Übung zur Vorlesung Architekturen und Entwurf von Rechnersystemen

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 16/17
Übungsblatt 5

In dieser Übung entwickeln Sie als Erweiterung für die, in der vorherigen Übung erstellte, RGB zu GrayScale Konvertierungseinheit einen einfachen Bildfilter.

Hinweis:

Die normale Version dieser Übung arbeitet nur auf 3×3 großen Kerneln über einem 20×20 großen Bild. Möchten Sie diese Übung mit erhöhtem Schwierigkeitsgrad bearbeiten, entwickeln Sie alle folgenden Module generisch, das heißt, ihre Implementierung kann über Typparameter für beliebigen Bildgrößen und Kernelgrößen konfiguriert werden.

Aufgabe 5.1 Median

In dieser Übung wird ein 3×3 Median-Filter entwickelt. Wie der Name impliziert, muss der Median eines neunelementigen Vektors gefunden werden. Realisieren Sie ein Modul, das einen solchen Median findet. Nutzen Sie dafür eine Pipeline von Vergleichern mit jeweils drei Eingängen und Ausgängen. Die Ausgänge des eines Vergleichers sind der höchste Wert der Eingabe, der niedrigste Wert der Eingabe und der mittlere Wert der Eingabe. Pro Takt soll immer nur ein Vergleich sequentiell (beliebig viele Parallel) ausgeführt werden.

```
1 module mkMedian(Server#(Vector#(9, GrayScale), GrayScale));
2   ...
3 endmodule
```

Testen Sie Ihre Implementierung mit Hilfe von BlueCheck. Als einfache (aber ineffiziente) Vergleichsoperation können Sie die Listen-Funktion `sort` verwenden. Einen Vektor können Sie in eine Liste umwandeln mit der Funktion `toList` (BSV-Reference-Guide 259ff.).

Lösungsvorschlag

```
1  package Median;
2
3      import Types :: *;
4      import ClientServer :: *;
5      import GetPut :: *;
6      import Vector :: *;
7      import List :: *;
8      import FIFO :: *;
9
10     import BlueCheck :: *;
11
12     typedef Vector#(9, GrayScale) FilterKernel;
13     typedef Server#(Vector#(9, GrayScale), GrayScale) Median;
14
15     module mkMedian(Median);
16         FIFO#(FilterKernel) in <- mkFIFO();
17         FIFO#(GrayScale) out <- mkFIFO();
18
```

```
        Vector#(7, Server#(Vector#(3, GrayScale), Sorted)) sortingNetwork <-
    ↪ replicateM(mkSort());

        rule firstStage;
            let t = in.first(); in.deq();

            Vector#(3, Vector#(3,GrayScale)) c = unpack(pack(t));

            for(Integer i = 0; i < 3; i = i + 1) begin
                sortingNetwork[i].request.put(c[i]);
            end
        endrule

        rule secondStage;
            Vector#(3, Vector#(3, GrayScale)) sorted;
            for(Integer i = 0; i < 3; i = i + 1) begin
                let tVal <- sortingNetwork[i].response.get();
                sorted[i] = unpack(pack(tVal));
            end

            for(Integer i = 0; i < 3; i = i + 1) begin
                Vector#(3, GrayScale) iSort;
                for(Integer j = 0; j < 3; j = j + 1) begin
                    iSort[j] = sorted[j][i];
                end
                sortingNetwork[3 + i].request.put(iSort);
            end
        endrule

        rule thirdStage;
            Vector#(3, GrayScale) sorted;
            for(Integer i = 0; i < 3; i = i + 1) begin
                let tVal <- sortingNetwork[3 + i].response.get();
                Vector#(3, GrayScale) tSorted = unpack(pack(tVal));
                sorted[i] = tSorted[2 - i];
            end
            sortingNetwork[6].request.put(sorted);
        endrule

        rule fourthStage;
            let tVal <- sortingNetwork[6].response.get();
            out.enq(tVal.med);
        endrule

        interface Put request = toPut(in);
        interface Get response = toGet(out);
    endmodule

    typedef struct {
        GrayScale max;
        GrayScale med;
        GrayScale min;
    } Sorted deriving(Bits, Eq);

