Übung zur Vorlesung Architekturen und Entwurf von Rechnersystemen

Prof. Dr-Ing. A. Koch
Jaco Hofmann, MSc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 16/17

Übungsblatt 1 - Lösungsvorschlag

Diese Übung bietet eine allgemeine Einführung in grundlegende Bluespec Sprachkonzepte.

Aufgabe 1.1 Hello Bluespec

Die erste Aufgabe dieser Übung führt in das Kompilieren von Bluespecmodulen sowie deren Simulation ein. Es wird vorausgesetzt, dass Bluespec vollständig eingerichtet ist und die benötigten Programme (`bsc` etc.) ausgeführt werden können. Dies sollte in den ISP-Rechnerpools der Fall sein. Bei Fragen zur Einrichtung stehen zum Beispiel das d120 Forum oder die der Bluespec Distribution beiliegenden Literatur zur Verfügung.

Zum Kennenlernen der Bluespec-Toolchain kann folgender Bluespec Code verwendet werden.

```
1 package HelloBluespec;
2   module mkHelloBluespec(Empty);
3     rule helloDisplay;
4       $display("(%0d) Hello World!", $time);
5     endrule
6   endmodule
7 endpackage
```

Der Dateiname muss mit dem Namen des Packages übereinstimmen und hat die Endung „.bsv“, in diesem Fall „HelloBluespec.bsv“. Zum Kompilieren und Simulieren mit Hilfe des Bluespec eigenen Simulators wird `bsc` verwendet. Als erster Schritt werden die Bluespec Module kompiliert. Dies geschieht mit dem Befehl

```
bsc -u -sim -g mkHelloBluespec HelloBluespec.bsv
```

Die eigentliche Simulation wird erstellt mit dem Befehl

```
bsc -sim -o out -e mkHelloBluespec
```

Mit Hilfe des `-o` Parameters kann der Name der Ausgabedatei verändert werden. Wurden beide Befehle erfolgreich ausgeführt, kann die Simulation mit `./out` gestartet werden.

Wie zu erwarten ist, wird in jedem Taktzyklus `Hello World!` ausgegeben. Die Simulation kann mit der Tastenkombination `Strg-C` beendet werden.

Aufgabe 1.2 Blinky

Während in Programmiersprachen häufig das „Hello World“ Programm die ersten Gehversuche darstellt, findet man in hardwarenahen Umgebungen häufig LED-Blink Programme. Dabei wird eine LED in einer festgelegten Frequenz an- und ausgeschaltet. Diese Frequenz ist typischerweise wesentlich langsamer als die Betriebsfrequenz der benutzten Hardware. Für dieses Beispiel wird angenommen, ihr Modul wird an ein 100 MHz Taktsignal angeschlossen. Um den Schaltvorgang der LED als Mensch sehen zu können, muss eine Möglichkeit gefunden werden, um aus dem schnellen Takt einen langsameren zu machen.

Aufgabe 1.2.1 Zähler

Eine einfache Möglichkeit das Taktsignal zu verlangsamen ist die Nutzung eines Zählers. Der Zähler wird mit jedem Takt um eins erhöht. Erreicht der Zähler einen Schwellenwert, wird das gewünschte Ereignis ausgeführt und der Zähler zurückgesetzt.

Ergänzen Sie das Beispiel aus Aufgabe 1.1 um einen solchen Zähler, der die Botschaft „Hello World“ alle 2^{25} Taktzyklen ausgibt. Verwenden Sie dafür ein Register des Typs `Reg#(UInt#(25))` als Zähler.

```
package HelloBluespec;
  module mkHelloBluespec(Empty);
    Reg#(UInt#(25)) counter <- mkReg(0);

    rule helloDisplay (counter == 25'h1ffffff);
      $display("(%0d) Hello World!", $time);
    endrule

    rule count;
      if(counter == 25'h1ffffff) counter <= 0;
      else                       counter <= counter + 1;
    endrule
  endmodule
endpackage
```

Aufgabe 1.2.2 LED

Erweitern Sie das Modul `mkHelloBluespec` aus Aufgabe 1.1 um einen Ausgang für die LED-Ansteuerung. Binden Sie dafür das Interface `HelloBluespec` ein.

```
interface HelloBluespec;
  (* always_enabled, always_ready *) method Bool led();
endinterface
```

Die Attribute `always_enabled` und `always_ready` spezifizieren dabei, dass die folgende Methode keine Signale zum Handshake benötigt. Dies ist hier der Fall, da die LED sich ständig in einem der beiden Zustände befindet.

