Z1-Addierer Kreislaufmodell

Belegarbeit

vorgelegt von

Alexander Preuß & Ralf Zücker

Einrichtung:	Hochschule Zittau/Görlitz
Modul:	Web Engineering (WE 1)
Betreuender Professor:	Prof. Dr. rer. nat. Christian Wagenknecht

Görlitz, 31. Juli 2016

Inhaltsverzeichnis

1 Einleitung

Alexander Preuß

Entwickelt werden sollte ein Addierermodell, welches die grundlegende Arbeitsweise des von Konrad Zuse gebauten Z1 veranschaulicht. Als Vorlage diente ein bereits vorhandenes mechanisches Modell des 2Bit-Addierers. Dieses sollte erweitert werden, sodass ein Arbeiten im Kreislauf möglich ist und die Summanden und Zwischenergebnisse stets rückübertragen werden. Das zu entwickelnde Modell sollte die Arbeitsweise des Addierers der Z1 Maschine nur nachbilden, eine originalgetreue Struktur und Bauweise der Einzelteile ist nicht zu realisieren, da über die tatsächlich verwendeten Bauteile der Z1 keine bzw. nur unvollständige Aufzeichnungen bestehen. Auch aus didaktischen Gründen weicht das entwickelte Modell ab, es wird flächig und nicht dreidimensional dargestellt, um es einfacher erkunden zu können. Im Folgenden wird der Entstehungsprozess vom Konzept zum fertigen Modell beschrieben.

2 Konzept der parallelen Addition

Alexander Preuß

2.1 Prinzip

Da eine Rechenmaschine wie die Z1 ständig Berechnungen ausführt, entwickelte Konrad Zuse eine Möglichkeit, diese möglichst schnell zu realisieren. Sein Prinzip der parallelen bzw. stellenweisen Addition basiert auf der Idee, dass jede Rechenstelle bzw. jedes Bit unabhängig von seinem Vorgänger bzw. dem nächsten niederwertigeren Bit berechnet werden kann. So kann die Addition für alle Stellen parallel bearbeitet werden und muss nicht mehr sequentiell durch alle Stellen arbeiten. Ermöglicht wird dies durch den einschrittigen Übertrag. Mit diesem kann die Addition in drei Schritten erfolgen, wobei alle Stellen gleichzeitig bearbeitet werden und der Übertrag immer in nur einem Schritt geschieht. Die Berechnung wird im Folgenden, so wie sie auch im Modell erfolgt, im Binärsystem beschrieben.

Berechnungschritt 1 Im ersten Schritt werden die Eingabebits für jede Stelle addiert, man erhält dann die Summenbits. Ist in einer Stelle ein originärer Übertrag entstanden (wenn beide Bits den Wert 1 hatten), dann wird die Stelle markiert. Mit einer zweiten Markierung werden die Stellen der Summe versehen, die den Wert 1 haben, da bei diesen eventuell aus der vorherigen Stelle ein Übertrag durchgekoppelt werden muss.

Berechnungsschritt 2 Im zweiten Schritt werden nun die Markierungen bearbeitet. Die Markierungen für eine eventuelle Durchkopplung werden zu Originären ergänzt, wenn die Vorgängerstelle mit einer Markierung für den originären Übertrag versehen ist. Die Überträge werden nun in einem Schritt verrechnet.

Berechnungsschritt 3 Im dritten und letzten Schritt berechnet man das Ergebnis aus den Stellen der Summe und den Überträgen. In der Markierungszeile werden alle originären Markierungen durch den Wert 1 ersetzt, alle leeren oder nur kopplungsmarkierten Stellen werden durch den Wert 0 ersetzt. Nun werden die Übertragsstellen um eine Stelle nach rechts verschoben (shift right). Danach addiert man wieder stellenweise ohne die Überträge zu berücksichtigen und erhält so das Ergebnis.

2.2 Beispielberechnung

Als Beispiel folgt die Berechnung der Addition für die Zahlen 39 (dezimal) und 42 (dezimal).

$$
\begin{aligned}
100111 \quad & & (2.1)\\
101010 \quad & & (2.2)\\
001101 \quad & \mathit{Summenbits} & (2.3)\\
\mathit{xooxo} \quad & & (2.4)\\
\mathit{xxxxo} \quad & & (2.5)\\
101110 \quad & & (2.6)\\
1011100 \quad & & (2.7)\\
001101 \quad & \mathit{Summenbits} & (2.8)\\
1010001 \quad & \mathit{Ergebnis} & (2.9)
\end{aligned}
$$

Zunächst werden die beiden Zahlen 39 (2.1) und 42 (2.2) nach Schritt 1 addiert. Daraus entstehen wie in (2.3) zu sehen die Summenbits und die erste Markierungszeile (2.4). In Schritt 2 werden nun die Markierungen bearbeitet (2.5). Schritt 3 ersetzt die Markierungen (2.6) und verschiebt um eine Stelle nach rechts (2.7). Die verschobene Übertragszeile wird mit den Summenbits wieder stellenweise addiert und man erhält das Ergebnis der Addition, in diesem Fall 81 (2.9).

3 Analyse des physischen Aufbaus

Alexander Preuß

3.1 Struktur des vorhandenen Modells

3.1.1 Aufbau

Das Modell arbeitet rein mechanisch, alle Rechnungen und Zwischenschritte werden also durch Bewegungen realisiert und angezeigt. Um dies zu ermöglichen, arbeitet das Modell mit mehreren Ebenen von Platten (aus Metall oder Kunststoff), die sich an bestimmten Stellen beeinflussen können. Dies wird durch Aussparungen in denen sich Stifte (im Folgenden Sticks) befinden, die die Platten verschieben realisiert.

3.1.2 Bauteile

Das vorhandene Modell lässt sich in zwei Bauteilgruppen aufteilen, Verbindungsteile und Schaltglieder. Zu den Verbindungsteilen gehören Übertragungsbleche, Taktbleche sowie Ein- und Ausgangsbleche. Die Gruppe der Schaltglieder umfasst drei Bauteile, das logische AND, das logische XOR und das ONE-Bauteil.
Das AND-Bauteil besitzt drei Ebenen. Die unterste besteht aus dem Takteingangsblech und Ausgangsblech, die mittlere beinhaltet die Linkerbleche und die oberste enthält die Eingabebleche. Alle drei Ebenen sind durch einen Stick verbunden. Die Aussparungen der verschiedenen Bleche sind nun so gewählt, dass wenn das Takteingangsblech verschoben wird, die Eingabebleche das Ausgangsblech so schieben oder nicht verschieben, dass es einem logischen AND gleichzusetzen ist. Das AND-Bauteil wird für den erzwungenen Übertrag genutzt.
Nach demselben Prinzip arbeitet auch das XOR-Bauteil. Hier sind die Aussprarun-

gen natürlich anders gewählt, damit das Betätigen des Taktes am Ausgangsblech eine Verschiebung hervorruft die dem logischen XOR gleichkommt. Die logischen Werte 1 und 0 werden dargestellt durch ein Blech in Ausgangslage (0) bzw. das Blech verschoben in eine Richtung (1). Das XOR-Bauteil wird für die Addition ohne Übertrag genutzt.
Die ONE-Bauteile haben verschiedene Funktionen. Sie ermöglichen einerseits ein Umschalten zwischen Addition und Subtraktion, realisieren aber auch ein Durchkoppeln der Überträge. Auch sie besitzen drei Ebenen. Die Ausgangsbleche spiegeln den Zustand des Eingabebleches zum Start des Taktes wieder. Sie dienen zum Durchkoppeln eines Zwischenergebnisses.

