# FPGA-basiertes Template-Matching mit Distanztransformierten Bildern

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Diplom-Physiker Stefan Hezel
aus Tuttlingen

Mannheim, 2004

| | |
|---|---|
| Dekan: | Professor Dr. Jürgen Potthoff, Universität Mannheim |
| Referent: | Professor Dr. Reinhard Männer, Universität Mannheim |
| Korreferent: | Professor Dr. Peter Fischer, Universität Mannheim |

Tag der mündlichen Prüfung: 03. Mai 2004

# Kurzfassung

In dieser Arbeit werden Implementierungsstrategien für einen von D. Gavrila gegebenen Algorithmus zur Erkennung von Verkehrszeichen für FPGAs entwickelt. Als Zielplattform wurde das PCI-Board MPRACE ausgewählt, welches mit einem modernen Virtex-II XC2V3000-FPGA bestückt ist. Der Algorithmus benutzt als Merkmale Kanten und deren Orientierung im Bild. Acht distanztransformierte (DT) Bilder, die den acht diskretisierten Richtungen der Kanten zuzuordnen sind, werden berechnet. Zum Auffinden der Ränder der Verkehrszeichen wird ein Template-Matching-Verfahren auf den DT-Bildern durchgeführt. Sowohl die Berechnungen der DT-Bilder als auch des Template-Matchings sind zeitkritisch. Ziel dieser Arbeit ist es, die Extraktion der Merkmale und das Template-Matching für insgesamt 36 kreisförmige und dreieckige Templates für Bilder der Größe $512 \times 512$ in Echtzeit auf auf dem MPRACE-Koprozessor durchzuführen.

Die Algorithmen zur Bestimmung der DT-Bilder erweisen sich als gut parallelisierbar. Alle hierfür entwickelten Module sind nach dem Pipeline-Prinzip aufgebaut und können zu zwei größeren Pipelines zusammengefasst werden. Die Ressourcen des FPGAs des MPRACE-Prozessors sind hierfür zu 12% ausgelastet. Im Hauptteil dieser Arbeit werden Implementierungsstrategien für das Template-Matching untersucht. Es zeigt sich, dass die Module für das Matching in einer großen Pipeline zusammengefasst werden können. Die für das Matching relevanten Pixel der DT-Bilder werden in einer Matrix von Registern gespeichert, und die Ähnlichkeitsmaße für einen Bildpunkt werden mit binären Addiererbäumen für alle Templates gleichzeitig bestimmt. Diese Methode führt allerdings zu einem sehr hohen FPGA-Ressourcenbedarf. Mit der direkten Implementierung dieses Verfahrens können die DT-Bilder und das Matching von höchstens 24 Templates auf dem eingesetzten FPGA berechnet werden. Die Berechnungen der DT-Bilder und des Template-Matchings sind problemlos mit 30 Hz auf dem FPGA durchführbar.

Zur Reduzierung des FPGA-Ressourcenbedarfs für das Template-Matching werden vier Optimierungsstrategien entwickelt. Es wird untersucht, wie benachbarte Register, auf die nicht von den Addiererbäumen zugegriffen wird, ressourcensparend zu ersetzen sind. Die Addiererbäume können derart aufgebaut werden, dass Teilsummen, die für mehrere Templates zu berechnen sind, zusammengefasst und nur einmal ausgeführt werden. Hierzu wird ein iterativer Algorithmus zum Aufbau von optimierten Addiererbäumen für unverschobene Templates entwickelt und für Templates mit verschobenen Templateelementen erweitert. Eine weitere Strategie zur Verringerung des Ressourcenbedarfs ist, die Anzahl der Templateelemente der Templates zu reduzieren. Für die reduzierten und ursprünglichen Templates werden Strategien für deren Verschiebungen, entwickelt. Für Templates mit einzeln verschobenen Templateelementen lässt sich der FPGA-Ressourcenbedarf für die optimierten Registerfelder und optimierten Addiererbäume um einen Faktor $3.5$ verringern.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Einleitung

Der beste Bildverarbeiter, den wir kennen, ist in den meisten Fällen der Mensch selbst. Als Verkehrsteilnehmer ist er in der Lage, das gesamte Verkehrsgeschehen zu erfassen, vorausgesetzt er ist aufmerksam. Dabei können z.B. Verkehrszeichen weit weg oder nah, in gutem Zustand oder leicht beschädigt, ganz sichtbar oder teilweise verdeckt sein oder sogar auf dem Kopf stehen. Der Mensch hält es für selbstverständlich sich in seiner Umwelt unter unterschiedlichsten Bedingungen bestens zurechtzufinden und ist sich dabei der Komplexität dieser Aufgabe nicht bewusst, zumindest so lange nicht, bis er versucht, dieses Maschinen beizubringen. Die Bildverarbeitung bzw. Bildanalyse beschäftigt sich damit, wie Maschinen ihre Umwelt sehen, Objekte Ihres Interesses erkennen und vernünftige Entscheidungen über die Zugehörigkeit der Objekte in eine Objektklasse treffen können.

Bildverarbeitungssysteme übernehmen in vielen Bereichen unseres Lebens Aufgaben der automatischen Bildauswertung. Als Beispiele aus der Wissenschaft und Wirtschaft seien die Spuranalyse in Detektorbildern in der Hochenergiephysik, oder der Qualitätskontrolle im Fertigungsprozess, z.B. Laserschweißen, genannt. In Zukunft wird dem Einsatz von bildverarbeitenden Systemen im Fahrzeug zur Unterstützung der Verkehrsteilnehmer im Straßenverkehr eine hohe Bedeutung zugemessen werden. Neben dem Erkennen der Fahrbahn oder anderer Verkehrsteilnehmer wie Fahrzeuge oder Fußgänger, ist das Erkennen von Verkehrsschildern ein wichtiger Teilaspekt. Die Untersuchung von Methoden, Algorithmen und Implementierungen auf FPGA-basierter hybrider Hardware zur Erkennung von Verkehrszeichen in Echtzeit ist Gegenstand dieser Arbeit.

Obwohl die Grundlagen vieler Methoden und Algorithmen schon in der zweiten Hälfte des letzten Jahrhunderts gelegt wurden ist die praktische Lösung mit diesen Verfahren von komplexen Anwendungen auch heute noch schwierig. Dies liegt zum einen daran, dass es eine große Anzahl unterschiedlicher Methoden gibt, welche je nach Problemstellung mehr oder weniger zufrieden stellende Ergebnisse liefern und sich teilweise sehr stark in ihrer Komplexität und damit auch im Bedarf von Rechenressourcen unterscheiden. Außerdem sind oft zeitliche Rahmenbedingungen, in denen die Algorithmen berechnet werden müssen, von der Anwendung vorgegeben. Beide Aspekte, die sich gegenseitig beeinflussen, sind gleichzeitig zu betrachten: eine geeignete Auswahl bzw. Entwicklung von Methoden, die für die Anwendung zufrieden stellende Ergebnisse liefern, sowie eine geeignete Auswahl von Hardware, auf der die Algorithmen in der vorgegebenen Zeit möglichst effizient ausgeführt werden können.

Der kantenbasierte Algorithmus von D. Gavrila [1] zu Erkennung von Verkehrszeichen, der auf Grauwertbildern arbeitet und Grundlage dieser Arbeit ist, lässt sich in zwei große Teile gliedern. Einerseits dem Erkennen von Ort und Größe der Form potentieller Verkehrszeichen und andererseits der Zuweisung dieser in eine bestimmte Verkehrszeichenklasse oder ggf. einer Zurückweisung. Im ersten Schritt wird ein Matching-Algorithmus eingesetzt der mit binären Templates auf distanztransformierten Kantenbildern arbeitet. Als zusätzliche Merkmale werden die Orientierungen der Kanten berücksichtigt. Das Matching wird auf Bildern der Größe $512 \times 512$ und mit jeweils zwölf kreisförmigen Templates, dreieckigen Templates mit Spitze nach unten und dreieckigen Templates mit Spitze nach oben durchgeführt. Unter diesen Rahmenbedingungen ist das Matching sehr rechenintensiv und daher prädestiniert zur Ausführung auf parallelen Hardwarearchitekturen. Im zweiten Schritt wird ein statistischer RBF-Klassifikator eingesetzt. Die Klassifikation der potentiellen Kandidaten ist jedoch nicht rechenzeitkritisch. Daher war eine Untersuchung von FPGA-Implementierungsstrategien nur für den ersten Schritt, der Detektion der Verkehrszeichen, vorgesehen.

Oft werden in der Praxis sog. hybride Systeme, bestehend aus einem Standard-Prozessor und speziellen Prozessoren wie z.B. ASICS oder rekonfigurierbaren Bausteinen wie FPGAs, eingesetzt. FPGAs erlauben den effizienten Aufbau paralleler Rechenarchitekturen nach dem Pipeline-Prinzip. Algorithmen mit guten Parallelisierungseigenschaften lassen sich auf FPGAs gegenüber PCs oft um ein bis zwei Größenordnungen schneller ausführen. Gegenüber den höher taktenden, jedoch fest verdrahtete ASICS besitzen FPGAs den Vorteil, dass der Algorithmus jederzeit, z.B. nach Erfahrungen im Einsatz im Straßenverkehr, geändert werden kann. Die Einsatzmöglichkeiten von programmierbaren FPGAs sind daher ähnlich flexibel wie die von Standard-Prozessoren. Jedoch ist das Erstellen oder Ändern von Programmen für Standard-Prozessoren mit geringerem Aufwand als für FPGAs durchzuführen. Man ist daher interessiert nur die wirklich zeitkritischen Teile des Algorithmus auf FPGAs abzubilden.

Als Hardware-Plattformen werden der FPGA-Koprozessor *microEnable* von der Firma Silicon Software und der am Lehrstuhl entwickelte FPGA-Koprozessor MPRACE [37] eingesetzt. Beide sind jeweils mit einem rekonfigurierbaren FPGA der Firma Xilinx ausgestattet und verfügen über eine Anbindung zum Host Computer via PCI-Bus. Über Stecker, die direkte physikalische Verbindungen mit dem FPGA besitzen, kann zur Bildaufnahme eine Kamera angeschlossen werden. Die Programmierung der FPGA-Module erfolgt mit der von K. Kornmesser an der Universität Mannheim entwickelten C++ basierten Hardwarebeschreibungssprache CHDL [70].

Das Ziel dieser Arbeit besteht darin, den von D. Gavrila [1] vorgeschlagenen Algorithmus zur Verkehrszeichenerkennung auf einem hybriden Bildverarbeitungssystem, bestehend aus einem Host-Computer mit einem Standard-Prozessor sowie einem oder mehreren der oben vorgestellten FPGA-Koprozessoren, effizient abzubilden. Es wird untersucht, inwiefern die sequentiell und teilweise rekursiv implementierten Algorithmen für eine parallele FPGA-Hardwarearchitektur abzubilden und ggf. anzupassen sind. Strategien für die Parallelisierung der Algorithmen nach dem Pipeline-Prinzip unter Berücksichtigung der zeitlichen Anforderungen bei einem möglichst niedrigen Bedarf an FPGA-Ressourcen werden entwickelt. Änderungen am Algorithmus, die zu einer effizienteren FPGA-Implementierung führen, sollen berücksichtigt werden. An die FPGA-Implemen-

tierung des Template-Matching-Algorithmus [1] werden daher folgende Anforderungen gestellt:

- Das Template-Matching für die in dieser Arbeit eingesetzten 36 kreisförmigen und dreieckigen Templates und die Erzeugung der DT-Bilder der Größe $512 \times 512$ soll in Echtzeit mit 30 Bilder/sec ausgeführt werden können.
- Die Implementierung des Algorithmus für das Template-Matching sollte mit einem möglichst geringen Ressourcenverbrauch für das FPGA erfolgen. Dies minimiert die Hardwarekosten nicht nur für das Prototyping sondern für den späteren Einsatz im Fahrzeug.
- Das FPGA-Design sollte modular aufgebaut werden. Die Module sollten nach Möglichkeit hoch parametrisierbar sein. Die Einstellung von wichtigen Parametern sollte außerdem während der Laufzeit möglich sein.

Diese Arbeit gliedert sich wie folgt:

Im ersten Kapitel werden grundlegende Methoden der Bildverarbeitung bzw. -analyse vorgestellt, die für die Detektion von Merkmalen bzw. Objekten und deren Klassifizierung in Objektklassen von Bedeutung sind und zur Erkennung von Verkehrszeichen eingesetzt werden. Anschließend wird ein Überblick über Algorithmen zur Erkennung von Verkehrszeichen gegeben, die auf Grauwertbildern arbeiten. Das Kapitel schließt mit einer ausführlichen Beschreibung der im Rahmen dieser Arbeit verwendeten FPGA-Koprozessoren sowie deren Programmierung.

Der von D. Gavrila entwickelte Algorithmus zur Erkennung von Verkehrszeichen, der Grundlage dieser Arbeit ist, wird in Kapitel 2 ausführlich beschrieben. Zusätzlich wird auf mögliche Verbesserungen einzelner Schritte des Algorithmus eingegangen und es werden Auswirkungen auf die Rechenzeit diskutiert.

In Kapitel 3 werden die FPGA-Implementierungen zur Erzeugung der distanztransformierten Bilder beschrieben. Es wird untersucht, wie die einzelnen Module aufzubauen und günstig zu Pipelines zusammenzufassen sind. Diese sind im Detail beschrieben und illustriert. Die benötigten Rechenzeiten und FPGA-Ressourcen werden diskutiert. Außerdem werden die verwendeten Module zur Bildaufnahme vorgestellt.

Kapitel 4 beschäftigt sich mit der Umsetzung des Template-Matchings für FPGAs für die verwendeten 36 Templates. Die für die Templates relevanten Daten werden in einer Matrix von Registern auf dem FPGA gespeichert. Das Ähnlichkeitsmaß für das Matching wird für alle Templates gleichzeitig mit binären Addiererbäumen durchgeführt. Alternative Implementierungsstrategien werden diskutiert. Außerdem wird ein Überblick über die Gesamtschaltung der Module zur Berechnung der DT-Bilder und des Template-Matchings angegeben.

