Vorlesung Architekturen und Entwurf von Rechnersystemen

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2022/2023
Übungsblatt 2

In dieser Übung lernen Sie folgende Konzepte und ihre Umsetzung in BSV kennen:

- Subinterfaces
- FIFOs
- FSMs
- Tagged Unions

Aufgabe 2.1: Einfache Pipeline

Wir wollen eine simple Berechnungspipeline implementieren, welche die Formel $res = ((((x + a) \cdot b) \cdot c)/4) + 128$ berechnet. Dabei ist x ein Eingabedatum und a, b, c sind Konfigurationsparameter, welche wir zur Laufzeit im Modul ändern wollen. Wir nehmen hierfür an, dass alle Operationen in einem Takt ausgeführt werden.

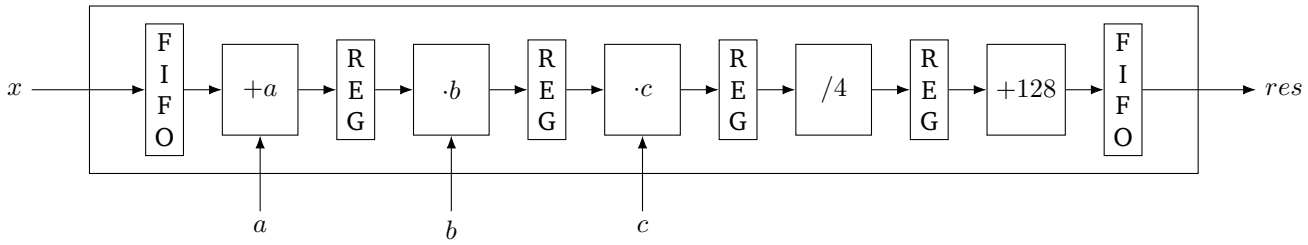

Abbildung 1: Pipeline aus Aufgabe 2.1.

Aus objektorientierten Programmiersprachen kennen Sie die Möglichkeit, dass eine Klasse mehrere Interfaces implementiert. Dieses Konzept macht man sich auch im Hardware-Entwurf zunutze und verwendet dafür in BSV das Konzept der Subinterfaces. Jedes Interface, das in Bluespec deklariert wurde, kann als Teil eines größeren Interfaces verwendet werden. Dies ermöglicht es, ein Modul mit mehreren anderen Modulen kommunizieren zu lassen oder ein Modul mit zusätzlicher Funktionalität zu erweitern, es aber in vereinfachter Form zu behandeln. Letztere Variante betrachten wir in dieser Aufgabe.

Wenn wir die Pipeline in Abbildung 1 betrachten sehen wir, dass das zu implementierende Modul folgende Funktionalitäten bereistellen muss:

1. Input des Wertes x
2. Output des Ergebnisses
3. Setzen der Parameter a, b, c.

Hierbei kann Unterschieden werden in Funktionalität zum Ein- und Ausgeben der Daten (Punkte 1 & 2) und Funktionalität zum Konfigurieren der Parameter (Punkt 3). Für eine modulare Gestaltung wollen wir die Punkte 1 & 2 in ein eigenes Interface auslagern. Dieses werden wir dann später an anderer Stelle wiederverwenden können. Dieses Interface können wir wie folgt in BSV formalisieren:

```
interface CalcUnit;
    method Action put(Int#(32) x);
    method ActionValue#(Int#(32)) result();
endinterface
```

Das Interface für unsere Pipeline sieht dann wie folgt aus:

```
interface Pipeline;
    interface CalcUnit calc;
    method Action setParam(UInt#(2) addr, Int#(32) val);
endinterface
```

Dieses Interface verwendet das vorher definierte Interface **CalcUnit** als Subinterface und erlaubt das Ein- und Ausgeben der Daten. Zusätzlich ermöglicht es das Konfigurieren der Parameter über die Methode **setParam()**. Dabei soll für **addr** $= 0$ der Wert von a gesetzt werden, für **addr** $= 1$ der Wert von b und für **addr** $= 2$ der Wert von c.

