Vorlesung Architekturen und Entwurf von Rechnersystemen

Lösungsvorschlag

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2022/2023
Übungsblatt 1

Diese Übung bietet eine allgemeine Einführung in grundlegende Bluespec Sprachkonzepte.

Aufgabe 1.1: Installation

Es gibt mehrere Möglichkeiten, wie Sie den Bluespec Compiler nutzen können. Wir stellen Ihnen die einzelnen Möglichkeiten kurz vor.

a) Nutzung auf den ISP-Poolrechnern (über SSH)

b) Lokale Nutzung mit Docker

c) Lokale Nutzung mit eigener Installation

1.1a) Nutzung auf den ISP-Poolrechnern (über SSH)

Folgen Sie zunächst der Anleitung zur Einrichtung des SSH-Zugriffs auf die ISP-Rechner (https://support.rbg.informatik.tu-darmstadt.de/wiki/de/doku/computerhilfe/ssh). Anschließend können Sie sich über SSH auf einem der Clients einloggen, unter Linux z.B. mit

```
ssh TUID@clientssh1.rbg.informatik.tu-darmstadt.de
```

und die korrekte Installation des BSC mit dem Befehl

```
bsc
```

verifizieren. Sie sollten die Hilfsnachricht des Compilers sehen.

1.1b) Lokale Nutzung mit Docker

Zunächst müssen Sie sicherstellen, dass Docker installiert ist. Dafür können Sie der Anleitung unter https://docs.docker.com/engine/install/ für Ihr System folgen. Unter Linux lohnt es sich, noch die Post-Installation Steps durchzuführen, um Docker auch ohne **sudo** nutzen zu können. Nach erfolgreicher Docker-Installation können Sie das Docker-Image `esatu/bsc` verwenden. Mit folgendem Befehl kann ein Docker-Container gestartet werden:

```
docker run -it --rm -v $PWD:/root/share esatu/bsc:latest
```

Das aktuelle Verzeichnis wird unter /root/share gemounted und beim Beenden wird der Container automatisch gelöscht. Sie können die korrekte Installation testen und diverse Hinweise des Compilers sehen, indem Sie in der nun offenen Shell den Befehl

```
bsc
```

eingeben.

1.1c) Lokale Nutzung mit eigener Installation

Alternativ kann der Bluespec-Compiler direkt aus den Quellen kompiliert werden (https://github.com/B-Lang-org/bsc). Für Arch-Linux existiert auch das Paket `bluespec-git`. Dieser Schritt funktioniert nur für Linux und Mac OS. Unter Windows müssen Sie zunächst eine virtuelle Maschine, z.B. mit Ubuntu, einrichten und dort dann wie gewohnt der Installationsanleitung aus der README-Datei folgen.

Bei offenen Fragen zur Einrichtung steht Ihnen das Moodle-Forum zur Verfügung.

Aufgabe 1.2: Grundlegende Sprachkonzepte

In dieser Aufgabe lernen Sie die wichtigsten Sprachelemente von Bluespec System Verilog (BSV) kennen. Wir gehen dabei inkrementell vor und versuchen, die Konzepte Modularität, Hierarchie und Regularität [1], die Sie aus Digitaltechnik kennen sollten, an Code-Beispielen zu verdeutlichen. Bevor wir irgendetwas Sinnvolles implementieren können, muss zunächst ein Package für den Code erstellt werden. Packages werden genutzt, um darin zusammengehörige Komponenten/Interfaces/Typen zu sammeln. Für die Erzeugung eines Packages sind die folgenden Schritte notwendig:

1. Erstellung einer BSV-Datei mit dem Namen **PACKAGENAME.bsv**
2. Öffnen Sie die Datei im Texteditor Ihrer Wahl (Visual Studio Code stellt z.B. eine Erweiterung für Bluespec-Syntax zur Verfügung)
3. Schreiben Sie in die neu erstellte Datei

```
package PACKAGENAME;
    // Packageinhalt
endpackage
```

Alle Inhalte des Packages kommen zwischen diese beiden Codezeilen. Sie können einzeilige Kommentare mit `//` schreiben und mehrzeilige Kommentare mit `/*` **Kommentarinhalt**`*/`.

