

Vorlesung Architekturen und Entwurf von Rechnersystemen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Lösungsvorschlag

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

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.

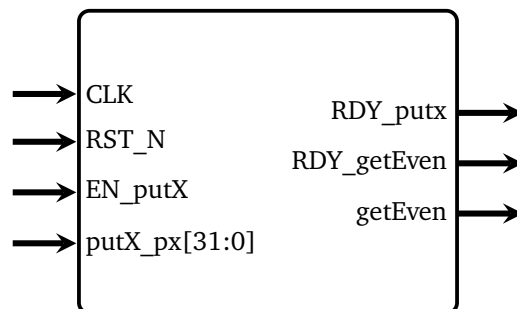
2.1.1 BSV zu Verilog

a) Gegeben sei folgender BSV-Code:

```
1 interface Foo;
2     method Action putX(Int#(32) px);
3     method Bool getEven();
4 endinterface
5
6 module mkEven(Foo);
7     /*
8     Some fancy code...
9     */
10 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.

Lösungsvorschlag:



b) Gegeben sei folgender BSV-Code:

```
1 interface Bar;
2     (* always_ready, prefix="" *)
3     method Action putX((* port="x" *)UInt#(32) px);
4     method UInt#(32) getY();
5 endinterface
6
7 module mkBar(Bar);
8     /* Hardware that solves all human problems... */
9 endmodule
```

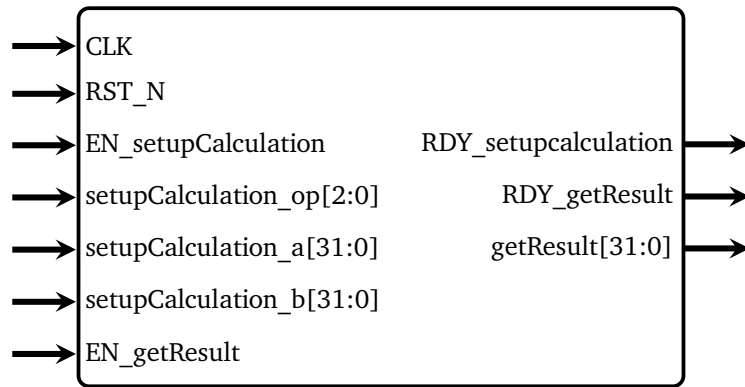
Geben Sie für den Bluespec-Code den erzeugten Verilog-Code des Interfaces an.

Lösungsvorschlag:

```
1 input CLK;
2 input RST_N;
3
4 // action method putX
5 input [31:0] x;
6 input EN_putX;
7
8 // value method getY
9 output [31:0] getY;
10 output RDY_getY;
```

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.

Lösungsvorschlag:

```

1 interface HelloALU;
2     method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b);
3     method ActionValue#(Int#(32)) getResult();
4 endinterface

```

b) Gegeben sei folgender Verilog-Code:

```

1 input CLK;
2 input RST_N;
3
4 input ack_pirq;
5 output interrupt;
6 input valid_v;
7 output ready;
8 input [15 : 0] set_px;
9 input EN_get;
10 output [31 : 0] get;
11 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.

Lösungsvorschlag:

```

1 interface FooBar;
2     (* always_enabled *)
3     method Action ack(Bool pirq);
4     (* always_ready *)
5     method Bool interrupt();
6     (* always_enabled *)
7     method Action valid(Bool v);
8     (* always_ready *)
9     method Bool ready();

```

```

10      (* always_enabled *)
11      method Action set(Int#(16) px);
12      method ActionValue#(Int#(32)) get();
13  endinterface