Implementieren Sie nun ein entsprechendes Modul **mkSimplePipeline**, welches das Interface **Pipeline** implementiert und die oben beschriebene Formel berechnet. Nutzen Sie dabei zum Speichern der Eingaben und Ausgaben FIFOs, zum Zwischenspeichern der Ergebnisse Register (wie in Abbildung 1 dargestellt). Sie können sich dabei an der Implementierung der Shifter Pipeline aus der Vorlesung orientieren (2. Realisierung)[1]. Welchen Typ hat die soeben implementierte Pipeline (dynamisch vs statisch; elastisch vs starr)?

Aufgabe 2.2: Testen mit FSMs

In der vorherigen Übung haben wir eine einfache regelbasierte Testbench verwendet. Wenn unsere Testbench aber viele kurze Schritte sequenziell abarbeiten soll, wird dieses Regelkonstrukt schnell unübersichtlich. Eine elegantere Alternative stellt die Nutzung des Packages **StmtFSM**[2] dar. Dabei erfolgt die Erstellung einer FSM in zwei Schritten:

1. Erstellung eines Statements, das den zeitlichen Ablauf der FSM beschreibt
2. Instanziierung einer FSM mit dem Statement aus (1)

2.2a) Erstellung des Statements

Machen Sie sich zunächst mit dem **StmtFSM**-Package vertraut. Sie finden einige Beispiele zur Nutzung in *BSV by Example*[2] und die Grammatik zur Konstruktion von Statements im *BSV Reference Guide*[3][4]
In der Datei **U2_2_template.bsv** finden Sie das Codegerüst der Testbench für unser Pipeline-Modul. Wir geben Ihnen einen Vector mit gültigen Testergebnissen vor. Erstellen Sie in der Testbench ein Statement, das folgendes Verhalten abbildet:

1. Der Parameter a soll auf den Wert 42 gesetzt werden
2. Der Parameter b soll auf den Wert 2 gesetzt werden
3. Der Parameter c soll auf den Wert 13 gesetzt werden
4. Anschließend soll eine Nachricht mit dem Inhalt „Initialized parameters, starting test...“ ausgegeben werden.
5. Nun sollen die folgenden zwei Auflistungen parallel ablaufen:
   a) • Die Pipeline wird zehn Takte lang mit Testeingaben gefüttert. Diese sind die Zahlen 0 bis einschließlich 9.

[1] Foliensatz BSV; Folien 99-103
[2] BSV by Example; Kapitel 14.3; S. 185ff
[3] BSV Language Reference Guide; Kapitel 12; S. 99ff.
[4] BSV Library Reference Guide; Kapitel 3.6; S. 172ff.

- Anschließend wird a auf den Wert 7 gesetzt, b auf den Wert 3 und c auf den Wert 17.
- Daraufhin soll die Pipeline wieder mit Testeingaben gefüttert werden, den Zahlen 10 bis einschließlich 19.

b)
- Die FSM soll 20 mal:
  - Die Nachricht „Checking results for INPUT“ ausgeben, wobei INPUT mit dem tatsächlichen Input-Wert ersetzt wird.
  - Das entsprechende Ergebnis aus der Pipeline auslesen
  - Überprüfen ob das ausgelesene Ergebnis mit dem passenden Eintrag aus **testvec** übereinstimmt und entsprechend einen Zähler für korrekte Tests um 1 erhöhen.
  - Sollte das ausgelesene Ergebnis Falsch sein soll die Nachricht „Expected CORRECT, got ACTUAL“ ausgegeben werden, wobei CORRECT mit dem entsprechenden Eintrag aus **testvec** ersetzt wird und ACTUAL mit dem zuvor ausgelesenen Ergebnis.
  - Dies soll alles in einem Takt geschehen.

6. Anschließend soll die Nachricht „N_CORR of 20 tests passed.“ ausgegeben werden, wobei N_CORR mit der Anzahl der korrekten Tests ersetzt werden soll.
7. Wenn alle Tests erfolgreich waren, soll anschließend die Nachricht „ SUCCESS!“ ausgegeben werden, ansonsten die Nachricht „FAILURE“.
8. Zuletzt soll die Simulation mit **$finish()** beendet werden.

