Vorlesung Architekturen und Entwurf von Rechnersystemen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2018/2019

Übungsblatt 4

Prof. Andreas Koch, Jaco Hofmann

In den folgenden Übungen soll ein Vorteil von Berechnungen auf Hardwarebeschleunigern besonders hervorgehoben werden. Dazu soll nach und nach ein streambasierter FAST-Bildfilter entworfen werden.

Für die folgenden Aufgaben kann es vorteilhaft sein, dass Sie den entsprechenden Teil erst in C(++) programmieren und diesen dann portieren.

Aufgabe 4.1 Vergleich für FAST

In dieser Teilaufgabe soll der Vergleicher für jeweils n (12) Pixel gegen den mittleren Pixel + einen Threshold (0) implementiert werden. Entscheiden Sie sich zuerst für ein passendes Interface, hier kann ein bereits bekanntes wiederverwendet werden. Ein Pixel besteht aus 8 Bit, das Bild ist nur in grau. Nutzen Sie dafür:

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

Am Anfang des Entwurfs müssen Sie schon einige Entscheidungen treffen. Sollen alle 12 Vergleiche seriell ablaufen oder alle gleichzeitig? Sollten die Vergleiche seriell geschehen, dann reicht 1 8-Bit Vergleicher. Die Implementierung kann aber auch eine Pipeline nutzen: 12 8-Bit Vergleicher, jeder davon vergleicht einen anderen Wert pro Takt. Außerdem ist es von Interesse, ob Standard-FIFOs, ByPassFIFOs etc. verwendet werden. In was unterscheiden sich diese beiden?

Aufgabe 4.2 Testen der Implementierung

Testen Sie Ihre Implementierung mit einer herrkömmlichen Testbench. Der Vergleicher soll `True` zurückliefern, falls alle Pixel größer als der mittlere sind.

Aufgabe 4.3 Testen der Implementierung mit Bluecheck

Testen Sie auch Ihre Implementierung mit Hilfe vom Bluecheck. Um einen verlässlichen Wert zu bekommen, können Sie hier `List#(t)` zusammen mit `toList()` und `sort()` verwenden. Diese Funktionen sind allerdings sehr unperformant, weswegen sie nicht in normalen Modulen eingesetzt werden sollten.

Aufgabe 4.4 Parametrisieren der Implementierung

Bis jetzt sind die Anzahl der Pixel und der Threshold fest reingeschrieben. Sie können folgend ein Modul bzw. ein Interface parametrisieren. Achten Sie dabei auf die Konvertierung zwischen den verschiedenen Typen. Auch zu beachten ist, dass es in Bluespec `numeric type` gibt. Diese sind Zahlen, die zu Typen "wurden". Ein Beispiel dazu finden Sie weiter unten. In Typdeklarationen werden `numeric type` gebraucht, keine `Integer`. Als letztes ist es manchmal notwendig, an parametrisierte Typen Anforderungen zu stellen. Z.B., dass ein `numeric type` höchstens 8 ist. Dies erreichen Sie mit `provisos`.

- `unpack`: Konvertiert `Bit#(...)` in einen bitrepräsentierbaren Typ.
- `pack`: Konvertiert einen bitrepräsentierbaren Typ in `Bit#(...)`.
- `fromInteger`: Konvertiert `Integer` in `Bit#(...)` oder ähnliches.
- `valueOf`: Konvertiert einen `numeric type` in `Integer`.

```
typedef 8 N; // numeric type
typedef Bit#(N) GrayScale; // type

interface MyServer#(type a, type b);
    interface Put#(a) request;
    interface Get#(b) response;
endinterface: MyServer

module mkTest#(Integer a, Integer b [...])(Server#(n, m));
    [...]
endmodule
```