Übung zur Vorlesung Einführung in Computer Microsystems

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 15

Übungsblatt 5 - Lösungsvorschlag

Diese Übung behandelt die Implementierung eines „Streaming Image Processor“ zur Reduzierung der Auflösung eines Eingabebildes. Ein Eingabebild der 2160p Auflösung 3840 × 2160 würde ein Ausgabebild der Auflösung 1080p 1920 × 1080 produzieren (Halbierung der Auflösung Horizontal und Vertikal).

Das Interface des Streaming Prozessors erwartet auf der einen Seite Pixeldaten in einem geeigneten Format wie 8 bit pro Farbe RGB. Die Ausgabeseite gibt Pixeldaten in eben jenem Format aus. Des weiteren soll die Breite einer Zeile des Ursprungsbildes über eine Methode auf beliebige gerade Auflösungen bis 2160p veränderbar sein.

Aufgabe 5.1 Datentypen

Definieren Sie mit Hilfe von `typedef` einen geeigneten Datentyp für die Pixeldaten des Prozessors. Benutzen Sie entweder RGB(A) oder YCbCr Formate.

```
typedef 2160 RowWidthMax;

typedef struct {
    UInt#(8) r;
    UInt#(8) g;
    UInt#(8) b;
} Pixel deriving(Bits, Eq);
```

Aufgabe 5.2 Interface

Implementieren Sie ein Interface für den oben beschriebenen Prozessor. Nutzen Sie wenn möglich standardisierte Interfaces (z.B. `ClientServer`).

```
interface ImageProcessor;
    interface Server#(Pixel, Pixel) server;
    method Action setRowWidth(UInt#(12) rowWidth);
endinterface
```

Aufgabe 5.3 Zwischenspeicherung der Eingabedaten

Überlegen Sie sich eine Möglichkeit die Eingabepixel bis zur Bearbeitung zu speichern. Versuchen Sie zu jeder Zeit nicht mehr als zwei Zeilen des Eingabebildes zu Speichern. Benutzen Sie zur Zwischenspeicherung das Package `RegFile`. Die Zeilenbreite soll bis 2160p einstellbar sein.

```
module mkImageProcessor(ImageProcessor);

    Reg#(UInt#(12)) rowWidthMax <- mkReg(fromInteger(valueOf(RowWidthMax)));
    FIFOF#(Pixel) outFIFO <- mkSizedFIFOF(16);
    FIFO#(Tuple4#(Pixel, Pixel, Pixel, Pixel)) storeInter <- mkFIFO();

    // Store the position which is used to calculate new pixels
```

```
8       Reg#(UInt#(12)) currentCalcPosition <- mkReg(0);
9
10      Reg#(Bool) doCalc[2] <- mkCReg(2, False);
11
12      RegFile#(UInt#(12), Pixel) mem <- mkRegFileFull();
13
14      // Calculation
15      ....
16      //
17
18      Reg#(Bool) firstLine <- mkReg(True);
19      // Store the position where new pixels are stored
20      Reg#(UInt#(12)) currentStoragePosition <- mkReg(0);
21
22      Reg#(UInt#(12)) pixelThisLine <- mkReg(0);
23
24      interface Server server;
25          interface Put request;
26              method Action put(Pixel p) if(outFIFO.notFull());
27                  mem.upd(currentStoragePosition, p);
28                  if(pixelThisLine == (rowWidthMax * 2) - 1) begin
29                      pixelThisLine <= 0;
30                      currentStoragePosition <= 0;
31                      doCalc[1] <= True;
32                  end else if(pixelThisLine >= rowWidthMax) begin
33                      if(currentStoragePosition == rowWidthMax) begin
34                          currentStoragePosition <= rowWidthMax + 1;
35                      end else begin
36                          currentStoragePosition <= rowWidthMax;
37                          doCalc[1] <= True;
38                      end
39                      pixelThisLine <= pixelThisLine + 1;
40                  end else begin
41                      currentStoragePosition <= currentStoragePosition + 1;
42                      pixelThisLine <= pixelThisLine + 1;
43                  end
44              endmethod
45          endinterface
46
47          interface Get response = toGet(outFIFO);
48      endinterface
49
50      method Action setRowWidth(UInt#(12) rowWidth);
51          // Set row width
52          if(pack(rowWidth)[0] == 0) rowWidthMax <= rowWidth;
53          // Reset module operation
54          outFIFO.clear();
55          currentStoragePosition <= 0;
56          currentCalcPosition <= 0;
57      endmethod
58  endmodule
59  endpackage
```

