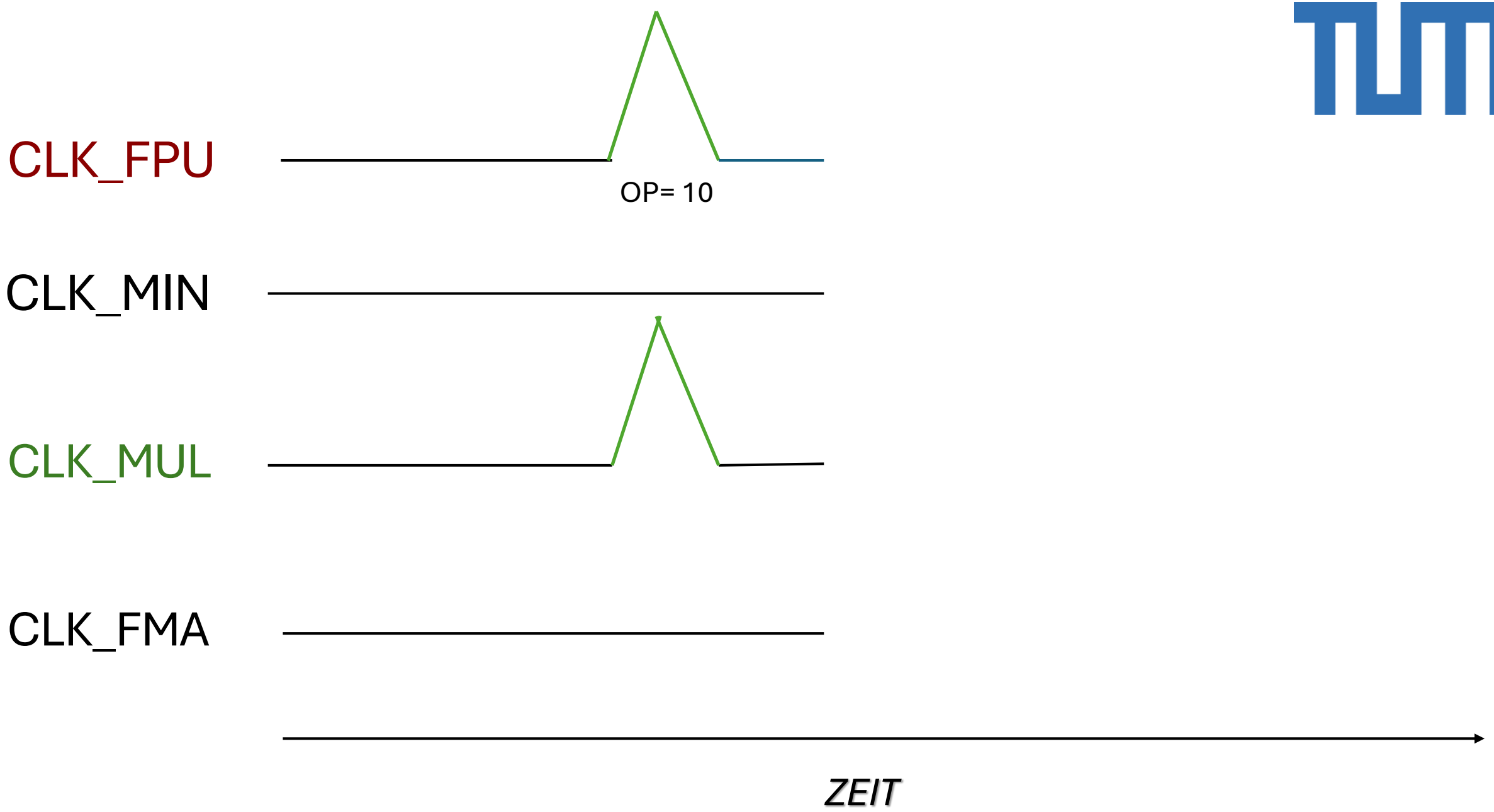
- Jedes SystemC SubModul (Min,Max,AddSub, Mul,FMA) hat seine eigene Clock-Leitung (clk_min,clk_max,..).

