Vorlesung Architekturen und Entwurf von Rechnersystemen

Prof. Andreas Koch, Yannick Lavan, Johannes Wirth, Mihaela Damian

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2022/2023
Übungsblatt 2

Diese Übung befasst sich mit der Übersetzung von Bluespec zu Verilog und dem Zeitverhalten einzelner Module.

2.1 Interface Übersetzung

Gängige Design-Tools, wie zum Beispiel Xilinx Vivado, können nicht direkt Hardware aus BSV Code erzeugen. Der Bluespec-Code muss vorher zu Verilog kompiliert werden, damit die Toolchain damit arbeiten kann. In dieser Aufgabe beschäftigen Sie sich nur mit der Übersetzung von Bluespec Interfaces zwischen BSV und Verilog. Verwenden Sie zur Lösung dieser Aufgabe ***nicht*** den Bluespec Compiler. Stattdessen können die Folien 152ff im BSV Foliensatz und Kapitel 14.2 im BSV Reference Guide als Unterstützung dienen.

2.1.1 BSV zu Verilog

a) Gegeben sei folgender BSV-Code:

```
interface Foo;
        method Action putX(Int#(32) px);
        method Bool getEven();
endinterface

module mkEven(Foo);
        /*
        Some fancy code...
        */
endmodule
```

Skizzieren Sie das Modul als einen [Block] und geben Sie die synthetisierten Signale an. Zeichnen Sie mit Pfeilen ein, ob es sich bei einem Signal um Input oder Output handelt.

b) Gegeben sei folgender BSV-Code:

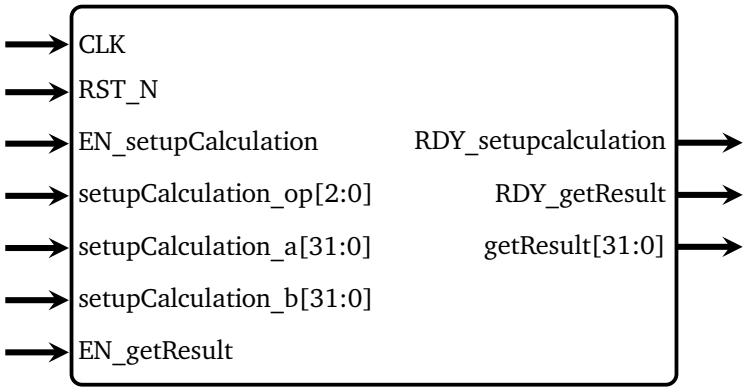
```
interface Bar;
        (* always_ready, prefix="" *)
        method Action putX((* port="x" *)UInt#(32) px);
        method UInt#(32) getY();
endinterface

module mkBar(Bar);
        /* Hardware that solves all human problems... */
endmodule
```

Geben Sie für den Bluespec-Code den erzeugten Verilog-Code des Interfaces an.

2.1.2 Verilog zu BSV

a) Gegeben sei folgendes Modul:



Geben Sie das Interface des dargestellten Moduls in BSV an. Sie können hierbei davon ausgehen, dass keine Attribute für die Umbenennung von Signalen verwendet wurden. Weiterhin können Sie davon ausgehen, dass für skalare Werte der Typ `Int` verwendet wurde. Sie dürfen das Interface nach eigenem Ermessen benennen.

b) Gegeben sei folgender Verilog-Code:

```
input  CLK;
input  RST_N;

input  ack_pirq;
output interrupt;
input  valid_v;
output ready;
input  [15 : 0] set_px;
input  EN_get;
output [31 : 0] get;
output RDY_get;
```

Rekonstruieren Sie aus dem gegebenen Code das Bluespec Interface. Auch hier wurden keine Compiler-Attribute verwendet, die den Namen von Signalen ändern. Weiterhin können Sie für skalare Werte den Typ `Int` annehmen und davon ausgehen, dass die Methodennamen keine Sonderzeichen enthalten.

2.2 Zeitverhalten

Die Entwicklung von Hardware-Beschleunigern ist häufig mit Zeitanforderungen behaftet. Um feststellen zu können, wo noch potenzieller Optimierungsbedarf besteht, ist es wichtig das Zeitverhalten eines Moduls abschätzen zu können. In dieser Aufgabe werden Sie das Zeitverhalten verschiedener Bluespec Module analysieren.

2.2.1 Maximale Taktfrequenz

Zeitanforderungen sind häufig über eine Frequenz formuliert, die mindestens erreicht werden muss. Um zu überprüfen, ob ein Modul eine Frequenz mindestens erreicht, kann die maximale Taktfrequenz eines Moduls berechnet werden. Gegeben sei folgendes BSV-Modul:

```
module mkDeriver(Derivatives);
        Wire#(Int#(16)) x <- mkDWire(0);
        Wire#(Int#(32)) fx <- mkDWire(0);
        Wire#(Int#(32)) dfx <- mkDWire(0);
        Wire#(Int#(32)) x2 <- mkDWire(0);

        rule foo;
                x2 <= extend(x)*extend(x);
        endrule

        rule calcF;
                fx <= 3*x2*extend(x) + 42;
        endrule

        rule calcDf;
                dfx <= 9*x2;
        endrule

        method Action putX(Int#(16) px);
                x <= px;
        endmethod

        method Int#(32) f();
                return fx;
        endmethod

        method Int#(32) df();
                return dfx;
        endmethod
endmodule
```

1. Skizzieren Sie den Datenpfad des Moduls.
2. Berechnen Sie für jeden Ausführungszweig des in (1) skizzierten Datenpfads die Ausführungszeit. Nehmen Sie dafür an, dass eine Addition 3ns und eine Multiplikation 6ns (jeweils mit zwei Operanden) benötigt.
3. Geben Sie nun unter Betrachtung des kritischen Pfades die maximale Taktfrequenz für das Modul mkDeriver an.

