Architekturen und Organisation von Rechensystemen

Andreas Koch

Wintersemester 2018-2019

Inhaltsverzeichnis

Verhaltens-beschreibung, Struktur	Bluespec	Synthetisierbare RTL HDL	High-Level Synthese
Verhalten	Atomare Regeln	Synchrone Schaltungen	Sequentielle Programmierung.
Schnittstellen	Atomare Methoden	Im wesentlichen Drähte	In der Regel nur auf oberster Hierarchieebene.
Einfluss auf Architektur	Stark	Stark	Schwach
Typprüfungen	Stark	Schwach bis Mittel (VHDL)	
Typen	Mächtig, auch benutzerdefiniert	Bits, schwach benutzerdefiniert	Mittel
Parametrisierung	Mächtig	Schwach	Schwach

Tabelle 1: Vergleich vom Hardware-Beschreibungssprache

1 Einführung in Bluespec

Bluespec ist eine Hardwarebeschreibungssprache, mit der sowohl das *Verhalten* von Hardware (was soll eine Struktur tun?), als auch deren *Struktur* (wie soll es aufgebaut sein?) beschrieben werden kann.

1.1 Verhaltensbeschreibung

Bluespec nutzt *atomare Transaktionen*, um parallele Abläufe zu Formulieren. Tabelle 1 zeigt die Unterschiede zu anderen Systemen.

1.2 Strukturbeschreibung

Die Strukturbeschreibung ist angelehnt an die mächtige Programmiersprache Haskell. Dadurch hat es ein ausdrucksstarkes Typsystem, strenge Typprüfung und eine mächtige Parametrisierung. Tabelle 1 zeigt die Unterschiede zu anderen Systemen. Bluespec erlaubt es, eigene Typen zu definieren und zu benutzen, und das mächtige Typensystem kann diese Prüfen.

1.3 Workflow

Wie funktioniert eigentlich der Workflow von Bluespec-Projekten? Der Bluespec Quelltext beinhaltet sowohl die Beschreibung als auch Tests. Der Bluespec-Compiler

kann ein Projekt nativ kompilieren, um es mit Bluesim oder dem SystemC Plugin ausführen zu können. Ebenso kann der Code in eine andere *Register Transfer Language*, beispielsweise Verilog, übersetzt werden. Der Verilog-Code kann dann ebenso zur Simulation, oder aber zur FPGA bzw. ASIC-Synthese genutzt werden.

Ein Vorteil davon, dass der Bluespec-Compiler den Code nativ mit Bluesim simulieren kann ist, dass die Simulationsgeschwindigkeit wesentlich schneller ist, als generierten Verilog-Code zu simulieren.

1.4 Weitere Informationen

Es gibt, als Teil der Bluespec-Distribution, sowohl Englische Folien, als auch eine Sprachspezifikation, ein Benutzerhandbuch und das Buch *Bluespec by Example* inklusive Beispiele.

1.5 Multiplikation

Jetzt schauen wir uns Bluespec anhand eines Beispiels an.

```
interface Mult_ifc;
    method Action               put_x (int xx);
    method Action               put_y (int yy);
    method ActionValue #(int)   get_w ();
endinterface: Mult_ifc
```

Hier haben wir ein neues Interface definiert, welches wir `Mult_ifc` genannt haben. Dieses Interface besitzt drei Methoden. Die Methoden `put_x` und `put_y` nehmen jeweils einen Integer und setzen die Multiplikanden (denn das sind *Action*-Methoden), und die Methode `get_w`, die das Ergebnis der Multiplikation nach Außen liefert (denn es ist eine *ActionValue*-Methode).

```
module mkTestbench (Empty);
    Mult_ifc m <- mkMult;

    rule gen_x;
        m.put_x (9);
    endrule

    rule gen_y;
        m.put_y (5);
    endrule

    rule drain;
        let w <- m.get_w ();
        $display ("Product = %d", w);
```

```
        $finish ();
    endrule
endmodule: mkTestbench
```

Hier haben wir eine Testbench. Dieser Testrahmen wird mit dem Namen `mkTestbench` definiert, wobei `mk` ein Präfix ist, der bei Bluespec-Projekten generell immer für module benutzt wird. Diese Testbench bekommt einen Multiplier, und sie hat Regeln, die den Multiplier mit Input versorgen und den Output auslesen.