Hinweis: Machen Sie sich anhand von http://wiki.bluespec.com/Home/Data-Types/Data-Type-Conversion-Functions klar, wie Sie einen `UInt` zu einem `Int` konvertieren können.

2.2b) Instanziierung der FSM

Wir können **StmtFSM**s auf verschiedene Arten instanziieren.

1. FSM fsm <- mkFSM(stmt) - Diese FSM können wir beliebig oft mit **fsm.start()** innerhalb einer Rule starten und mit **fsm.done()** prüfen, ob sie ihr Statement abgearbeitet hat.
2. mkAutoFSM(stmt) - Diese FSM wird direkt bei der Instanziierung gestartet und kann nicht nochmal ausgeführt werden.
3. FSM fsm <- mkFSMWithPred(stmt, bool) - Diese FSM verhält sich wie mkFSM, mit dem Unterschied, dass sie nach dem Start nur ausgeführt wird wenn die Bedingung **bool** erfüllt ist.

Instanziieren Sie Ihre FSM mittels **mkAutoFSM**. Sie können Ihre Testbench nun kompilieren und Ihre Pipeline aus Aufgabe 2.1 testen. Sollte der Simulationsprozess irgendwo stecken bleiben, feuern Teile Ihrer Regeln oder Methoden nicht. Sie können sich die CAN_FIRE-Bedingungen Ihrer Regeln und Guards Ihrer Methoden anzeigen lassen, indem Sie dem ersten Compiler-Aufruf das Flag **-show-schedule** übergeben. Die Schedule Datei hat die Endung **.sched** und die CAN_FIRE steht hinter dem Punkt **Predicate**.

Hinweis: Damit ihre implementierte Pipeline korrekt funktioniert, müssen sie **bsc** möglicherweise mit dem Flag **-aggressive-conditions** aufrufen.

2.2c) Auswertung von FSMs

Gegeben sei das nachfolgende Modul mit drei verschiedenen FSM-Statements. Werten Sie die Statements *auf dem Papier* aus und geben Sie für jedes Statement die angezeigten Werte des Registers **ctr** an oder, ob es in dem Statement zu einem Compilerfehler kommt (und warum). Betrachten Sie die Statements individuell, d.h. Veränderungen durch die anderen Statements dürfen ignoriert werden.

```
module mkFSMEx(Empty);
    Reg#(int) ctr <- mkReg(0);

    Stmt s1 = seq
        $display("%t : %d", $time(), ctr);
        ctr <= ctr + 1;
        $display("%t : %d", $time(), ctr);
        par
            seq
                $display("%t : %d", $time(), ctr);
            endseq
            seq
                action
                    ctr <= ctr + 1;
                    $display("%t : %d", $time(), ctr);
                endaction
                $display("%t, %d", $time(), ctr);
            endseq
        endpar
    endseq;

    Stmt s2 = seq
        $display("%t : %d", $time(), ctr);
        ctr <= ctr + 1;
        $display("%t : %d", $time(), ctr);
        par
            seq
                $display("%t : %d", $time(), ctr);
            endseq
            seq
                ctr <= ctr + 1;
                $display("%t : %d", $time(), ctr);
                $display("%t, %d", $time(), ctr);
            endseq
        endpar
    endseq;

    Stmt s3 = seq
        $display("%t : %d", $time(), ctr);
        ctr <= ctr + 1;
        $display("%t : %d", $time(), ctr);
        par
            seq
                $display("%t : %d", $time(), ctr);
                action
                    ctr <= ctr + 1;
                    ctr <= ctr + 1;
                endaction
                $display("%t : %d", $time(), ctr);
            endseq
            seq
                ctr <= ctr + 1;
                $display("%t : %d", $time(), ctr);
                $display("%t, %d", $time(), ctr);
            endseq
        endpar
    endseq;
endmodule
```

Aufgabe 2.3: Modulare Pipeline

Wir wollen nun eine alternative Implementierung für die Pipeline aus Aufgabe 2.1 erstellen. Dabei wollen wir insbesondere die Berechnungen der einzelnen Pipelinestufen in separate (Unter-)Module ("Calc Units") auslagern.

2.3a) Calc Units

Zuerst implementieren wir die für die einzelnen Berechnungen zuständigen Module.