```

```
72      module mkSort(Server#(Vector#(3, GrayScale), Sorted));
73          FIFO#(Vector#(3, GrayScale)) in <- mkFIFO();
74          FIFO#(Sorted) out <- mkFIFO();
75
76          rule sort;
77              let i = in.first(); in.deq();
78              let xored = i[0] ^ i[1] ^ i[2];
79              Sorted tVal;
80              tVal.max = max(i[0], max(i[1], i[2]));
81              tVal.min = min(i[0], min(i[1], i[2]));
82              tVal.med = xored ^ tVal.max ^ tVal.min;
83              out.enq(tVal);
84          endrule
85
86          interface Put request = toPut(in);
87          interface Get response = toGet(out);
88      endmodule
89
90      module [BlueCheck] mkMedianSpec ();
91          Median impl <- mkMedian;
92
93          FIFO#(Vector#(9, GrayScale)) specFIFO <- mkFIFO();
94
95          function ActionValue#(GrayScale) getMedian();
96              actionvalue
97                  let s = specFIFO.first(); specFIFO.deq();
98                  List#(GrayScale) l = toList(s);
99                  l = sort(l);
100
101                 return l[4];
102             endactionvalue
103         endfunction
104
105         equiv("put", specFIFO.enq, impl.request.put);
106         equiv("get", getMedian, impl.response.get);
107     endmodule
108
109     module [Module] mkMedianChecker ();
110        blueCheck(mkMedianSpec);
111     endmodule
112
113 endpackage
```

Aufgabe 5.2 Stream Kernelbearbeiter (Schwierig)

Entwickeln Sie ein Modul, das einen Stream von `GrayScale` Daten als Eingabe erhält. Innerhalb des Moduls sollen 3×3 Kernel erzeugt und an das Median Modul übergeben werden. Die Ausgabe des Moduls ist das Median gefilterte Bild als Stream von `GrayScale` Werten. Das Eingabebild wird Zeile für Zeile von links nach rechts und oben nach unten eingelesen.

Hinweis:

Das Bild wird durch die Bearbeitung nicht verkleinert. Beachten Sie die Ränder des Bildes nicht.

Benutzen Sie zwei `mkSizedFIFO` zum Zwischenspeichern der Bildreihen wie in Abbildung 1 skizziert. Die Pfeile geben die Datenflussrichtung an. Wenn der erste Pixel das mittlere Register (Reg) erreicht können Sie den ersten Kernel (alle Register) an das Median Modul übergeben.

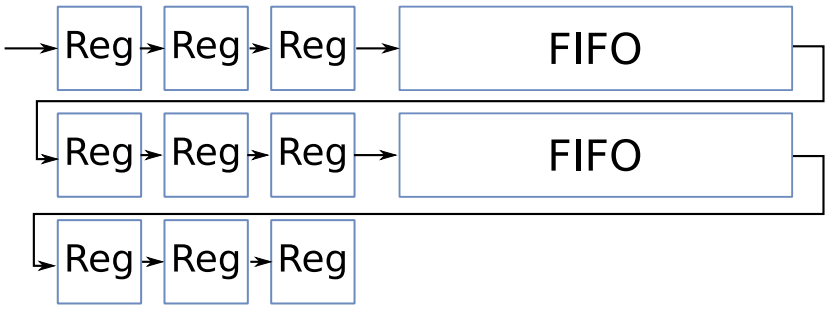

Abbildung 1: Effiziente Buffer für Bildzeilen für die Bildverarbeitung in Hardware

Testen Sie die Module angemessen.