Optimierungen bezüglich dem FPGA-Ressourcenverbrauch der Implementierung für das parallele Template-Matching sind Gegenstand von Kapitel 5. Zum einen wird untersucht, wie die Anzahl der Register verringert werden kann, falls benachbarte Register, auf die nicht von den Addiererbäumen zugegriffen wird, zusammengefasst werden und durch ressourcensparende Verzögerungsglieder (SRLTs) ersetzt werden. Des Weiteren

wird untersucht, wie die Addiererbäume aufzubauen sind, falls mehrere Templateelemente unterschiedlicher Templates überlappen und deren Partialsummen gemeinsam ausgeführt werden können. Ein neuer Algorithmus für den gemeinsamen Aufbau von Addiererbäumen wird zunächst für unverschobene Templates vorgestellt, dann für Templates, die als Ganzes verschoben werden, und schließlich für Templates mit einzeln verschobenen Templateelementen erweitert. Die Verschiebungen der Templateelemente für die in dieser Arbeit verwendeten 36 Templates, so dass sich viele Überdeckungen ergeben, werden angegeben und die resultierenden Templates illustriert. Als weitere Strategie zur Verringerung des FPGA-Ressourcenbedarfs wird zugelassen, die Anzahl der Templateelemente der Templates zu reduzieren. Die Auswirkungen dieser vier Optimierungsstrategien auf den FPGA-Ressourcenbedarf werden diskutiert.

Die Arbeit schließt mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf mögliche Verbesserungen der Methoden.

Kapitel 1

# Grundlagen

In diesem Kapitel werden in Abschn. 1.1 allgemeine Methoden aus der Bildverarbeitung und -analyse vorgestellt, die für das Verständnis von Algorithmen zur Erkennung von Verkehrszeichen von Bedeutung sind. Algorithmen hierzu, die auf Grauwertbildern arbeiten, werden in Abschn. 1.2 diskutiert.

In Abschn. 1.3 wird ein geeignetes Bildverarbeitungssystem, bestehend aus rekonfigurierbaren FPGA-Prozessoren beschrieben und auf dessen Einsatzmöglichkeiten und Eigenschaften eingegangen.

Der Algorithmus zur Erkennung von Verkehrszeichen [1], der im Rahmen dieser Arbeit für FPGAs implementiert wurde, wird dann in Kapitel 2 im Detail beschrieben.

## 1.1 Digitale Bildverarbeitung und -analyse

Digitale Bildverarbeitung bzw. -analyse lässt sich in die drei Teilschritte *Sehen*, *Erkennen* und *Entscheiden* unterteilen [6].

Als erster Teilschritt ist die Bildaufnahme bzw. -repräsentation zu betrachten, die im Rahmen dieser Arbeit mit einer digitalen Kamera erfolgt. Die Bildaufnahme ist eine Projektion aus unserer Umwelt, einem dreidimensionalen Raum, in eine zweidimensionale Bildebene. Man spricht in diesem Zusammenhang auch von 3D-Welt- und 2D-Bildkoordinaten. Der mit der Bildaufnahme verbundene Informationsverlust wird vom Menschen durch sein visuelles System ausgeglichen, welches ihm einen dreidimensionalen Eindruck vermittelt.

Der zweite Teilschritt auf dem Weg zur Klassifikation bzw. Bildanalyse ist die Merkmalsextraktion. Viele dieser allgemeinen Konzepte zur Merkmalsreduktion sind in Bildverarbeitungsbüchern wie z. B. Jähne [6, 7] oder [8] erläutert. Sie dienen im Wesentlichen dem Erkennen bzw. Trennen von Wichtigem von Unwichtigem. Vorverarbeitende Schritte haben die Aufgabe Objekte zu segmentieren oder vom Hintergrund hervorheben, Rauschen zu unterdrücken oder beseitigen oder die Objekte zu normalisieren. Die Merkmale werden häufig in Merkmalsbildern oder Merkmalsvektoren dargestellt. Der Merkmals-

vektor ist eine Repräsentation von Objekten in der wirklichen Welt. Merkmale, die im Rahmen dieser Arbeit verwendet werden, sind z.B. Kanten und deren Orientierungen.

Der dritte Teilschritt der Bildanalyse ist die Klassifikation. Deren Aufgabe ist die Zuweisung von Objekten in bestimmte Objektklassen. Muster bzw. Objekte im Bild können oft nicht direkt klassifiziert werden, weil die auszuführenden Rechenschritte enorme Ausmaße annehmen würden. Daher wird die Klassifikation häufig auf den Merkmalsbildern oder -vektoren durchgeführt. Die Resultate der Klassifikation sind dabei stark abhängig von der Wahl der Merkmale. Das Ergebnis ist die Zuweisung eines Objektes in eine Klasse oder deren Zurückweisung. Bei dem in dieser Arbeit eingesetzten Template-Matching wird ausschließlich eine Vorentscheidung darüber getroffen, ob ein Bildbereich, welcher der Größe eines Templates entspricht, für eine weitergehende Untersuchung interessant ist. Die letztendliche Entscheidung trifft ein RBF (*Radial-Basis-Funktion*)-Klassifikator.

Die inhaltliche Dreiteilung der Schritte der Bildverarbeitung bzw. -analyse ist zusammenfassend in Abb. 1.1 illustriert.

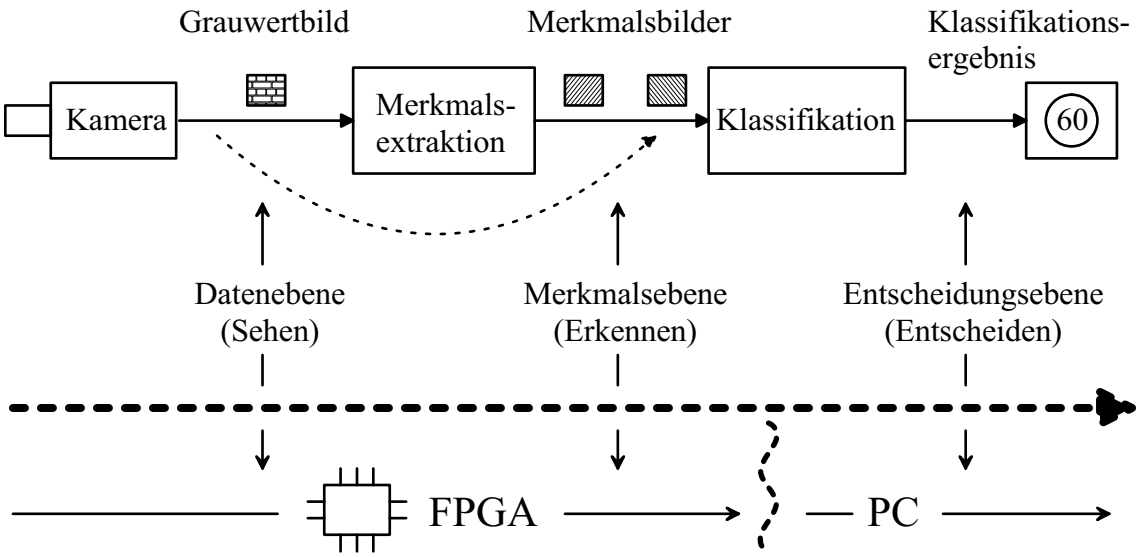



**Abbildung 1.1:** Teilschritte der Bildverarbeitung bzw. -analyse.

Wird ein Algorithmus zur Bildanalyse auf einem hybriden Bildverarbeitungssystem bestehend aus einem PC und einem FPGA-Koprozessor abgebildet, so stellt sich die Frage, wie die Teilschritte auf die einzelnen Komponenten der Hardware zu unterteilen sind. Man wird i.A. versuchen, ausschließlich die rechenzeitintensiven Teile des Algorithmus auf den FPGA abzubilden, weil die Implementierung für FPGAs mit einem wesentlich höheren Aufwand als bei einer Softwarelösung verbunden ist.

Der weitere Verlauf dieses Abschnitts gliedert sich wie folgt:

- *Sehen*: Bildaufnahme bzw. -repräsentation in Abschn. 1.1.1
- *Erkennen*: Merkmalsextraktion in Abschn. 1.1.2
- *Entscheiden*: Klassifikation in Abschn. 1.1.3.

### 1.1.1 Bildrepräsentation

Als Ausgangspunkt der Betrachtungen in diesem Abschnitt soll ein ideales oder fehlerfreies Bild $g(x, y)$ dienen. Die Funktion $g$ soll dabei reellwertig auf einem Teilbereich der reellen Zahlen sein. Der Wert $g(x, y)$ heißt auch Grauwert oder Farbwert an der Stelle $(x, y)$.

In der Bildverarbeitung wird mit einem diskreten Bildmodell gearbeitet, welches man aus einer zweistufigen Diskretisierung (Digitalisierung) enthält

$$\begin{array}{ccc} g(x,y) & \rightarrow & G_{h,w} \\ x, y \in I\!R & & h, w \in I\!Z. \end{array} \tag{1.1}$$

Die Diskretisierung der Argumentbereichs (Ortsbereichs) heißt Abtastung. Dabei werden die Argumente auf endliche Teilintervalle der natürlichen Zahlen $\{0, \ldots, H-1\}$ und $\{0, \ldots, W-1\}$ abgebildet. Die Diskretisierung des Wertebereichs heißt Quantisierung. Die kontinuierlichen Grauwerte werden auf eine begrenzte Zahl $G$ diskreter Grauwerte abgebildet. Die Rasterung der Werte entsteht durch Mittelung, Rundung oder Schwellwertbildung. Hierauf soll nicht näher eingegangen werden.

Die Bildpunkte sind in einem quadratischen Gitter angeordnet und werden auch als Pixel[1] bezeichnet. Das Bild kann auch als eine $H \times W$ Matrix diskreter Werte $G_{h,w}$ aufgefasst werden mit $h = 0, \ldots, H-1$, $w = 0, \ldots, W-1$, siehe Abb. 1.2.

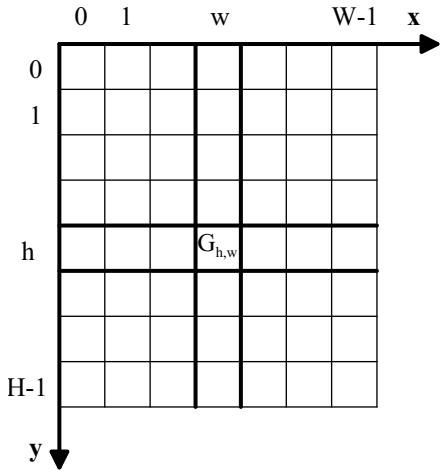



**Abbildung 1.2:** Repräsentation von Digitalbildern durch Felder diskreter Punkte auf einem quadratischen Gitter (nach [6]).

Kleine Werte sind bei einem Grauwertbild dunkel, hohe Werte hell. Grauwertbilder sind oft mit acht bis zehn Bit Auflösung dargestellt, was einem Wertebereich von 256 bis 1024 Grauwerten entspricht. Es hat sich herausgestellt, dass für das menschliche Empfinden 256 verschiedene Grauwertstufen ausreichend sind. Ein ideales Bild und das durch die Diskretisierung gewonnene Grauwertbild sind in Abb. 1.3 (a),(b) veranschaulicht.

[1] Engl. picture element.

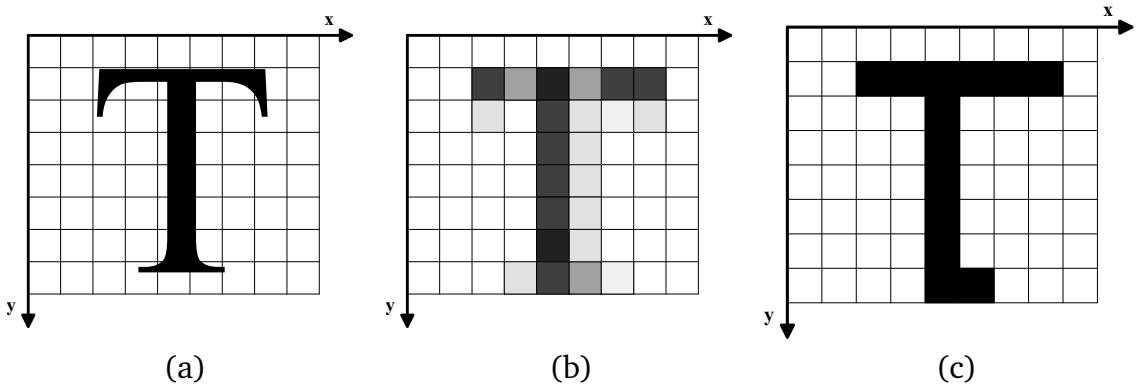



**Abbildung 1.3:** (a) Ideales Bild (b) diskretisiertes Grauwertbild (c) Binärbild.

In Abb. 1.3(c) ist das Binärbild des zugehörigen Grauwertbildes dargestellt. Dieser Schritt kann z.B. durch Schwellwertbildung realisiert werden. Der Wertebereich eines Binärbildes ist durch $\{0, 1\}$ gegeben. Dies entspricht einer Auflösung von einem Bit. Ein Pixel mit Wert 0 kennzeichnet ein Merkmal bzw. ein Objekt im Bild und eines mit dem Wert eins den Hintergrund.

In Abb. 1.4 ist die Repräsentation eines binären Templates gegeben. Die Templatepunkte besitzen den Wert 0 und die nicht zum Template gehörenden Punkte den Wert eins.

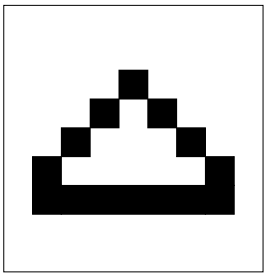


**Abbildung 1.4:** Repräsentation eines binären Templates nach ( [1]).

Ein weiterer wichtiger Begriff ist die Nachbarschaft eines Pixels. Sind die Bildpunkte bezüglich einer gemeinsamen Kante (direkten Nachbarn) benachbart, so spricht man von einer 4er-Nachbarschaft, sind sie zusätzlich bezüglich gemeinsamer Ecken (indirekte Nachbarn) benachbart, so spricht man von einer 8er-Nachbarschaft, siehe Abb. 1.5.

Das Konzept eines Bildes kann auch auf einen dreidimensionalen Argumentbereich erweitert werden. Ist die dritte Dimension die Zeit, so spricht man von Bildfolgen, und ist dies dagegen eine Ortskoordinate, von Volumen-Bildern. Die Gitterpunkte der Volumen-Bilder werden als Voxel[2] bezeichnet.

[2]Engl. volume element.

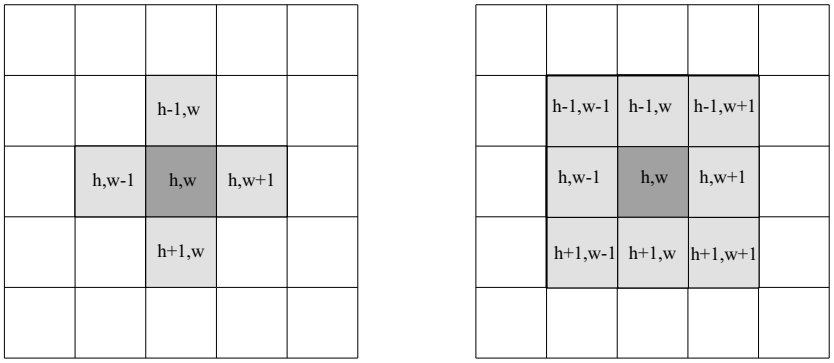



**Abbildung 1.5:** Nachbarschaften auf einem quadratischen Gitter: (a) 4er-Nachbarschaft (b) 8er-Nachbarschaft (nach [6]).

