ASIC Implementierung einer Zweidimensionalen Flugbahnberechnung mittels Heun-Verfahren und Stokes Reibung

Julius Gilka-Bötzow

WiSe 2021-22

Zusammenfassung

In diesem Beleg wird der Entwurfs- und Testprozess eines Application-specific integrated circuit (ASIC) dargestellt. Entstanden ist das Ganze als Praktikum im Fach Schaltkreis- und Systementwurf an der TU Dresden.

Inhaltsverzeichnis

1 Aufgabenstellung

Der Rahmen für die mir selbst gestellte Aufgabe ist wie folgt.

Ich wurde von einem Rüstungsunternehmen gebeten einen ASIC für schnelle Flugbahnberechnung zu entwerfen. Dieser soll in den Computer eines Artillerie-Geschützes eingebaut werden.

Da es sich um das Militär als Kunden handelt, sind Kosten und somit Fläche keine Limitation. Fokus liegt auf der Geschwindigkeit, Genauigkeit und Energieeffizienz, weil es möglicherweise mit einer Batterie betrieben wird.

Damit der Chip mit dem Rest des Systems funktionieren kann, gibt es folgende E/A-Vorgaben.

Eingegeben werden eine Startposition und -geschwindigkeit (jeweils mit x und y Koordinaten), der Radius und die Masse des Geschosses, sowie die Länge der Zeitschritte um die Berechnungsgenauigkeit anpassen zu können.

Ausgegeben wird eine Liste von Werten mit x und y Postionen, die in einen dem ASIC externen Speicher zur späteren weiterverarbeitung gespeichert werden. Der Zeitstempel ist implizit in der Stelle des Elementes im Speicher.

Um diese Aufgabe zu lösen habe ich zuerst eine Implementierung in Rust geschrieben und diese Anschließend überarbeitet um sie auf einen ASIC übertragen zu können.

2 Programmablauf

2.1 Initiale Version

Das Programm nutzt den Heun-Algorithmus, der Schrittweise von einem Startwert und seiner Ableitung eine Kurve aus Punkten entwickelt. Diesen habe ich überlagert mit dem Gesetzt von Stokes um die Luftreibung zu simulieren.

In der ersten Version (Listing 1) sind keine Optimierungen vorgenommen. Sowohl für die Anzahl der Operationen als auch die Umsetzbarkeit für ASICs.

```
        // Constants will be read from memory
        const PI: f64 = 3.142;
        const VISC: f64 = 0.000018215;
        // Dynamic Viscosity of air at 20 degree Celsius in kg/(m*s)

        // position: tuple of x,y in meters
        // velocity: tuple of x,y in meters per second
        // radius: value in meters
        // delta_t: value in seconds for timesteps
        // mass: value in kilogram
        // return: Vec of tuple of pos_x, pos_y, vel_x, vel_y, time
        pub fn get_list(position: (f64, f64), velocity: (f64, f64),
            radius: f64, delta_t: f64, mass: f64) -> Vec<(f64, f64, f64, f64, f64)>{
            // Set initial values
            let mut rx = position.0;
            let mut ry = position.1;
            let mut vx = velocity.0;
            let mut vy = velocity.1;
            let mut t = 0.0;
            let dt = delta_t;
            let gx = 0.0;
            let gy = -9.81;

            // Save initial values in new Vector
            let mut vec = vec![(rx, ry, vx, vy, t)];
            loop {
                // Calculate next velocity with gravity and stokes
                let next_vx = vx + (gx -6.0*PI*radius*VISC * vx / mass)*dt;
                let next_vy = vy + (gy -6.0*PI*radius*VISC * vy / mass)*dt;
                // Calculate next position with median of speeds
                let newrx = rx + (vx + next_vx)/2.0 * dt;
                let newry = ry + (vy + next_vy)/2.0 * dt;
                // Assign new values
                t += dt;
                rx = newrx;
                ry = newry;
                vx = next_vx;
                vy = next_vy;
                // break when reaching the "ground" aka y==0
                if ry < 0.0{
                    break;
                }
                vec.push((rx, ry, vx, vy, t));
            }
            return vec;
        }

```

Listing 1: Initialer Code.