Lösungsvorschlag

```
package MedianFilter;

    import Types :: *;
    import ClientServer :: *;
    import GetPut :: *;
    import Vector :: *;
    import FIFO :: *;
    import Connectable :: *;

    import BlueCheck :: *;
    import Median :: *;
    import ColorConverter :: *;

    import StmtFSM :: *;

    typedef 20 ImageWidth;
    typedef 20 ImageHeight;
    typedef 3 KernelSize;
    typedef TMul#(ImageHeight, ImageWidth) ImagePixels;
    typedef TAdd#(ImagePixels, TMul#(2, ImageWidth)) ImagePixelsTotal;
    typedef TSub#(ImageWidth, KernelSize) BufferSize;

    module mkMedianFilter(Server#(GrayScale, GrayScale));
        Server#(GrayScale, Vector#(9, GrayScale)) kernel <- mkKernelBuffer();

        Median median <- mkMedian();

        FIFO#(GrayScale) in <- mkFIFO();

        Reg#(UInt#(TLog#(TMul#(2, ImagePixelsTotal)))) inputCounter[2] <- mkCReg(2, 0);

        function UInt#(TLog#(TMul#(2, ImagePixelsTotal))) incrWithMax(UInt#(TLog#(TMul#(2,
↪  ImagePixelsTotal))) r);
            let t = r + 1;
            if(t == fromInteger(2 * valueOf(ImagePixelsTotal))) t = 0;
            return t;
```

```
36          endfunction
37
38          rule inForward if(((inputCounter[0] & 1) == 0)
39                                  && inputCounter[0] < fromInteger(2 * valueOf(ImagePixels))
40                                  );
41              kernel.request.put(in.first());
42              in.deq();
43              inputCounter[0] <= incrWithMax(inputCounter[0]);
44          endrule
45
46          rule inFeed if(((inputCounter[0] & 1) == 0)
47                              && inputCounter[0] >= fromInteger(2 * valueOf(ImagePixels))
48                          );
49              kernel.request.put(0);
50              inputCounter[0] <= incrWithMax(inputCounter[0]);
51          endrule
52
53          rule outDiscard if(((inputCounter[1] & 1) == 1)
54                              && inputCounter[1] < fromInteger(2 * valueOf(ImageWidth))
55                              );
56              let d <- kernel.response.get();
57              inputCounter[1] <= incrWithMax(inputCounter[1]);
58          endrule
59
60          rule outForward if(((inputCounter[1] & 1) == 1)
61                                  && inputCounter[1] >= fromInteger(2 * valueOf(ImageWidth))
62                                  );
63              let d <- kernel.response.get();
64              median.request.put(d);
65              inputCounter[1] <= incrWithMax(inputCounter[1]);
66          endrule
67
68          interface request = toPut(in);
69          interface response = median.response;
70      endmodule
71
72      module mkMedianFilterTest(Empty);
73
74          Server#(GrayScale, GrayScale) medianFilter <- mkMedianFilter();
75
76          Reg#(UInt#(32)) cntr <- mkReg(0);
77          Reg#(UInt#(32)) cntr2 <- mkReg(0);
78
79          Stmt fsm = {
80              par
81                  $display("Starting test");
82                  for(cntr <= 0; cntr < fromInteger(2 * valueOf(ImagePixels)); cntr <= cntr + 1)
    ↪  seq
83                      action
84                          $display("Putting in %d", cntr);
85                          medianFilter.request.put(pack(truncate(cntr + 1)));
86                      endaction
87                  endseq
```

```
                for(cntr2 <= 0; cntr2 < fromInteger(2 * valueOf(ImagePixels)); cntr2 <= cntr2 + 1) seq
                    action
                        let d <- medianFilter.response.get();
                        $display("Fetching %d in %d", d, cntr2);
                    endaction
                endseq
            endpar
        };

        mkAutoFSM(fsm);
    endmodule

    module mkKernelBuffer(Server#(GrayScale, Vector#(9, GrayScale)));
        Reg#(Bool) valueFetched[2] <- mkCReg(2, False);

        FIFO#(GrayScale) in <- mkFIFO();
        FIFO#(Vector#(9, GrayScale)) out <- mkFIFO();

        Reg#(Vector#(9, GrayScale)) bufferReg <- mkReg(unpack(0));

        FIFO#(Vector#(2, GrayScale)) bufferFIFO <- mkAlwaysFullSizedFIFO();

        Wire#(Vector#(2, GrayScale)) bufferWire <- mkWire();

        rule feedBufferRegular;
            Vector#(9, GrayScale) tmpBuf;

            tmpBuf[0] = in.first(); in.deq();
            for(Integer i = 1; i < 9; i = i + 1) begin