Erweitern Sie das Modul um ein Register für den LED Zustand vom Typ `Bool`. Dieses Register soll alle 2^{25} Taktzyklen negiert werden. Erweitern Sie das Modul um eine Implementation der Methode `led`, die den aktuellen Zustand der LED zurückgibt.

```
package HelloBluespec;
    interface HelloBluespec;
        (* always_enabled, always_ready *) method Bool led();
    endinterface

    module mkHelloBluespec(HelloBluespec);
        Reg#(Bool) ledStatus <- mkReg(False);
        Reg#(UInt#(25)) counter <- mkReg(0);

        rule helloDisplay (counter == 25'h1ffffff);
            $display("(%0d) Hello World!", $time);
            ledStatus <= !ledStatus;
        endrule

        rule count;
            if(counter == 25'h1ffffff) counter <= 0;
            else                       counter <= counter + 1;
        endrule

        method Bool led();
            return ledStatus;
        endmethod
    endmodule
endpackage
```

Aufgabe 1.2.3 Testbench

Aktuell wird die Simulation bis zur Unterbrechung mit `Strg-C` ausgeführt. Um die Ausführung der Simulation zu steuern und das zu testende Modul beeinflussen zu können, kann eine Testbench verwendet werden. In Bluespec wird dafür ein weiteres Modul erstellt, das eine Instanz des zu testenden Moduls enthält.

Erstellen Sie ein Modul `mkHelloTestbench` mit `Empty` als Interface. Binden Sie das Modul `mkHelloBluespec` ein. Die Testbench soll für 2 Sekunden laufen (100 MHz Takt s.o.). Die Simulation kann mit `$finish()` beendet werden.

Passen Sie die Befehle zur Kompilierung entsprechend auf das neue Topmodul `mkHelloTestbench` an (`bsc --help` listet die möglichen Parameter und deren Bedeutung auf).

```
bsc -u -sim -g mkHelloTestbench HelloBluespec.bsv
bsc -sim -o out -e mkHelloTestbench
./out
```

```
module mkHelloTestbench(Empty);
  Reg#(UInt#(32)) counter <- mkReg(0);

  HelloBluespec uut <- mkHelloBluespec();

   rule endSimulation (counter == 200000000);
      $finish();
   endrule

   rule counterIncr;
      counter <= counter + 1;
   endrule
endmodule
```

Aufgabe 1.2.4 LED Ausgabe

Erweitern Sie die Testbench um eine Ausgabefunktion, die „`LED an.`“ oder „`LED aus.`“ ausgibt, wann auch immer die LED im Modul `mkHelloBluespec` an-/ausgeschaltet wird. Benutzen Sie dafür ein Register, das den letzten Zustand der LED speichert und vergleichen Sie diesen mit dem aktuellen Zustand der LED.

```
Reg#(Bool) ledLastCycle <- mkReg(False);
rule checkLedStatus;
  ledLastCycle <= uut.led();
  if(ledLastCycle == True && uut.led() == False) $display("LED aus.");
  else if(ledLastCycle == False && uut.led() == True) $display("LED an.");
endrule
```

Aufgabe 1.2.5 Analysieren

Bluespec bietet die Möglichkeit, die Simulation auf Waveform Ebene nachzuvollziehen. Der Parameter `-V filename.vcd` veranlasst die Simulation die Datei `filename.vcd` zu erzeugen, die alle zur Simulation gehörenden Waveforms beinhaltet. Diese Datei kann mit einem geeigneten Anzeigeprogramm wie GTKWave geöffnet werden. Standardmäßig werden bei der Kompilierung des BSV-Codes viele interne Signale wegoptimiert. Häufig ist es zur Fehlersuche hilfreich, wenn diese Signale erhalten bleiben. Mit Hilfe des `bsc` Parameters `-keep-fires` kann man die Wegoptimierung der Signale verhindern.

Vergleichen Sie die Waveforms der Simulation mit und ohne `-keep-fires`.

Achtung: Die Simulation mit BlueSim ist bei Verwendung des Parameters `-V` um ein vielfaches langsamer.