Aufgabe:
Legen Sie ein Package mit dem Namen **U1** an. Es bietet sich an, für jedes Übungsblatt einen eigenen Ordner anzulegen, um die Übersicht zu behalten.

Lösungsvorschlag:

Wir legen zunächst eine Datei **U1.bsv** an. Der Inhalt von **U1.bsv** ist:

```
package U1;

endpackage
```

Für einen besseren Überblick kann man noch den Packagenamen am **endpackage** anbringen.

```
package U1;

endpackage : U1
```

1.2a) Interfaces und Methoden

Interfaces definieren Hardware-Schnittstellen von Modulen. Im Gegensatz zu Verilog, wo die Interfaces direkt durch Signale beschrieben werden, verwendet BSV **Methoden**, um diese Funktionalität umzusetzen. Die Verwendung von Methoden hat den Vorteil, dass sehr komplexe Daten übergeben werden können, ohne dass Entwickelnde sich lange damit aufhalten müssen, Signale zu zählen. Die für AER ausreichende Syntax zur Deklaration von Interfaces ist:

```
interface INTERFACENAME;
  // Methoden
endinterface
```

Die Namen von Interfaces beginnen **immer** mit einem Großbuchstaben.

Methoden werden mit folgender Syntax deklariert:

```
method RÜCKGABETYP METHODENNAME(PARAMETERTYP1 PARAMETER1, PARAMETERTYP2 PARAMETER2,...);
```

Werden keine Parameter übergeben, bleiben die Klammern leer. Weiterhin sind folgende Rückgabetypen vorhanden:

1. `Action` - Die Methode liefert keine Werte über Signale nach außen zurück, sondern führt Zustandsänderungen im Modul durch.
2. `ActionValue#(TYP)` - Die Methode führt im Modul eine Zustandsänderung durch **und** liefert ein Signal vom Typ **TYP**.
3. `TYP` - Die Methode führt im Modul **keine** Zustandsänderungen durch und liefert lediglich Signale vom Typ **TYP** nach außen.

TYP ist an dieser Stelle ein beliebiger durch Bits darstellbarer Datentyp, der auch für Parametertypen verwendet werden könnte, z.B. **int** oder **Bool**. Interfaces geben **keine** konkrete Implementierung der Methoden vor.

Aufgabe:
Wir wollen nun im Verlauf dieses Übungsblatts ein Modul schreiben, das eine blinkende LED steuern soll. Das Modul benötigt nach außen eine Schnittstelle, die die Steuerung des Moduls modelliert. Deklarieren Sie dafür im Package **U1** ein Interface mit dem Namen **Blinky** und geben Sie Methoden mit folgenden Namen an:

1. blink - Diese Methode soll ein einzelnes Signal ausgeben, das 1 ist, falls die LED sich im angeschalteten Zustand befindet.
2. start - Diese Methode soll die Steuerung in den angeschalteten Zustand versetzen.
3. stop - Diese Methode soll die Steuerung und die LED in den ausgeschalteten Zustand versetzen und ein Signal ausgeben, das angibt, wie oft die LED geleuchtet hat.

Hinweis: Sie finden weitere Informationen zu Interfaces und Methoden im Buch *BSV by Example* auf den Seiten 29ff, und zu Datentypen auf den Seiten 33ff.

Lösungsvorschlag:

```
interface Blinky;
  method Bool blink();
  method Action start();
  method ActionValue#(int) stop();
endinterface
```

Die Methode **blink** führt selbst keine Zustandsänderung der Steuerung durch, sondern spiegelt den Zustand nach außen. Entsprechend ist diese Methode eine sogenannte **Value-Methode**. Die Methode **start** liefert keine Werte nach außen, sondern führt eine Zustandsänderung durch. Sie ist eine **Action-Methode**. Die Methode **stop** führt eine Zustandsänderung durch **und** liefert einen Wert über Signale nach außen. Es handelt sich hierbei um eine sogenannte **ActionValue-Methode**.

1.2b) Module

Module ermöglichen es, komplexe Systeme in leicht verständliche und testbare Bestandteile aufzuteilen. Im Rahmen dieser Veranstaltung reicht es aus, wenn Sie sich folgende Grundsyntax zur Definition von Modulen zu merken:

```
module mkMODULNAME(INTERFACENAME);
    // Modulinhalte
endmodule
```

Jedes Modul implementiert ein Interface. Sofern kein Interface angegeben wird, verwendet der BSC das Interface **Empty**, welches keine Schnittstellen nach außen ermöglicht.