Hinweis: Bedenken Sie auch welche Operationen parallel zueinander ausgeführt werden können.

2.2.2 Maximale Taktfrequenz 2

Gegeben sei folgendes BSV-Modul:

```
interface Foo;
        method Action putX(Int#(16) px);
        method Action putY(Int#(16) py);
        method Action putZ(Int#(16) pz);
        method ActionValue#(Int#(32)) getF1();
        method ActionValue#(Int#(32)) getF2();
        method ActionValue#(Int#(32)) getF3();
```

```
endinterface

module mkFoo(Foo);
        Reg#(Int#(32)) x <- mkReg(0);
        Reg#(Int#(32)) y <- mkReg(0);
        Reg#(Int#(32)) z <- mkReg(0);
        Reg#(Bool) gotX <- mkReg(False);
        Reg#(Bool) gotY <- mkReg(False);
        Reg#(Bool) gotZ <- mkReg(False);
        Reg#(Int#(32)) f1 <- mkReg(0);
        Reg#(Int#(32)) f2 <- mkReg(0);
        Reg#(Int#(32)) f3 <- mkReg(0);
        Reg#(Bool) gotF1 <- mkReg(False);
        Reg#(Bool) gotF2 <- mkReg(False);
        Reg#(Bool) gotF3 <- mkReg(False);
        Reg#(UInt#(3)) i <- mkReg(0);
        Reg#(UInt#(3)) j <- mkReg(0);
        Reg#(UInt#(3)) k <- mkReg(0);
        Wire#(Int#(32)) x2 <- mkDWire(0);
        Wire#(Int#(32)) y2 <- mkDWire(0);
        Wire#(Int#(32)) z2 <- mkDWire(0);

        function Action setReg(Reg#(Int#(32)) r, Reg#(Bool) got, Int#(16) val);
                return action r <= extend(val); got <= True; endaction;
        endfunction

        function ActionValue#(Int#(32)) retF(Reg#(Int#(32)) f, Reg#(Bool) gotF,
 ↪  Reg#(UInt#(3)) count);
                return(actionvalue
                        gotF <= False;
                        count <= 0;
                        return f;
                endactionvalue);
        endfunction

        rule x_sqr (gotX);
                x2 <= x * x * x;
        endrule

        rule lin_y (gotY);
                y2 <= 42 * y + 1337;
        endrule

        rule sthZ (gotZ);
                z2 <= (z + 14) << 2;
        endrule

        rule calcF1 (!gotF1 && i < 7);
                f1 <= f1 + 2 * x2 + 3 * z2;
                i <= i + 1;
        endrule

        rule calcF2 (!gotF2 && j < 5);
                f2 <= f2 + x2 + y2 + z2;
                j <= j + 1;
```

```
62          endrule
63
64          rule calcF3 (!gotF3 && k < 6);
65                  f3 <= f3 + x2 * y2 + (z2 - 42) << 2;
66                  k <= k + 1;
67          endrule
68
69          rule finI (i == 7 && !gotF1);
70                  gotF1 <= True;
71          endrule
72
73          rule finJ (j == 5 && !gotF2);
74                  gotF2 <= True;
75          endrule
76
77          rule finK (k == 6 && !gotF3);
78                  gotF3 <= True;
79          endrule
80
81          method Action putX(Int#(16) px) if(!gotX) = setReg(x, gotX, px);
82
83          method Action putY(Int#(16) py) if(!gotY) = setReg(y, gotY, py);
84
85          method Action putZ(Int#(16) pz) if(!gotZ) = setReg(z, gotZ, pz);
86
87          method ActionValue#(Int#(32)) getF1() if(gotF1) = retF(f1, gotF1, i);
88
89          method ActionValue#(Int#(32)) getF2() if(gotF2) = retF(f2, gotF2, j);
90
91          method ActionValue#(Int#(32)) getF3() if(gotF3) = retF(f3, gotF3,
    ↪  k);
92  endmodule
```

Berechnen Sie die maximale Taktfrequenz des Moduls mkFoo. Nehmen Sie dabei die Verzögerungen aus der folgenden Tabelle an. Gehen Sie weiterhin davon aus, dass arithmetische Operationen immer paarweise (also mit zwei Eingängen) ausgeführt werden. Sie können hierbei vorgehen wie in Teilaufgabe (a). Betrachten Sie außerdem die Multiplexer, die durch explizite Guards erzeugt werden.

- `Addition`: 3 ns
- `Multiplikation`: 6 ns
- `Division`: 8 ns
- `AND, OR, XOR, NOT, ...`: 1 ns
- `Shift` (feste Distanz): 0 ns
- `Shift` (variable Distanz): 1 ns
- `Multiplexer`: 2 ns
- `Vergleich`: 4 ns
- `Dequeue, Enqueue, reg.write, reg.read`: 0 ns

Hinweis: Bedenken Sie auch welche arithmetischen Operationen parallel zueinander ausgeführt werden können.