### 1.1.2 Merkmalsextraktion

Bei der Merkmalsextraktion ist man bestrebt, effiziente und trennscharfe Merkmale bereitzustellen und deren Anzahl so gering wie möglich zu halten. Merkmalsräume hoher Dimensionalität können dabei von Nachteil sein, weil dies die Anzahl der Klassifikationsfehler erhöhen kann. Für die nachfolgende Klassifikation ist es wünschenswert, wenn die Merkmalsvektoren einer Klasse einen kompakten Bereich bilden und wenn die Bereiche verschiedener Klassen weit auseinander liegen. Oft sind die erzeugten Merkmale in Merkmalsbilder gespeichert. Außerdem wird anhand der Merkmale versucht, das Bild auf einen interessanten Bereich (ROI[3]) einzuschränken.

Die zu extrahierenden Merkmale lassen sich in differentielle Merkmale oder in integrative Merkmale einteilen.

#### Differentielle Merkmale

Differentielle Merkmale werden oft durch lokale Operatoren auf kleinen Nachbarschaften im Bild generiert [6]. Eine Maske wird als ein lokaler Operator auf einer kleinen Nachbarschaft bezeichnet. Durch Berechnung einer diskreten Faltung der Maske mit einem Bild, können Glättungs- oder Kantenbilder generiert werden. Jeder Bildpunkt im Bereich der Filtermaske wird hierbei mit den Koeffizienten der Maske multipliziert und deren Summe ist für den Zentralpixel der Maske der Wert im gefilterten Bild. Die Rechenvorschrift für eine Faltung ist in Abschn. 2.1.1 in Gl. 2.1 gegeben und in Abb. 2.2 illustriert. Ein einfacher Filter zur Differentiation erster Ordnung ist der Sobel-Operator, der in dieser Arbeit verwendet wird und in Abschn. 2.1.1 ausführlich beschrieben wird. Häufig eingesetzt wird auch der Laplace-Operator, ein Ableitungsfilter 2. Ordnung oder der sog. Canny-Kantendetektor [11]. Auf dem Differenzenbild wird anschließend eine Binarisierung durchgeführt. Das Ergebnis ist ein binäres Kantenbild.

Als weitere Merkmale werden häufig die Orientierung bzw. Richtung von Kanten verwendet. Diese können effizient mittels Sobel-Operatoren ermittelt werden. Diese Me-

[3] Region of interest.

thode findet in dieser Arbeit Verwendung. Auf weitere Methoden zur Bestimmung der Orientierung wird in Abschn. 2.1.2 eingegangen.

Eine weitere Klasse von Nachbarschaftsoperationen sind morphologische Operatoren [10], welche die Form von Objekten beeinflussen. Die morphologischen Operatoren sind binäre Nachbarschaftsoperationen auf Binärbildern, die meistens eine binäre Maske verwenden, die Strukturelement genannt wird. Die binäre Faltung wird mit logischen *Und*- bzw. *Oder*-Operationen ausgeführt. Grundlegende Operatoren sind *Dilatation* (Oder) oder *Erosion* (Und) eines binären Objektes, welche ein Objekt ausdehnen oder zusammenziehen. Die Kombination aus *Erosion* und *Dilatation* wird *Opening* genannt. Durch die Ausdehnung des Objektes werden kleine Löcher gefüllt und die Umrandung des Objektes geglättet. Anschließend wird das Objekt auf die Ausgangsgröße zusammengezogen. Die Kombination aus *Dilatation* und *Erosion* wird *Closing* genannt. Durch das Zusammenziehen der Objekte verschwinden solche im Bild, die kleiner als die Maske sind. Anschließend werden die Objekte wieder auf die Ausgangsgröße ausgedehnt. Mit dem *Closing*-Operator können z.B. im Bild vorhandene Störpixel eliminiert werden. Im Rahmen dieser Arbeit wird diese Aufgabe vom sog. *Cleaning*-Operator übernommen, siehe Abschn. 2.1.1. Ein weiterer morphologischer Operator, die Distanztransformation (DT), ist in Abschn. 2.1.3 ausführlich beschrieben. Auf den DT-Bildern, die jeweils einem unterschiedlichen Richtungsbereich zuzuordnen sind, wird das Template-Matching ausgeführt.

### Integrative Merkmale

Einfache integrative Merkmale, die häufig Verwendung finden, sind z.B. mittlerer Grauwert (Energie), lokale Varianz, Zentralmomente höherer Ordnung oder Entropie (erwartete Information) für einen Bildausschnitt [8]. Dabei ist die lokale Varianz ein Moment zweiter Ordnung und beschreibt die mittlere quadratische Abweichung eines Grauwertes vom Mittelwert.

Ein wichtiger Teilbereich der Merkmalsextraktion ist die Segmentierung. Das Ziel der Segmentierung besteht darin, inhaltlich zusammengehörige Regionen bzw. Segmente zu generieren. Es gibt eine Vielzahl von Segmentierungsverfahren, von denen einige kurz erläutert werden.

Eine einfache Methode ist die pixelbasierte Segmentierung. Objekte, deren Grauwerte gute Trennungseigenschaften besitzen, können z.B. mit einem globalen Schwellwertverfahren segmentiert werden.

Bei regionenbasierten Verfahren wird ein Bildpunkt nicht nur aufgrund seines Grauwertes einem Objekt zugewiesen, sondern in Abhängigkeit von dem Wert seines Nachbarpixels. Als Beispiele seien das Bereichswachstumverfahren[4] oder eine hierarchische Segmentierung mittels Gauß- und Laplace-Pyramiden genannt.

Kantenbasierte bzw. konturbasierte Segmentierungsverfahren beziehen die Grauwertänderungen benachbarter Pixel und nicht von absoluten, globalen Grauwerten ein. Eine Kontur- oder Linienverfolgung erfolgt z.B. entlang des Nulldurchgangs der zweiten Ableitung

[4] Engl. region growing.

oder entlang dem Maximum bei einem Differenzenbild erster Ordnung. Letztere wird beim sog. Canny-Kantendetektor eingesetzt.

Ein Beispiel für einen älteren modellbasierten Ansatz ist die Hough-Transformation [13]. Die Idee der Hough-Transformation besteht darin, alle zum Template gehörenden Kantenpixel im Bild auf einen Punkt im Transformationsraum abzubilden. Die Punkte im Transformationsraum sind die Parameter der Templates. Je mehr Kantenpixel zu einem Template an einem Punkt im Bild gehören, desto deutlicher wird der zum Template gehörende Punkt im Transformationsraum zu einem lokalen Maximum. Nach der Durchführung der Transformation werden die lokalen Maxima ermittelt und auf das Bild zurückgerechnet, womit die Templates bzw. deren Parameter bestimmt sind. Mit der Hough-Transformation können auch unvollständige Templates im Bild erkannt werden.

Des Weiteren sind texturbasierte, netzwerkbasierte, diffusionsbasierte Verfahren oder Regularisierung zur Segmentierung zu nennen.

### 1.1.3 Klassifikation

Die Zuweisung von Objekten in bestimmte Objektklassen ist Aufgabe der Bildanalyse, welche ein Teilgebiet des allgemeineren Forschungsgebiets der Mustererkennung, ist [6]. Zum einen sind hierbei die Beziehungen zwischen Merkmalen und Objektklassen zu untersuchen. Zum anderen gilt es, einen optimalen Satz von Merkmalen zu bestimmen, mit denen die unterschiedlichen Objektklassen möglichst fehlerfrei eingeteilt werden können. Nach Möglichkeit sollte dies mit niedrigem Aufwand durchzuführen zu sein.

Die Klassifikation kann als eine Zuweisung von der Menge der Merkmalsvektoren in die Menge der Objektklassen angesehen werden. Ein einfacher pixelbasierter Klassifikator besteht z.B. aus einem Schwellwertverfahren welches jedem Bildpunkt oder Punkt im Merkmalsraum das Label Objekt oder Hintergrund zuweist. Dieser einfache Klassifikator kann eingesetzt werden, falls sich die verschiedenen Objekte im Bild nicht überlappen und gut vom Hintergrund unterscheiden.

#### Template-Matching

Das Template-Matching ist einer der einfachsten und frühesten Ansätze zur Formerkennung bzw. -analyse [18]. Ein Template bzw. eine Templatemaske, siehe Abb. 1.4, ist ein lokales Vergleichsmuster und wird zur Suche nach Mustern im Bild eingesetzt. Die Templates werden z.B. gegen ein binäres Bild “gematcht” wobei ein Ähnlichkeitsmaß berechnet wird. Bei einer hohen Ähnlichkeit hat man ein Muster im Bild gefunden. Hier kann vom Benutzer ein Schwellwert vorgegeben oder automatisch durch Training gelernt werden.

Zur Steigerung der Effizienz des Matchings wird häufig ein hierarchisches Matching durchgeführt. In [1] ist ein Ansatz beschrieben, der mit einer Templatehierarchie arbeitet, die aus drei Stufen von Templates besteht. Dieser Ansatz ist in Abschn. 2.2.4 ausführlich beschrieben. Ein anderer Ansatz beim hierarchischen Matching besteht darin, die Templates z.B. aus mehreren Kreisbögen oder aus Teilen von Linien mit unterschiedlicher Steigung zusammenzusetzen [12]. Diese Ansätze haben jedoch den Nachteil, dass

durch die Templatehierarchie oder durch das Zusammenfügen der Ergebnisse von Teilen des Matchings zusätzliche Fehler entstehen. Eine grundlegende Arbeit für das hierarchische Matching ist in [50] beschrieben.

Eine Methode, größere Strukturen bzw. Merkmale im Bild zu finden, die sich ausschließlich in der Skalierung gegenüber den Templates unterscheiden, besteht darin, auf dem sog. Skalenraum zu arbeiten [6]. Dieser wird durch eine Glättung des Bildes, z.B. durch mehrmalige Faltung des Bildes mit dem Binomialfilter, und anschließender Größenreduktion erzeugt. Der Skalenraum ist eine Folge von Bildern mit unterschiedlicher Auflösung. Große Strukturen im Bild bleiben dann in allen betrachteten Skalen sichtbar. In der Praxis erfolgt die Mehrgitterrepräsentation häufig mittels Gauß-Pyramide. Die Auflösung nimmt von einer Ebene zur nächsten um die Hälfte ab. Lokale Nachbarschaftsoperationen mit kleinen Kernen, wie z.B. das Template-Matching, können dann auf unterschiedlichen Skalen effizient berechnet werden.

Im Folgenden wird ein kurzer Überblick über weitere Verfahren zur Klassifikation gegeben. Für diese erfolgten in dieser Arbeit keine FPGA-Implementierungen.

### Statistische Klassifikatoren

Konzepte aus der statistischen Entscheidungstheorie werden eingesetzt um lineare oder nichtlineare Entscheidungsgrenzen zwischen den Objektklassen aufzubauen. Der Entscheidungsprozess kann folgendermaßen beschrieben werden: Ein gegebenes Muster basierend auf einem Merkmalsvektor ist in eine der Objektklassen zuzuweisen oder zurückzuweisen [14]. Dabei soll die Komplexität und Dimensionalität der Klassifikation möglichst gering gehalten werden.

Die Modellierung des Merkmalsraums erfolgt aus einer Menge von bekannten Merkmalsvektoren, den Trainingsdaten. Bilden die Merkmalsvektoren einen kompakten Bereich im Merkmalsraum, so spricht man von einer Punktwolke oder einem Cluster. Ideal ist es, wenn die Cluster weit auseinander liegen und somit gut trennbar sind. Eine einfache Möglichkeit, einen Cluster zu beschreiben, kann durch dessen Schwerpunkt erfolgen. Bei der Klassifikation wird ein Merkmalsvektor derjenigen Objektklasse zugeordnet, zu deren Schwerpunkt er den kürzesten Abstand[5] hat.

Bei wenigen Testvektoren ist diese geometrische Methode oft nicht ausreichend zur Merkmalsbewertung und die Zuordnung unbekannter Merkmalsvektoren zu einer Objektklasse ist nicht möglich bzw. mit einer hohen Unsicherheit verbunden.

Eine Möglichkeit zur Modellierung des Merkmalsraums kann mit statistischen Wahrscheinlichkeitsdichtefunktionen erfolgen. Bei den statistischen Klassifikatoren werden die Trennfunktionen durch die Wahrscheinlichkeitsverteilungen der Zugehörigkeit der Muster in die Klassen bestimmt. Zunächst muss jedoch die Verteilung der Merkmalsvektoren spezifiziert oder gelernt werden. Wahrscheinlichkeitsdichtefunktionen werden häufig durch Gaußfunktionen angenähert, deren Parameter aus den Trainingsdaten zu bestimmen sind. Ein anderer Ansatz besteht darin, die Dichtefunktionen mittles der sog. *Parzen-Window*-Methode zu schätzen [14]. Die Klassifikation wird folgendermaßen

[5] Wird auch als *nearest neighbour (NNB)* Methode bezeichnet.

durchgeführt: Der Merkmalsvektor wird zu derjenigen Objektklasse mit der höchsten Wahrscheinlichkeitsdichte[6] zugewiesen.

### RBF-Klassifikator

Der RBF-Klassifikator ist eine lokale, kernelbasierte Methode [14]. Die Grundlegende Idee von Kernel-Methoden besteht darin, die kontinuierliche Wahrscheinlichkeitsdichtefunktion durch eine lineare Kombination von radialen Basis-Funktionen aus den gegebenen Trainingsdaten zu approximieren. Allgemein haben Kernelfunktionen die Eigenschaft, in den Stützpunkten[7] den maximalen Wert zu besitzen und nach außen hin flacher zu werden. Die Basisfunktionen werden häufig durch einfache Funktionen approximiert. Beim RBF-Klassifikator werden die Basisfunktionen oft durch Gauß-Kernels repräsentiert.