Dabei gibt es zwei Typen von Berechnungen: Die letzten beiden Berechnungen (vgl. Abbildung 1) benötigen **keinen** Parameter. Daher können wir hier das bereits bekannte Interface **CalcUnit** nutzen:

```
interface CalcUnit;
    method Action put(Int#(32) x);
    method ActionValue#(Int#(32)) result();
endinterface
```

Die anderen drei Berechnungen benötigen jeweils einen Parameter (a, b bzw. c). Wir benötigen also ein neues Interface:

```
interface CalcUnitChangeable;
    interface CalcUnit calc;
    method Action setParameter(Int#(32) param);
endinterface
```

Beachten Sie, dass wir hier das Interface **CalcUnit** wiederverweden, anstatt dessen zwei Methoden auch in das neue Interface zu schreiben.

In **U2_3_template.bsv** finden Sie die oben deklarierten Interfaces. Benennen Sie die Datei zunächst korrekt um. Implementieren Sie anschließend die Module in Tabelle 1 mit den passenden Interfaces. Dabei sollen die konfigurierbaren Parameter a, b und c jeweils in einem Register gespeichert werden. Sie können Wires[5][6] verwenden, um die Berechnungen mit kombinatorischer Logik umzusetzen. Sollten Sie sich für die kombinatorische Implementierung entscheiden, müssen Sie die Action **noAction;** in der Definition der Methode **result** aufrufen, um eine gültige ActionValue-Methode zu implementieren. Warum müssen Sie nur ein Multiplikationsmodul implementieren?

Modul	Interface
mkAddA	CalcUnitChangeable
mkMul	CalcUnitChangeable
mkDiv4	CalcUnit
mkAdd128	CalcUnit

Tabelle 1: Teilmodule und zugehörige Interfaces.

2.3b) Pipeline Modul

In dieser Teilaufgabe wollen wir nun die einzelnen Teilmodule zu einer modularen Pipeline kombinieren. Diese soll das **Pipeline**-Interface aus Aufgabe 2.1 implementieren:

```
interface Pipeline;
    interface CalcUnit calc;
    method Action setParam(UInt#(2) addr, Int#(32) val);
endinterface
```

[5] BSV by Example; Kapitel 8.2; S. 114ff.
[6] BSV Libraries Reference Guide; Kapitel 2.4.4; S. 51ff.

Implementieren Sie das entsprechende Modul.

Hinweis: Sie können die Pipeline-Funktionalität jetzt nicht mehr in einer einzelnen Regel implementieren, sondern müssen die Funktionalität auf mehrere Regeln aufteilen. Sie können Ihren Code mit geschickter Nutzung von for-Schleifen[7][8] und **Vector**[9] deutlich kompakter halten.

Testen Sie die modulare Pipeline mit der Testbench aus Aufgabe 2.2. Sie sollten dafür keine Änderungen an der Testbench vornehmen müssen - abgesehen vom Tausch des DUT.

2.3c) Pipelinetypen

Welchen Typ hat die soeben implementierte, modulare Pipeline (dynamisch vs statisch; elastisch vs starr)?

Können Sie in der modularen Pipeline bspw. das mkMult-Teilmodul durch eine alternative Implementierung ersetzen, welche eine variable Anzahl an Takten zur Berechnung des Produkts benötigt (abhängig von den Eingabedaten)[10]? Ändert dies etwas am Typ der Pipeline?

Ist dies auch bei der einfachen Pipeline aus Aufgabe 2.1 möglich?

2.3d) Zusatzaufgabe

Erstellen Sie eine alternative Implementierung der modularen Pipeline, welche FIFOs zum Speichern der Zwischenergebnisse nutzt (anstatt Registern). Ändert dies etwas am Pipelinetyp? Können Sie auch mit dieser Implementierung CalcUnits mit variabler Latenz nutzen?