Aufgabe 1.3 Bluespec ALU

Eine ALU (Arithmetic Logic Unit) ist ein Rechenwerk, das häufig in Prozessoren zum Einsatz kommt. In dieser Aufgabe wird eine einfache ALU mit den Funktionen

- Multiplizieren
- Dividieren
- Addieren
- Subtrahieren
- Logisches Und
- Logisches Oder

implementiert.

Aufgabe 1.3.1 Das Interface

Das Bluespec Modul soll das folgende Interface besitzen:

```
interface HelloALU;
    method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b);
    method ActionValue#(Int#(32))  getResult();
endinterface
```

`AluOps` ist dabei der folgende Typ:

```
typedef enum{Mul,Div,Add,Sub,And,Or} AluOps deriving (Eq, Bits);
```

Die Anmerkung `deriving` kann in Bluespec genutzt werden, um Typklassen zu verwenden. In diesem Fall wird AluOps den Typklassen `Eq` und `Bits` zugewiesen. Der Bluespec Compiler wird automatisch dafür sorgen, dass sich der Typ AluOps als Bits darstellen lassen und verglichen werden kann. Weitere Informationen zu Typklassen finden Sie in der nächsten Übung oder in der Bluespec Referenz.

Aufgabe 1.3.2 Implementierung

Implementieren Sie auf Basis des oben vorgestellten Interfaces ein Modul `mkSimpleALU`. Die Methode `getResult` soll dabei so lange blockieren, bis das Ergebnis der Berechnung vorliegt. Erstellen Sie des Weiteren eine Testbench, die automatisch das aktuelle Ergebnis ausgibt, wenn es sich ändert, und alle Funktionen testet.

```
typedef enum{Mul,Div,Add,Sub,And,Or} AluOps deriving (Eq, Bits);

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

module mkHelloALU(HelloALU);
    Reg#(Bool) newOperands <- mkReg(False);
    Reg#(Bool) resultValid <- mkReg(False);
    Reg#(AluOps) operation <- mkReg(Mul);
    Reg#(Int#(32)) opA    <- mkReg(0);
    Reg#(Int#(32)) opB    <- mkReg(0);
    Reg#(Int#(32)) result <- mkReg(0);

    rule calculate (newOperands);
        Int#(32) rTmp = 0;
        case(operation)
            Mul: rTmp = opA * opB;
            Div: rTmp = opA / opB;
            Add: rTmp = opA + opB;
```

```
            Sub: rTmp = opA - opB;
            And: rTmp = opA & opB;
            Or:  rTmp = opA | opB;
        endcase
        result <= rTmp;
        newOperands <= False;
        resultValid <= True;
    endrule

    method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b) if(!newOperands);
        opA <= a;
        opB <= b;
        operation <= op;
        newOperands <= True;
        resultValid <= False;
    endmethod

    method ActionValue#(Int#(32)) getResult() if(resultValid);
        resultValid <= False;
        return result;
    endmethod
endmodule

module mkALUTestbench(Empty);
    HelloALU uut             <- mkHelloALU();
    Reg#(UInt#(8)) testState <- mkReg(0);

    rule checkMul (testState == 0);
        uut.setupCalculation(Mul, 4,5);
        testState <= testState + 1;
    endrule

    rule checkDiv (testState == 2);
        uut.setupCalculation(Div, 12,4);
        testState <= testState + 1;
    endrule

    rule checkAdd (testState == 4);
        uut.setupCalculation(Add, 12,4);
        testState <= testState + 1;
    endrule

    rule checkSub (testState == 6);
        uut.setupCalculation(Sub, 12,4);
        testState <= testState + 1;
    endrule

    rule checkAnd (testState == 8);
        uut.setupCalculation(And, 32'hA,32'hA);
        testState <= testState + 1;
    endrule

    rule checkOr (testState == 10);
        uut.setupCalculation(Or, 32'hA,32'hA);
        testState <= testState + 1;
```

```
77      endrule
78
79      rule printResults;
80          $display("Result: %d", uut.getResult());
81          testState <= testState + 1;
82      endrule
83
84      rule endSim (testState == 11);
85          $finish();
86      endrule
87  endmodule
```