Zur Klassifikation der Ergebnisse des Template-Matchings für den in dieser Arbeit verwendeten RBF-Klassifikator (RBF-Netzwerk) wird für die Basisfunktionen ein sehr einfacher Ansatz gewählt [4]. Aus Effizienzgründen wird der Gauß-Kernel durch eine radiale Treppenfunktion approximiert. Die linearen Treppenfunktionen (Rampen) sind in einem Bereich um den Stützpunkt konstant, und fallen dann linear auf Null ab. Für die Darstellung jeder Funktion werden dann zwei Parameter (Radien) benötigt. Zunächst werden die Referenzvektoren (Schwerpunktvektoren) der Cluster im Merkmalsraum bestimmt. Die beiden Parameter einer Rampe werden gelernt mittels nächsten Abstands zu einem Referenzvektor der eigenen Klasse und dem nächsten Abstand zu einem Referenzvektor einer anderen Klasse. Die Anzahl der Referenzvektoren variiert zwischen 50 bis 250 in Abhängigkeit der Verkehrszeichen-Klasse [2].

### Neuronale Netze

Neuronale Netze [17] werden häufig direkt auf den Bildern zur Formerkennung bzw. -analyse eingesetzt.

Im Folgenden wird ausschließlich auf vorwärtsgekoppelte Netzwerke eingegangen, die aus Perzeptronen[8] aufgebaut sind [20]. Ein Perzeptron besteht aus einer festen Anzahl von Elementen, denen Eingabemuster zugeführt werden. Die Komponenten des Eingabemusters können (normalisierte) Pixelgrauwerte eines Bildes bzw. Bildausschnitts oder durch Vorverarbeitungsschritte extrahierte Merkmale sein. Die Ausgangsaktivität des Perzeptrons ist Eins (aktiv), falls die gewichtete, lineare Summation der Komponenten des Eingangsmusters bzw. -vektors einen Schwellwert überschreitet, ansonsten Null (inaktiv). Anstelle der Schwellwertfunktion wird häufig eine sog. sigmoidale Aktivierungsfunktion mit einer kontinuierlichen Ausgangsaktivität zwischen Null und Eins eingesetzt. Ein einziges Perzeptron ermöglicht eine lineare Trennung zweier Musterklassen mit einer Gerade im zweidimensionalen Raum oder einer Hyperebene in höherdimensionalen Räumen. Der Einsatz von einem Netzwerk, welches aus einer Schicht von Perzeptronen

[6] In der englischsprachigen Literatur auch als *maximum likelihood* Methode bekannt.
[7] Engl. support points.
[8] Das Perzeptron wurde 1958 erstmals von Rosenblatt vorgeschlagen [19].

besteht, führt zu stückweise linearen Trennfunktionen. Jede beliebige nichtlineare Trennfunktion lässt sich aus einem geschichteten Netztwerk bestehend aus drei Schichten approximieren. Die Merkmalsextraktion erfolgt bei diesen durch die inneren Schichten.

Bei Neuronalen Netzen ist eine implizite Abbildung in einen höherdimensionalen Raum gegeben. Die Anzahl der Neuronen der Zwischenschichten ist von vorneherein nicht bekannt und schwierig zu bestimmen. Zu wenige Perzeptronen führen zu schlechten Trennungseigenschaften, zu viele Perzeptronen zu Overfitting. Letzteres bedeutet, dass gute Klassifikationsergebnisse für den Trainingsdatensatz aber schlechte für die unbekannten zu klassifizierenden Eingangsdaten erzielt werden. Die Vorteile von Neuronalen Netzen sind in derer geringen Abhängigkeit von domain-spezifischem Wissen zu sehen im Gegensatz zu modellbasierten oder statistischen Ansätzen. Dieser Abschnitt schließt mit einem Zitat über die implizite Ähnlichkeit von Neuronalen Netzen mit statistischen Klassifikatoren von Anderson et. al. [21]: “Neuronale Netze sind Statistiken für Amateure... Die meisten Neuronalen Netze verbergen die Statistiken des Benutzers.”

## 1.2 Algorithmen zur Verkehrszeichenerkennung mit Grauwertbildern

Zunächst werden in Abschn. 1.2.1 Algorithmen zur Erkennung von Verkehrszeichen vorgestellt. Diese werden nach Methoden der Merkmalsextraktion bzw. Detektion und Klassifikation getrennt untersucht und es wird auf Schwierigkeiten bei farbbasierten Ansätzen eingegangen. In Abschn. 1.2.2 wird begründet, warum der von D. Gavrila vorgestellte Algorithmus ausgewählt und für FPGAs implementiert wurde.

### 1.2.1 Algorithmen zur Verkehrszeichenerkennung

#### Merkmalsextraktion (Detektion)

Bei grauwertbasierten Algorithmen zur Verkehrszeichenerkennung werden in einem ersten Schritt fast ausschließlich differentielle Merkmale, die in Abschn. 1.1.2 und 1.1.3 beschrieben wurden, eingesetzt. Binäre Kantenbilder werden bei [1] und [12] mittles Sobel-Operatoren in x- und y-Richtung und anschließender Binarisierung erzeugt. Als zusätzliche Merkmale finden die Orientierungen der Kanten Verwendung, die aus den beiden mittels Sobel-Operatoren erzeugten Kantenbildern abgeleitet sind. [24] verwendet den sog. Canny-Kantendetektor zur Bestimmung der Kantenbilder. Zur weiteren Detektion bzw. Vorklassifizierung wird bei fast allen Algorithmen ein Template-Matching durchgeführt. In [1] und [24] wird das Matching mit den Templates als Ganzes ausgeführt. Die Templates enthalten als abgeleitetes Merkmal die äußere Form der Verkehrszeichen. In [1] wird das Matching auf DT-Bildern durchgeführt. Das Matching auf DT-Bildern führt bei fehlenden Pixeln einer Kontur in den Kantenbildern zu robusteren Ergebnissen. Eine Korrelation zwischen einem normalisierten Bildausschnitt und einem Template, welches direkt dem Verkehrszeichen entspricht, wird von [24] eingesetzt. Ein hierarchisches Matching auf einem binärem Bild wird in [47] durchgeführt. Zunächst

wird nach Teilstücken, aus denen die Templates zusammengesetzt sind, im Bild gesucht. In einem nächsten Schritt wird für die ermittelten ROIs das Template als Ganzes überprüft. In [24] geht dem Matching eine Formanalyse einzelner Segmente voraus. Für Dreiecke werden Kanten mit verschiedenen Steigungen von $0°, 60°$ und $-60°$ genauer betrachtet.

Integrative Merkmale wie Mittelwert, Energie oder Entropie finden in [26] Verwendung.

Die vorgestellten Algorithmen basieren fast ausschließlich auf differentiellen Merkmalen und einer anschließenden Formanalyse. Diese sind im Hinblick auf die verschiedenen Beleuchtungsverhältnisse, die im Straßenverkehr auftreten können, relativ robust.

### Klassifikation

Die Klassifikation (Abschn. 1.1.3) erfolgt häufig direkt auf den Bilddaten. Dem Klassifikator wird dann ein auf eine feste Größe normalisierter Bildausschnitt übergeben. In dieser Arbeit wird der in [1] und im vorigen Abschnitt beschriebene RBF-Klassifikator eingesetzt. Der Gauß-Kernel des RBF-Klassifikators wird hierbei durch radiale Treppenfunktionen approximiert. Ein Nearest-Neighbour (NNB)-Klassifikator kommt in [27] und Neuronale Netze kommen in [32] und [28] zum Einsatz.

In [26] wird ein *Parzen-Window*-Klassifikator mit Produkt-Laplace-Kernel vorgeschlagen. Der Laplace-Kernel ist dem Gauß-Kernel sehr ähnlich. Die Glättungsparameter für den Laplace-Kernel konvergieren jedoch für die verwendeten Verkehrszeichen um Größenordnungen schneller als beim Gauß-Kernel.

Außerdem kommen in der Verkehrszeichenerkennung Methoden wie Simulated-Annealing [27] und Genetische Algorithmen [28] zum Einsatz. Diese sind einerseits robust, andererseits jedoch sehr rechenintensiv und daher nicht für Echtzeitanwendungen interessant.

### Algorithmen mit Farbinformation

Wie grauwertbasierte benutzen auch farbbasierte Ansätze Form oder differentielle Merkmale [32]. Häufig werden integrale Merkmale wie Segmentierung eingesetzt, die z.B. mittels CCC[9] Code [31] generiert werden. Typisch ist anschließend die Suche nach gewissen Formen auf zusammenhängenden Flächen gleicher Farbe. Der Suchraum (ROI) wird z.B. in [24] durch farbbasierte Methoden grob eingeschränkt, indem der Himmel und die Fahrbahn im Bild detektiert werden.

Allerdings gibt es bei der Farbsegmentierung einige Schwierigkeiten. Die Farben der Verkehrszeichen variieren stark abhängig von ihrer Beschaffenheit und den Beleuchtungsbedingungen bei Tag, Nacht, Nebel, Regen oder Sonnenschein. Daher muss ein großer Bereich im Farbraum (RGB) der Basisfarben rot, gelb und blau abgedeckt werden. In [26] wird von einer i.A. höheren Erkennungsrate bei schlechter Beleuchtung (Nebel, Dämmerung, Dunkelheit, Nacht) bei Verwendung von Grauwertbildern berichtet. Eine falsche Farbsegmentierung kann fatale Auswirkungen für das Klassifikationsergebnis haben.

[9] Colour Connected Components.

### 1.2.2 Auswahl des Algorithmus und der Hardware-Plattform

#### Auswahl des Algorithmus

Die bisher vorgestellten Algorithmen zur Erkennung von Verkehrszeichen sind bezüglich ihrer Erkennungsrate schwierig zu vergleichen, weil keine Standardbildsequenzen definiert bzw. verfügbar sind. In den bisher vorgestellten Arbeiten werden unterschiedliche Verkehrszeichen untersucht, deren Anzahl und Templateklassen stark variieren. Außerdem sind die Verkehrszeichen aus unterschiedlichen Ländern verschieden.

Für den Algorithmus von D. Gavrila wird in [1] für eine Datenbank bestehend aus 1000 Bildern von Erkennungsraten von über $95$ % berichtet, die sich bei schlechten Umweltbedingungen (z.B. Regentropfen oder direkte Sonneneinstrahlung in die Kamera) oder teilweise Verdeckungen der Verkehrszeichen (z.B. durch die Scheibenwischer) um bis zu $15$ % verringern können. In [24] wurden Experimente für 600 Bilder, die ein oder mehrere dreieckige Muster beinhalten, durchgeführt. Es konnten Erkennungsraten von $92$ % erzielt werden. Für kreisförmige Templates, bei denen die Bilder in einfache und schwierige eingeteilt sind, wurden in Experimenten Erkennungsraten von $96$ % bzw. $93$ % ermittelt.

In dieser Arbeit wurde der Algorithmus von D. Gavrila [1] eingesetzt, weil seit Ende 1998 eine Kooperation zwischen dem Lehrstuhl für Informatik V der Universität Mannheim und Daimler-Chrysler[10] besteht. Von Seiten des Lehrstuhls sollte der Algorithmus für FPGAs implementiert werden. Der Algorithmus ist aufgrund der zusätzlichen Auswahl von Merkmalen sehr robust und effizient und führt zu hohen Erkennungsraten. Dieser wird in Kap. 2 ausführlich dargestellt und diskutiert und besteht im Wesentlichen aus den folgenden Teilschritten:

- Für die Bestimmung der Kanten im Bild wird ein einfacher Sobel-Operator in x- und y-Richtung mit anschließender Binarisierung und Eliminierung einzelner Störpixel eingesetzt. Diese Methode, die ebenfalls in [47] eingesetzt wird, liefert gute Ergebnisse für die Kanten und ist nicht besonders rechenintensiv.

- Als zusätzliches Merkmal werden die Orientierungen der Kanten verwendet, was zu weniger Fehlzuweisungen beim Template-Matching führt. Die Zuweisung der Kantenpixel in einen diskretisierten Richtungsbereich basiert auf den Pixeln der beiden Sobel-Bilder (ähnlich wie in [47]) und ist nicht rechenintensiv.

- Der Algorithmus ist der einzige zur Verkehrszeichenerkennung, der das Matching mit binären Templates auf DT-Bildern und nicht auf einem binären Kantenbild ausführt. Dies führt zu wesentlich weniger “falschen” Ergebnissen beim Template-Matching und daher zu wenigeren Rückweisungen durch den Klassifikator. Da für jeden Richtungsbereich ein DT-Bild berechnet wird, ist hierfür ein mittlerer Rechenaufwand zu veranschlagen.

- Ein weiterer Vorteil besteht darin, dass das Matching der Templates als Ganzes ausgeführt wird, und nicht zuerst nach einzelnen Segmenten der Muster gesucht

[10] Image Understanding Systems, Daimler-Chrysler Forschungszentrum, Ulm.

wird, die dann wieder zusammenzusetzen sind wie in [24, 47]. Die eingesetzte Methode führt i.A. bei einem höheren Rechenaufwand zu robusteren Ergebnissen beim Matching und zu weniger Rückweisungen durch den Klassifikator.

- Außerdem wird ein hierarchisches Template-Matching eingesetzt (Abschn. 2.2.4) bestehend aus einer Templatehierarchie und einem Bildraster aus jeweils drei Ebenen. Bei diesem Ansatz kann es eher vorkommen, dass nicht alle im Bild vorhandene Muster erkannt werden. Außerdem ist das hierarchische Matching für Bilder, die sehr viele Kanten beinhalten, nicht wesentlich schneller auszuführen als ein Matching, welches mit allen Templates an allen Bildpunkten erfolgt. Eine FPGA-Implementierung für letzteres wurde daher in dieser Arbeit bevorzugt.
- Die letztendliche Klassifikation der Verkehrszeichen im Bild wird nicht durch ein Matching [24] oder Neuronales Netzwerk [32] sondern durch ein RBF-Netzwerk realisiert, welches sich effizient implementieren lässt. Die Klassifikation ist nicht rechenzeitkritisch, wenn das vorausgegangene Matching nur wenige Ergebnispunkte liefert, und wird daher im Rahmen dieser Arbeit nicht weiter untersucht.

Die robuste Detektion von ROIs im Bild wirkt sich positiv auf die Rechenzeiten für den Algorithmus aus.

**Rechenzeiten** Für die Ausführung des Algorithmus von D. Gavrila auf einem Dual-Pentium-II mit 266 MHz und MMX[11] wird in [1] für 36 Templates und Bilder der Größe $360 \times 288$ mit einem hierarchischen Matching, welches in Abschn. 2.2.4 ausführlich beschrieben ist, von einer Bildwiederholrate von 6-8 Hz berichtet. Die Ausführungszeiten beim hierarchischen Matching sind jedoch stark vom Bildinhalt abhängig, was zu Raten von deutlich weniger als 6 Hz führen kann[12].