```

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:

```

1  module mkDeriver(Derivatives);
2      Wire#(Int#(16)) x <- mkDWire(0);
3      Wire#(Int#(32)) fx <- mkDWire(0);
4      Wire#(Int#(32)) dfx <- mkDWire(0);
5      Wire#(Int#(32)) x2 <- mkDWire(0);
6
7      rule foo;
8          x2 <= extend(x)*extend(x);
9      endrule
10
11     rule calcF;
12         fx <= 3*x2*extend(x) + 42;
13     endrule
14
15     rule calcDf;
16         dfx <= 9*x2;
17     endrule
18
19     method Action putX(Int#(16) px);
20         x <= px;
21     endmethod
22
23     method Int#(32) f();
24         return fx;
25     endmethod
26
27     method Int#(32) df();
28         return dfx;
29     endmethod
30 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.

Lösungsvorschlag:

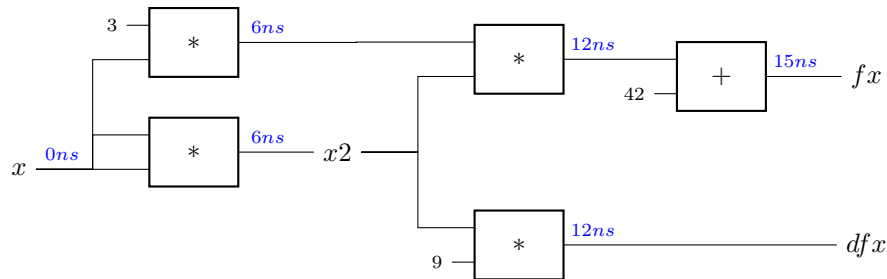


Abbildung 1: Datenpfad des Moduls aus Aufgabe 2.2.1.

- Siehe Abbildung 1. Hinweis: Es wird hier nur das gegebene Modul betrachtet. Für die finale Bestimmung der Taktperiode müsste auch die Verzögerungen für den Aufruf von putX sowie die Verwendung der Ergebnisse von f und df betrachtet werden. Außerdem wurden keine Multiplexer für die Wires benutzt, da die WILL_FIRE-Bedingungen aller Rules True ist.
- In der Abbildung sind jeweils in blau die (kumulierten) Verzögerungen eingetragen.
- Der kritische Pfad ergibt sich dementsprechend hier durch den oberen Pfad mit einer Taktperiode von 15 ns. Die maximale Taktfrequenz beträgt also $\frac{1}{15\text{ns}} = 66.67\text{ MHz}$.

2.2.2 Maximale Taktfrequenz 2

Gegeben sei folgendes BSV-Modul:

```
1 interface Foo;  
2     method Action putX(Int#(16) px);  
3     method Action putY(Int#(16) py);  
4     method Action putZ(Int#(16) pz);  
5     method ActionValue#(Int#(32)) getF1();  
6     method ActionValue#(Int#(32)) getF2();  
7     method ActionValue#(Int#(32)) getF3();  
8 endinterface  
9  
10 module mkFoo(Foo);  
11     Reg#(Int#(32)) x <- mkReg(0);  
12     Reg#(Int#(32)) y <- mkReg(0);  
13     Reg#(Int#(32)) z <- mkReg(0);  
14     Reg#(Bool) gotX <- mkReg(False);  
15     Reg#(Bool) gotY <- mkReg(False);  
16     Reg#(Bool) gotZ <- mkReg(False);  
17     Reg#(Int#(32)) f1 <- mkReg(0);  
18     Reg#(Int#(32)) f2 <- mkReg(0);  
19     Reg#(Int#(32)) f3 <- mkReg(0);  
20     Reg#(Bool) gotF1 <- mkReg(False);  
21     Reg#(Bool) gotF2 <- mkReg(False);  
22     Reg#(Bool) gotF3 <- mkReg(False);  
23     Reg#(UInt#(3)) i <- mkReg(0);  
24     Reg#(UInt#(3)) j <- mkReg(0);  
25     Reg#(UInt#(3)) k <- mkReg(0);
```

```

26 Wire#(Int#(32)) x2 <- mkDWire(0);
27 Wire#(Int#(32)) y2 <- mkDWire(0);
28 Wire#(Int#(32)) z2 <- mkDWire(0);
29
30 function Action setReg(Reg#(Int#(32)) r, Reg#(Bool) got, Int#(16) val);
31     return action r <= extend(val); got <= True; endaction;
32 endfunction
33
34 function ActionValue#(Int#(32)) retF(Reg#(Int#(32)) f, Reg#(Bool) gotF,
→ Reg#(UInt#(3)) count);
35     return(actionvalue
36         gotF <= False;
37         count <= 0;
38         return f;
39         endactionvalue);
40 endfunction
41
42 rule x_sqr (gotX);
43     x2 <= x * x * x;
44 endrule
45
46 rule lin_y (gotY);
47     y2 <= 42 * y + 1337;
48 endrule
49
50 rule sthZ (gotZ);
51     z2 <= (z + 14) << 2;
52 endrule
53
54 rule calcF1 (!gotF1 && i < 7);
55     f1 <= f1 + 2 * x2 + 3 * z2;
56     i <= i + 1;
57 endrule
58
59 rule calcF2 (!gotF2 && j < 5);
60     f2 <= f2 + x2 + y2 + z2;
61     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.

Lösungsvorschlag:

Wir stellen zunächst fest, dass die put-Methoden und get-Methoden lediglich Register lesen und schreiben und damit nicht für das Zeitverhalten der Schaltung relevant sind. Wir stellen weiterhin fest, dass die fin-Rules lediglich Register schreiben und somit nicht für das Zeitverhalten der Schaltung relevant sind. Es bleiben als nur sechs Rules übrig, für die man das Scheduling betrachten muss.