Aufgabe 2.4: Tagged Unions

Aktuell kann die Pipeline aus Aufgabe 2.3 nur vorzeichenbehaftete Integerwerte verarbeiten. Wir wollen die Module und Interfaces nun so erweitern, dass sowohl vorzeichenbehaftete (signed), als auch nicht-vorzeichenbehaftete (unsigned) Integerwerte verarbeitet werden können. Dabei soll die Vorzeichenbehaftung der Ein- und Ausgabedaten zur Laufzeit überprüfbar sein. Bluespec stellt hierfür als Mittel der Wahl die so genannten **Tagged Unions** zur Verfügung. Diese sind Strukturen, deren Daten zur Laufzeit mit einem **Tag** versehen sind, wodurch die Schaltung interpretieren kann welcher Datentyp gerade anliegt.
Für diese Aufgabe definieren wir das Tagged Union **SignedOrUnsigned** mit

```
typedef union tagged {UInt#(32) Unsigned;
                      Int#(32)  Signed;
                     } SignedOrUnsigned deriving(Eq, Bits);
```

Der obige Code erledigt dabei folgende Dinge:

- Er legt ein Tagged Union mit den beiden Tags **Unsigned** und **Signed** an.
- Er setzt den Typnamen dieses Tagged Unions mittels **typedef** auf **SignedOrUnsigned**.
- Er übergibt dem Bluespec Compiler mit **deriving** die Information, dass der Typ mittels == vergleichbar ist
- Er übergibt dem BSC die Information, dass der Typ mit Bits darstellbar ist, also die Funktionen **pack** und **unpack** unterstützt, mit.

Wir können diesem Typen nun mit folgendem Code **Signed** oder **Unsigned** Werte zuweisen:

[7] BSV by Example; Kapitel 11; S. 151ff
[8] BSV Language Reference Guide; Kapitel 9.7; S. 74f.
[9] BSV by Example; Kapitel 13; S. 163ff.
[10] vgl. Foliensatz BSV; Folie 21

```
SignedOrUnsigned us = tagged Unsigned 42;
SignedOrUnsigned s = tagged Signed -42;
```

Durch das verwendete **deriving**-Statement erstellt der Compiler automatisch die in Abbildung 2 gezeigte Bitrepresentation für die beiden Variablen.

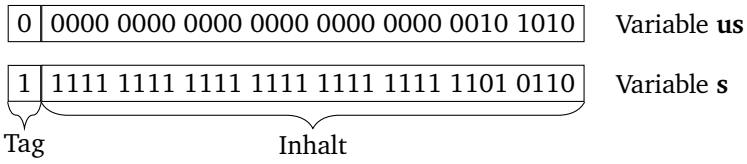

Abbildung 2: Standard Bitrepresentation von Tagged Unions

Der Inhalt des Typen **SignedOrUnsigned** kann mittels Pattern Matching[11] interpretiert werden. Nehmen wir an, es gäbe drei Register **a**, **b** und **c**, die jeweils den Typen **SignedOrUnsigned** beinhalten und **a** und **b** seien auf Operanden gesetzt, während **c** das Ergebnis beinhalten soll. Dann könnte man folgende Variante des Pattern Matchings wählen:

```
rule doAdd (pack(a)[32] == pack(b)[32]);
    case (a) matches
        tagged Unsigned .usa: begin
                UInt#(32) usb = unpack(pack(b)[31:0]);
                c <= tagged Unsigned (usa+usb);
            end
        tagged Signed .sa: begin
                Int#(32) sb = unpack(pack(b)[31:0]);
                c <= tagged Signed (sa + sb);
            end
    endcase
    $display("%h", c);
endrule
```

Dabei erledigt der obige Code folgendes:

- Die explizite Guard der Regel überprüft, ob die Tag-Bits der beiden Operandenregister übereinstimmen. Nur in diesem Fall feuert die Rule.
- Das Case-Statement überprüft das Tag-Bit von **a**. Ist das Tag **Unsigned**, wird der **UInt**-Wert in einer Variable namens **usa** gespeichert, andernfalls in **sa**.
- Weiterhin wird der Inhalt des Registers **b** passend zum Tag von **a** in einem **UInt** oder **Int** gespeichert. Dabei wird **b** erst mit **pack** zum Typ **Bit** konvertiert, wodurch wir auf einzelne Bitvektoren zugreifen können. Anschließend wird der 32-Bit Vektor mittels **unpack** zum Typen **UInt** oder **Int** konvertiert und in einer passenden Variable gespeichert.
- Zuletzt wird im Register **c** das Ergebnis der Addition der beiden Registerinhalte und das passende Tag gespeichert.