Bei Bluespec müssen alle Ausdrücke (Expressions) in solchen *Rules*, also Regeln, stehen. Wenn nichts anderes angegeben wird, dann werden diese Regeln parallel ausgeführt.

```
module mkMult (Mult_ifc);
    Reg #(int)  w     <- mkRegU;
    Reg #(int)  x     <- mkRegU;
    Reg #(int)  y     <- mkRegU;
    Reg #(Bool) got_x <- mkReg (False);
    Reg #(Bool) got_y <- mkReg (False);

    rule compute ((y != 0) && got_x && got_y);
        if (lsb(y) == 1) w <= w + x;
        x <= x << 1;
        y <= y >> 1;
    endrule

    method Action put_x (int xx) if (! got_x);
        x <= xx; w <= 0; got_x <= True;
    endmethod

    method Action put_y (int yy) if (! got_y);
        x <= xx; w <= 0; got_x <= True;
    endmethod

    method ActionValue #(int) get_w () if ((y == 0) && got_x && got_y);
        got_x <= False; got_y <= False;
        return w;
    endmethod
endmodule: mkMult
```

Hier haben wir jetzt das Modul definiert, welches die Multiplikation ausführt. Wie auch die Testbench haben wir dem Module einen Namen mit dem Präfix `mk` gegeben. Außerdem haben wir angedeutet, dass es das Interface `Mult_ifc` implementiert. Wir können sehen dass das Module mehrere Register besitzt. Die ersten drei Register, `w`, `x` und `y`, sind alles `RegU`, was *Uninitialized Register* bedeutet, diese haben also keinen Startwert. Anders bei `got_x` und `got_y`, diese sind beide Booleans mit dem Startwert *False*.

1.6 Bluespec Praxis

Um dieses Module praktisch auszuprobieren, organisiert man das Ganz typischerweise in zwei Dateien auf. In die erste Datei, die die Testbench beinhaltet, nennt man `Testbench.bsv` und fügt ein paar Zeilen hinzu. Dabei steht die Endung `.bsv` für *Bluespec SystemVerilog*, denn Bluespec hatte ursprünglich einen Haskell-Syntax, welcher aber nicht wirklich gut angenommen wurde, und man deswegen schlussendlich zu einen SystemVerilog-ähnlichen Syntax gewechselt hat.

```
package Testbench;
import Mult::*;

module mkTestbench (Empty);
    ...
endmodule: mkTestbench
endpackage: Testbench
```

Hiermit definiert man ein Package namens *Testbench*, in welche man das Modul `mkTestbench` hineinpackt. Dies ist ähnlich wie Namespaces in C++, zum Beispiel. Außerden kann man mit der `import`-Anweisung ein anderes Package importieten. Hier importieren wir das Package names *Mult*. Damit können wir auch gleich die andere Datei, die wir `Mult.bsv` nennen,

1.7 Bluespec Ausführung

Wie kann man Bluespec-Code denn jetzt eigentlich ausführen? Das geht relativ leicht, aber nur auf den RBG-Rechnern, da Bluespec kommerziell und Closed-Source ist. Dazu muss man einmal die RBG-Lizenz initialisieren.

```
$ export LM_LICENSE_FILE=27002@license.rbg.informatik.tu-darmst…
```

Sobald das erledigt ist, kann man den Bluespec-Compiler ausführen.

```
$ bsc -sim -g mkTestbench -u Testbench.bsv
```

Die Optionen hier teilen dem Compiler mit, dass er Code für den internen Simulator erzeugen soll (`-sim`), dass das oberste Module `mkTestbench` ist, und dass er auch Untermodule übersetzen soll. Der Bluespec-Compiler geht davon aus, dass jedes Package in einer Datei mit dem Packagenamen existiert. Bevor man den Code jetzt simulieren kann, muss diese erstmal gelinked werden. Dazu nutzt man ebenso den Bluespec-Compiler, aber mit anderen Optionen.

```
$ bsc -sim -e mkTestbench -o myFirstModel
```

Hierbei muss man dem Compiler mitteilen, dass man an einer Simulation interessiert ist mit dem Parameter `-sim`, den Einsprungpunkt des Moduls mit dem Parameter `-e` bestimmen (in diesem Fall `mkTestbench`) und den Dateinamen, in dem das

Simulationsprogramm geschrieben werden soll. Hier generiert Bluespec Maschinencode. Um die Simulation dann zu started, muss einfach das resultierende Programm ausgeführt werden.

```
$ ./myFirstModel
Produkt =           45
```

Wichtig ist hier, dass in der `mkTestbench` die Anweisung `$finish();` vorkommt, ansonsten hält die Simulation nicht an. Der Kompilationsprozess kann mit einer `Makefile` automatisiert werden, oder man kann die Bluespec IDE benutzen, die in der Vorlesung aber nicht benutzt wird.

> **Anmerkung**
> Professionelle Programmierer, die mit Vim arbeiten, können sich ein Vim-Plugin herunterladen, mit dem man Syntax-Highlighting und Auto-Indentation bekommt. Das macht das Arbeiten mit dem Code wesentlich angenehmer[1].

[1] Siehe https://github.com/mtikekar/vim-bsv.