Universität Bremen

Praktische Informatik 3 WS 22/23

Christoph Lüth
Raphael Baass
Thomas Barkowsky
Tede von Knorre
Alexander Krug
Tarek Soliman

3. Übungsblatt

Ausgabe: 07.11.22
Abgabe: 13.11.22

Wir können in einer funktionalen Sprache nicht nur Dinge wie Krämerläden und Eiscafés behandeln, sondern auch ganz handfeste Dinge wie Hardware modellieren. Deshalb wollen wir in dieser Übung Hardware in Haskell implementieren — nicht nur weil es so schön alliteriert.

3.1 *Bits and Bytes.* *3 Punkte*

Wir fangen an mit Signalen (einzelnen Bits). Diese können `O` (low oder *false*) und `I` (high oder *true*) sein

```
data S = O | I   deriving (Eq, Show)
```

Für Signale gibt es vordefiniert zwei Konvertierungsfunktionen von und nach `Int`:

```
s2i :: S→ Int
i2s :: Int→ S
```

sowie Funktionen für Konjunktion, Disjunktion und Negation (im Gegensatz zu den Funktionen auf `Bool` sind diese *strikt*):

```
(.&.) :: S→ S→ S
(.|.) :: S→ S→ S
neg :: S→ S
```

Listen von Signalen sind *Wörter*. Implementieren Sie zwei Funktion, welche Wörter in ganze Zahlen und zurück übersetzen. Die Wörter sollen LSB-first sortiert sein, d.h. das niedrigstwerte Bit (LSB) steht am Anfang der Liste (links):[1]

```
w2i :: [S]→ Int
i2w :: Int→ Int→ [S]

w2i [O, I, I, O]  ⇝  6
i2w 6 11  ⇝  [I,I,O,I,O,O]
```

Das erste Argument von `i2w` ist die *Wortbreite*, das Ergebnis wird mit `O` rechts aufgefüllt oder ggf. abgeschnitten. Generell ist `length (i2w n xs) == n`.

Jetzt wollen wir Schaltkreise modellieren, konstruieren, und simulieren. Alle Schaltkreise bestehen aus folgenden simplen drei Grundelementen, oder sind ein konstantes Signal:

```
data G = AND G G   -- AND-Gatter
       | OR  G G   -- OR-Gatter
       | NOT G     -- NOT-Gatter
       | CONST S   -- Konstantes Signal S
       deriving (Eq, Show)
```

Einen Schaltkreis können wir *simulieren*, indem wir konstante Signale auswerten, und die Gatter mit den entsprechenden Funktionen auf Signalen (Konjunktion, Disjunktion, Negation) auswerten. Implementieren Sie diese Funktion:

```
sim :: G→ S

sim (AND (NOT (CONST I)) (OR (CONST I) (CONST O)))  ⇝  O
```

Aus diesen einfachen Schaltkreisen können wir anspruchsvollere konstruieren. Als erstes konstruieren wir ein exklusives Oder. Das exklusive Oder ist wie folgt definiert:

$$A \oplus B = (A \wedge (\neg B)) \vee (\neg A \wedge B)$$

[1] Das vereinfacht die Konversion.

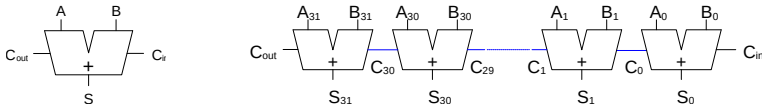

Abbildung 1: 1-Bit Volladdierer (links), 32-Bit Ripple-Carry-Addierer (rechts)
Quelle: Harris & Harris, *Digital Design and Computer Architecture*, 5^{th} Edition, Elsevier 2022.

Definieren Sie eine Funktion, welche für zwei Eingänge das exklusive Oder (Xor) berechnet:

```
xor :: G→ G→ G
```

Man beachte, dass dieser Schaltkreis zwei Schaltkreise als Argumente (d.h. Eingänge) hat, genau wie der Konstruktor `AND` auch zwei Schaltkreise als Argumente (Eingänge) hat.

Mit der Simulationsfunktion können wir unseren Schaltkreis simulieren. Es sollte beispielsweise gelten

```
sim (xor (CONST I) (CONST I))  ⇝  O
sim (xor (CONST I) (CONST O))  ⇝  I
```

3.2 *Addition* *3 Punkte*

Jetzt wollen wir einen Schaltkreis konstruieren, der rechnet —konkret zwei Binärzahlen kodiert als Wörter gleicher Länge *addiert*. Die Konstruktion erfolgt in zwei Schritten. Zuerst konstruieren wir einen Volladdierer, der zwei Bits A, B und einen Übertrag C_{in} addiert, und als Resultat ein Bit S und einen Übertrag C_{out} ausgibt:

```
fullAdder :: G→ G→ G→ (G, G)
```

Die Ausgaben des Volladdierers (siehe Abb. 1 links) sind wie folgt definiert:

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = (A \wedge B) \vee (A \wedge C_{in}) \vee (B \wedge C_{in})$$

Der Volladdierer kann mit der vordefinierten Funktion `sim2` simuliert werden, mit beispielsweise folgenden Ergebnissen:

```
sim2 :: (G, G)→ (S, S)
sim2 (g1, g2) = (sim g1, sim g2)

sim2 (fullAdder (CONST O) (CONST I) (CONST O))  ⇝  (I, O)
sim2 (fullAdder (CONST I) (CONST O) (CONST I))  ⇝  (O, I)
```