Zu beachten ist hierbei, dass ist den Vector anschließend in eine Textdatei geschrieben habe und diese anschließend mit R geplotted. Dazu ist die Zeit als weitere Ausgabe notwendig.

Der Zweck des Plottens war es zu sehen ob der Algorithmus das erwartete Resultat bringt und ob die Reibung einen merkbaren Einfluss hat oder vernachlässigt werden kann.

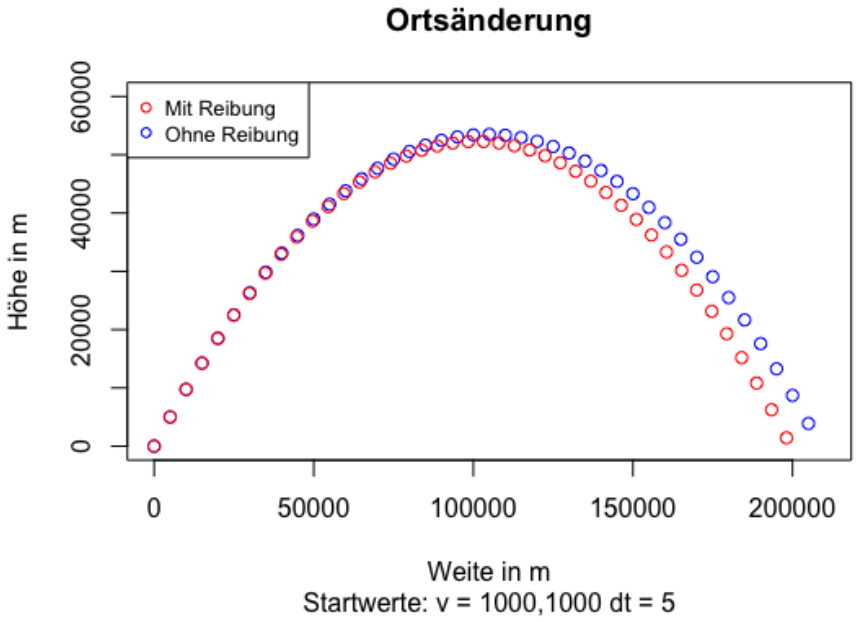

Abbildung 1: Plot des Initialen Algorithmus.

2.2 Verbesserte Version

Anschließend habe ich einige Optimierungen vorgenommen. Als erstes habe ich Zeile Code aufgeteilt um jeweils nur eine Operation pro Zeile zu haben. Dabei ist mir aufgefallen, dass einige Operationen vor den Loop gebracht können werden, wenn man die Gleichungen etwas umstellt. Dadurch entstehen mehr Operationen am Start aber pro Loop-Durchlauf werden Operationen gespart.