3.2 Funktion

Das Modell rechnet im 2Bit Bereich. Für die Eingabezahlen gibt es zwei zusammengeschlossene AND- & XOR-Bauteile. An der einen Einheit werden die höherwertigen Bits der Eingabezahlen über die Eingangsbleche eingestellt, an der anderen die niederwertigen Bits. Durch Betätigen des ersten Taktes werden die Summenbits und die Überträge über die Verbindungsbleche weitergeschoben. Beim Betätigen des zweiten Taktes werden über die ONE-Bauteile die Überträge weitergekoppelt und an weitere XOR-Teile geleitet. Diese addieren durch Betätigen von Takt 3 wieder einstellig und haben so an ihrem Ausgangsblech das Ergebnis der jeweiligen Bitstelle anliegen.

3.3 Erweiterung zum Kreislaufmodell

Im erweiterten Modell wird das bestehende ergänzt durch zwei weitere AND-Bauteile, deren Eingangsbleche mit den Ausgangsblechen der Ergebnis-XORs verbunden sind, und mit einem beweglichen Teil, welches das An-/Ausschalten der Rückkopplung aktiviert. Ist das Rückkopplungsblech in den 1-Zustand geschoben, so werden beim Betätigen des neuen Takt 4 die Ergebnisbits auf die Eingabebleche der AND&XOR-Einheit verschoben und man kann in Folge dessen mit diesem Ergebnis gleich weiterrechnen, indem man die zweite Zahl z. B. verändert und der Kreislauf mit Takt 1 wieder von vorn beginnt. Dazu benötigt das neue Modell

weitere Bleche zum Zurücksetzen der vorherigen Eingaben, damit auch bei jeder Rechnung neu übertragen wird und nicht von einer alten Rechnung noch falsche Überträge beispielsweise durchgekoppelt werden. Mit dem erweiterten Modell ist man nun in der Lage, lange Berechnungsketten realisieren.

4 Programmiertechnische Abbildung

Ralf Zücker

4.1 Grundidee

Im physischen Modell werden Kraftübertragungen dadurch realisiert, dass bewegte Elemente durch Kollision andere Elemente bewegen. Im einfachen Modell gibt es solche Kollisionen nur zwischen fest verbundenen Elementen und durch Stifte, die sich in Aussparungen nicht frei bewegen können und somit das Bauteil mitziehen oder schieben. Durch Verkettung dieser einzelnen Kollisionen entstehen komplexe Bewegungen, die den Additionsvorgang darstellen.
Die Grundidee der programmiertechnischen Nachbildung dieses Prozesses besteht darin, die einzelnen Bauteile als eigenständige Entitäten mit definierten Freiheitsgraden zu betrachten, die Abhängigkeiten untereinander haben. So kann die manuell ausgelöste Bewegung eines Elements zunächst durch die eigene Bewegungsfreiheit und anschließend durch die Freiheiten der abhängigen Bauteile bestätigt bzw. blockiert werden.
Anstatt das Modell explizit "nach zu programmieren" hat diese Methode den Vorteil, dass nur das Bewegungsgrundprinzip programmiert werden muss und die eigentliche Bauteilkombination nur ein konfigurierter Anwendungsfall ist. Dadurch entsteht eine Art Baukasten, der es erlaubt, ohne weitere Logik-Programmierung auch andere Automaten zu konstruieren.

4.2 Technische Basis

Da das Projekt mit HTML5 umgesetzt wird, müssen im Endeffekt graphische HTML Elemente bewegt werden. Diese bilden damit das Frontend und repräsentieren die Entitäten (Bauteile) graphisch. In den folgenden Code-Ausschnitten werden

svg Elemente verwendet, da diese auch für die Umsetzung verwendet wurden. Jedes svg Element hat eine eindeutige id. Um an einem Element die Freiheitsgrade und Abhängigkeiten zu definiere,n werden data-Attribute verwendet. Jedes Bauteil kann verschiedene Stellungen einnehmen. Im Attribut data-bit wird die aktuelle Stellung codiert. Mit dem hinterlegten Wert kann die logische Stellung eines Bauteils für Auswertungen benutzt werden. Das Attribut data-bitpos definiert die aktuelle räumliche Stellung des Bauteils. Mit dem data-move Attribut wird die Liste der beeinflussten Bauteile definiert.

Listing 4.1: Benutzerdefinierte Attribute

```
1 <svg id="elementID" x="0" y="0"
2     data-bit="0"
3     data-bitpos="up"
4     data-move="otherElement1 otherElement2">
```

Die Definition in Listing 4.1 beschreibt ein Element mit dem internen Wert 0, welches sich momentan im Zustand up befindet, also nur nach unten (und somit ausschließlich vertikal) beweglich ist. In data-move sind die IDs der Bauteile gelistet, welche durch diese Bewegung u. U. mit bewegt werden.
Für die Eingangs- und Ausgangsbleche von Schaltgliedern (Setter und Trigger) reichen diese Angaben aus, um deren Bewegung und Abhängigkeit vollkommen zu beschreiben. Aber nicht alle Elementtypen haben nur eindimensionale Bewegungsrichtungen und einige benötigen zusätzliche Definitionen. Im Folgenden werden die speziellen Elementtypen näher beschrieben.

4.2.1 Sticks

Sticks sind die Stifte, welche sich in den Aussparungen der Bauteile bewegen. Sie sind horizontal und vertikal beweglich und benötigen daher keine Angabe von data-bitpos. Genau genommen gibt es vier verschiedene Positionen, die diese einnehmen können. Diese Positionen werden durch die Werte 1 bis 4 codiert. Im absoluten Koordinatensystem bedeutet der Wert 1 links oben, 2 rechts oben, 3 rechts unten und 4 links unten. Diese Definition ist damit vollkommen unabhängig von der Lage der umgebenden Elemente.

Listing 4.2: Definition Stick

```
1 <svg id="stick1" x="95" y="115"
2     data-bit="1"
3     data-move="linker1 trigger1">
```

Die Definition in Listing 4.2 beschreibt eine Stift, der sich in seiner linken oberen Position befindet und Einfluss auf die Elemente `linker1` und `trigger1` ausüben kann.

4.2.2 Linker

Linker sind die zentralen Verbindungsbleche, die die Logik eines Schaltgliedes umsetzen. Sie sind nur eindimensional beweglich. Deren Bewegung kann also durch die Attribute `data-bit` und `data-bitpos` vollkommen beschrieben werden. Mittels `data-move` Attribut können die beeinflussten Elemente angegeben werden.
Um jedoch die aus der Aussparung resultierende Übertragungslogik zu beschreiben, gibt es ein weiteres Attribut `data-type`.

Listing 4.3: Definition Linker

```
1 <svg id="linker1" x="80" y="100"
2     data-bit="0"
3     data-bitpos="up"
4     data-move="stick1 linker2"
5     data-type="124">
```

Die Definition in Listing 4.3 beschreibt einen vertikal beweglichen Linker mit dem Ausschnitt `124`. Die Zahlen `1` bis `4` codieren hier wieder die vier möglichen Positionen eines Sticks. Die in `data-type` angegebenen Zahlen definieren die Positionen, die durch den Stick eingenommen werden können. In diesem Fall ist nur die untere rechte Ecke nicht ausgeschnitten.
Ein einzelnes Kopplungsblech besitzt nach Grams [Gra15] zumeist zwei Ausschnitte. In dem hier beschriebenen Modell besteht ein solches Blech aus zwei separaten Elementen mit jeweils einem Blech, die eine direkte gegenseitige Beeinflussungsbeziehung haben. Damit lassen sich beliebig ausgesparte Kopplungsbleche als Kombination definieren.

4.3 Auswertungslogik

Wird die Bewegung eines Elements ausgelöst, muss geprüft werden, ob sich das Element in die entsprechende Richtung bewegen kann. Da sich einzelne Schaltglieder auch in einer blockierenden Position befinden können, ist eine Bewegung nicht immer möglich.