Aufgabe:
Definieren Sie im Package U1 ein Modul **mkBlinky**, welches das Interface **Blinky** implementiert. Sie müssen in dieser Aufgabe noch keine Modulinhalte angeben.

Lösungsvorschlag:

```
module mkBlinky(Blinky);

endmodule
```

1.2c) Modulkomponenten

Module können aus weiteren Untermodulen bestehen und so das Gesamtsystem hierarchisch aufbauen. Untermodule sollten am Anfang des Moduls festgelegt werden. Eine Modulinstanziierung folgt der Syntax:

```
INTERFACENAME modulreferenz <- mkMODULNAME();
```

Variablennamen, bzw. Modulreferenzen, beginnen in BSV immer mit einem Kleinbuchstaben, um sie von Typen und Interfaces zu unterscheiden. An dieser Stelle sehen wir auch einen BSV-Zuweisungsoperator: **<-**. Diesen Zuweisungsoperator kann man sich als eine Zuweisung mit Seiteneffekt vorstellen. In diesem Fall ist der Seiteneffekt die Erstellung des Untermoduls und der zugewiesene Wert die Referenz zum Interface dieses Moduls. Der **<-** Operator wird außerdem bei der Zuweisung der Ergebnisse von ActionValue-Methoden verwendet.

Aufgabe:
Register sind eine der am häufigsten genutzten Komponenten in modernen Schaltungen. Auch unsere LED-Steuerung soll Register verwenden, um ihre Aufgabe zu erfüllen. Wir benötigen ein Register, das angibt, ob das Modul eingeschaltet ist, ein Register, das den Status der LED angibt und ein weiteres Register, das zählt, wie oft die LED eingeschaltet wurde. Informieren Sie sich in *BSV by Example* über das Interface **Reg** und die Registermodule **mkReg** und **mkRegU**. Instanziieren Sie anschließend drei Register **ctrl_on**, **led_on** und **blink_ctr** im Modul **mkBlinky**. Die Steuerung und LED sollen sich zunächst im ausgeschalteten Zustand befinden.

Lösungsvorschlag:

```
module mkBlinky(Blinky);
    Reg#(Bool) ctrl_on <- mkReg(False);
    Reg#(Bool) led_on <- mkReg(False);
    Reg#(int) blink_ctr <- mkReg(0);
endmodule
```

1.2d) Regeln

Regeln (Rules) implementieren Zustandsübergänge und folgen mehreren Eigenschaften, die Sie in der Vorlesung noch vertiefen werden. Für diese Übung reicht es aus, zu wissen, dass der Körper einer Rule ein atomarer Block aus Actions ist, die alle gleichzeitig durchgeführt werden. Eine Regel hat die Form:

```
rule RULENAME(GUARD);
    // Regelinhalt
endrule
```

Der Name einer Rule beginnt immer mit einem Kleinbuchstaben. Die Guard stellt eine Bedingung (Bool Expression) dar, die erfüllt sein muss, damit die Regel feuern kann. Wenn eine Regel keine explizite Guard hat, können die Klammern weggelassen werden und die Regel feuert, sofern sie nicht anderweitig daran gehindert wird (z. B. durch in der Vorlesung besprochene Konflikte), immer.

Aufgabe:
Wir wollen nun das Konzept der Regeln nutzen, um die Hauptfunktionalität des Moduls zu implementieren: Den Zustand der LED ändern und mitzählen, wie oft diese eingeschaltet wurde. Informieren Sie sich in *BSV by Example* darüber, wie in Registern Werte zugewiesen werden, und über grundlegende arithmetische und logische Operatoren. Die LED soll, sofern die Steuerung eingeschaltet ist, taktweise abwechselnd ein- und ausgeschaltet werden. Wenn die LED in Takt t eingeschaltet ist, so soll sie in Takt $t + 1$ ausgeschaltet sein, in $t + 2$ wieder eingeschaltet, usw. Schreiben Sie eine Regel, die dieses Verhalten umsetzt.
Nun muss noch das Zählregister erhöht werden, sofern die LED eingeschaltet wurde. Informieren Sie sich in *BSV by Example* über die Nutzung von if-Statements und fügen Sie Ihrer Regel die entsprechende Funktionalität hinzu.