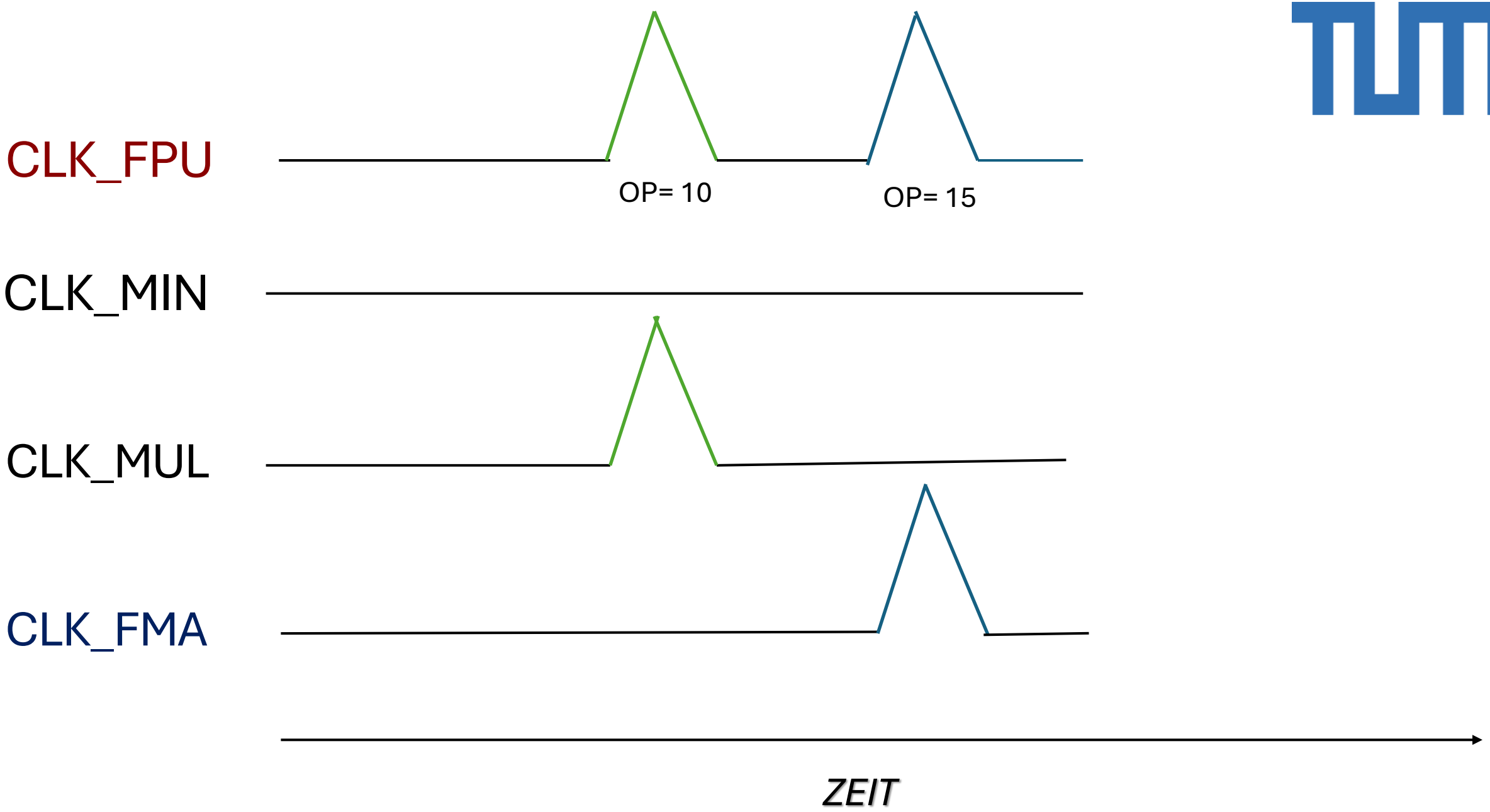
- Statt alle SubModul zu aktivieren und das Ergebnis danach anhand dem 4-Bit-Opcode auswählen , wird nur der Takt für das benötigte Modul aktiviert, alle anderen bleiben im Leerlauf.