Listing 4.4: Auslösung einer Bewegung

```
var dir = swapDir(item.getAttribute("data-bitpos"));
var moveItems = checkMove(root, [],item, dir)
if (moveItems != null){
    moveItems.forEach(function(entry){
        moveItem(root, entry, dir);
    });
}
```

Listing 4.4 ist zu entnehmen, dass sich die Bewegungsrichtung aus der aktuellen `data-bitpos` des Elements ergibt. Die Prüfung der Beweglichkeit des Elements liefert entweder `null` (bei Blockierung) oder die Liste der Elemente, die sich mitbewegen sollen. In letzterem Fall muss nur noch die Bewegung ausgelöst werden. Die eigentliche Prüfung wird rekursiv durchgeführt, denn die Beweglichkeit eines Elements hängt von der Beweglichkeit der verknüpften Elemente ab.

Listing 4.5: Beweglichkeitsprüfung

```
// root: die gesamte Modell-Konstruktion
// parents: Kaskade der Auslöser
// item: das frei bewegte Element
// dir: die Bewegungsrichtung
function checkMove(root, parents, item, dir){
    var result = [];
    // Das aktuelle Element in die Auslöserkaskade einfügen
    parents.push(item);
    // IDs der abhängigen Elemente ermitteln
    var depends = item.getAttribute("data-move").split(" ");
    for(var i = 0; i < depends.length; i++){
        // Beeinflusstes Element
        var dep = $("#"+depends[i],root)[0];
        // Kreisläufe verhindern
        if (parents.indexOf(dep)!= -1) continue;
        // Prüfung der Beweglichkeit von dep
        ...
```

```
18          // rekursive Prüfung der Unterelemente
19          var subItems = checkMove(root,parents,dep,dir);
20          // mindestens ein Unterelement blockiert
21          if (subItems == null) return null;
22          // bewegte Unterlemente in Ergebnisliste
23          result.push.apply(result,subItems);
24      }
25      // item und alle Subelemente lassen sich verschieben
26      result.push(item);
27      return result;
28  }
```

Listing 4.5 zeigt die Rekursion. Über das `data-move` Attribut werden die verknüpften Elemente ermittelt. Da die Beziehungen zwischen den Elementen meist gegenseitig sind, bilden sich Ringe, die eine endlose Rekursion zur Folge hätten. Durch Merken der aktuellen Bewegungskaskade wird verhindert, dass solche Ringe betreten werden. Der rekursive Aufruf liefert nun die Liste der beweglichen verknüpften Elemente oder `null` bei Blockierung. Eine einzige Blockierung reicht aus, um den gesamten Bewegungsprozess zu blockieren. Bei freier Beweglichkeit werden die Elemente in die Ergebnisliste aufgenommen. Liefert keines der Unterelemente eine Blockierung, wird das zu prüfende `item` selbst noch der Ergebnisliste hinzugefügt vor dem rekursiven Aufstieg.
Eine Sonderstellung bei der Beweglichkeitsprüfung nehmen Beziehungen zwischen Sticks und Linkern ein, denn nur hier können Blockierungen auftreten.

Listing 4.6: Stick bewegt Linker

```
1  // item: Stick
2  // dep: Linker
3  // Art der Linker-Aussparung ermittlen
4  var type = dep.getAttribute("data-type");
5  var linkerdir = getLinkerDir(dep,dir);
6  var stickBit = getStickBit(dep,item.getAttribute("data-bit"));
7  var linkerBit = dep.getAttribute("data-bit");
8  var m = canMoveStick2Link(type,linkerBit,stickBit,linkerdir);
9  if (m == null){ // Blockade
10     parentMoveItems.push(dep);
11     highlight_chain(parentMoveItems, dir);
12     return null;
13 }
14 else if (m == "F") // Stick frei beweglich
15     continue;
```

Beeinflusst ein Stick einen Linker (Listing 4.6) sind die Art der Aussparung am Linker (Attribut `data-type`) und die Stellungen von Linker und Stick von entscheidender Bedeutung. Nur in bestimmten Konstellationen sind Bewegungen möglich. Bei freier Bewegung (Stick bewegt sich frei innerhalb der Linkeraussparung) gibt es keine Wirkung auf den Linker (Fall `"F"`). Ist die Bewegung des Sticks nur möglich, wenn auch der Linker verschoben werden kann (Fall `"T"`), wird die rekursive Prüfung der Unterelemente wie in Listing 4.5 ersichtlich durchgeführt. Ist die Bewegung gar nicht möglich (Fall `null`), wird die komplette Auslösekaskade (inklusive blockierendem Linker) grafisch hervorgehoben (`highlight_chain`).In diesem Fall wird auch die weitere Prüfung abgebrochen.
Nahezu identisch ist der Fall, das ein Linker einen Stick verschiebt wie Listing 4.7 zeigt.

Listing 4.7: Linker bewegt Stick

```
1  // item: Linker
2  // dep: Stick
3  // Art der Linker-Aussparung ermittlen
4  var type = item.getAttribute("data-type");
5  var linkerdir = getLinkerDir(item,dir);
6  var stickBit = getStickBit(item,dep.getAttribute("data-bit"));
7  var linkerBit = item.getAttribute("data-bit");
8  var m = canMoveLink2Stick(type,linkerBit,stickBit,linkerdir);
9  if (m == null){ // Blockierung
10     parentMoveItems.push(dep);
11     highlight_chain(parentMoveItems, dir);
12     return null;
13 }
14 else if (m == "F") // Linker frei beweglich
15     continue;
```

Im Fall `"F"` kann sich der Linker bewegen, ohne dass der Stick beeinflusst wird. Im Fall `"T"` müssen weitere Abhängigkeiten des bewegten Sticks rekursiv geprüft werden. Und bei Blockade wird die Auslösekaskade grafisch hervorgehoben. Die Funktionen `canMoveStick2Link` und `canMoveLink2Stick` greifen lediglich auf ein statisches Array mit den Bewegungsmöglichkeiten zu, um in Abhängigkeit der aktuellen Situation das richtige Ergebnis zu liefern.
Eine weitere Möglichkeit für vorzeitigen Abbruch des rekursiven Abstiegs bieten orthogonale Bewegungen (außer zwischen Stick und Linker). Trifft eine horizontale Bewegung auf ein nur vertikal bewegliches Element (oder anders herum), dann

ist diese Bewegung immer möglich (Listing 4.8). Das betrifft Sticks, die sich in Aussparungen von Triggern und Settern frei bewegen können. Jeder andere Fall ist eine Fehlkonfiguration der Abhängigkeiten und wird hier nicht explizit geprüft.

Listing 4.8: Orthogonale Beweglichkeit

```
var bitpos = dep.getAttribute("data-bitpos");
// Bei Sticks ist bitpos == null >> niemals continue möglich
if (bitpos != null && bitpos != dir && bitpos != swapDir(dir))
    continue; // Ortogonal zur Element-Verschieberichtung
```

Nur wenn `checkMove` eine Liste mit zu bewegenden Elementen geliefert hat, wird die eigentliche Bewegung ausgeführt. Das hat den Vorteil, dass die Bewegungen sehr schnell aufeinander folgend ausgeführt werden können, was optisch den Eindruck erweckt, gleichzeitig zu sein. Für die Animationen wird Snap.svg[Bar16] benutzt.

Listing 4.9: Bewegungsanimation

```
var isStick = item.className.baseVal.indexOf("stickItem")==0;
var stickBit = item.getAttribute("data-bit");
if (dir == "down"){
    Snap(item).animate({y:'+=30'},durat);
    if (isStick)
        item.setAttribute("data-bit",stickBit==1 ? 4 : 3)
}
...
if (!isStick){
    swapAttribute(item, "data-bit");
    swapAttribute(item, "data-bitpos");
}
```

Neben der eigentlichen Bewegung müssen auch die internen Zustände der Elemente (`data-bit` und `data-bitpos`) der Bewegung entsprechend angepasst werden. In Listing 4.9 ist zu erkennen, dass das für alle nur eindimensional beweglichen Elemente eine einfache Invertierung ist. Beim Stick, der vier interne Stellungen hat, muss `data-bit` in Abhängigkeit von Bewegungsrichtung und aktueller Stellung gesetzt werden.