Lösungsvorschlag:

```
rule regel(ctrl_on);
    Bool led_new = !led_on;
    if(led_new) begin
        blink_ctr <= blink_ctr + 1;
    end
    led_on <= led_new;
endrule
```

1.2e) Methodendefinition

Damit das Modul wie erhofft nutzbar ist, müssen wir noch die internen Zustände nach außen sichtbar machen. Der letzte Block eines Moduls ist (falls das Interface nicht Empty ist) **immer** die Definition des Interfaces, also die konkrete Implementierung der Methoden. Eine Methodendefinition folgt der Syntax:

```
method RÜCKGABETYP METHODENNAME(PARAMETERLISTE);
    // Implementierung
endmethod
```

Value- und ActionValue-Methoden haben am Ende ihrer Methode ein **return**-Statement, mit dem der Rückgabewert an ein Ausgangssignal angelegt wird.

Aufgabe:
Implementieren Sie nun im Modul mkBlinky das Interface Blinky. Erinnern Sie sich daran, wie sich die Methoden auf den Zustand der Schaltung auswirken sollen.

Lösungsvorschlag:

```
method Bool blink();
    return led_on;
endmethod

method Action start() if(!ctrl_on);
    ctrl_on <= True;
    blink_ctr <= 0;
endmethod

method ActionValue#(int) stop() if(ctrl_on);
```

```
    ctrl_on <= False;
    led_on <= False;
    return blink_ctr;
endmethod
```

Aufgabe 1.3: Testen

Zuletzt müssen wir unser Steuerungsmodul testen. Dafür verwenden wir Testbenches. In dieser Übung verwenden wir eine rein regelbasierte Testbench. In der nächsten Übung werden Sie ein hilfreiches Bluespec-Feature kennenlernen, mit dem Testbenches etwas angenehmer zu schreiben sind. Eine Testbench ist ein Bluespec Modul, welches das Interface **Empty** implementiert und später als Top-Level Modul an den BSC übergeben wird. Sie instanziiert das zu testende Modul, generiert Testeingaben und wertet Ausgangssignale aus, um potenzielle Fehler zu erkennen.

Aufgabe:
In der Datei **U1Tb_template.bsv** finden Sie eine zum Großteil vorgefertigte Testbench. Benennen Sie zunächst die Datei so um, dass der Dateiname und der Packagename übereinstimmen. Machen Sie sich anschließend mit dem Inhalt der Datei vertraut. Die Grundidee dieser Testbench-Implementierung ist, das Steuerungsmodul in einer Regel zu starten und es anschließend für **cycle_limit** Takte laufen zu lassen. Daraufhin wird das Modul gestoppt und ausgelesen, wie oft die LED angeschaltet wurde. Daraufhin wird der erwartete Wert mit dem tatsächlich ausgelesenen Wert verglichen und die Testbench anschließend mit **$finish()** beendet.
In der Testbench befinden sich einige **TODO**-Kommentare. Erledigen Sie die TODOs und kompilieren Sie anschließend die Testbench. Dies geschieht im Allgemeinen mit dem Befehl:

```
bsc -u -sim -g mkMODULNAME PACKAGE.bsv
bsc -u -sim -e mkMODULNAME -o EXECUTABLE_NAME
```

Das Flag **-sim** legt fest, dass wir für den Bluespec Simulator kompilieren. Das Flag **-g** gibt an, dass das dahinter genannte Modul das Top-Level Modul ist. PACKAGE.bsv ist die Datei, die das Modul enthält. Wollen Sie nur geänderte Quelldateien neu kompilieren, können Sie dem ersten BSC-Aufruf noch das **-u** Flag mitgeben. Die Testbench verwendet ein Makro namens **`CYCLES**, um die Anzahl der Takte festzulegen, für die der Simulator ausgeführt werden soll. Sie können den Wert des Makros im ersten BSC-Aufruf definieren, indem Sie das Flag **-D "CYCLES=WERT"** übergeben. Daraufhin ersetzt der Compiler das Makro im Quellcode mit dem String **WERT**. Sie können die Simulation anschließend mit

```
./EXECUTABLE_NAME
```