Im Folgenden wird die Anzahl der auszuführenden Rechenoperationen für die Berechnung der DT-Bilder und das Template-Matching von 36 Templates, die im Mittel etwas mehr als 100 Templateelemente besitzen, angegeben. Hierbei sind ca. 920 Mio. Additionen für ein Bild der Größe $512 \times 512$ auszuführen. Dies wird in Abschn. 4.2.1 genauer begründet. Sind 30 Bilder pro Sekunde zu berechnen, so ist eine Rechenleistung von ungefähr 27600 Mio. Festkomma-Additionen/sec notwendig.

### Auswahl der Hardware-Plattform

**Einsatz von CPUs und DSPs** Ein Standard-Prozessor (CPU[13]), ist die zentrale Rechen- und Steuereinheit eines Computers bzw. hybriden Bildverarbeitungssystems und nach der von-Neumann-Architektur aufgebaut. Nach der ursprünglichen Definition existiert ein Speichersegment, welches sowohl das Programm als auch die Daten enthält. Die Befehle des Programms, welches in Software erstellt wird, werden von der CPU interpretiert und ausgeführt. Moderne CPUs sind hoch optimiert für Gleitkomma-Arithmetik und

[11] Multi Media Extension.
[12] Aus persönlichem Gespräch mit D. Gavrila.
[13] Central Processing Units.

benötigen für diese nur wenige Prozessorzyklen. Die von Intel entworfene SIMD[14] Technik erlaubt die Anpassung von Integer-Daten an feste Bitbreiten von 8, 16 oder 32 Bit mit einer schnelleren Verarbeitung der Daten. Digitale Signalverarbeitungsprozessoren (DSPs[15]), die in Software programmiert und häufig in der Bildverarbeitung eingesetzt werden, sind für bestimmte Rechenoperationen geschwindigkeitsoptimierte CPUs. Bei schnellen DSPs ist der Programmspeicher physikalisch vom Datenspeicher getrennt, was einen gleichzeitigen Zugriff auf beide Speichersegmente erlaubt. Moderne DSPs besitzen mehrere Rechenwerke (ALUs) bzw. Multiply-Accumulate-Rechenwerke (MACs), die eine Multiplikation und anschließende Addition des Ergebnisses in einem Prozessorzyklus ausführen. Eine effiziente Ausnutzung der ALUs bei geringerer Rechengenauigkeit und Festpunkt-Arihtmetik ist mit 8, 16, 32 oder 64-Bit Daten möglich. Zur Kommunikation mit externem RAM stehen bei modernen DSPs zwei externe Speicher-Interfaces mit einem Datenbus von insgesamt 80 (64 und 16) Bit zur Verfügung.

Moderne Standard-Prozessoren können mit einer Frequenz von etwas mehr als 3 GHz getaktet werden. Werden in jedem Takt vier 8-Bit Festkomma-Additionen (mit SIMD-Optimierung) auf der CPU ausgeführt, so sind in einer Sekunde maximal 12000 Mio. 8-Bit Festkomma-Additionen möglich. Moderne DSPs können mit Taktfrequenzen bis zu 1000 MHz betrieben werden. Der TMS320C6416T-DSP[16] kann in einem Takt (1 ns) bis zu 24 Festkomma-Additionen (8-Bit) gleichzeitig ausführen. Dies führt zu einer max. Rechenleistung von 24000 Mio. 8-Bit Festkomma-Additionen/sec. Die Schwierigkeit bei den Standard-Prozessoren und DSPs besteht jedoch darin, dass die Daten auf denen ein Algorithmus arbeitet, den Rechenwerken nicht hinreichend schnell zugeführt werden können und sich eine reale Rechenleistung ergibt, die wesentlich geringer ist als die maximale.

**Einsatz von ASICs und FPGAs** ASICs[17] sind leistungsstarke Bausteine auf denen es möglich ist eine digitale Schaltung mit einer sehr hohen Packungsdichte abzubilden, die für einen anwendungsspezifischen Algorithmus konzipiert ist. Bei Änderungen am Algorithmus muss ein neues Design entworfen und ein neuer Chip produziert werden, was mit einem hohen Kostenaufwand verbunden ist. Ein FPGA ist ein frei programmierbarer Logikbaustein auf dem sich beliebige digitale Schaltungen abbilden lassen. Sind Änderungen am Algorithmus erwünscht, so ist ein neues Design zu erstellen und auf das FPGA, dessen Schaltkreise veränderbar sind, abzubilden. ASICs sind jedoch wesentlich schneller als FPGAs mit Gatter-Durchlaufzeiten im Picosekunden Bereich. FPGAs kommen häufig zur Prototyp-Entwicklung für Hardware-Implementierungen für ASICs zum Einsatz. Bei hohen Stückzahlen sind ASICs gegenüber FPGAs preiswerter. Sowohl bei FPGAs als auch ASICs ist die parallele Verarbeitung der Daten an den Algorithmus anpassbar. Wird dieser in Teilprobleme zerlegt, so können diese gleichzeitig von speziell hierfür konzipierten Verarbeitungseinheiten bzw. -ketten ausgeführt werden (Pipelining). Außerdem erlauben ASICs oder FPGAs die effiziente Ausführung eines Algorithmus, weil sich die Verarbeitungseinheiten bitgenau an die Wortbreite der Daten anpassen lassen. Die Nachteile von diesen bestehen jedoch in höheren Entwicklungskosten und längeren Entwicklungszeiten.

[14] Single Instruction Multiple Data.
[15] Digital Signal Processors.
[16] TMS320C6416T Fixed-Point Digiatal Signal Processor von Texas Instruments mit 1000 MHz.
[17] Application Specific Integrated Circuits.

Wird für den in dieser Arbeit eingesetzten Virtex-II XC2V3000 FPGA (mit 28672 LUTs) ein Design erzeugt, in welchem abwechselnd 8-Bit Festkomma-Addierer und 8-Bit Register miteinander verbunden sind, so lassen sich max. 1792 Festkomma-Addierer (8-Bit) auf das FPGA abbilden, bei einer angenommenen Ressourcenauslastung des FPGAs von $50\%$. Wird das Design mit einer Frequenz von 100 MHz getaktet, so können auf diesem maximal 179200 Mio. 8-Bit Festkomma-Additionen/sec ausgeführt werden.

Für die Ausführung des Algorithmus von D. Gavrila in Echtzeit sind FPGAs oder ASICs geeignet. Moderne Standard-Prozessoren und DSPs besitzen zu geringe Rechenkapazitäten um 27600 Mio. Additionen/sec für das Matching mit den 36 Templates ausführen zu können. Außerdem besteht für diese die Schwierigkeit beim Template-Matching darin, die Pixel der DT-Bilder den Rechenwerken (MACs) in der vorgegebenen Zeit zuzuführen. Bei FPGAs oder ASICs kann die sehr hohe Rechenleistung realisiert werden, weil auf diesen eine genaue Anpassung an die geringe Bit-Breite der zu verarbeitenden Daten möglich ist und diese effizient den Verarbeitungseinheiten zugeführt werden können. Dies wird in Kap. 4 im Einzelnen begründet. Ein weiterer Vorteil von FPGAs oder ASICS gegenüber CPUs ist deren wesentlich geringere Leistungsaufnahme. Dies hat zum einen Auswirkungen auf einen niedrigeren Kraftstoffverbrauch im Fahrzeug. Zum anderen entsteht weniger Wärme, die im Fahrzeug abzuleiten ist.

Für die Auswahl von FPGAs gegenüber ASICs werden folgende Gründe angeführt. Zunächst sollte in dieser Arbeit die Machbarkeit einer Prototyp-Entwicklung auf Hardware für den Algorithmus von D. Gavrila gezeigt werden im Hinblick auf Echtzeitfähigkeit und günstigem Hardware-Ressourcenbedarf. Des Weiteren können auf dem ASIC keine Änderungen am Algorithmus zur Verkehrszeichenerkennung im Einsatz, nachdem die Hardware in das Fahrzeug eingebaut wurde, durchgeführt werden. Sollen Schwachstellen am Algorithmus auf einer FPGA-basierten Hardware geändert werden, so ist dies leicht möglich, während bei ASICs ein neuer Chip produziert und die Hardware ausgetauscht werden müsste. Außerdem ist eine evtl. Anpassung des Algorithmus für unterschiedliche Länder auf FPGAs ohne Mehrkosten realisierbar. Bei ASICs müssten für verschiedene Länder eigene Chips entwickelt und produziert werden. Es wird sich zeigen, dass die Geschwindigkeit auf dem FPGA für die parallele Implementierung des Matchings für die in dieser Arbeit verwendeten Templates ausreichend und daher ein Einsatz von ASICs nicht notwendig ist.

## 1.3 FPGA-Architektur

In diesem Abschnitt wird die im Rahmen dieser Arbeit verwendete Hardware vorgestellt. In den ersten drei Jahren dieser Arbeit kam der rekonfigurierbare Koprozessor *microEnable*[18] zum Einsatz, ab Ende 2001 das an der Universität Mannheim entwickelte MPRACE-Board. Hauptbestandteil dieser beiden Koprozessoren sind jeweils ein FPGA (Field Programmable Gate Array) der Firma Xilinx, deren Aufbau, Eigenschaften und Art der Programmierung in den folgenden Abschnitten beschrieben wird. Zusätzlich werden zwei allgemeine Strategien zur hardwareseitigen Beschleunigung von Algorithmen erläutert.

[18] Silicon Software GmbH, Mannheim.

Anschließend werden die beiden im Rahmen dieser Arbeit verwendeten hybriden Bildverarbeitungssysteme, bestehend aus einem oder mehreren FPGA-Koprozessoren, einem handelsüblichen PC und einer Digitalkamera vorgestellt.

### 1.3.1 Verwendete FPGA-Bausteine

FPGAs sind Bausteine programmierbarer Logik. Auf diesen lassen sich beliebige digitale Schaltungen abbilden, vorausgesetzt, die Ressourcen des FPGAs sind hierfür ausreichend. Sie wurden erstmals 1985 von der Firma Xilinx [33] auf den Markt gebracht. Der grundlegende Aufbau der beiden im Rahmen dieser Arbeit verwendeten Bausteine von Xilinx wird im Folgenden beschrieben.

#### Xilinx XC4000

Das Kernstück eines FPGAs der XC4000er Familie [34] ist eine Matrix von programmierbaren Logikblöcken. Die Anbindung dieser CLBs (Configurable Logic Blocks) nach außen erfolgt über die sog. IOBs (In/Out Blocks). Für die innere Verdrahtung der CLBs und IOBs steht ein veränderbares Verbindungsnetzwerk zur Verfügung.

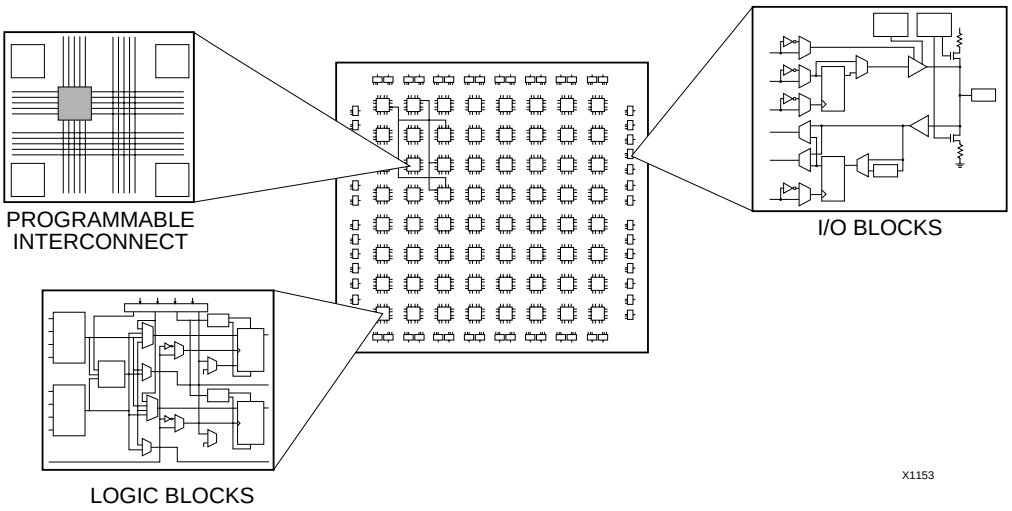



**Abbildung 1.6:** Aufbau eines FPGAs der Xilinx XC4000er Familie.

In Abb. 1.6 ist die Anordnung dieser unterschiedlichen Elemente zu sehen. Die Anzahl der CLBs eines FPGAs der 4000er Serie hat sich in nur wenigen Jahren vervielfacht. Der kleinste Baustein (XC4002XL) aus dem Jahre 1991 enthält 76 CLBs, der größte (XC40250XV) aus dem Jahre 1999 enthält 8464 CLBs. Die Komplexität von FPGAs hat sich im Mittel in der letzten Dekade pro Jahr verdoppelt. Dabei ist eine Zunahme der max. Frequenz, mit der die Designs getaktet werden, um ca. 20% pro Jahr zu beobachten. Insgesamt wächst die Rechenleistung von FPGAs schneller als bei Standard-Prozessoren.

In Abb. 1.7 ist das Blockschaltbild einer XC4000 CLB dargestellt. Sie besteht im Wesent-

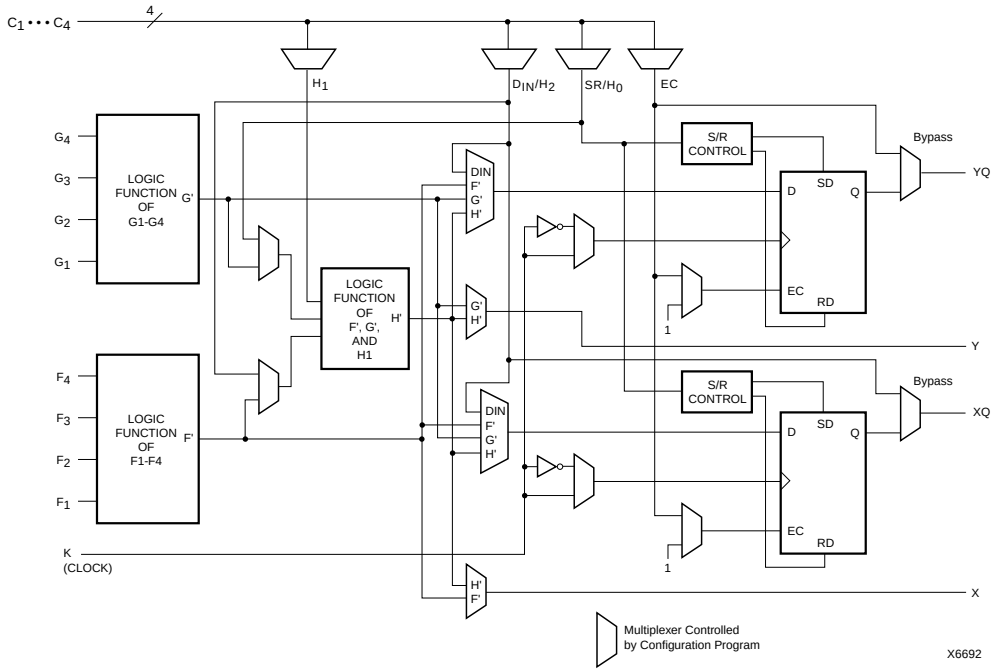



**Abbildung 1.7:** Struktur einer CLB der XC4000-FPGA Familie.

lichen aus zwei Funktionsgeneratoren $F$ und $G$ mit jeweils vier Eingängen sowie zwei Flip-Flops (FFs). Auf die beiden Funktionsgeneratoren (FGs) können beliebige logische Funktionen mit max. vier Eingangsvariablen abgebildet werden. Ein dritter Funktionsgenerator $H$ mit nur drei Eingängen verbindet die beiden vorigen FGs und ermöglicht mit einem noch freien Eingang den Aufbau beliebiger logischer Funktionen mit insgesamt neun Variablen. Dies wird durch den Einsatz von N-Bit-ROM Elementen als Look-Up-Tables (LUTs) realisiert.