4.4 Erweiterung zum Kreislaufmodell

Ein Kreislauf erfordert eine Rücksetzung der Taktbleche (Trigger), da diese in ihrem gesetzten Zustand die Eingabebleche (Setter) blockieren. Das Zurückziehen der Taktbleche bewirkt jedoch aufgrund der gegenseitigen Beziehungen auch ein Zurückziehen der Ausgangsbleche, welche gleichzeitig als Eingabe für den folgenden Takt dienen. Folglich müssen die Stellungswechsel bei Bewegung eines Taktblechs erhalten bleiben, wenn das Taktblech zurückgezogen wird.
In der Modellbeschreibung[Gra15] von Grams wurden die Schaltglieder insofern modifiziert, dass die Eingabebleche, wenn sie durch einen Takt gesetzt sind, in ihrer gesetzten Lage verharren, wenn das Taktblech zurückgezogen wird. Dazu wurden die Eingabebleche um eine weitere orthogonale Aussparung erweitert, in welcher ein Stift zwei Takte später das Blech wenn nötig in seine Ausgangslage zurückzieht. So kann der Folgetakt die veränderten Stellungen des vorherigen nutzen, welche erst im nächsten Takt wieder in die Ausgangslage gebracht werden.
Aus Gründen der Übersichtlichkeit des graphischen Modells und dem Bestreben das Programmiermodell so einfach und verständlich wie möglich zu halten, wurde hier ein Ansatz ohne zusätzliche Stifte (Sticks) gewählt. Es wird eine andere Art von Abhängigkeitsbeziehung eingeführt.

Listing 4.10: data-push Attribute

```
1 <svg id="triggerA1" x="-180" y="90"+
2     data-bit="0"
3     data-bitpos="left"
4     data-move="stickA1"
5     data-pushright="setterA2">
```

Mit dem `data-push...` Attribut können, analog zu `data-move`, Elemente definiert werden, auf die eine Bewegung des `svg` Elements Einfluss haben kann, jedoch mit dem Unterschied, dass die Einflussnahme nur dann stattfindet, wenn es die Bewegungsrichtung und die interne Stellung der Elemente zulässt. Das Attribut `data-push...` als solches muss immer um eine Richtung erweitert werden, in die eine Einflussnahme möglich ist. In Listing 4.10 wird ein horizontal bewegliches Taktblech beschrieben, welches nur bei Bewegung nach rechts Einfluss auf das Element `setterA2` ausüben kann. Bei einer Bewegung nach links bleibt `setterA2` unangetastet, was der zu erreichenden Situation des Verharrens entspricht.

Listing 4.11: Push-Erweiterung

```
var pushes = item.getAttribute("data-push"+dir).split(" ");
for(var i = 0; i < pushes.length; i++){
    var push = $("#"+pushes[i],root)[0];
    if (parentMoveItems.indexOf(push)!= -1) continue;
    var pos1 = item.getAttribute("data-bitpos");
    var pos2 = push.getAttribute("data-bitpos");
    if (pos1 == pos2){ // Bauteile grenzen aneinander
        var subItems = checkMove(root, parents,push,dir);
        if (subItems == null) return null;
        result.push.apply(result,subItems);
    }
}
```

Wird in `checkMove`(Listing 4.5) ein Element geprüft welches ein `data-push` Attribut für die aktuelle Bewegungsrichtung enthält, dann wird die Routine in Listing 4.11 ausgeführt. In einer Schleife werden die beeinflussbaren Elemente geprüft. Befinden sich die in Beziehung stehenden Elemente in der gleichen Position (`data-bitpos` identisch), so grenzen sie aneinander und das Unterelement erfährt die Wirkung. Es wird damit Teil der Kaskade und muss rekursiv auf weitere Verschiebungen bzw. Blockierungen geprüft werden.
Durch dieses Prinzip können Bewegungsabhängigkeiten, die nur in eine Richtung wirken, realisiert werden.

4.5 Sonstige Modell-Features

Standard-Werte Bei Initialisierung eines SVG Modells werden sämtliche relevanten Eigenschaften eines Bauteils in einem `default` Attribut hinterlegt. Listing 4.12 zeigt die veränderbaren Eigenschaften und Attributwerte eines Elements.

Listing 4.12: Standardwerte speichern

```
jQuery.each(svgRoot.getElementsByTagName("svg"), function(i, e){
    e.default = { x: e.x.baseVal.value,
                  y: e.y.baseVal.value,
                  bit: e.getAttribute("data-bit"),
                  bitpos: e.getAttribute("data-bitpos"),
    };
});
```

Zur Wiederherstellung der gespeicherten Werte gibt es eine weitere Funktion.

Listing 4.13: Standardwerte wiederherstellen

```
1 // Jedes SVG Element erhält wieder die gespeicherten Werte
2 jQuery.each(svgRoot.getElementsByTagName("svg"), function(i, e){
3     e.setAttribute("data-bit",e.default.bit);
4     if (e.default.bitpos != null)
5         e.setAttribute("data-bitpos",e.default.bitpos);
6     Snap(e).animate({x:e.default.x},durat);
7     Snap(e).animate({y:e.default.y},durat);
8 });
```

Aus Listing 4.13 wird ersichtlich, dass optische Rücksetzungen als Animation erfolgen. Außerdem müssen Sticks, welche keine `data-bitpos` haben, gesondert behandelt werden.

Previous-Werte Um einzelne Aktionen rückgängig machen zu können, müssen die relevanten Eigenschaften eines Bauteils vor der Ausführung der Bewegung gespeichert werden.

Listing 4.14: Previous-Werte speichern

```
1  // Bereits gespeicherte Werte an allen Elementen zurücksetzen
2  jQuery.each(svgRoot.getElementsByTagName("svg"), function(i, e){
3      e.previous = null;
4  });
5  // an den gewünschten Elementen die aktuellen Werte sichern
6  items.forEach(function(entry){
7      entry.previous = {
8          x: entry.x.baseVal.value,
9          y: entry.y.baseVal.value,
10         bit: entry.getAttribute("data-bit"),
11         bitpos: entry.getAttribute("data-bitpos"),
12     };
13 });
```

Listing 4.14 zeigt, wie nach dem Entfernen möglicherweise vorhandener Werte, ein `previous` Objekt an den betroffenen Bauteilen mit den zur Wiederherstellung benötigten Informationen erzeugt wird. Zur animierten Wiederherstellung des gespeicherten Zustands gibt es die in Listing 4.15 gezeigte Funktion.

Listing 4.15: Previous-Werte wiederherstellen

```
// Jedes SVG Element erhält wieder die gespeicherten Werte
jQuery.each(svgRoot.getElementsByTagName("svg"), function(i, e){
    if (e.previous != null) {
        e.setAttribute("data-bit",e.previous.bit);
        if (e.previous.bitpos != null)
            e.setAttribute("data-bitpos",e.previous.bitpos);
        Snap(e).animate({x:e.previous.x},durat);
        Snap(e).animate({y:e.previous.y},durat);
    }
});
```

Pulsieren Für die graphische Hervorhebung bestimmter Bauteile durch Pulsieren gibt es eine Klasse `pulsing` mit der entsprechenden CSS-Definition. Mit den Funktionen `resetPulsing` und `setPulsing` wird die Klasse von allen Elementen entfernt bzw. den übergebenen Elementen hinzugefügt, um das Pulsieren an- bzw. auszuschalten.

5 Graphische Darstellung

Alexander Preuß

Die graphische Darstellung wird überwiegend durch SVGs realisiert. Die Vorteile der Nutzung von SVG-Grafiken sind die freie Skalierbarkeit ohne Qualitätsverlust und die geringe Dateigröße, welche eine schnelle Ladezeit ermöglicht. Beide Vorteile sind für die Nutzung auf mobilen Endgeräten notwendig. Das Modell selbst ist als eine große SVG Datei gespeichert. Die einzelnen Bauteile mit ihren verschiedenen Attributen sind in diese größere SVG eingebettet. Ein Beispiel eines solchen Bauteils ist in Listing 5.1 zu sehen.