- x_sqr
- lin_y
- sthZ
- calcF1
- calcF2
- calcF3

In den Abbildungen 2 bis 7 sind die Datenpfade dieser Rules abgebildet. In den Abbildungen sind jeweils in blau die akkumulierten Verzögerungen annotiert. Beachten Sie, dass die ersten drei Rules **Wires** schreiben, die in den anderen Rules gelesen werden. Die Verzögerung bis zum Schreiben der Wires muss daher in den lesenden Rules berücksichtigt werden.

In den Abbildungen lässt sich dann ablesen, dass die größte Verzögerung 25 ns beträgt (in calcF1 beim Schreiben von f1 und calcF3 beim Schreiben von f3). Die Verzögerung des kritischen Pfads beträgt also 25 ns. Damit ergibt sich $f_{max} = 40 \text{ MHz}$.

Hinweis: In einigen Rules wurde die Reihenfolge von (zueinander kommutativen) Operationen umgestellt, um die Verzögerung zu minimieren.

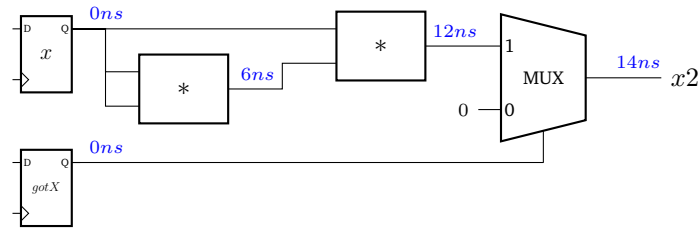


Abbildung 2: Datenpfad der Rule x_sqr des Moduls mkFoo 2.2.2.

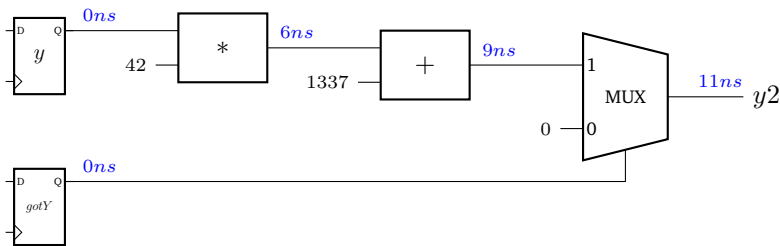


Abbildung 3: Datenpfad der Rule lin_y des Moduls mkFoo 2.2.2.

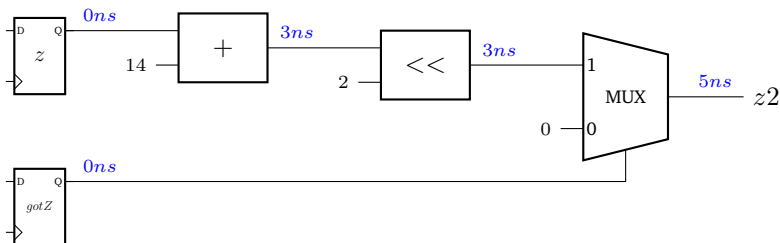


Abbildung 4: Datenpfad der Rule sthZ des Moduls mkFoo 2.2.2.

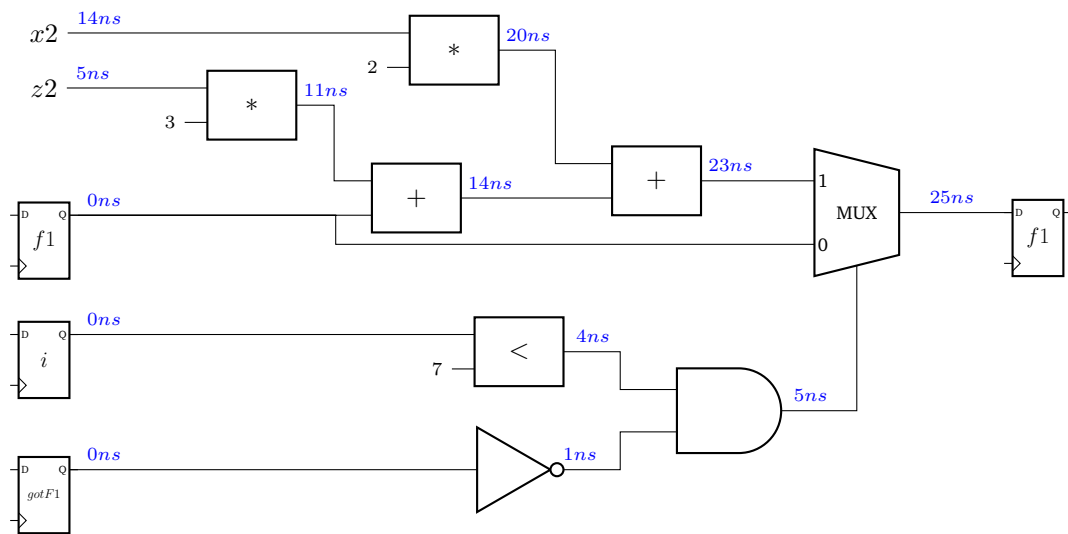


Abbildung 5: Datenpfad der Rule calcF1 des Moduls mkFoo 2.2.2.