starten. Bei **kurzen** Simulationen (Simulationen, die über wenige Takte laufen) können Sie das Flag **-V** beim Aufruf Ihrer ausführbaren Simulation übergeben und so Signal-Dumps erzeugen. Diese können Sie dann zum Beispiel mit GTKWave analysieren und so Ihre Module debuggen. Die Dumps werden in der Datei **dump.vcd** abgelegt. Kompilieren Sie die Testbench einmal so, dass sie 10 Takte lang ausgeführt wird und einmal so, dass sie 2^{32} Takte ausgeführt wird. Erzeugen Sie **keine** Signal-Dumps für die 2^{32} Takte Simulation! Was fällt Ihnen auf?
Hinweis: Sie können Pakete mit dem Statement **import PAKETNAME::KOMPONENTENNAME;** importieren. Wenn alle Paketinhalte importiert werden sollen, können Sie den Komponentennamen mit einem Stern ersetzen.

Lösungsvorschlag:

```
package U1Tb;
    import U1::*;

    module mkU1Tb();
        Blinky dut <- mkBlinky();

        Reg#(Bool) started <- mkReg(False);
        Reg#(UInt#(33)) cycles <- mkReg(0);
        Reg#(Bool) checked <- mkReg(False);

        UInt#(33) cycle_limit = `CYCLES;
```

```
        UInt#(33) n_blinks = (cycle_limit+1) / 2;

        rule start (!started);
            dut.start();
            started <= True;
        endrule

        rule let_blink (cycles <= cycle_limit);
            Bool led = dut.blink();
            if (cycle_limit < 100)
                $display("LED status: %b", led);
            cycles <= cycles + 1;
        endrule

        rule check (cycles > cycle_limit);
            let blinked <- dut.stop();
            if (extend(blinked) != unpack(pack(n_blinks))) begin
                $display("LED count != Reference value. Expected %d, got %d",
                         n_blinks, blinked);
            end
            else
                $display("Test successful!");
            checked <= True;
        endrule

        rule shutdown (checked);
            $finish();
        endrule
    endmodule
endpackage
```

Wir kompilieren die Testbench einmal mit

```
bsc -u -sim -D "CYCLES=10" -g mkU1Tb U1Tb.bsv
bsc -sim -e mkU1Tb -o test1
```

und einmal mit

```
bsc -u -sim -D "CYCLES=(1 << 32)" -g mkU1Tb U1Tb.bsv
bsc -sim -e mkU1Tb -o test2
```

Wenn die Testbench nur zehn Takte läuft, gelingt der Test. Wenn die Testbench 2^{32} Takte läuft ist die LED so oft angeschaltet worden, dass im Counter das most significant bit gesetzt wurde und die Counter-Werte damit negativ werden. Dadurch schlägt der Test fehl. Das Problem entsteht durch die schlechte Wahl des Datentyps **int** ($\hat{=}$ **Int#(32)**) für einen Zähler. Wenn man den Typen im Blinky-Interface und mkBlinky-Modul zu **UInt#(32)** ändert, ist das Problem gelöst. Natürlich findet auch hier nach 2^{32} Takten ein Überlauf statt. Die Interpretation der Werte bleibt aber für die gesamte Auflösung des Signals gleich.

Aufgabe 1.4: Zusammenfassung

In dieser Übung haben Sie gelernt wie Sie:

- ein BSV Interface deklarieren
- ein Modul definieren
- andere Module innerhalb eines Moduls instanziieren
- einfache Regeln schreiben

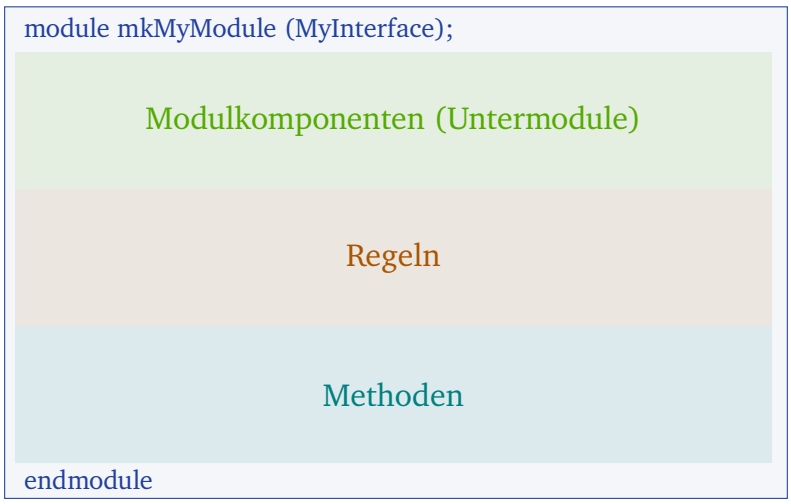

Abbildung 1: Allgemeiner Aufbau eines BSV-Moduls.