Eine weitere, sehr wichtige Verwendung finden die Funktionsgeneratoren als Single-Port oder Dual-Port RAM. Das Single-Port RAM kann als $32 \times 1$, $16 \times 2$ oder $16 \times 1$ Bit-RAM, das Dual-Port RAM nur als $16 \times 1$ Bit-RAM konfiguriert werden. Die nachgeschalteten Multiplexer erlauben eine Durchleitung der Ausgänge der FGs oder Eingänge der CLB zu den nachfolgenden Registern oder Ausgängen der CLB. Die beiden Flip-Flops (FFs), die als flankengetriggerte D-Flip-Flops realisiert sind, finden Verwendung als Datenspeicher. Die beiden FFs sind an ein separates *Clock*- und *CE* (*Clock Enable*)-Verbindungsnetzwerk angeschlossen, das ein globales und asynchrones Setzen bzw. Zurücksetzen aller FFs ermöglicht.

Die Verdrahtung der IOBs und CLBs erfolgt über ein Verbindungsnetzwerk, welches aus einem Gitter von Verbindungskanälen und sog. Switch-Matrizen besteht. Es bestehen direkte Verbindungen von benachbarten Switch-Matrizen (*Single-Length-Lines*), Verbindungen, die einen Switch überspringen (*Double-Length-Lines*) und Verbindungen, die über den gesamten FPGA geführt werden (*Long-Lines*). Als Besonderheit steht eine sehr effiziente Fast-Carry-Logik zur Verfügung, welche den Aufbau schneller Arithmetik wie Addierer, Subtrahierer, Inkrementer oder Zähler ermöglicht. Jeweils zwei Bits für diese Rechenoperationen können in einer CLB verarbeitet werden.

### Xilinx Virtex-II

Eine Weiterentwicklung der 4000er Familie der Firma Xilinx ist der Virtex II-FPGA [35], der im Jahr 2001 auf den Markt kam und dessen Aufbau in Abb. 1.8 dargestellt ist. Neben CLBs, IOBs und Verbindungsnetzwerk, die der Funktionsweise der 4000er Familie

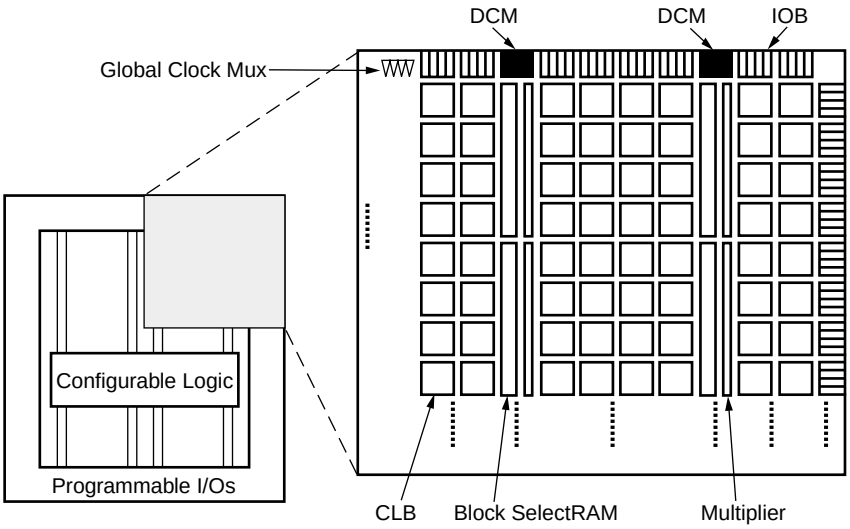



**Abbildung 1.8:** Virtex-II Schema.

ähnlich sind, enthält der Virtex-II zusätzlich Bänke von 18-kBit-BlockRAM und dieselbe Anzahl an Bänken von $18 \times 18$-Bit Multiplizierern. Das BlockRAM ist als synchrones Single- und Dual-Port RAM konfigurierbar. Die Datenbreite kann für jeden RAM-Block zwischen einem und 36 Bit gewählt werden, was eine entsprechende Anpassung der Adresstiefe erfordert.

Jede CLB enthält vier Slices, deren funktionale Darstellung in Abb. 1.9 gegeben ist. Jedes

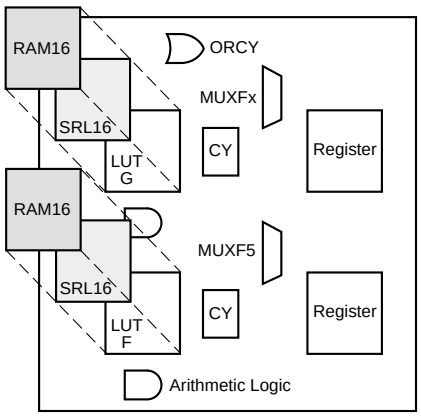



**Abbildung 1.9:** Virtex-II Slice.

Slice besteht im Wesentlichen aus zwei Funktionsgeneratoren und zwei FFs und ist somit

dem Aufbau einer CLB der 4000er Familie sehr ähnlich. Die beiden Funktionsgeneratoren lassen sich max. als $8$-Bit LUTs, synchrones $32$-Bit Single- und $16$-Bit Dual-Port-RAM und zusätzlich als $32 \times 1$-Bit Shift Register, die als Shift-Register-Lookup-Tables (SRLTs)[19] bezeichnet werden, verwenden. Innerhalb einer CLB ermöglichen spezielle Leitungen, die sog. CLB-*Feedback*-Pfade, ein effizientes Zusammenarbeiten der Slices. Als Verbindungen außerhalb einer CLB existieren *Single-Lines*, *Double-Lines*, *Hex-Lines* und *Long-Lines*.

Die IOBs erlauben die Anpassung an 15 verschiedene Signalstandards, wie z.B. LVDS[20]. Außerdem existieren bis zu zwölf voneinander unabhängige globale DCMs (Digital Clock Manager). Hiermit können Laufzeitunterschiede zwischen externer und interner Clock im FPGA mit DLLs (Delay-Locked-Loops) kompensiert werden. Die Taktfrequenz der externen Clock kann mit dem Frequenz Synthesizer verdoppelt oder in einem eingeschränkten ganzzahligem Verhältnis verändert werden.

Eine quantitative Beschreibung einiger FPGAs der Virtex-II Familie ist in Tab. 1.1 angegeben.

| XC2V... | 500 | 1000 | 1500 | 2000 | 3000 | 4000 | 6000 |
|---|---|---|---|---|---|---|---|
| Logikgatter | 500 k | 1 M | 1.5 M | 2 M | 3 M | 4 M | 6 M |
| Logikblöcke (Slices) | 3072 | 5120 | 7680 | 10752 | 14336 | 23040 | 33792 |
| Block-Select-Ram (kBits) | 576 | 720 | 864 | 1008 | 1728 | 2160 | 2592 |
| Muliplizierer (18 Bit) | 32 | 40 | 48 | 56 | 96 | 120 | 144 |
| I/O (max) | 264 | 432 | 528 | 624 | 720 | 912 | 1104 |

**Tabelle 1.1:** Ressourcen einiger FPGAs der Virtex-II Familie von Xilinx.

### 1.3.2 Verwendete FPGA-Prozessoren

Im Rahmen dieser Arbeit wurden der kommerziell erhältliche FPGA-Koprozessor *microEnable* sowie das am Lehrstuhl für Informatik V entwickelte MPRACE-Board eingesetzt. Beide lassen sich über den PCI-Bus des PCs konfigurieren und sind mit jeweils einem FPGA der Firma Xilinx bestückt und gehören somit zu den preisgünstigeren Varianten aus einer Vielzahl kommerziell erhältlicher FPGA-Koprozessoren.

*microEnable*

In Abb. 1.10 ist der schematische Aufbau des *microEnable*-Koprozessors [36] zu sehen.

Das *microEnable*-Board ist mit unterschiedlichen FPGAs aus der XC4000er-Familie von Xilinx bestückbar, wobei im Rahmen dieser Arbeit der XC4085-XLA FPGA eingesetzt wurde. Zusätzlich ist eine asynchrone SRAM[21]-Speicherbank zwischen 512 kByte und 2 MByte verfügbar mit einer Datenbreite von $36$ Bit, welche mit einer Taktfrequenz von bis zu 82 MHz betrieben werden kann. Über zwei CMC[22] Stecker, deren 64 Pins als Ein- oder

[19] Diese Funktionalität ist für Optimierungen beim parallelen Template-Matching sehr wichtig.
[20] Low Voltage Differential Signaling.
[21] Static Random Access Memory.
[22] Common Mezzanine Card, Steckerstandard nach IEEE P1396 Spezifikation.

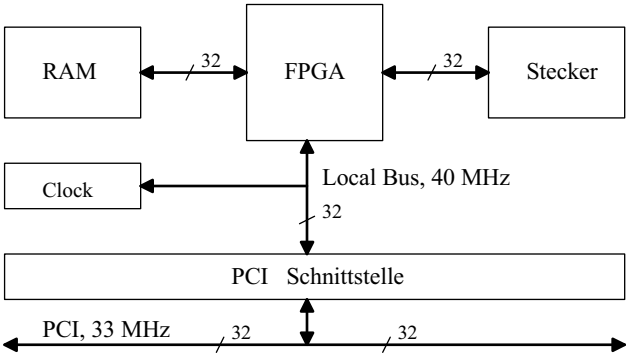



**Abbildung 1.10:** Aufbau des FPGA-Koprozessors *microEnable* der Firma Silicon Software.

Ausgänge konfigurierbar sind, können weitere Zusatzkarten angeschlossen werden, die z.B. weitere SRAM-Bänke oder einen passenden Anschluss für eine Digitalkamera enthalten. Die Anbindung an den PCI-Bus des PC erfolgt über den PLX9080 Chip. Auf dem lokalen Bus steht eine theoretische Bandbreite von $40$ Bit$\times 33$ MHz $= 165$ MByte/s, rechnerseitig eine Bandbreite von $32$ Bit $\times 33$ MHz $= 132$ MByte/s zur Verfügung. Gemessen wurden auf dem *microEnable* Board sehr gute Transferraten von bis zu $120$ MByte/s. Außerdem stehen eine *Local-Bus*-Clock, die den Takt mit Frequenzen von $10$ bis $40$ MHz für den lokalen Bus liefert, sowie zwei weitere programmierbare Taktgeneratoren mit Frequenzen von $1$ bis $120$ MHz zur Verfügung, mit denen das eigentliche Design gespeist wird. Dabei ist Clock 2 phasensynchron zu Clock 1 im wahlfreien Teilungsverhältnis 2 oder 4.

### MPRACE

Der MPRACE[23]-Koprozessor [37] (Abb. 1.11) ist eine 64-Bit PCI-Steckkarte für PCs. In Abb. 1.12 sind schematisch die wesentlichen Bausteine des MPRACE-Boards sowie die Bit-Breiten der Datenverbindungen eingezeichnet. Hauptbestandteil ist ein moderner, rekonfigurierbarer Virtex-II-FPGA vom Typ XC2V3000[24], dessen Eigenschaften in Abschnitt 1.3.1 beschrieben wurden.

Auf dem MPRACE-Board sind vier identische, 36 Bit breite, synchrone ZBT[25]-RAM Bänke mit jeweils einer Größe von 2 MByte und einer max. Taktrequenz von 133 MHz vorhanden. Eingesetzt wird die gepipelinte Version, die drei Zyklen sowohl für das Schreiben als auch das Lesen eines Datums benötigen. Im Gegensatz zum asynchronen SRAM haben Schreib- und Lesezugriffe nun dasselbe Timing. Als weiterer Datenspeicher steht ein Modul von SDRAM mit 64 Datenbits und max. 512 MByte zur Verfügung.

Insgesamt kann über vier Steckerverbindungen mit dem Virtex-II extern kommuniziert werden. In Abb. 1.12 sind die beiden Stecker A und B mit jeweils 96 Datenleitungen zu sehen, die den physikalischen Kontakt zu den Pins des FPGA herstellen. An die

[23] Multi-Purpose Reconfigurable Accelerator Computing Engine.
[24] Am Lehrstuhl sind außerdem Boards mit XC2V6000 vorhanden. Das Board kann mit Virtex-II-Typen von XC2V2000 bis XC2V8000 bestückt werden.
[25] Zero Bus Turnaround.