Eine Alternative hierzu, die uns die Verwendung des expliziten Casts von **b** erspart, wäre die Verwendung von zwei Regeln, die in Ihren Guards direkt das Pattern-Matching mit der Wertzuweisung durchführen.

```
rule doAddSigned (a matches tagged Signed .sa &&& b matches tagged Signed .sb);
    c <= tagged Signed (sa+sb);
endrule

rule doAddUnsigned (a matches tagged Unsigned .usa &&& b matches tagged Unsigned
 ↪  .usb);
    c <= tagged Unsigned (usa+usb);
endrule
```

[11] BSV by Example; Kapitel 10.1.6; S. 141f.

2.4a) Interfaces

Erstellen Sie ein Interface **TU_Pipeline** mit dem auf **SignedOrUnsigned** angepassten Interface aus. Legen Sie weiterhin ein Package **CalcUnits** an und definieren Sie dort die Interfaces **CalcUnit** und **CalcUnitChangeable** und passen Sie diese auf den Typen **SignedOrUnsigned** an. Denken Sie dran, dass Sie das Package **CalcUnits** in **TU_Pipeline.bsv** importieren müssen, um die dort deklarierten Interfaces nutzen zu können.

2.4b) Module

Implementieren Sie nun erneut die Pipeline aus Aufgabe 2.3 und Ihre Teilmodule mit den entsprechenden neuen Interfaces. Implementieren Sie dabei die Untermodule im Package **CalcUnits**.

2.4c) Tests

Wir stellen Ihnen ein Template **U2_4c_CustomTest.bsv** und eine angepasste **U2_4c_Testbench.bsv** zur Verfügung. Das Modul **mkCustomTest** wurde mit Parametern so erweitert, dass man Testfälle einfach bei der Erstellung des Moduls übergeben kann. Innerhalb von **mkCustomTest** können diese Parameter wie gewöhnliche BSV-Variablen genutzt werden.

Erweitern Sie in **mkCustomTest** das Statement **s**, sodass dieses den Parameter **in** in die Pipeline eingibt und danach das Ergebnis der Pipeline ausliest.

Wenn das Ergebnis der Pipeline nicht mit dem Parameter **expOut** übereinstimmt, soll der Text „Test INDEX failed. Expected expOut, got result“ ausgegeben werden. Dabei sollen die beiden Ergebniswerte in hexadezimaler Darstellung angezeigt werden. Der **INDEX** soll dabei der Index des Testfalls sein. Dieser ist der Inhalt des Parameters **in**.

Wenn das erwartete Ergebnis mit dem tatsächlichen Ergebnis übereinstimmt, soll der Text „Test INDEX succeeded!“ ausgegeben werden. Der **INDEX** entspricht auch hier dem Index des Testfalls.

Aufgabe 2.5: Zusatzaufgabe: ALU mit FSM-Testbench

Das Modul `mkAutoFSM` eignet sich hervorragend zur Erstellung von Testbenches.

Schreiben Sie eine Testbench für das Modul `mkSimpleALU` aus der ersten Übung. Nutzen Sie dabei einen Vektor, der alle Testdaten beinhaltet. Lagern Sie häufig genutzte Teile (Operanden eingeben und Ergebnis überprüfen) in eine extra FSM aus, indem Sie `mkAutoFSM` und `mkFSM` kombinieren. Einen Vektor können Sie folgendermaßen erzeugen:

```
typedef struct {
   Int#(32) opA;
   Int#(32) opB;
   AluOps    operator;
   Int#(32) expectedResult;
 } TestData deriving (Eq, Bits);
 ...
 Vector#(20, TestData) myVector;
 myVector[0] = TestData {opA: 2, opB: 4, operator: Add, expectedResult: 6};
 ...
 myVector[19] = TestData {opA: 4, opB: 2, operator: Div, expectedResult: 2};
```

Aufgabe 2.6: Zusatzaufgabe: ALU mit Tagged Unions

Erweitern Sie die ALU aus Übung 1, sodass sie Signed und Unsigned Integer verarbeiten kann.