Nach einigen Experimenten mit Festkommadarstellungen um Platz zu sparen, habe ich mich entschieden bei der floating point Darstellung zu bleiben. Hauptgrund war die Kundenvorgabe der Genauigkeit aber auch die einfachere Umsetzung war ein Faktor.

```rust
const PI: f64 = 3.142;
const VISC: f64 = 0.000018215;

const STOKES: f64 = 0.000343345;

pub fn get_list(position: (f64, f64), velocity: (f64, f64),
                radius: f64, delta_t: f64, mass: f64)
                -> Vec<(f64, f64, f64, f64, f64)>{

    let mut rx = position.0;
    let mut ry = position.1;
    let mut vx = velocity.0;
    let mut vy = velocity.1;
    let mut t = 0.0;

    // Helpful "constants"
    let stokes_const_1 = STOKES * radius;
    let stokes_const_2 = stokes_const_1/mass;
    let stokes_const = stokes_const_2 * delta_t;
    let earth_speed_y = -9.81 * delta_t;
    let half_step = delta_t/2.0;

    // Save initial values in new Vector
    let mut vec = vec![(rx, ry, vx, vy, t)];
    loop {
        // Calculate next velocity with gravity and stokes
        let next_vx_1 = stokes_const * vx;
        let next_vx = vx - next_vx_1;
        let next_vy_1 = vy + earth_speed_y;
        let next_vy_2 = stokes_const * vy;
        let next_vy = next_vy_1 - next_vy_2;
        // Calculate next position with median of speeds
        let newrx_1 = vx + next_vx;
        let newrx_2 = newrx_1 * half_step;
        let newrx = rx + newrx_2;
        let newry_1 = vy + next_vy;
        let newry_2 = newry_1 * half_step;
        let newry = ry + newry_2;
        // Assign new values
        t += delta_t;
        rx = newrx;
        ry = newry;
        vx = next_vx;
        vy = next_vy;
        // break when reaching the "ground" aka y==0
        if ry < 0.0{
            break;
        }
        vec.push((rx, ry, vx, vy, t));
    }
    return vec;
}

```

Listing 2: Optimierter Code.

Um zu überprüfen ob der Code das selbe verhalten hat, habe ich das Ergebnis wieder als Plot dargestellt.

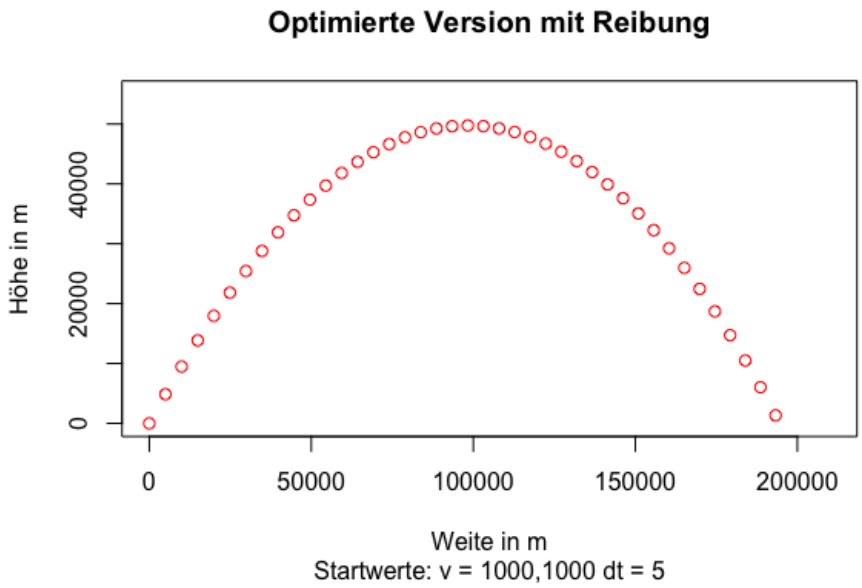

Abbildung 2: Plot des Optimierten Algorithmus.

Da das Verhalten des Optimierten Algorithmus das gleiche ist das des initialen ist, nutze ich den optimierten Code als Grundlage für das weitere Vorgehen. Namentlich den DFG im nächsten Abschnitt.

3 Datenpfadarchitektur

3.1 Datenflussgraph

Der DFG ...

3.2 Register-Transfer-Folge

Diese erste Version der RT-Folge besteht aus LOAD, PREP, LOOP, FULL und END. Bei LOAD werden am Anfang die nötigen Daten aus dem Speicher geladen. PREP bezeichnet die Takte nach dem Lade aber vor dem Beginn des Loop. LOOP meint den Anfang des Loops, wenn die Pipeline noch nicht voll gelaufen ist. FULL sind zwei abwechselnde States, bei denen die Pipeline schon voll gelaufen ist. END ist ein simpler Endzustand.