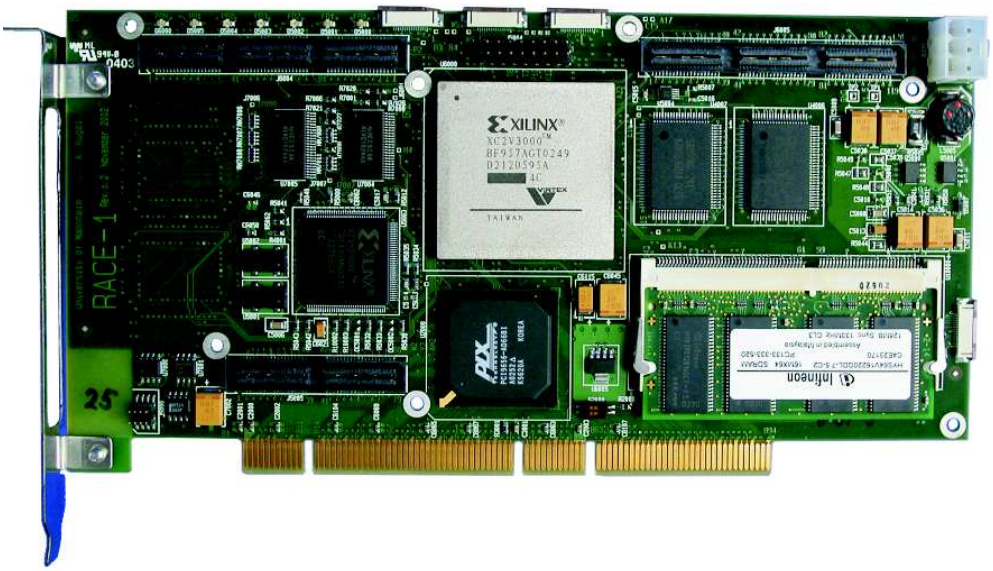



**Abbildung 1.11:** Der eingesetzte Koprozessor MPRACE.

Funktionalität der Zusatzkarten existieren keinerlei Einschränkungen, da das Kommunikationsmodell frei bleibt. Von A. Kugel [37] wurde die Zusatzkarte *RACE-1-MenableIo* entwickelt, die den Anschluss von Digitalkameras ermöglicht, die mit der *microEnable*-Zusatzkarte kompatibel sind. Auf beide Zusatzkarten wurde im Rahmen dieser Arbeit zurückgegriffen. Zusätzlich existieren zwei sehr schnelle, serielle Board-zu-Board Verbindungen.

Die Kommunikation mit dem PC übernimmt die PLX 9656 PCI-*Bridge*, die zur PCI Seite mit einer theoretischen Bandbreite von 64 Bit $\times$ 66 MHz = 528 MByte/s und zur FPGA Seite mit 32 Bit $\times$ 66 MHz = 264 MByte/s arbeitet. Gemessen wurde eine Bandbreite von ca. 250 MByte/s [37].

Das MPRACE-Board besitzt eine *Local-Bus*-Clock und eine System-Clock. Letztere speist den FPGA und sämtliche Speicher und Verbindungsstecker. Die *Local-Bus*-Clock kann über den PLD auf 16, 32 oder 64 MHz eingestellt werden. Die *Local-Bus*-Clock und System-Clock können synchron in einem Verhältnis 1:1 oder 1:2 laufen. Außderem kann die System-Clock über den *Frequency* Synthesizer des DCM des Virtex-II eingestellt werden. Die System-Clock kann dann unabhängig von der *Local-Bus*-Clock mit einer max. Frequenz von 125 MHz laufen.

### 1.3.3 Programmierung von FPGA-Prozessoren

Der Entwurf komplexer digitaler Schaltungen für FPGAs erfolgt üblicherweise mit den Hardware-Beschreibungssprachen VHDL[26] und Verilog. Diese HDLs[27] ermöglichen neben einer strukturellen auch eine verhaltensbasierte Beschreibung. Bei der strukturellen Be-

[26] VHSIC HDL.
[27] Hardware Description Languages.

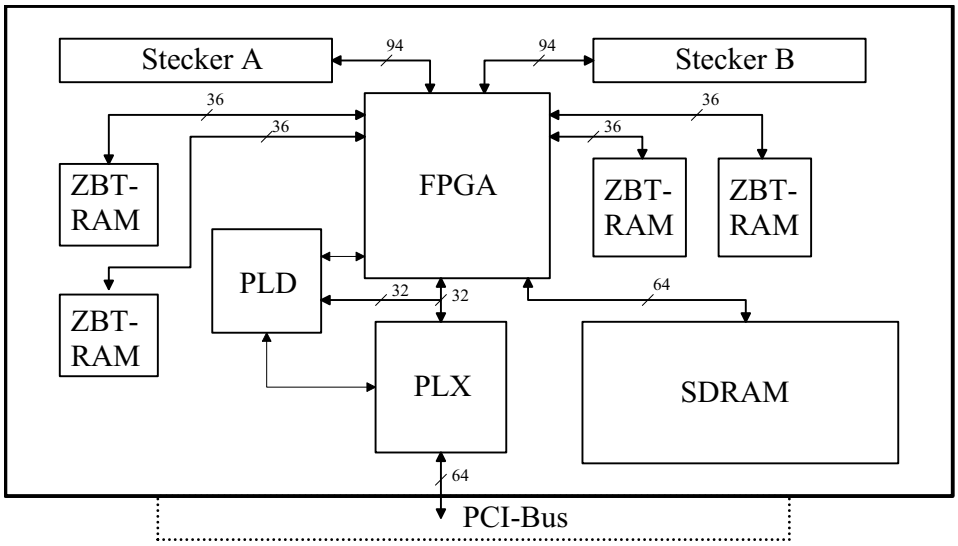



**Abbildung 1.12:** Aufbau und Verbindungen des MPRACE-Boards.

schreibung werden einzelne Module bzw. die gesamte digitale Schaltung aus grundlegenden Komponenten oder bestehenden Modulen zusammengeschaltet. Bei der verhaltensbasierten Beschreibung wird aus den Zuständen der Eingangssignale eines Moduls das Zeitverhalten der Ausgangssignale angegeben. Dabei sind sequentielle oder nebenläufige[28] Ausführungen der Anweisungen erlaubt. Diese grundlegenden Bausteine findet man meist in Standard-Bibliotheken zusammengefasst.

Der Ablauf der Designentwicklung ist in Abb. 1.13 beschrieben. Durch eine funktionale

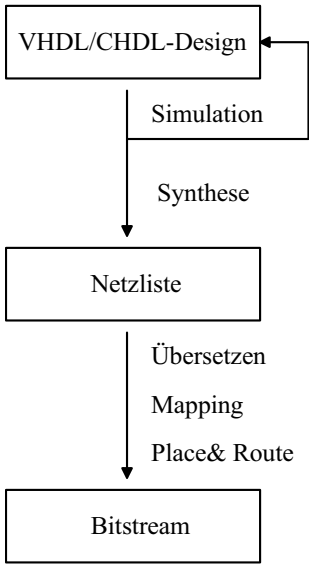



**Abbildung 1.13:** Ablauf des FPGA-Designentwurfs.

Simulation der digitalen Schaltung können Fehler im Design erkannt werden. Bei der

[28] engl. *concurrent*.

Synthese des Designs wird eine Netzliste generiert, wobei boolsche Gleichungen optimiert werden und eine Abbildung auf Register-Transfer-Ebene erfolgt.

Die folgenden Schritte erfolgen mit den Werkzeugen der FPGA-Hersteller. Zunächst wird die Syntax der Netzliste auf Ihre Korrektheit überprüft und die Elemente allgemein auf die CLBs und IOBs des FPGA abgebildet (*Mapping*). Anschließend erfolgt eine möglichst optimale Platzierung (*Place*)[29] auf die Logikblöcke des FPGAs mit dem Ziel die Logikblöcke optimal zu verbinden (*Route*)[30]. Als Ergebnis wird die Konfigurationsdatei, der *Bitstream* erzeugt. Die Konfiguration des FPGA erfolgt durch Laden des *Bitstreams*.

### Hardwarebeschreibungssprache CHDL

Für die in dieser Arbeit implementierten Algorithmen wurde die von K. Kornmesser [70] am Lehrstuhl für Informatik V entwickelte Hardwarebeschreibungssprache CHDL[31] verwendet. Diese erlaubt ausschließlich eine strukturelle Beschreibung der digitalen Schaltung in reinem C++ Programmcode. Die hierfür notwendigen grundlegenden Bauelemente sind in einer C++ Klassenbibliothek enthalten. Eine Auswahl vordefinierter C++ Klassen-Elemente für die 4000er und Virtex-II FPGAs sind:

- Flip-Flops und Register
- Logik-Operatoren wie AND, OR, XOR, ...
- allg. Funktionsgeneratoren
- Multiplexer
- Ladbare Zähler in Vorwärts- und Rückwärtsrichtung, Inkrementer, Dekrementer
- Komparatoren, Addierer, Subtrahierer
- FIFOs[32], synchron und asynchron
- asynchrones CLB-RAM (*distributed memory*).

Für die Virtex-II FPGAs stehen außerdem die folgenden Komponenten zur Verfügung

- SRLTs (Shift Register Lookup Tables)
- 18-kBit Block-RAM mit unterschiedlichen Datentiefen
- 18*18 Bit-Multiplizierer

[29] Der Entwickler kann dabei vorgeben, auf welche CLBs bzw. IOBs die jeweiligen Funktionen platziert werden sollen. Hiervon wird aber fast nie gebrauch gemacht, weil dadurch ein optimales *Mapping* meist eingeschränkt wird.

[30] Der Entwickler hat keinen direkten Einfluss auf das Verschalten der Switchelemente des Verbindungsnetzwerks. Er nimmt nur indirekt Einfluss, indem er Beschränkungen für max. Signallaufzeiten z.B. zwischen Registern festlegt.

[31] C++ based Hardware Description Language.

[32] First-In-First-Out.

- DCM (Digital Clock Manager).

Für die externe Kommunikation sind für beide FPGAs u.A. folgende Bauteile vorhanden:

- PCI_SLAVE für *microEnable* (Interface für Datenaustausch über den 9080 PLX-Chip mit dem PCI-Bus des Host-Rechners)
- PCI_SLAVE für MPRACE (Interface für Datenaustausch über den 9656 PLX-Chip mit dem PCI-Bus des Host-Rechners)
- S2RAM für *microEnable* (0/2 *Waitstates* beim Schreiben/Lesen)
- ZBTRAM für MPRACE (3/3 *Waitstates* beim Schreiben/Lesen).

Die Beschreibung der Verbindungen der einzelnen Bausteine erfolgt mit dem Zuweisungsoperator, der hierfür überladen wurde. Die Eigenschaft der Hierarchisierung von C++, und damit von CHDL, erlaubt es neue Klassen aus einer Vielzahl von bestehenden Klassen zu definieren. Somit können modulare, hierarchische Designs erstellt werden, die auch im komplexen Fall überschaubar bleiben. Außerdem ist das Einbinden von leistungsfähigen *Core Tools* wie z.B. Multiplizierer oder Dividierer, die vom FPGA-Hersteller angeboten werden, möglich. Dem Anwender bleibt es überlassen ein passendes Simulationsmodell zu schreiben.

Methoden zur Simulation der aus den Modulen zusammengesetzten Schaltung werden von CHDL zur Verfügung gestellt. Ebenfalls ist die Simulation der mit dem FPGA kommunizierenden Komponenten wie PCI-Slave, externes RAM in Abhängigkeit des Steuerprogramms auf dem Host-Rechner möglich. Bei der Instanziierung der C++ Objekte wird eine globale Netzliste erzeugt und in einer Datei im `.xnf` oder `.edif` Format gespeichert. Aus dieser wird wiederum mit den Werkzeugen des FPGA Herstellers das *Bistream*-File generiert, siehe hierzu Abb. 1.13.

### 1.3.4 Hardware-Beschleunigung

In diesem Abschnitt werden die beiden Strategien Parallelisierung und Pipelining zur Beschleunigung von Algorithmen auf Hardware beschrieben [40].

#### Parallelisierung

Um eine Verarbeitungssteigerung der Daten zu erzielen, wich man sehr früh von der von-Neumann-Rechner-Architektur ab.

Prinzipielle Möglichkeiten der Beschleunigung liegen sowohl in der Parallelisierung der Daten als auch der Verarbeitungseinheiten (VE). Die VEs lassen sich auf FPGAs gut realisieren, vorausgesetzt, die digitale Schaltung passt in das FPGA.

Der “Flaschenhals” bei parallel arbeitenden Verarbeitungseinheiten liegt oft darin, die Daten den VEs zuführen und deren Ergebnisse zu speichern, besonders dann, wenn sämtliche Daten im externen RAM des Koprozessors liegen, deren Zugriff nicht systematisch

erfolgt und nur schwer im FPGA zwischengespeichert werden können. Die max. Anzahl der Verarbeitungseinheiten ist von der Größe des FPGAs und vom Datentyp abhängig. Es können beispielsweise wesentlich mehr N-Bit Integer-Additionen als N-Bit Gleitkomma-Multiplikationen ausgeführt werden.

Diese beiden Probleme wurden durch die neu hinzugefügten Elemente beim Virtex-II FPGA entschärft, siehe Abschn. 1.3.2. Durch die Einführung des BlockRAM auf dem Virtex-II FPGA können wesentlich mehr Daten im FPGA zwischengespeichert werden. Eine ressourcensparende Realisierung von Gleitkommaarithmetik erlauben die $18 \times 18$ Bit-Multiplizierer des Virtex-II.

#### Pipelining

Eine weitere Strategie zur Verarbeitungssteigerung der Daten ist das Pipelining, bei welchem der Prozessor die Daten in mehreren Stufen verarbeitet. Hierzu wird die Aufgabe in mehrere Teilaufgaben zerlegt, die schnell und effizient gelöst werden können. Lange bevor ein Datum abgearbeitet ist, d.h. durch die letzte Stufe der Pipeline hindurch ist, kann die Pipeline bereits mit der Bearbeitung des nächsten Datums beginnen. Im Idealfall lässt sich so in jedem Takt ein neues Datenwort nachschieben, bei entsprechender Parallelisierung auch mehrere. Die Idee des Pipelinings ist es, jedes Modul möglichst gleichmäßig und ohne Unterbrechungen arbeiten zu lassen.

Die eigentliche Durchlaufzeit (Latenzzeit) kann dabei mehrere Takte benötigen, während der Performance-relevante Datendurchsatz meistens bei einem Ergebnis pro Takt liegt.

### 1.3.5 FPGA-basiertes Bildverarbeitungssystem

Im diesem Abschnitt werden die in dieser Arbeit verwendeten hybriden Bildverarbeitungssysteme, bestehend aus FPGA-Koprozessoren, Host-Rechner und Digitalkamera vorgestellt, siehe Abb. 1.14.