Listing 5.1: Ausschnitt eines eingebetten Bauteils

```
1 <svg id="CONNECTOR1" class="triggerItem" x="70" y="450" data-bit="
    0" data-bitpos="up" data-move="AND1trigger2" data-pushdown="
    ONE1setter">
2     <path class="connector" data-draw="connector-short"/>
3 </svg>
```

Jedes Bauteil besitzt eine einzigartige `id`. Das Attribut `class` dient zur Verarbeitung durch die Programmlogik. Die X- und Y-Koordinaten sind im jeweiligen Attribut vorgegeben und bestimmen die feste Position des Teils im Modell. Die mit "data-" beginnenden Attribute dienen zur Verarbeitung. Im `path` wird ein weiteres class-Attribut angegeben, welches zum Styling mittels eingebundenem CSS verwendet wird. Das Attribut `data-draw` gibt letztendlich einen Namen vor, der bestimmt, wie das Bauteil gezeichnet wird. Dazu wurde mittels JavaScript ein Array `subPath` erzeugt, welches den Inhalt des `path` dem passenden Namen zuordnet. Die geschieht durch Aufruf der Funktion `addSVGPaths`. Listing 5.2 zeigt einen Ausschnitt.

Listing 5.2: Ausschnitt Z1-SVGPath.js

```
var subPath = [];
subPath["connector-short"]  = "M 0 0 L 100 0 100 420 0 420 Z";
subPath["connector-long"]   = "M 0 0 L 100 0 100 370 0 370 Z";

function addSVGPaths(svgRoot){
    jQuery.each(svgRoot.getElementsByTagName("path"), function(
       index, entry){
        var drawPath = entry.getAttribute("data-draw");
        if (drawPath != null)
            entry.setAttribute('d', subPath[drawPath]);
    });
}
```

Durch Verwendung eigens definierter “data”-Attribute ist es möglich, ein Modell beliebig zusammenzustellen. Die Logik übernimmt sämtliche Bewegungen und die grafische Darstellung ist frei wählbar. Das definierte Modell kann so optisch völlig verändert werden, behält seine Arbeitsweise jedoch bei. Das automatisierte Hinzufügen des Zeichenpfads besitzt zwei Vorteile. Zum einen bleibt der Code selbst übersichtlich und zum anderen sind Änderungen an der Form der Teile schnell zu bewerkstelligen. Durch das eingebundene Stylesheet erhält man gleichermaßen eine Trennung zwischen dem Aussehen der Teile und ihrer Funktion im Modell.

6 Umsetzung als Web-Applikation

6.1 HTML Grundgerüst

Alexander Preuß

Das Layout der Webapplikation wird durch eine Tabelle vorgegeben. Listing 6.1 zeigt einen Ausschnitt der Layouttabelle mit eingebettetem SVG des Modells und Button.

Listing 6.1: Ausschnitt Tabellenlayout

```
<tr>
    <td class="controlarea">
        <embed class="zuseAutomaton" id="zuseModel"
            src="svg/Z1-Prototype.svg"
            width="1400" align="left"/>
    </td>
    <td bgcolor="#111111">
        <table id="controlarea">
        <tr>
            <td>
                <button id="toggleOverlay" title="toggle overlay"
                    class="z1button" href="#"
                    disabled="false">
                    <img height="100%" src="pic/overlay.png"/
                </button>
            </td>
            ...
```

Diese beinhaltet das als `<embed>` eingebettete SVG des Modells selbst, als auch eine weitere Tabelle `controlarea`, welche die Steuerelemente für das Modell enthält. Dort sind weitere SVG-Elemente eingebettet, die eine Sieben-Segment-Anzeige darstellen. Diese dienen zur besseren Verständlichkeit der einzelnen Rechenschrit-

te. Die Steuerfunktionen des Modells werden über z1buttons in der controlarea-Tabelle realisiert, diese sprechen den ZuseController an, welcher die Veränderung im Modell hervorruft. Weiterhin werden infobuttons definiert, die beim Anklicken ein Popup mit weiteren Details zu den verschiedenen Bauteilen öffnen. Das Popup wird als -Element realisiert, welches zunächst nicht gerendert wird. Durch Klick auf einen infobutton wird die -Box gerendert und ist nun über dem Modell zu sehen. Diese infoBox enthält ein weiteres -Element infoBoxContent, dieses enthält ähnlich wie zuvor eine Tabelle, in der ein Bauteil als SVG mit <embed> eingebettet ist und Buttons zur Manipulation des selbigen. Die in der Webapp verwendeten Buttons haben ein Bild hinterlegt, welches durch das <img>-Tag definiert wird. Eine Hilfestellung bieten zusätzlich die verwendeten title-Attribute.

6.2 Controller

Ralf Zücker

Mit dem ZuseController wird die Funktionsweise des Modells gekapselt. Zugriffe auf das Modell, wie Bewegen eines bestimmten Elementes, werden somit immer durch den Controller ausgelöst. Auf diese Weise wird die Funktionalität des Modells dahinter in geeigneter Weise eingeschränkt und koordiniert, sodass eine gewisse Endbenutzerfreundlichkeit erreicht wird.
Bei Initialisierung der Web-Applikation wird unter Angabe des Automaten-SVGs ein ZuseController erstellt.

Listing 6.2: Controller-Initialisierung

```
1  window.onload = function () {
2      // Controller steuert das Modell
3      z1 = new ZuseController($('#zuseModel')[0]);
4      z1.init(); // SVG initialisieren (paths laden)
5      jQuery.each($('.z1button'), function(index, entry) {
6          z1.addButton(entry); // Steuerbuttons bekannt machen
7      });
8      z1.updateButtonStates();
9      z1.pulseClock();
10     ...
11 }
```

Anschließend werden diesem die zugehörigen Steuerelemente des HTML Gerüsts mittels `addButton` bekannt gemacht und initiale Startzustände definiert. Listing 6.2 zeigt, wie dieser Vorgang mittels jQuery[Fou16] umgesetzt wurde.
Die `id` eines `z1button` kodiert dessen Funktionalität. Der Controller beinhaltet ein Array, welches für jede unterstützte `id` ein Objekt mit den benötigten Funktionalitäten bereit hält.

Listing 6.3: Aktionen-Array

```
1  // Array der Aktionen (id entscheidet über Funktionalität)
2  this.controlItems = [];
3  // Overlays ein-/ausschalten
4  this.controlItems["toggleOverlay"] = {
5      isEnabled: function(controller) { // Funktion möglich?
6          return true; // geht jederzeit
7      },
8      doAction: function(controller){ // Funktion ausführen
9          toggleOverlay(controller.baseItem);
10     }
11 };
```

Listing 6.3 zeigt die Implementation der `toggleOverlay` Aktion. Jede Aktion enthält jeweils eine Funktion zur Ermittlung, ob die Aktion gerade ausführbar ist (`isEnabled`) und eine Funktion, welche die eigentliche Aktion ausführt (`doAction`). Beim Hinzufügen eines Steuerelements mit `addButton` wird wie in Listing 6.4 ersichtlich, nur der Click-Listener initialisiert sowie Steuerelement und Aktion bzw. Controller miteinander verknüpft.