Um Wörter der Länge n zu addieren, werden jetzt n Volladdierer verknüpft (siehe Abb. 1).[2] Der i-te Volladierer addiert die i-ten Bits der Eingabewörter und den Übertrag des $i-1$-ten Volladdierers (für den ersten ist der Übertrag `O`), das Ergebnis ist das i-te Bit des Ergebniswortes und der Übertrag geht in den nächsten Volladdierer. Folgende Funktion konstruiert einen n-bit Ripple-Carry-Addierer (wobei n die Länge der beiden Eingabelisten ist; die Funktion soll undefiniert sein, wenn beide Eingabelisten ungleiche Länge haben):

```
rcAdder :: [G]→ [G]→ ([G], G)
```

Um den Addierer zu simulieren, implementieren Sie eine `sim` Funktion für Listen:

```
sims :: [G]→ [S]
```

Damit können wir den Addierer simulieren, mit folgenden Ergebnissen:

```
sim3 :: ([G], G)→ ([S], S)
sim3 (g1, g2)= (sims g1, sim g2)

sim3 (rcAdder [CONST I, CONST O] [CONST O, CONST I])  ⇝  ([I,I], O)
sim3 (rcAdder [CONST I, CONST O, CONST I] [CONST O, CONST I, CONST I])  ⇝  ([I, I, O], I)
```

[2] Dies ist ein *ripple-carry-adder*; der Vollständigkeit halber sei darauf hingewiesen, dass es effizientere Addierer gibt.

3.3 *Testen ist gut, erschöpfendes Testen ist besser.* *4 Punkte*

Wir haben oben unsere Schaltkreise an einzelnen Eingaben getestet; jetzt wollen wir das systematischer angehen. Dazu schreiben wir zuerst eine *Spezifikation*, welche für jede mögliche Eingabe die erwünschte Ausgabe berechnet:

```
xorSpec    :: S→ S→ S
adderSpec  :: S→ S→ S→ (S, S)
rcAdderSpec :: [S]→ [S]→ ([S], S)
```

Die Spezifikation für Xor ist einfach eine Wertetabelle (vorgegeben). Für den Volladdierer ist das Ergebnis

$$R = (\texttt{s2i}(A) + \texttt{s2i}(B) + \texttt{s2i}(C_{in})) \mod 2$$
$$C_{out} = (\texttt{s2i}(A) + \texttt{s2i}(B) + \texttt{s2i}(C_{in})) \div 2$$

(Hier berechnet $x \mod 2$ das letzte Bit des von x, und $x \div 2$ das erste Bit von x; $\div$ ist die ganzzahlige Operation.) Für den RC-Addierer für zwei Wörter v, w der Länge n entsprechend

$$R = (\texttt{w2i}(V) + \texttt{w2i}(W)) \mod 2^n$$
$$C_{out} = (\texttt{w2i}(V) + \texttt{w2i}(W)) \div 2^n$$

(wobei hier $x \mod 2^n$ die ersten n Bits von x berechnet, und $x \div 2^n$ das $n+1$-te Bit.) Wichtig ist hierbei, dass die Spezifikation die Addition auf den ganzen Zahlen durchführt, und so die mathematische Korrektheit der Operation sicherstellt. Beispielwerte sind

```
adderSpec I O I  ⇝  (O,I)   -- 1+0+1 = 2
rcAdderSpec [I, O, I, I, O] [I, O, I, O, I]  ⇝  ([O,I,O,O,O],I) -- 13+21 = 34
```

Jetzt müssen wir nur noch testen. Dazu schreiben wir zuerst eine Funktion

```
allWords :: Int→ [[S]]
```

welche alle Wörter der Länge n erzeugt (wobei die Reihenfolge der Wörter in der Liste keine Rolle spielt):

```
allWords 2  ⇝  [[O,O],[O,I],[I,O],[I,I]]
length (allWords 16)  ⇝  65536
```

Mit dieser Funktion lassen sich drei Test-Funktionen implementieren:

```
testXor :: [(S, S)]
testAdder :: [(S, S, S)]
testRCAdder :: Int→  [([S], [S])]
```

Diese sollen für alle möglichen Eingabekombinationen den simulierten Wert mit der Spezifikation vergleichen, und die Eingabekombinationen zurückgeben, bei denen Spezifikation und simuliertes Ergebis *nicht gleich* sind. Dabei müssen wir aus den Signalen mit dem Konstruktor `CONST` Eingaben für den Schaltkreise; für `xor` ist das beispielsweise für die Signale `a` und `b`:

```
sim (xor (CONST a) (CONST b)) == xor_spec a b
```

Für den `rc_adder` ist der Parameter n die Wortbreite; hier benötigen wir eine zu implementierende Hilfsfunktion

```
w2g :: [S]→ [G]
```

die ein Word (eine Liste von Signalen) in eine Liste von `CONST`-Schaltkreisen umwandelt.

Hinweis: Für den `rc_adder` benötigen wir alle möglichen Kombinationen aus zwei Wörtern der Länge n. Das können wir am einfachsten erzeugen, indem wir alle Wörter der Länge $2n$ erzeugen, und diese mit der vordefinierten Funktion `splitAt` in der Mitte teilen.

Wenn alles funktioniert, sollten diese Funktionen jeweils eine leere Liste zurückgeben. Das kleine Beispiel zeigt auch, das erschöpfendes Testen keine Antwort ist; der RC-Addierer benötigt auf meinem Rechner schon bei 8 Bits Wortbreite 70 Sekunden und 53 GB Speicher (und über eine Stunde für 10 Bits).