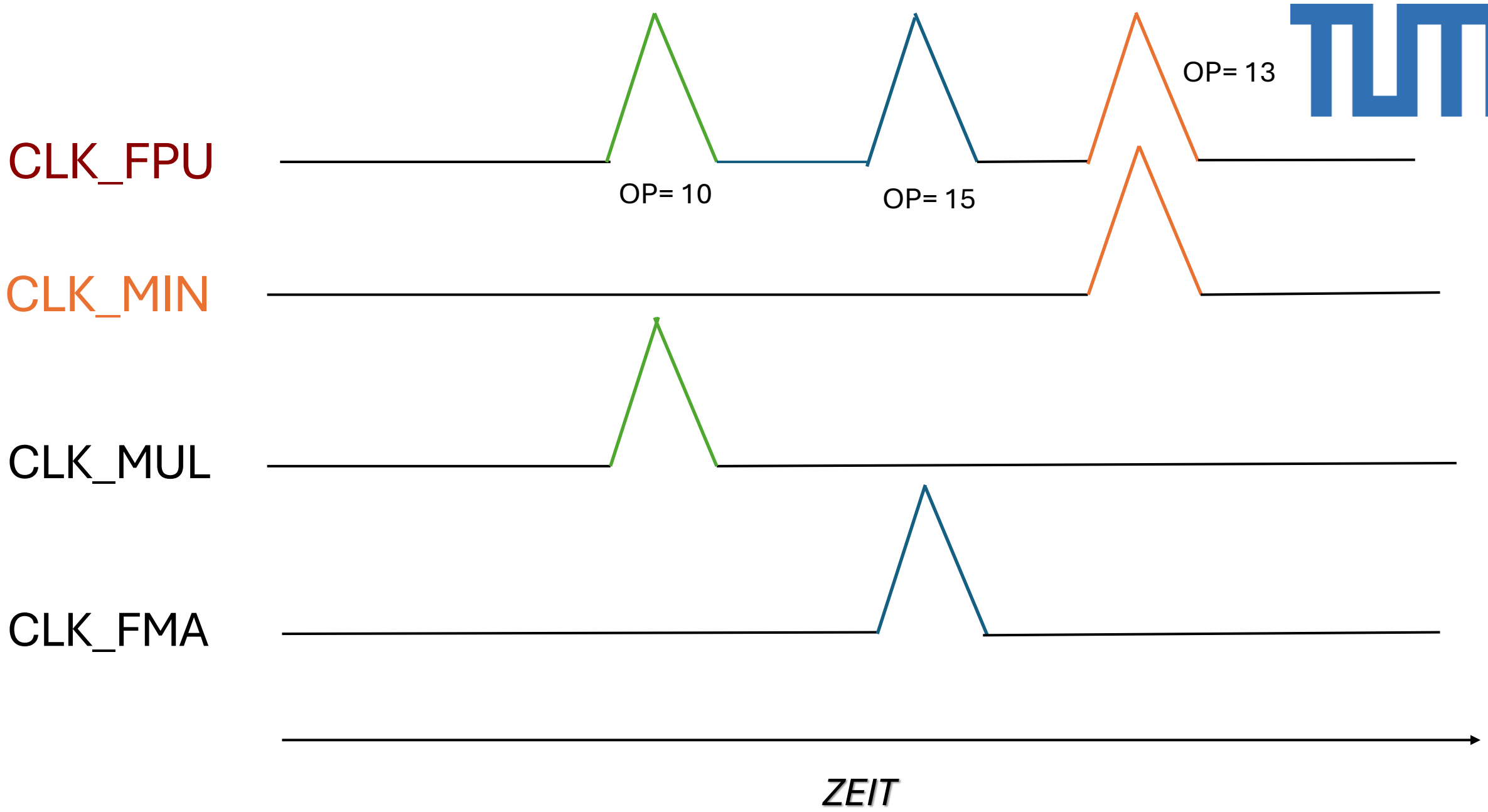
- Das spart Energie und schont Ressourcen.

