Übung zur Vorlesung Architekturen und Entwurf von Rechnersystemen

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 16/17

Übungsblatt 4 - Lösungsvorschlag

Ab dieser Übung wird ein kleines System on Chip (SoC) entwickelt. Dabei lernen Sie Techniken wie Direct-Memory-Access (DMA) und Bussysteme in der Praxis kennen. Als Einstieg in diesen Themenkomplex wird in dieser Übung ein Stream-basierter Bildfilter entwickelt.

Aufgabe 4.1 Bildfilter mit ClientServer Interface

Aus den Vorlesungen kennen Sie bereits generische Interfaces wie das `GetPut` Interface. Damit die Bildfilter, die in den kommenden Übungen entwickelt werden, modular und einheitlich aufgebaut sind, benutzen wir ein solches generisches Interface.

```
typedef Bit#(8) Color;
typedef Bit#(8) GrayScale;

typedef struct {
        Color r;
        Color g;
        Color b;
} RGB deriving(Bits, Eq, FShow);
 module mkGray(Server#(RGB, GrayScale));
 ...
 endmodule
```

Die Details zum `Server` Interface finden Sie im BSV-Reference-Guide oder in BSV-by-example.

Implementieren Sie ein Modul, das RGB-Pixel eines Bildes über das `Server` Interface erhält und verwandeln Sie diesen Pixel in Grauwerte. Verwenden Sie dazu die Methode, die unter “Luma coding in video systems” auf https://en.wikipedia.org/wiki/Grayscale beschrieben wird:

$$Y = 0.299R + 0.587G + 0.114B$$

Die Größe von Floating-Point Einheiten in Hardware macht diese uninteressant in vielen Anwendungen. Anstatt von Floating-Point wird ein (U)Q8.8 Fixed-Point Format für die Berechnung verwendet. Information über “Q”-Floating-Point finden Sie auf https://en.wikipedia.org/wiki/Q_(number_format). Gibt es in den Bluespec Libraries Unterstützung für arithmetische Operation auf diesem Format?

Lösungsvorschlag

```
package ColorConverter;

import ClientServer::*;
import GetPut::*;
import FIFO::*;

typedef Bit#(8) Color;
typedef Bit#(8) GrayScale;

typedef struct {
```

```
    Color r;
    Color g;
    Color b;
    } RGB deriving(Bits, Eq, FShow);

module mkGray(Server#(RGB, GrayScale));
        FIFO#(GrayScale) outputValue <- mkFIFO;
        FIFO#(RGB) inputValue <- mkFIFO;

        function GrayScale toGray(RGB rgb);
            UInt#(16) scale = 0;
            UInt#(16) r = extend(unpack(rgb.r)) << 8;
            UInt#(16) g = extend(unpack(rgb.g)) << 8;
            UInt#(16) b = extend(unpack(rgb.b)) << 8;

            UInt#(16) factorR = 76;  // floor(0.299 * (1 << 8))
            UInt#(16) factorG = 150; // floor(0.587 * (1 << 8))
            UInt#(16) factorB = 29;  // floor(0.114 * (1 << 8))

            // Multiply color with factors and normalize
            UInt#(24) foo = extend(r) * extend(factorR); // Enough space for multiplcation
↪  result
            r = truncate(foo >> 8);
            foo = extend(g) * extend(factorG);
            g = truncate(foo >> 8);
            foo = extend(b) * extend(factorB);
            b = truncate(foo >> 8);

            scale = r + g + b;

            return pack(scale)[15:8];
        endfunction

        rule calc;
            let color = inputValue.first; inputValue.deq;
            let gray = toGray(color);

            outputValue.enq(gray);
        endrule

        interface Put request = toPut(inputValue);
        interface Get response = toGet(outputValue);
endmodule

endpackage
```

Aufgabe 4.2 Testen des Moduls

Testen Sie das von Ihnen entwickelte Modul mithilfe einer klassischen Testbench und zusätzlich mit BlueCheck.

Lösungsvorschlag

Die klassische Testbench testet die Unit-Under-Test gegen vorgegebene Werte:

```
import ColorConverter::*;

import ClientServer::*;
import GetPut :: *;
import Vector :: *;

typedef 4 TotalTests;

module mkTestbench(Empty);
        Server#(RGB, GrayScale) uut <- mkGray;

        Vector#(TotalTests, Tuple2#(RGB, GrayScale)) testvalues;
        testvalues[0] = tuple2(RGB { r:0, g:0, b:0 }, 0);
        testvalues[1] = tuple2(RGB { r:255, g:255, b:255 }, 254); // Rounding errors/Round
↪  towards zero
        testvalues[2] = tuple2(RGB { r:42, g:10, b:50 }, 23);
        testvalues[3] = tuple2(RGB { r:128, g:5, b:240 }, 68);

        Reg#(Bool) sendTest <- mkReg(True);
        Reg#(UInt#(32)) testCntr <- mkReg(0);

        rule sendTestValue if(sendTest && testCntr < fromInteger(valueOf(TotalTests)));
                uut.request.put(tpl_1(testvalues[testCntr]));
                sendTest <= False;
        endrule

        rule checkTestValue if(!sendTest && testCntr < fromInteger(valueOf(TotalTests)));
                let r <- uut.response.get();
                if(r != tpl_2(testvalues[testCntr])) begin
                        $display("Error at %d: %d != %d", testCntr, r,
↪  tpl_2(testvalues[testCntr]));
                end
                testCntr <= testCntr + 1;
                sendTest <= True;
        endrule

        rule endTest if(testCntr == fromInteger(valueOf(TotalTests)));
                $finish();
        endrule

endmodule
```

Die BlueCheck Variante kann in diesem Fall zum Beispiel zum finden von Überläufen bei einer effizienten Implementierung verwendet werden:

```
import BlueCheck :: *;

// Very simple/inefficient implementation
module mkSimpleGray(Server#(RGB, GrayScale));
    FIFO#(GrayScale) outputValue <- mkFIFO;

    interface Put request;
        method Action put(RGB rgb);
            UInt#(32) factorR = 76;  // floor(0.299 * (1 << 8))
```

```
            UInt#(32) factorG = 150; // floor(0.587 * (1 << 8))
            UInt#(32) factorB = 29;  // floor(0.114 * (1 << 8))

            UInt#(32) r = extend(unpack(rgb.r)) << 8;
            UInt#(32) g = extend(unpack(rgb.g)) << 8;
            UInt#(32) b = extend(unpack(rgb.b)) << 8;

            outputValue.enq(truncate(pack((r * factorR + g * factorG + b * factorB) >> 16)));
        endmethod
    endinterface

    interface Get response = toGet(outputValue);
endmodule

module [BlueCheck] mkColorConverterSpec ();
    Server#(RGB, GrayScale) spec <- mkSimpleGray;
    Server#(RGB, GrayScale) impl <- mkGray();

    equiv("put", spec.request.put, impl.request.put);
    equiv("get", spec.response.get, impl.response.get);
endmodule

module [Module] mkColorConverterChecker ();
    blueCheck(mkColorConverterSpec);
endmodule
```

Als Alternative: Häufig wird die BlueSpec Implementierung mit Hilfe von BDPI gegen eine Software-Version verglichen. Dies wird in Übung 8 näher beschrieben (oder kann im Reference-Guide nachgelesen werden).