Listing 6.4: Steuerelemente registrieren

```
1  this.addButton = function(bItem){
2      // Button muss seinen Controller kennen
3      bItem.controller = this;
4      // onAction-Listener an jeden Button
5      bItem.addEventListener("click", this.onAction);
6      // DOM Element merken
7      this.controlItems[bItem.id].controlButton = bItem;
8  }
```

Listing 6.5: Steuerelement-Aktion

```
this.onAction = function(){ // irgendein Button wurde gedrückt
    // this ist hier das DOM-Element da Event-Listener
    var ctrl = this.controller; // controller hängt am Button
    // Aktion des Button aufrufen
    ctrl.controlItems[this.id].doAction(ctrl);
    // Status der Buttons aktualisieren
    ctrl.updateButtonStates();
}
this.updateButtonStates = function(){ // Buttons (de-)aktivieren
    var keys = Object.keys(this.controlItems);
    // Durch die Buttons iterieren
    for(var i = 0; i < keys.length; i++){
        var e = this.controlItems[keys[i]];
        // IsEnabled Funktion liefert disabled Status
        e.controlButton.disabled = !e.isEnabled(this);
    }
}
```

Die in Listing 6.5 gezeigte Event-Listener Funktion `onAction` ist für alle Steuerelemente die gleiche. Aber anhand der `id` des Steuerelements wird die richtige dazugehörige Aktion ausgeführt. Anschließend werden die Zustände aller Steuerelemente mit `updateButtonStates` neu festgelegt. Hierbei wird für jede mögliche Aktion durch die `isEnabled` Funktion am entsprechenden Steuerelement die `disabled` Eigenschaft gesetzt.
Zusätzlich wurden spezielle Modellelemente im Modell klick-aktiv gemacht. Auch diese nutzen mehr oder weniger direkt die definierten Steuerelement-Aktionen wie in Listing 6.6 zu sehen.

Listing 6.6: Klick-aktive Modellelemente

```
jQuery.each($(this.z1Feedback).children("path, .clickabletext"),
   function(index,entry){
    entry.addEventListener("click", function(){
        z1.tryAction(z1.controlItems["toggleFeedback"]);
    });
});
```

Die im Aktionen-Array definierten Funktionen stellen teilweise Grundfunktionen dar. Listing 6.7 zeigt, wie diese sich auch gegenseitig verwenden, um doppelte Implementationen zu vermeiden.

Listing 6.7: doAction-Funktion von startLoop

```
ctrl.deactivateAutoLoop();
ctrl.controlItems["reset"].doAction(ctrl); // Alles zurücksetzen
ctrl.loopMode = 2; // Loop-Mode
ctrl.updateButtonStates(); // Buttons deaktivieren
setTimeout(updateDisplaysDependentOnClock,500,4);
setTimeout(ctrl.controlItems["nextStep"].doAction,1000, ctrl);
```

Der Start des Endlos-Kreislaufs beginnt mit einem Aufruf der `reset` Funktion. Nach Setzen des Modus und Aktualisierung der Steuerelementzustände wird die `nextStep` Funktion ausgeführt. Letzteres passiert verzögert, da `reset` im Inneren asynchrone Animationen startet, die erst beendet sein müssen.
Diese verzögerten Ausführungen sind elementar zum Erreichen gefälliger Animationen und müssen gut aufeinander abgestimmt sein. Ein weiteres Beispiel zeigt Listing 6.8.

Listing 6.8: Taktauslösung

```
// num: auszuführender Takt (1-4)
ctrl.currentClock=-1; // Flag "in Bewegung"
resetPulsing(ctrl.svgRoot);
var clockItem = ctrl.z1Clocks[num];
// Takt hin, sofort, undo-speichern
setTimeout(clickItem,0, clockItem, ctrl.svgRoot, true);
// Takt zurück, anschließend
setTimeout(clickItem,500, clockItem, ctrl.svgRoot);
// die Displays aktualisieren, zusammen mit Rückbewegung
setTimeout(updateDisplaysDependentOnClock, 500, num);
if (num == 4){ // letzter Takt >> Feedback übernehmen
    setTimeout(ctrl.setFeedbackInput,500, ctrl);
    ctrl.backEnabled = false; // kein Rückgangig mehr möglich
}
// Pulsieren des nächsten Takts auslösen
setTimeout(ctrl.pulseClock,1000,ctrl, num);
// gleichzeitig aktuellen Takt setzen
// und per TriggerNext Folgeaktionen prüfen
setTimeout(function(ctrl) {
        ctrl.currentClock=num;
        ctrl.triggerNext();
    },1000,controller);
```

Das Ausführen der eigentlichen Taktbewegung wird quasi ohne Verzögerung ausgeführt, beinhaltet aber eine asynchrone Animation über 400 ms. Die Rückbewegung des Taktbleches wird mit einer Verzögerung vom 500 ms somit erst im Anschluss an die erste Animation ausgeführt. Der Beginn des Pulsierens des Folgetaktes sowie Folgefunktionen werden mit 1000 ms Verzögerung also erst nach der asynchronen Rückbewegungsanimation ausgelöst.

Nach einer festgelegten Zeit ohne Benutzerinteraktion startet der Controller automatisch den Endloskreislauf mit Zufallszahlen. Es gibt zwei Funktionen zum Aktivieren und deaktivieren dieser Funktionalität.

Listing 6.9: Automatischer Kreislauf

```
this.deactivateAutoLoop = function(){
    if (this.interactionTimeout != null){
        clearTimeout(this.interactionTimeout);
        this.interactionTimeout = null;
    }
}
this.activateAutoLoop = function(){
    if (this.interactionTimeout == null)
    this.interactionTimeout = setTimeout(
        this.controlItems["startLoop"].doAction,
        this.loopAfterMilliseconds, this);
}
```

Listing 6.9 zeigt, dass beim Aktivieren ein Timer mit einer relativ langen Ablaufzeit gestartet wird. Die Funktion `deactivateLoop` stoppt diesen Timer. Bei jeder Benutzeraktion wird der Timer zurückgesetzt und wieder neu gestartet.

6.3 Anzeige der Berechnungsschritte

Alexander Preuß

Die Anzeige der Berechnungsschritte wird durch in der Haupttabelle eingebettete SVGs der Klasse `segmentdisplay` ermöglicht. Die verwendete “sevensegment.svg” besitzt, ähnlich wie die SVG-Datei des Modells, Subelemente, die mit `<svg>` versehen sind. Listing 6.10 zeigt ein Beispiel eines solchen Subelementes.

Listing 6.10: Segment-Subelement

```
<svg id="TOP" class="segment" x="10" y="20" visibility="visible">
  <path d="M 100 100 L 150 50 350 50 400 100 350 150 150 150 Z"/>
</svg>
```

Dieses Subelement repräsentiert ein einzelnes Segment der Anzeige (in diesem Fall das oberste). Durch das `visibility`-Attribut wird die Sichtbarkeit des Segmentes gesteuert.
Mittels JavaScript wird gesteuert, welche Segmente sichtbar geschaltet sein müssen, um welche Zahl zu realisieren. Dazu wird ein Array `segmentParts` verwendet, welches die Teile der darzustellenden Nummer zuordnet. Mithilfe der Methode `setActiveSegments` werden dann für das Display alle Segmente zunächst unsichtbar gesetzt und die gewünschten Segmente anschließend sichtbar geschaltet, indem man das `visibility`-Attribut jedes Segmentes manipuliert.
In der .html Datei der Webapp werden den eingebetteten Displays bestimmte "data"- Attribute vergeben, die anschließend bestimmen, von welchem Bauteil des Modell-SVGs die Daten für das Display genommen werden, ob die Anzeige binär oder dezimal ist und ein `data-name`, der dem Display eine kleine Beschriftung hinzufügt. Die Auswertung der vergebenen Attribute erfolgt in JavaScript.
Da im Modell zu bestimmten Momenten die Eingänge zurückgesetzt werden, würden in Folge die Displays auch zurückgesetzt werden. Um dies zu verhindern, werden die Displays abhängig vom aktuellen Takt aktualisiert, so kann man bis zum Abschluss eines Zyklus alle Zwischenergebnisse nachvollziehen. In der Methode `updateDisplaysDependentOnClock` werden also aus einem vordefinierten Array die `id`s der zu erneuernden Displays ausgelesen und das zugehörige Element ermittelt, um dann durch Hilfsmethoden die Segment-Subelemente im SVG so sichtbar/unsichtbar zu schalten, dass die korrekte Zahl angezeigt wird.