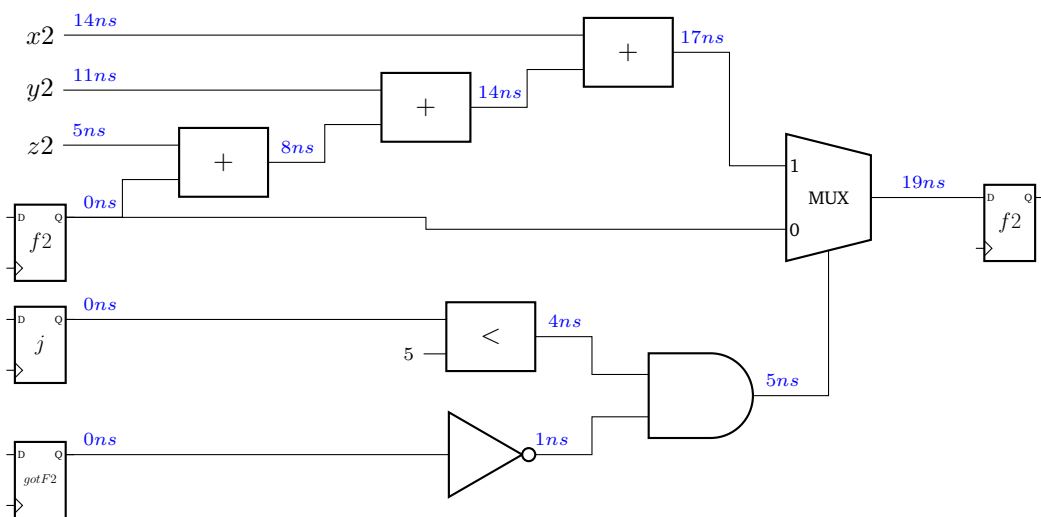


Abbildung 6: Datenpfad der Rule calcF2 des Moduls mkFoo 2.2.2.

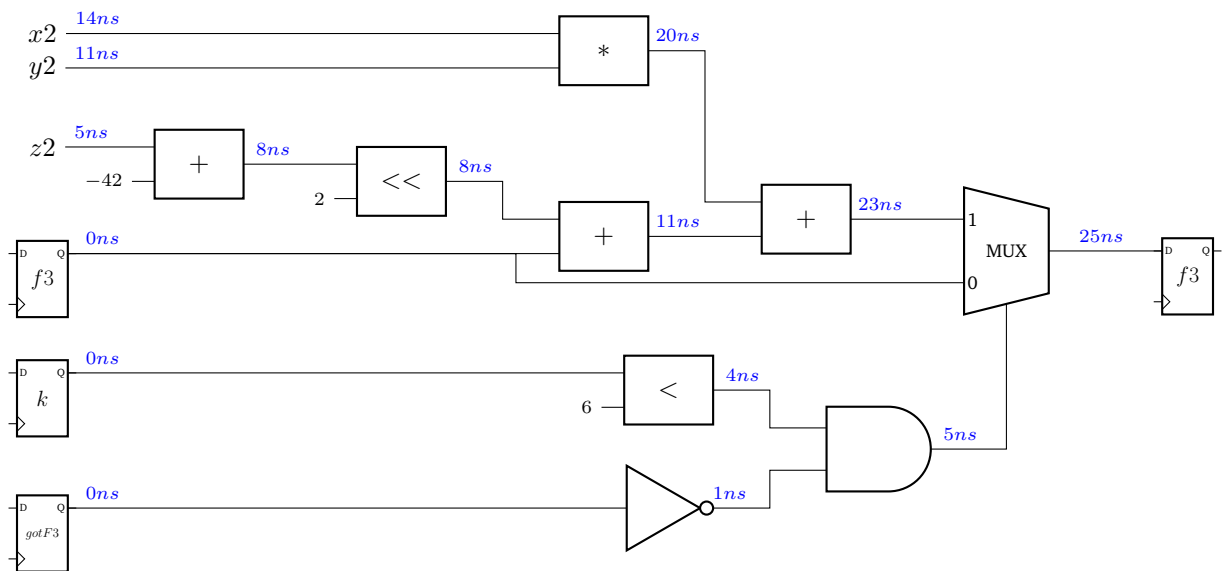


Abbildung 7: Datenpfad der Rule `calcF3` des Moduls `mkFoo 2.2.2`.