

Tutorium zu Computer-Engineering im SS19

Termin 2

Jakob Otto

HAW Hamburg

17. Oktober 2019

Ablauf

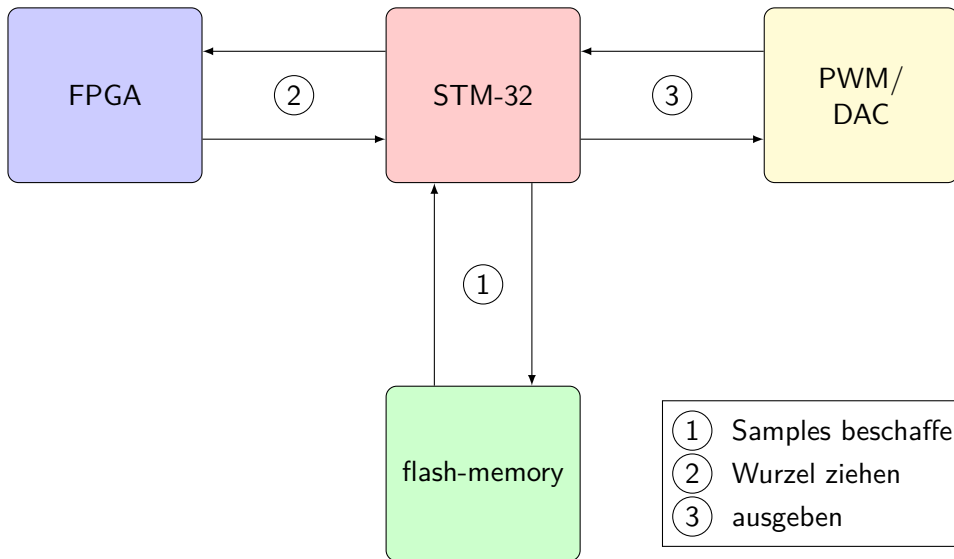
- Ausblick
- Praktikumsaufgabe
 - ▶ Trial-subtraction-Verfahren
 - ▶ Tipps
 - ▶ Testen



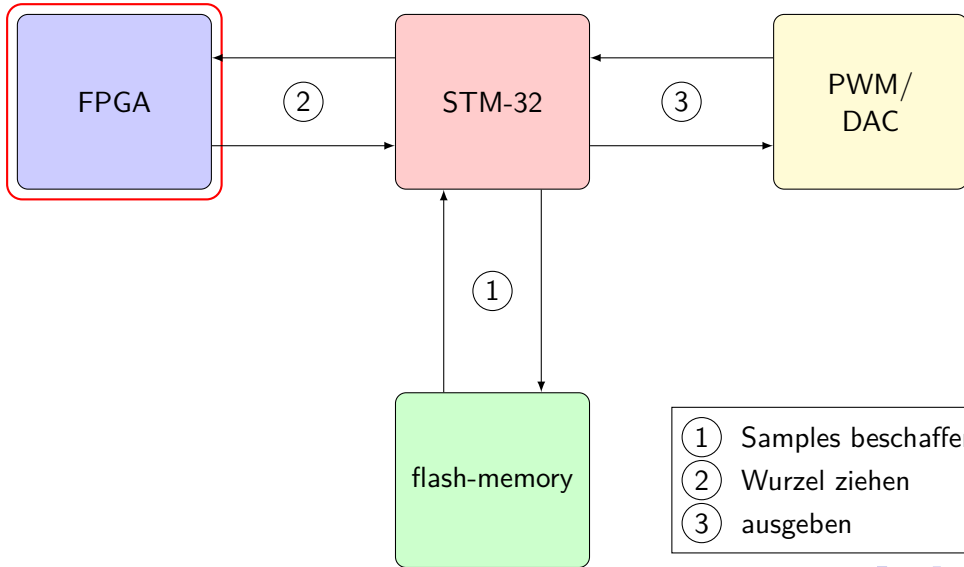
Ablauf

- ➊ Trial-Subtraction Algorithmus
 - ▶ effizientes Wurzelziehen aus Samples
- ➋ DAC spielereien
 - ▶ Ausgabe von Sound lernen
- ➌ Flash-speicher lesen/schreiben
 - ▶ Samples lesen lernen
- ➍ Alles zusammensetzen
 - ▶ Kommunikation zwischen FPGA/STM-32
 - ▶ Ausgabe übr PWM
 - ▶ Musik abspielen

Ausblick (II)



Ausblick (III)



Aufgabenzettel

Warum eigentlich?

- Wurzelziehen ist teuer
- Hardware-unterstützung hilfreich
- Bei uns durch Trial-Subtraction Algorithmus realisiert

Warum eigentlich?

Table 8. Some floating-point single-precision data processing instructions

Instruction	Description	Cycles
VABS.F32	Absolute value	1
VADD.F32	Addition	1
VSUB.F32	Subtraction	1
VMUL.F32	Multiply	1
VDIV.F32	Division	14
VCVT.F32	Conversion to/from integer/fixed-point	1
VSQRT.F32	Square root	14

Einschränkungen...

- Wir sind Stückkosten getrieben!
- Daher ein Addierer und ein Subtrahierer erlaubt
 - ▶ Addierer für Inkrementieren des Zustands
 - ▶ Subtrahierer für Algorithmus

Denkt an Aufgabe 3 aus DT → Addierer richtig einsetzen!