6.4 Überarbeitungen

Ralf Zücker

Um den Charakter einer Single-Page-Application zu unterstreichen, wurde eine automatische Größenskalierung hinzugefügt. Optimiert für die Darstellung im Querformat, wird die Applikation nun in den gängigen Browsern immer in der entsprechenden Zielgröße optimal angezeigt. Statt `table` wird nun `flexbox` zur Anordnung der Steuerelemente verwendet. Für eine gefälligere Präsentation gibt es eine Kopf- und Fußzeile mit Links zur Hochschule Zittau/Görlitz und dieser Belegarbeit.
Anstelle der bisher verwendeten Snap.svg[Bar16] Bibliothek zur SVG-Animation wird nun velocity.js[Sha16] verwendet. Neben der besseren Performance ist velocity.js robuster gegenüber Stromsparmechanismen des Host-Systems. Durch diese größere Stabilität des Javascript-Timers funktioniert der Automatikmodus auch fehlerlos, wenn sich der Browser-Tab im Hintergrund befindet. Die Einbindung von velocity.js sowie die unterschiedliche Behandlung von `embed` durch die verschiedenen Browser haben es erforderlich gemacht, die animierten SVGs mittels jQuery `load` nachzuladen.

7 Fazit und Ausblick

Ralf Zücker

Die hier vorgestellte HTML5 Anwendung bietet zunächst ein interaktives Modell zur Veranschaulichung der fortlaufenden Addition mit einschrittigem Übertrag nach Konrad Zuse. Die Mechanik der einzelnen Operationen ist dabei in jedem Schritt klar zu erkennen und nachvollziehbar. Die berechneten Binärergebnisse werden dem Zustand der Maschine entsprechend angezeigt und sind somit den einzelnen Takten klar zuzuordnen. Ein automatischer Kreislaufmodus rundet die für Demonstrationszwecke gedachte Applikation ab.
Durch den hohen Abstraktionsgrad der Implementation ist es möglich, nur durch geschicktes Zusammenstellen von SVG Grafiken weitere, auch komplexere Modelle zu erzeugen. Ebenso können auch verschiedene unabhängige SVGs auf der gleichen Seite eingebunden und beliebig skaliert werden.
Es ist ein universeller Baukasten zur Modellierung logischer Berechnungen mit mechanischen Bauteilen nach dem Vorbild von Konrad Zuse entstanden.

Literaturverzeichnis

[Bar16] Baranovskiy, Dmitry. *Snap.svg*. Januar 2016

[Fou16] jQuery Foundation. *jQuery*. Januar 2016

[Gra15] Grams, Timm. *Z1-Addierermodelle*. Juli 2015

[Sha16] Shapiro, Julian. *velocity.js*. Mai 2016

A Definition eigener Automaten

Ralf Zücker

SVG - Grundgerüst Jeder Automat muss als eigenständiges SVG definiert werden. Das SVG sollte als Datei mit Endung .svg vorliegen und einen ähnlichen Inhalt wie in Listing A.1 aufweisen.

Listing A.1: Einbettung des Automaten

```
1 <?xml-stylesheet href="Style.css" type="text/css"?>
2 <svg xmlns="http://www.w3.org/2000/svg"
3     width="100%" height="100%"
4     viewBox="0 0 1000 1000">
5     ...
6 </svg>
```

Eine SVG Datei muss zur Einbettung immer mit der `xml` Deklaration beginnen. Durch Import der css Datei werden die definierten graphischen Vorgaben für den Automaten übernommen. Das Basis-`svg`-Element muss unbedingt die zu nutzende SVG Spezifikation definieren. Mittels `viewbox` werden die logischen Rahmenkoordinaten für den Automaten festgelegt. Im Inneren des Basis-Elements können nun die einzelnen Bauteile definiert werden.

SVG - Einfache Bauteile Ein Bauteil besteht aus einem äußeren `svg` Element und einem inneren Zeichenobjekt. Das `svg` Element dient der Positionierung und Klassifizierung. Das innere Element wird zur graphischen Darstellung und für Klick-Detektion benutzt.

Listing A.2: Einfache Bauteile

```
1 <svg id="elementID" x="100" y="50"
2     data-bit="0" data-bitpos="right">
3     <path class="trigger" p="M 0 0 L 50 0 50 100 0 100 Z"/>
4     </svg>
```

Jedes `svg` Element sollte eine eindeutige `id` haben. Die Attribute `x` und `y` geben die Position des Bauteils innerhalb des Basis SVGs an. Das Attribut `data-bit` definiert den internen Startzustand und `data-bitpos` die graphische Startstellung des Elements. Es können auch mehrere Zeichenobjekte innerhalb eines `svg` Elements definiert werden, um komplexere Strukturen zu erzeugen. Bei nur eindimensional beweglichen Objekten gibt es die Zustände `0` und `1` und die Richtungen `left`, `right`, `up` und `down`. Mit jeder Bewegung des Elements wechselt Zustand und Stellung. Das in Listing A.2 definierte Bauteil ist ein einfaches Rechteck, welches sich beim ersten Klick nach links verschiebt und dabei seinen `data-bit` Wert auf `1` ändert. Eine solche Verschiebung wird intern immer dadurch realisiert, dass der Wert des `x` bzw. `y` Attributs um 30 Einheiten verringert oder erhöht wird.
Das innere Zeichenobjekt kann im Prinzip beliebig definiert werden. Das `class` Attribut ist jedoch wichtig für die Zuordnung der graphischen Repräsentation (CSS) sowie zur Klick-Detektion.

SVG - Sticks Sticks sind Elemente, die vier verschiedene Stellungen aufweisen können. Das Attribut `data-bit` kodiert die Anfangsstellung. Hierbei entsprechen die Werte 1, 2, 3 und 4 den absoluten Positionen links oben, rechts oben, rechts unten und links unten. Wie in Listing A.3 zu erkennen, muss das `svg` Element die Klasse `stickItem` haben, um als solches erkannt zu werden.

Listing A.3: Definition Stick

```
<svg id="AND1stick1" class="stickItem"
    x="95" y="165"
    data-bit="1"
    data-move="AND1linker1 AND1trigger1">
    <circle class="stick" cx="10" cy="10" r="10"/>
</svg>
```

Im `data-move` Attribut werden die `id`s der `svg` Elemente angegeben, die durch den Stick ziehend und schiebend beeinflusst werden können. Nach dem gleichen Prinzip können auch allen anderen `svg` Elemente entsprechende Abhängigkeiten gegeben werden.

SVG - Linker Linker sind die Elemente mit Eckausschnitten zur Kodierung der Schaltglieder. Die graphischen Ausschnitte in den Elementen sollten der logischen Definition entsprechen.

Listing A.4: Definition Linker

```
<svg id="AND1linker1" class="linkerItem"
    x="80" y="150"
    data-bit="0"
    data-bitpos="up"
    data-move="AND1stick1 AND1linker2"
    data-type="124">
    <path class="linker" data-draw="linker124-top"/>
</svg>
```

Listing A.4 zeigt, neben den üblichen Definitionen für `data-bit`, `data-bitpos` und `data-move`, dass ein Linker die Klasse `linkerItem` haben muss. Als einziger Bauteiltyp muss hier das Attribut `data-type` definiert werden. Es handelt sich hierbei um die logische Definition des Ausschnitts. `124` bedeutet, dass die Quadranten 1, 2 und 4 (links oben, rechts oben und links unten) ausgeschnitten sind. Dabei ist zu beachten, dass diese Quadrantenangabe relativ zur Lage des Linkers angegeben werden muss. Oben ist bei einem Linker immer dort, wo sich das `data-bit 0` befindet. In obigem Beispiel ist es oben. Hätte `data-bitpos` den Wert `down` oder wäre `data-bit 0`, dann müsste die `data-type` Definition um 180° gedreht werden. Die

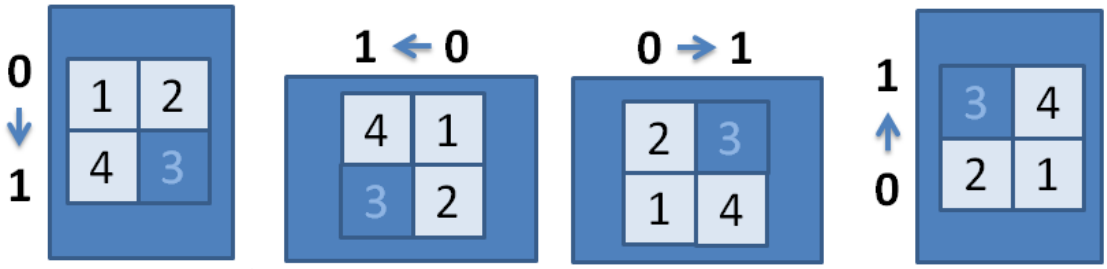

Abbildung A.1: Relative Linker-Ausschnitte

Abbildung A.1 zeigt den gleichen Ausschnitt (mit gleicher `data-type` Definition) in den unterschiedlichen Lagen des Elements.