- Methoden implementieren
- eine einfache Testbench implementieren

Sie können sich abschließend das Schema in Abbildung 1 für den Aufbau eines BSV-Moduls merken, damit Sie bei der Implementierung der nachfolgenden Übungen immer einen roten Faden haben, den Sie abarbeiten können.

Aufgabe 1.5: Enums und 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.

Wir kodieren diese Operationen im Enum **AluOps**.

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

Die `deriving`-Anweisung sorgt dafür, dass Werte des Enums vergleichbar (`==`/`!=`) und als Bits darstellbar (`pack`/`unpack`) sind.

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
```

1.5a) 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.
Hinweis: Case-Statements sind eine elegante Möglichkeit den Wert eines Enum-Typs zu überprüfen und passendes Verhalten zu erzeugen.

Lösungsvorschlag:

```
module mkSimpleALU(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 && !resultValid);
        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
```

1.5b) Testbench

Erstellen Sie eine Testbench zum Testen der ALU. Diese soll alle Funktionen testen, indem für jede Operation eine Berechnung durchgeführt wird. Dabei sollen automatisch die Ergebnisse als Text ausgeben werden.

Lösungsvorschlag:

```
module mkALUTestbench(Empty);
    HelloALU uut             <- mkSimpleALU();
```

```
    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'hB);
        testState <= testState + 1;
    endrule

    rule printResults (unpack(pack(testState)[0]));
        $display("Result: %d", uut.getResult());
        testState <= testState + 1;
    endrule

    rule endSim (testState == 12);
        $finish();
    endrule
endmodule
```

1.5c) Zusatzaufgabe: Power

Die ALU soll nun auch Potenzen berechnen können. Erweitern Sie zunächst das **AluOps**-Enum, implementieren Sie diese Funktionalität in einem Untermodul und instanziieren Sie dieses in der ALU. Erweitern Sie anschließend Ihre Testbench, sodass sie auch diese Funktionalität abdeckt.

Lösungsvorschlag:

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

interface Power;
    method Action   setOperands(Int#(32) a, Int#(32) b);
    method Int#(32) getResult();
```

```
endinterface

module mkPower(Power);
    Reg#(Bool) resultValid <- mkReg(False);
    Reg#(Int#(32)) opA    <- mkReg(0);
    Reg#(Int#(32)) opB    <- mkReg(0);
    Reg#(Int#(32)) result <- mkReg(1);

    rule calc (opB > 0);
        opB <= opB - 1;
        result <= result * opA;
    endrule

    rule calcDone (opB == 0 && !resultValid);
        resultValid <= True;
    endrule

    method Action setOperands(Int#(32) a, Int#(32) b);
        result <= 1;
        opA    <= a;
        opB    <= b;
        resultValid <= False;
    endmethod

    method Int#(32) getResult() if(resultValid);
        return result;
    endmethod
endmodule

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);

    Power pow             <- mkPower();

    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;
            Pow: rTmp = pow.getResult();
        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;
        if(op == Pow) pow.setOperands(a,b);
    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;
    endrule

    rule checkPow (testState == 12);
```

```
        uut.setupCalculation(Pow, 2, 12);
        testState <= testState + 1;
    endrule

    rule printResults (unpack(pack(testState)[0]));
        $display("Result: %d", uut.getResult());
        testState <= testState + 1;
    endrule

    rule endSim (testState == 14);
        $finish();
    endrule
endmodule
```

Literatur

[1] D. Harris und S.L. Harris. *Digital Design and Computer Architecture*. Engineering professional collection. Elsevier Science, 2012, S. 6. ISBN: 9780123944245. URL: https://books.google.de/books?id=-DG18Nf7jLcC.