Kleine Erinnerung (I)

SO NICHT!!

```
add: process (a_s, b_s, c_s, select_s) is
    variable res_v : <type>;
begin
    if (select_s = '0') then
        res_v := a_s + b_s;
    else
        res_v := b_s + c_s;
    end if;

    res_s <= res_v;
end process;
```

Kleine Erinnerung (II)

BESSER

```
add: process (a_s, b_s, c_s, select_s) is
    -- variablen opA_v, opB_v, res_v
begin
    if (select_s = '0') then
        opA_v := a_v;
        opB_v := b_v;
    else
        opA_v := b_v;
        opB_v := c_v;
    end if;

    res_v := opA_v + opB_v;
    res_s <= res_v;
end process;
```

Trial-Subtraction Algorithmus (I)

Ablauf

- ① $n = 1$
- ② r' und s' berechnen
- ③ wenn:
 - ① $r' \geq 0 \rightarrow s = s'$
 - ② $r' < 0 \rightarrow s = s$
- ④ sobald:
 - ① $r' = 0 \rightarrow \text{ende}$
 - ② sonst $\rightarrow n = n + 1$ und gehe zu 2

Trial-Subtraction Algorithmus (II)

Variablen

- $s \rightarrow$ Approximation der Wurzel
- $s' \rightarrow$ die neue Approximation
- $r \rightarrow$ der mögliche Rest
- $r' \rightarrow$ ein möglicher neuer Rest
- $n \rightarrow$ Der Zustand/die Phase
- $d \rightarrow$ Differenz $\rightarrow 2^n$

Trial-Subtraction Algorithmus (II)

Funktionen

$$r' = r - 2^n(2s + 2^n)$$

$$s' = s + 2^n$$

Trial-Subtraction Algorithmus (III)

Ursprüngliche Funktion

$$r' = r - 2^n \cdot (2s + 2^n)$$

Multiplikation mit 2 lässt sich durch shift darstellen
Addition in diesem Fall durch '|'

Vereinfacht

$$r' = r - ((s \ll 1) | (1 \ll n)) \ll n$$

Trial-Subtraction Algorithmus (IV)

Ursprüngliche Funktion

$$s' = s + 2^n$$

Multiplikation mit 2 lässt sich durch shift darstellen
Addition in diesem Fall durch '|'

Vereinfacht

$$s' = s | (1 \ll n)$$

Trial-Subtraction Algorithmus (V)

Bei uns etwas anders...

- Wir ziehen wurzel aus $Q_{0.15}$ Format
 - ▶ Zahlenbereich von $-1 \dots \sim 1$
 - ▶ In der Rechnung nur $0 \dots 1$
- Logik daher etwas anders
 - ▶ Statt $2^n \rightarrow 2^{-n}$
 - ▶ $0.5, 0.25, 0.125, \dots$

Trial-Subtraction Algorithmus (VI)

Eigentlich

$$\begin{aligned}r' &= r - (((s \ll 1) \mid (1 \ll n)) \gg n) \\s' &= s \mid (1 \ll n)\end{aligned}$$

Wird zu

$$\begin{aligned}r' &= r - (((s \ll 1) \mid (10000000000000000 \gg n)) \gg n) \\s' &= s \mid (10000000000000000 \gg n)\end{aligned}$$

Trial-Subtraction Algorithmus (VII)

Eigentlich

$$r' = r - (((s \ll 1) \mid (10000000000000000 \gg n)) \gg n)$$
$$s' = s \mid (10000000000000000 \gg n)$$

lässt sich noch verbessern

$$d = (10000000000000000 \gg n)$$
$$r' = r - (((s \ll 1) \mid d) \gg n)$$
$$s' = s \mid d$$

Trial-Subtraction Algorithmus (VIII)

Als Codebeispiel...

```
delta_v := to_stdlogicvector("1000000000000000" srl n_v);
operandB_v := to_stdlogicvector(to_bitvector(
    (s_v(msbPos-1 downto 0) & '0') or (delta_v)) srl n_v);
operandA_v := r_v;
result_v := opA_v - opB_v;
-- weiter interpretieren
```

Tipps (I)

Kein Modularisieren

- Versucht **NICHT** den Code modular zu gestalten
 - ▶ Ein Prozess, der die gesamte Logik enthält
- Modularisieren ist gut, allerdings 40 Zustände dadurch schwer

Tipps (II)

Wichtig!

- Bevor ihr eine Berechnung startet (Phase 0):
 - ▶ Eingabe auf VZ prüfen und ggf. positiv machen. $\rightarrow (0 - \text{Wert})$ **VZ merken!**
- Nach der Berechnung Ergebnis wieder Negativ machen $\rightarrow (0 - \text{Ergebnis})$

Tipps (III)

Eingaben

- Was tun mit Eingaben wie:
 - ▶ -1 → "10000000000000000"
 - ▶ 0 → "00000000000000000"

Tipps (III)

Eingaben

- Was tun mit Eingaben wie:
 - ▶ $-1 \rightarrow "100000000000000000"$
 - ▶ $0 \rightarrow "000000000000000000"$

Einfach durchreichen

- Was ist mit der 1???

Testen

```
for i in -32768 to 32767 loop
    x_s <= std_logic_vector(to_signed(i, x_s'length));
    req_s <= '1';

    -- warten, dass Berechnung gestartet wurde
    wait for fullClockCycle;
    req_s <= '0';

    -- warten auf Ende der Berechnung
    wait for 18 * fullClockCycle;
end loop;
```

Ende

Nächstes mal

- Aufgabe 3?
- Andere Wünsche?

Abschließendes

Fragen, Anmerkungen und Verbesserungen ausdrücklich erwünscht.
Ich bin auf euer Feedback angewiesen.