Beispiel


CLK_FPU
CLK_MIN
CLK_MUL
CLK_FMA
ZEIT


CLK_FPU
OP= 10
CLK_MIN
CLK_MUL
CLK_FMA
ZEIT


CLK_FPU
OP= 10
OP= 15
CLK_MIN
CLK_MUL
CLK_FMA
ZEIT


CLK_FPU
CLK_MIN
CLK_MUL
CLK_FMA
OP= 10
OP= 15
OP= 13
ZEIT

Speichersicherheit:

Für jedes Nutzereingabe-Pattern, das wir bereits beim Testing spezifiziert haben, haben wir AddressSanitizer und Valgrind verwendet, um verschiedene Speicherprobleme zu vermeiden – wie Buffer Overflows, Use-after-Free und viele weitere.
Zum Beispiel kann man für AddressSanitizer einfach den entsprechenden Flag im Makefile hinzufügen; dadurch wird das Programm mit `make asan` auf Speicherlücken überprüft

Beispiel:
make asan
./project test/test.csv

Output: "Simulation results:
0x40000000 0x00000000 0x4108a2b3
0x3f800000 0x40000000 0x7f800001 0x3fb00000
Cycles: 7
Signs: 0
Overflows: 0
Underflows: 0
Inexact: 1
NaNs: 1"

Vergleich mit anderen Lösungen der realen Welt

- Eine interessante Metrik zum Vergleich unserer Implementierung des FMA mit anderen Lösungen ist die Verwendung der Anzahl der benötigten Gates

Source/Company	Precision	Geschätzte Gate-Anzahl	Anmerkungen
Unsere FMA	Single	~60k–80k	Grundlegende Implementierung
Intel (Haswell/Skylake)	Double	~200k–300k	Optimiert für Durchsatz, Pipelines in voller Breite
IBM POWER9/10	Double	400k+	Aggressives SIMD mit hoher Präzision
Nvidia GPU SM core	FP32 FMA	~60k–100k (per lane)	Tausende parallele Einheiten

Quellen:

- Is there a document about in which hardware unit(ie. ALU FMU...) an instruction is executed? - CUDA / CUDA Programming and Performance - NVIDIA Developer Forums

- Microsoft Word - Enhanced Floating-Point Multiply-Add with Full Denormal Support.docx

- IBM Power microprocessors - Wikipedia

V- Zusammenfassung & Ausblick

Nun haben wir ein solides Verständnis dafür entwickelt was FMA ist , wie eine FPU aufgebaut ist,

wofür es verwendet wird und wie unserem Code funktioniert.

Auch wenn unsere Implementierung auf 32-Bit Gleitkommazahlen basiert, haben wir einen Blick in die Tiefe geworfen:
Was wäre, wenn wir in die Welt der 64-Bit Floating-Point Units eintauchen?

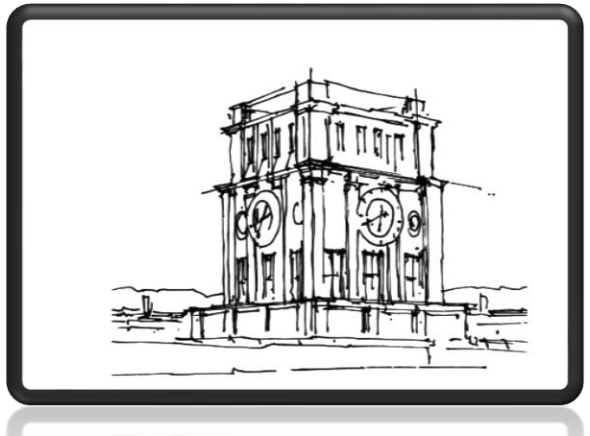
64bit

Doppelte Genauigkeit bringt:

- Noch mehr Präzision (besonders auch mit der Behandlung von Subnormals)
- Aber auch mehr Komplexität: breitere Register, größere Exponenten, aufwändigere Rundung

Ein spannender Schritt – vielleicht fürs nächste Projekt.

- Danke für die Aufmerksamkeit, und wir hoffen dass unsere Presentation euch gefällt hat ! 🌷



Ahmed Amine Loukil
Nourallah Gargouri
Saifeddine Guenanna

Gruppe T104