SVG - Sonstiges In Listing A.4 wurde das Attribute `data-draw` verwendet. Dieses wird bei Initialisierung des Elements in einen hinterlegten Pfad für das `p` Attribut des `svg` aufgelöst. Auf diese Weise müssen graphisch identische Bauteile nicht mehrfach im SVG definiert werden.
Analog zu den `data-move` Attribute können auch die `data-push...` Attribute verwendet werden, um Abhängigkeiten zu definieren, die nur drückend wirken und

nicht ziehend. Die Attributnamen `data-pushleft`, `data-pushright`, `data-pushup` und `data-pushdown` sind möglich. Zugewiesen werden, wie auch bei `data-move` die `id`s der abhängigen `svg`-Elemente.

HTML Dokument Das SVG kann an beliebiger Stelle beliebig skaliert in das HTML Dokument eingebunden werden, wie in Listing A.5 ersichtlich. Auch die Einbindung mehrerer unabhängiger SVGs ist möglich.

Listing A.5: Einbettung des Automaten

```
<embed id="zuseAutomaton"
    src="Z1-Prototype.svg"
    width="1500" height="1000"
    align="left" />
```

Es sind keine `id` oder `class` Definitionen notwendig, werden aber für den weiteren Zugriff und die Gestaltung mit CSS empfohlen.

Listing A.6: Initialisierung

```
window.onload = function () {
    jQuery.each($('.trigger, .setter'), function(index, entry) {
        entry.addEventListener("click", onClick);
    });
}
```

Die Funktion `loadSVG` kann unter Angabe des SVG-DOM-Elements aufgerufen werden, um eingebettete `data-draw` Attribute aufzulösen. Zur Benutzung des definierten Automaten reicht es aber aus, an den klickbaren Elementen die Funktion `onClick` zuzuordnen, wie Listing A.6 zeigt. Alle weiteren Abhängigkeiten werden direkt aus den Definition im SVG bezogen.

B Nutzungsanleitung

Alexander Preuß

Funktionsmodi Das Modell besitzt drei Funktionsmodi. Im automatischen Loopmodus werden automatisch Eingabezahlen gesetzt und das Modell führt die Berechnung im Kreislauf endlos fort. Dieser ist als eine Art Demomodus vorgesehen. Der Loopmodus kann per Button gestartet werden, beginnt nach einer Zeit der Inaktivität aber auch von selbst. Gestoppt werden kann er ebenfalls über einen Button.
Der zweite Modus ist der Automatikmodus, hierbei werden die vom Nutzer vorgegebenen Zahlen und Rechenoperation berücksichtigt und ein Zyklus der Rechnung wird komplett simuliert.
Der dritte Funktionsmodus ist der freie Modus. Hier kann der Nutzer selbst bestimmen welche Rechenoperation mit welchen Eingabewerten verwendet wird. Die nötigen Aktionen wie das Auslösen der Takte werden vom Nutzer selbst vorgenommen. Dazu kann man entweder die Elemente in der Darstellung direkt anklicken oder im Steuerfeld mit den jeweiligen Buttons die Aktion auslösen.
In allen drei Modi werden im Steuerfeld mithilfe der Displays die Zwischenergebnisse der Berechnung dokumentiert.

Buttons Im Steuerfeld findet sich eine Vielzahl von Buttons. Tabelle B.1 veranschaulicht die zugehörigen Funktionen.

Detailansichtfenster Im Detailansichtfenster kann der Nutzer weitere Informationen über ein Bauteil erhalten. Dazu ist das Bauteil selbst grafisch dargestellt und alle Bleche des Bauteils können durch Klicken verschoben werden. So kann der Nutzer die Funktionsweise des Bauteils und dessen Wirkung im großen Modell nachvollziehen.
Des Weiteren sind vier Buttons im Fenster enthalten. Der “Explosivdarstellung

an/ausschalten"-Button versetzt das Bauteil in eine Explosivdarstellung, in der weiterhin die Funktionsweise durch Klick erkundet werden kann. Ein erneuter Klick versetzt das Bauteil wieder in seine normale Ansicht.
Die anderen drei Buttons schalten die Sichtbarkeit der Schichten der Bleche an und aus. So kann man um einen besseren Blick auf die untere und mittlere Ebene der Bleche zu erhalten, mit dem "Sichtbarkeit der oberen Ebene umschalten"-Button die oberste Schicht unsichtbar schalten. Ein erneuter Klick stellt die Sichtbarkeit der jeweiligen Schicht wieder her.
Der "Detailansicht schließen"-Button in der oberen rechten Ecke schließt das Detailansichtfenster. Ein Klick auf den Bereich außerhalb des Detailansichtfensters hat die gleiche Reaktion zur Folge.

Tabelle B.1: Buttons und ihre Funktion

Button	**Funktion**
Overlay an/ausschalten	Schaltet das Bauteiloverlay an/aus, die Zahlen am Overlay zeigen den Status der Eingangsbleche an
Rückkopplung an/ausschalten	Schaltet die Rückkopplung an/aus
Simulation zurücksetzen	Versetzt die Simulation in ihren Anfangszustand
Loopmodus starten	Startet den Loopmodus
Simulation anhalten	Unterbricht die Simulation im Loop- & Automatikmodus
Zeit zwischen Aktionen verringern	Verringert die Zeit zwischen Aktionen im Loop- & Automatikmodus
Zeit zwischen Aktionen erhöhen	Erhöht die Zeit zwischen Aktionen im Loop- & Automatikmodus
Einen Zyklus simulieren	Startet den Automatikmodus
Zahl A erhöhen	Erhöht den Wert von A
Vorzeichen umschalten	Schaltet die Rechenoperation zwischen Addition und Subtraktion um
Zahl B erhöhen	Erhöht den Wert von B
Nächsten Schritt ausführen	Führt die nächste Aktion im Kreislauf aus (ähnlich einem Debugger im Einzelschrittmodus)
Letzten Schritt rückgängig machen	Setzt die Simulation um einen Schritt zurück
Takt 1 aktivieren	Löst Takt 1 aus
Takt 2 aktivieren	Löst Takt 2 aus
Takt 3 aktivieren	Löst Takt 3 aus
Takt 4 aktivieren	Löst Takt 4 aus
Detailansicht AND öffnen	Öffnet Detailansicht zum Bauteil AND
Detailansicht XOR öffnen	Öffnet Detailansicht zum Bauteil XOR
Detailansicht ONE Var. A öffnen	Öffnet Detailansicht zum Bauteil ONE Var. A
Detailansicht ONE Var. B öffnen	Öffnet Detailansicht zum Bauteil ONE Var. B

Eidesstattliche Erklärung

Hiermit erklären wir, dass wir diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Görlitz, 31. Juli 2016

Alexander Preuß & Ralf Zücker