Aufgabe 1.3.3 Power

Erweitern Sie das Modul mkHelloALU um eine Möglichkeit, a^b zu berechnen. Erstellen Sie dafür ein geeignetes Modul und Interface zur sequentiellen Berechnung. Binden Sie dieses in ihrer ALU ein. Eine mögliche C Implementation sieht dabei folgendermaßen aus:

```
1  int pow(int a, int b) {
2    int tmp = 1;
3    for(int i = 0; i < b; ++i) {
4      tmp *= a;
5    }
6    return tmp;
7  }
```

```
1   typedef enum{Mul,Div,Add,Sub,And,Or,Pow} AluOps deriving (Eq, Bits);
2
3   interface Power;
4       method Action   setOperands(Int#(32) a, Int#(32) b);
5       method Int#(32) getResult();
6   endinterface
7
8   module mkPower(Power);
9       Reg#(Bool) resultValid <- mkReg(False);
10
11      Reg#(Int#(32)) opA    <- mkReg(0);
12      Reg#(Int#(32)) opB    <- mkReg(0);
13      Reg#(Int#(32)) result <- mkReg(1);
14
15      rule calc (opB > 0);
16          opB <= opB - 1;
17          result <= result * opA;
18      endrule
19
20      rule calcDone (opB == 0 && !resultValid);
21          resultValid <= True;
22      endrule
23
24      method Action setOperands(Int#(32) a, Int#(32) b);
25          result <= 1;
26          opA    <= a;
27          opB    <= b;
28          resultValid <= False;
29      endmethod
30
```

```
31      method Int#(32) getResult() if(resultValid);
32          return result;
33      endmethod
34  endmodule
35
36  interface HelloALU;
37      method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b);
38      method ActionValue#(Int#(32))  getResult();
39  endinterface
40
41  module mkHelloALU(HelloALU);
42      Reg#(Bool) newOperands <- mkReg(False);
43      Reg#(Bool) resultValid <- mkReg(False);
44      Reg#(AluOps) operation <- mkReg(Mul);
45      Reg#(Int#(32)) opA    <- mkReg(0);
46      Reg#(Int#(32)) opB    <- mkReg(0);
47      Reg#(Int#(32)) result <- mkReg(0);
48
49      Power pow             <- mkPower();
50
51      rule calculate (newOperands);
52          Int#(32) rTmp = 0;
53          case(operation)
54              Mul: rTmp = opA * opB;
55              Div: rTmp = opA / opB;
56              Add: rTmp = opA + opB;
57              Sub: rTmp = opA - opB;
58              And: rTmp = opA & opB;
59              Or:  rTmp = opA | opB;
60              Pow: rTmp = pow.getResult();
61          endcase
62          result <= rTmp;
63          newOperands <= False;
64          resultValid <= True;
65      endrule
66
67      method Action setupCalculation(AluOps op, Int#(32) a, Int#(32) b) if(!newOperands);
68          opA <= a;
69          opB <= b;
70          operation <= op;
71          newOperands <= True;
72          resultValid <= False;
73          if(op == Pow) pow.setOperands(a,b);
74      endmethod
75
76      method ActionValue#(Int#(32)) getResult() if(resultValid);
77          resultValid <= False;
78          return result;
79      endmethod
80  endmodule
81
82  module mkALUTestbench(Empty);
83      HelloALU uut             <- mkHelloALU();
84      Reg#(UInt#(8)) testState <- mkReg(0);
85
```

```
86      rule checkMul (testState == 0);
87          uut.setupCalculation(Mul, 4,5);
88          testState <= testState + 1;
89      endrule
90
91      rule checkDiv (testState == 2);
92          uut.setupCalculation(Div, 12,4);
93          testState <= testState + 1;
94      endrule
95
96      rule checkAdd (testState == 4);
97          uut.setupCalculation(Add, 12,4);
98          testState <= testState + 1;
99      endrule
100
101     rule checkSub (testState == 6);
102         uut.setupCalculation(Sub, 12,4);
103         testState <= testState + 1;
104     endrule
105
106     rule checkAnd (testState == 8);
107         uut.setupCalculation(And, 32'hA,32'hA);
108         testState <= testState + 1;
109     endrule
110
111     rule checkOr (testState == 10);
112         uut.setupCalculation(Or, 32'hA,32'hA);
113         testState <= testState + 1;
114     endrule
115
116     rule checkPow (testState == 12);
117         uut.setupCalculation(Pow, 2, 12);
118         testState <= testState + 1;
119     endrule
120
121     rule printResults (unpack(pack(testState)[0]));
122         $display("Result: %d", uut.getResult());
123         testState <= testState + 1;
124     endrule
125
126     rule endSim (testState == 14);
127         $finish();
128     endrule
129 endmodule
```