Aufgabe 5.4 Ausgabebild

Berechnen Sie das Ausgabebild aus dem Eingabebild indem Sie den Durchschnitt der Farbwerte der entsprechenden Pixel des Eingabebildes berechnen. Diese sind die Pixel die am nächsten zum neuen Pixelwert gelegen sind. Für den Pixel $P_O(0,0)$ des Ausgabebildes benötigt man die Pixel $P_I(0,0), P_I(0,1), P_I(1,0), P_I(1,1)$ des Eingabebildes

$$P_O(0,0) = \frac{P_I(0,0) + P_I(0,1) + P_I(1,0) + P_I(1,1)}{4} \tag{1}$$

Geben Sie dieses Ausgabebild über das von Ihnen erstellte Interface aus.

```
1   function UInt#(12) addOneAndCap(UInt#(12) v);
2       if(v == rowWidthMax - 1) return 0;
3       else                     return v + 1;
4   endfunction
5
6   function UInt#(12) subOneAndCap(UInt#(12) v);
7       if(v == 0) return rowWidthMax - 1;
8       else       return v - 1;
9   endfunction
10
11  rule getPixels (doCalc[0]);
12      let pixelPos0 = currentCalcPosition;
13      let pixelPos1 = addOneAndCap(currentCalcPosition);
14      let pixelPos2 = rowWidthMax;
15      let pixelPos3 = rowWidthMax + 1;
16      let pix       = tuple4(mem.sub(pixelPos0),
17                             mem.sub(pixelPos1),
18                             mem.sub(pixelPos2),
19                             mem.sub(pixelPos3));
20      storeInter.enq(pix);
21      // Four Pixel were used for calculations
22      currentCalcPosition <= addOneAndCap(pixelPos1);
23      doCalc[0] <= False;
24  endrule
25
26  function UInt#(8) calcAverage(UInt#(8) c0, UInt#(8) c1, UInt#(8) c2, UInt#(8) c3);
27      UInt#(10) c0t = extend(c0);
28      UInt#(10) c1t = extend(c1);
29      UInt#(10) c2t = extend(c2);
30      UInt#(10) c3t = extend(c3);
31      return truncate((c0t + c1t + c2t + c3t) / 4);
32  endfunction
33
34  rule calc;
35      let data = storeInter.first(); storeInter.deq();
36      let pix0 = tpl_1(data);
37      let pix1 = tpl_2(data);
38      let pix2 = tpl_3(data);
39      let pix3 = tpl_4(data);
40      let r = calcAverage(pix0.r, pix1.r, pix2.r, pix3.r);
41      let g = calcAverage(pix0.g, pix1.g, pix2.g, pix3.g);
42      let b = calcAverage(pix0.b, pix1.b, pix2.b, pix3.b);
43      Pixel p = Pixel{r:r,g:g,b:b};
44      outFIFO.enq(p);
45  endrule
```

Aufgabe 5.5 Testbench

Testen Sie Ihr Modul mit Hilfe von Bluespec und einfachen Testdaten.

Aufgabe 5.6 (BONUS: Leicht)

Mit Hilfe des BRAM Moduls können Sie echte Bilder in Ihrer Testbench verwenden.

```
1 BRAM_Configure cfg = defaultValue ;
2 //declare variable cfg
3 cfg.memorySize = 1024*32 ; //new value for memorySize
4 cfg.loadFormat = tagged Hex "ram.txt"; //value for loadFormat
5 BRAM2Port#(UInt#(15), Bit#(16)) bram <- mkBRAM2Server (cfg) ;
6 //instantiate 32K x 16 bits BRAM module
```

Die Eingabedaten werden entsprechend als Hexadezimalwerte in der Datei „ram.txt“ erwartet. Schaffen Sie eine Möglichkeit (z.B. Python Script) ein Bild in das erwartete Format umzuwandeln.

Die Ausgabe kann zum Beispiel über die `$display` Funktion erfolgen.

Aufgabe 5.7 (BONUS: Mittel) C Code Testbench

Sie können C Funktionen in Ihre Testbench einbinden. Dieser Vorgang ist im Bluespec Reference Guide ab Kapitel 16 (ab Seite 146) beschrieben. Lesen Sie Bilder direkt über eine C Funktion ein und geben Sie das verkleinerte Bild über eine Grafikschnittstelle wie SDL aus.

Aufgabe 5.8 (BONUS: Schwer) Bicubic Resampling

Ein besserer Algorithmus zur Auflösungsreduktion ist das Bicubic Resampling. Tauschen Sie das oben verwendete Verfahren durch Bicubic Resampling aus wie es zum Beispiel auf `https://clouard.users.greyc.fr/Pantheon/experiments/rescaling/index-en.html#bicubic` oder `https://code.google.com/a/eclipselabs.org/p/bicubic-interpolation-image-processing/source/browse/trunk/libimage.c` beschrieben ist. Zur Implementierung können Sie die Floating-Point Implementation aus `$BLUESPECDIR/BSVSource/Math/FloatingPoint.bsv` (Achtung! Keine Dokumentation.) verwenden.