                tmpBuf[i] = bufferReg[i - 1];
            end

            Vector#(2, GrayScale) t = bufferFIFO.first(); bufferFIFO.deq();
            tmpBuf[3] = t[0];
            tmpBuf[6] = t[1];

            out.enq(tmpBuf);
            bufferReg <= tmpBuf;

            t[0] = bufferReg[2];
            t[1] = bufferReg[5];
            bufferWire <= t;

            valueFetched[1] <= False;
        endrule

        rule feedBuffer;
            bufferFIFO.enq(bufferWire);
        endrule

        interface request = toPut(in);
        interface response = toGet(out);
    endmodule
```

```
141
142     module mkKernelBufferTest(Empty);
143
144         Server#(GrayScale, Vector#(9, GrayScale)) kernel <- mkKernelBuffer();
145
146         Reg#(UInt#(32)) cntr <- mkReg(0);
147
148         Stmt fsm = {
149             seq
150                 $display("Starting test");
151                 for(cntr <= 0; cntr < 40; cntr <= cntr + 1) par
152                     action
153                         let d <- kernel.response.get();
154                         $display("Fetching in %d", cntr);
155                         $display(fshow(d));
156                     endaction
157                     action
158                         $display("Putting in %d", cntr);
159                         kernel.request.put(pack(truncate(cntr + 1)));
160                     endaction
161                 endpar
162             endseq
163         };
164
165         mkAutoFSM(fsm);
166     endmodule
167
168     module mkAlwaysFullSizedFIFO(FIFO#(t))
169         provisos(Bits#(t, t_sz));
170         Reg#(UInt#(TLog#(BufferSize))) writePntr[3] <- mkCReg(3, 0);
171         Reg#(UInt#(TLog#(BufferSize))) readPntr[3] <- mkCReg(3, 0);
172
173         Vector#(BufferSize, Reg#(t)) buffer <- replicateM(mkReg(unpack(0)));
174
175         function UInt#(TLog#(BufferSize)) incrWithMax(Reg#(UInt#(TLog#(BufferSize))) r);
176             let t = r + 1;
177             if(t == fromInteger(valueOf(BufferSize))) t = 0;
178             return t;
179         endfunction
180
181         method Action enq (t x) if(incrWithMax(writePntr[1]) == readPntr[1]);
182             writePntr[1] <= incrWithMax(writePntr[1]);
183             buffer[writePntr[1]] <= x;
184         endmethod
185
186         method Action deq if(readPntr[0] == writePntr[0]);
187             readPntr[0] <= incrWithMax(readPntr[0]);
188         endmethod
189
190         interface first = buffer[readPntr[0]];
191
192         method Action clear();
193             writePntr[2] <= 0;
194             readPntr[2] <= 1;
```

```
195             endmethod
196         endmodule
197
198         module mkAlwaysFullSizedFIFOTest(Empty);
199
200             FIFO#(UInt#(32)) f <- mkAlwaysFullSizedFIFO();
201
202             Reg#(UInt#(32)) cntr <- mkReg(0);
203
204             Stmt fsm = {
205                 seq
206                     $display("Starting test");
207                     for(cntr <= 0; cntr < 40; cntr <= cntr + 1) par
208                         action
209                             $display("Fetching %d in %d", f.first(), cntr);
210                             f.deq();
211                         endaction
212                         action
213                             $display("Putting in %d", cntr);
214                             f.enq(cntr + 1);
215                         endaction
216                     endpar
217                 endseq
218             };
219
220             mkAutoFSM(fsm);
221         endmodule
222
223
224         module mkRGBToMedian(Server#(RGB, GrayScale));
225             Server#(RGB, GrayScale) gray <- mkGray();
226             Server#(GrayScale, GrayScale) medianFilter <- mkMedianFilter();
227             mkConnection(gray.response, medianFilter.request);
228
229             interface request = gray.request;
230             interface response = medianFilter.response;
231         endmodule
232     endpackage
```

Erweiterung:
Behandeln Sie die Ränder des Bildes korrekt. Der Nutzer des Moduls soll den Wert der Ränder als Parameter übergeben können.

Aufgabe 5.3 Pipeline

Testen Sie den Farb-zu-Graustufen Konvertierer zusammen mit dem Median-Filter aus dieser Übung. Benutzen Sie `mkConnection` aus dem Paket `Connectable` für die Verbindung.