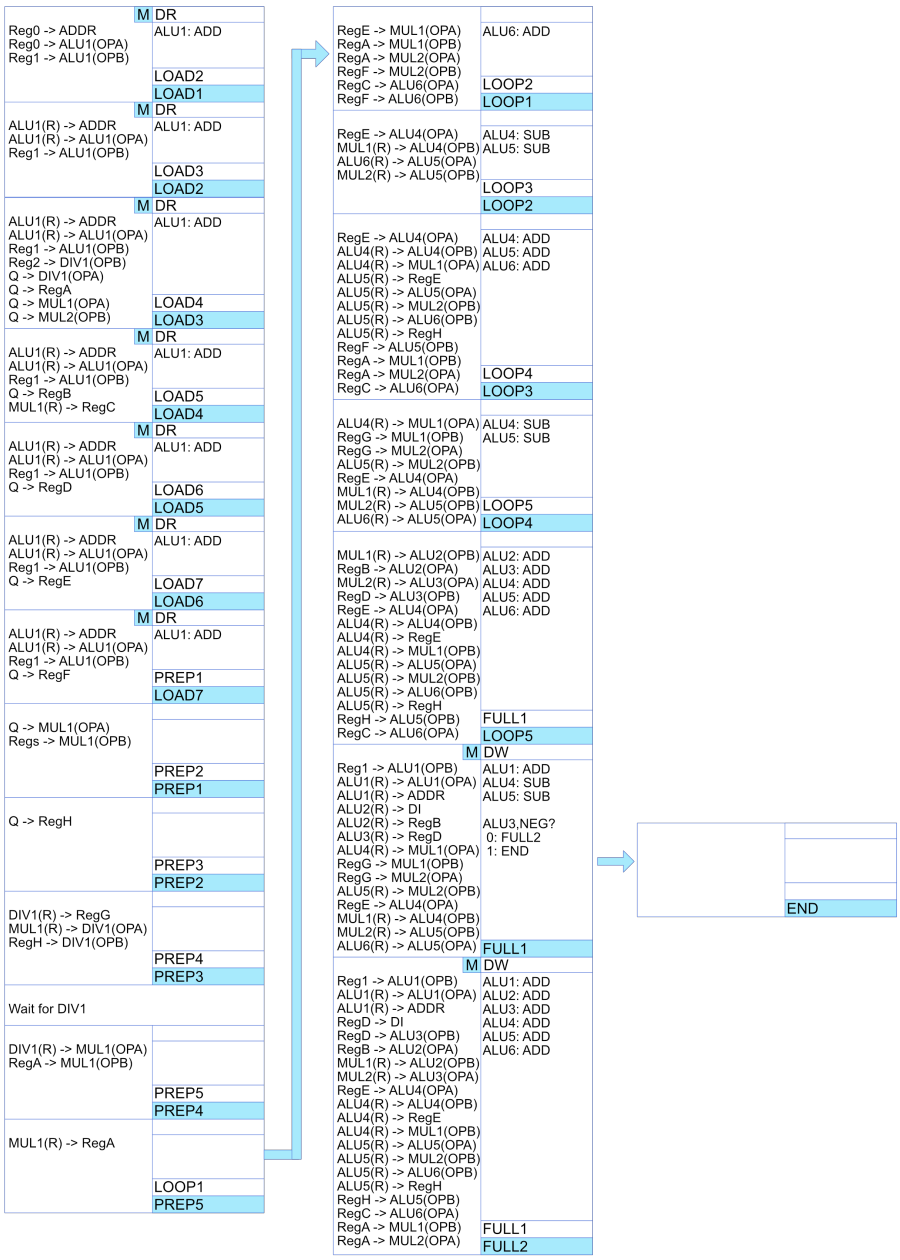

Abbildung 3: Register-Transfer-Folge