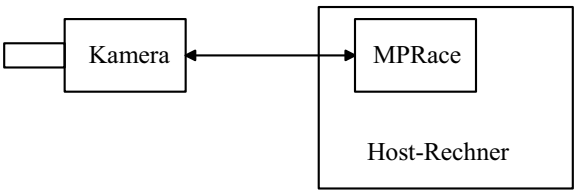



**Abbildung 1.14:** Schematischer Überblick über das verwendete Bildverarbeitungssystem mit MPRACE.

Zu Beginn dieser Arbeit wurde ein Bildverarbeitungssystem bestehend aus Digitalkamera, Host-Rechner und drei *microEnable*-Boards mit XC4085-XLA-FPGA, siehe Abschn. 1.3.2, eingesetzt. Das erste *microEnable*-Board diente ausschließlich zur Bildaufnahme, das zweite zur Berechnung der DT-Bilder und das dritte zur Ausführung des Template-Matching-Algorithmus. Host-Rechner war ein Dual-Pentium-III 500 MHz Rechner mit 896 MByte

Arbeitsspeicher. Ab Dezember 2001 wurden die drei *microEnable*-Boards durch ein MPRACE-Board mit Virtex-II XC2V3000 ersetzt. Das *Framegrabbing* und Template-Matching mit einer wesentlich höheren Anzahl an Templates kann nun auf *einem* FPGA-basierten Koprozessor durchgeführt werden. Hier kam ein Dual-Pentium-III 833 MHz Rechner mit 1 GByte Arbeitsspeicher zum Einsatz.

Die Aufgaben des Host-Rechners sind die Konfiguration und der Datenaustausch mit den FPGA-Koprozessoren, Ausführung des RBF-Klassifikators, sowie das Darstellen der Kamerabilder am Monitor. Zur Bildaufnahme stand eine digitale Pulnix-Kamera TM-1040 zur Verfügung, die im nächsten Abschnitt vorgestellt wird. Die Implementierungsdetails und die genaue Aufgabenverteilung für den in dieser Arbeit verwendeten Algorithmus zur Verkehrszeichenerkennung auf dem Bildverarbeitungssystem mit MPRACE-Koprozessor sind in den Kapiteln 3 und 4 zu finden.

### Verwendete Kamera

Die Kamera TM-1040 von Pulnix [39], siehe Abb. 1.15, liefert Grauwertbilder mit 10 Bit Auflösung pro Bildpunkt, die im *progressive scan*[33] Verfahren aufgenommen werden. Die Aufnahme gerader und ungerader Zeilen über den CCD[34]-Chip erfolgt gleichzeitig


**Abbildung 1.15:** Digitale Kamera TM-1040 von Pulnix.

(*non-interlaced*). Das $1024 \times 1024$ Pixel große Bild wird zeilenweise von links oben nach rechts unten über ein RS-422 Signal digital übertragen. Die Bildwiederholrate von 30 fps[35] bei einer Clock von 40 MHz kann nicht verändert werden. Der Aufnahmevorgang kann jedoch über das asynchrone Reset der TM-1040 jederzeit neu gestartet werden[36].

Die Pulnix-Kamera wird jeweils über zwei Subboards, siehe Abb. 1.16, an die FPGA-Boards angeschlossen. Die Kameraschnittstelle FG1CAM-RS422 von der Firma *Silicon Software* [38] wandelt die differentiellen, digitalen RS422 Signale in Standard-TTL-Signale um. Diese werden über ein 40-poliges Flachbandkabel an die Aufsteckkarten der Koprozessoren geführt. Für das *microEnable* Board wurde das CMC-Aufsteckmodul EXT-IDE [38] verwendet. Von [37] wurde die Aufsteckkarte *RACE-1-MenableIo* entwickelt,

[33] Immer ein ganzes Bild wird über die Schnittstelle übertragen.
[34] Charged Coupled Devices.
[35] Frames per second.
[36] Diese Funktionalität wird im Rahmen dieser Arbeit nicht unterstützt.

die einerseits zur Kameraschnittstelle FG1CAM-RS422 und andererseits zu den Steckern A und B des MPRACE-Boards kompatibel ist. Die FPGA-Implementierungsdetails der Ka-

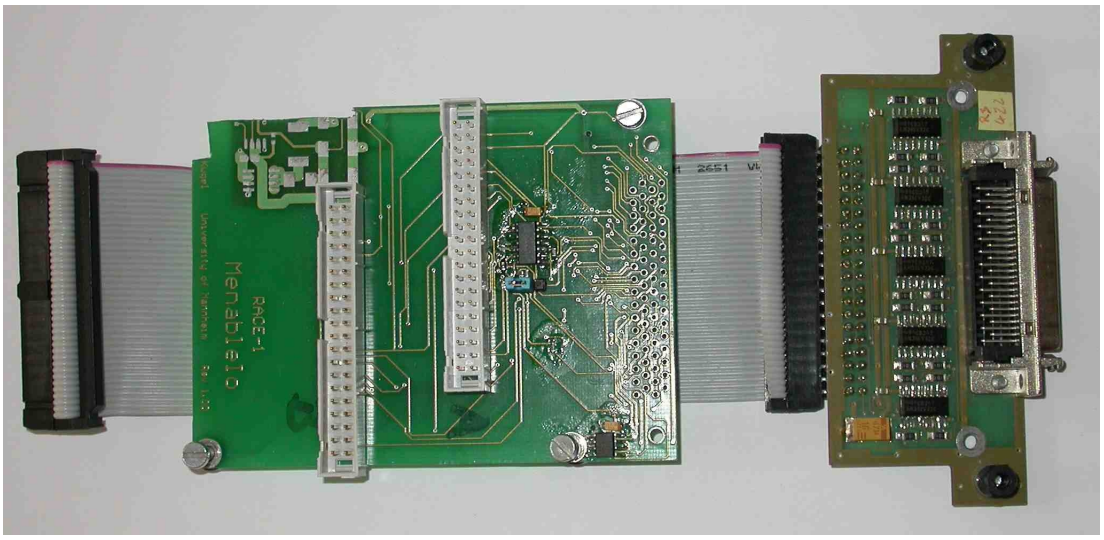



**Abbildung 1.16:** Mitte: MPRACE-Aufsteckkarte *RACE-1-MenableIo*, rechts: Kameraschnittstelle FG1CAM-RS422.

meraanbindung sind in Abschn. 3.2 zu finden.

**Kameras mit CMOS-Chip** Neben Digitalkameras mit CCD Chips, die den Vorteil einer hohen Bildqualität bei geringem Rauschen haben, werden immer häufiger solche mit CMOS[37] Chip eingesetzt. Wesentliche Vorteile sind wahlfreier Pixelzugriff, hohe Photoempfindlichkeit, hohe Helligkeitsdynamik (bis 150 db) und ein großer Temperaturbereich, die im tatsächlichen Einsatz im Fahrzeug von großem Vorteil sind. Qualitativ gute Bilder erhält man dann bei sich schnell ändernden Beleuchtungsverhältnissen wie bei einer Fahrt mit deinem Fahrzeug durch einen Tunnel oder ungünstigen Beleuchtungsverhältnissen wie nachts. Wesentlicher Nachteil ist ein hohes Rauschen verursacht durch einen Dunkelstrom, das um einen Faktor 5 bis 20 höher ist im Vergleich mit CCD Chips. Die Qualität der CMOS Kameras hat sich jedoch in den letzten Jahren stark verbessert, so dass der Einsatz dieser in der Praxis deutlich zunehmen wird.

[37] Complementary Metal-Oxide Semiconductor.

Kapitel 2

# Algorithmen für das Template-Matching

Grundlage dieser Arbeit ist ein Matching Algorithmus von D. Gavrila [1], basierend auf distanztransformierten orientierten Kantenbildern, der in diesem Kapitel ausführlich beschrieben wird. Auf mögliche Verbesserungen des Algorithmus unter Berücksichtigung des Rechenaufwands wird in den jeweiligen Abschnitten hingewiesen.

Die Templates, die beim Matching verwendet werden, sind Prototypen, die aus den abgeleiteten Merkmalen der Muster - in dieser Arbeit die Verkehrszeichen - bestehen. Als Merkmale der Verkehrszeichen finden deren Kanten und Orientierungen der äußeren Form Verwendung. Die Verkehrszeichen lassen sich in wenige Templateklassen mit den Formen Kreis, Dreieck, Rechteck, Sechseck und Raute einteilen. In dieser Arbeit werden ausschließlich kreisförmige und dreieckige Templates eingesetzt, die in Abschn. 2.2.1 dargestellt sind.

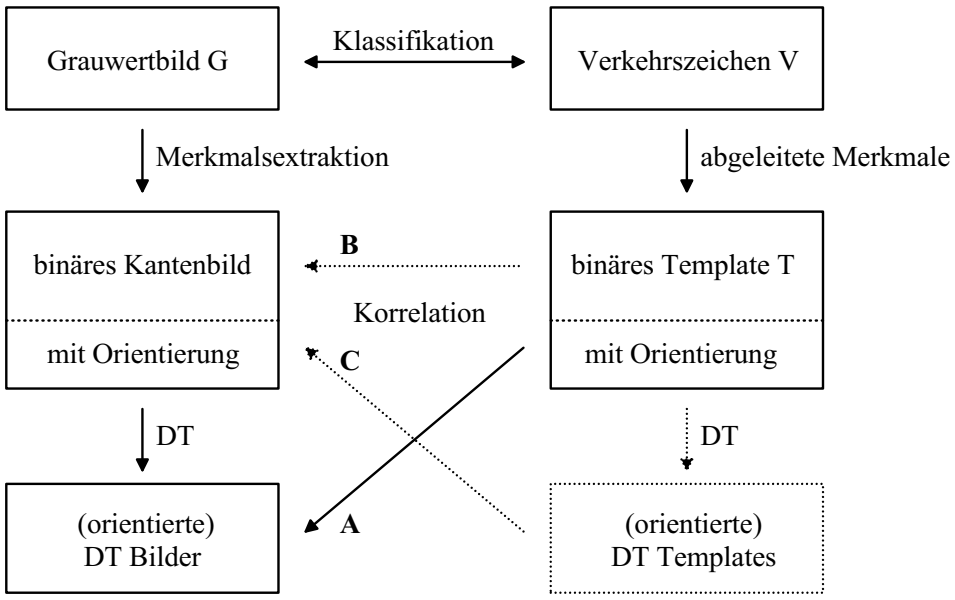



**Abbildung 2.1:** Template-Matching mit distanztransformierten (orientierten) Bildern.

Die Berechnung eines Ähnlichkeitsmaßes für das Template-Matching (Abschn. 1.1.3)

kann beispielsweise wie in Abb. 2.1 skizziert erfolgen über:

- A (Durchgehender Pfeil): In dieser Arbeit wird die Korrelation zwischen den Templates, die Orientierungsinformation enthalten, und den aus den Originalbildern erzeugten distanztransformierten orientierten Kantenbildern berechnet.
- B (Gestrichelter Pfeil): Ein weiterer möglicher Ansatz ist die Korrelation der binären Templates direkt auf den binären Kantenbildern. Dieser Ansatz führt jedoch zu vielen schwer zuzuordnenden und "falschen" Matching-Ergebnissen.
- C (Gestrichelter Pfeil): Des Weiteren besteht die Möglichkeit, das Matching mit distanztransformierten Templates auf binären Kantenbildern mit Richtungsinformation durchzuführen. Dies führt zu identischen Ergebnissen wie bei A bei deutlich höheren Rechenzeiten, weil die DT-Templates i.A. wesentlich mehr Templateelemente als die binären Templates enthalten.

Nach Berechnung der Ähnlichkeitsmaße werden die Ergebnisse des Template-Matchings an den RBF-Klassifikator übergeben, der auf dem ursprünglichen Grauwertbild $G$ arbeitet, siehe Abschn. 1.1.3. Insgesamt ist der Matching-Ansatz sehr rechenzeitaufwändig und prädestiniert für parallele Hardwarearchitekturen, weil er gut parallelisierbar ist.

Im folgenden Abschn. 2.1 wird beschrieben, wie die distanztransformierten orientierten Kantenbilder generiert werden. Dabei werden Erweiterungen bzw. Verbesserungen des Algorithmus diskutiert. In Abschn. 2.2 wird das Template-Matching zunächst ohne, später mit Orientierungsinformation beschrieben. Die Ergebnisse der jeweiligen Operationen auf ein Eingangsbild aus Abb. 2.14 sind in den Abb. 2.16 bis 2.19 illustriert. Diese befinden sich am Ende dieses Abschnitts.

## 2.1 Distanztransformierte orientierte Kantenbilder

### 2.1.1 Binäres Kantenbild

Wie bereits in Abschn. 1.1.2 erwähnt, sind Nachbarschaftsoperationen auf Bildern elementare Methoden zur Extraktion von Merkmalen (*Erkennen*) [6]. Eine sehr bedeutsame Klasse solcher Operationen sind Faltungen mit Filterkernen, mit denen näherungsweise Ableitungen verschiedener Ordnungen oder Mittelungen realisiert werden. Die diskrete Faltungsoperation auf dem Bild $G$ ist gegeben durch

$$G'_{hw} = \sum_{h'=-r}^{r} \sum_{w'=-r}^{r} H_{h'w'} G_{h-h',w-w'}, \tag{2.1}$$

wobei $H$ eine Filtermaske mit ungerader Größe $(2r+1)\times(2r+1)$ ist, siehe Abb. 2.2. Filter mit endlichem Faltungskern werden auch als FIR[1]-Filter bezeichnet. Ein wichtiger Aspekt ist das Design von Filtern mit dem Ziel einer hohen Rechengenauigkeit bei möglichst niedrigem Rechenaufwand.

[1] Finite Impulse Response.

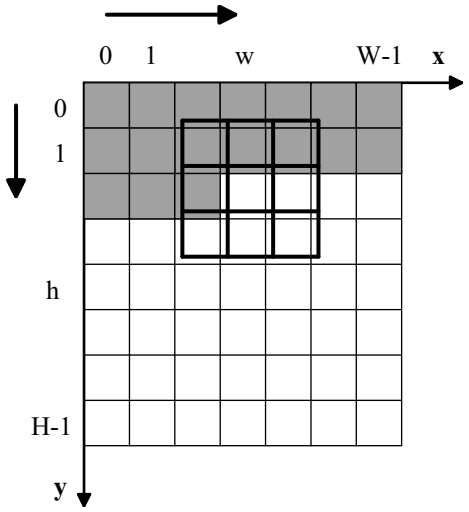



**Abbildung 2.2:** Diskrete Faltung eines Bildes durch zeilenweises Verschieben der Faltungsmaske (nach [6]).

### Glättung

Die Mittelwertberechnung erfolgt oft mit dem Gauß- bzw. Binomialfilter [6]. Das Gauß-Filter ist eine diskrete Version der Dichtefunktion für die Normalverteilung (Gaußsche Glockenkurve), wobei diese als kontinuierliche Verteilung der diskreten Binomialverteilung angesehen werden kann. Die Binomialkoeffizienten $\binom{n}{k}$ ergeben sich bekanntlich aus dem Pascalschen Dreieck [51]. Im Rahmen dieser Arbeit wird auf ein explizites Glättungsfilter verzichtet. Der Sobel-Operator, der im nächsten Abschnitt beschrieben wird, enthält jedoch eine implizite Glättung senkrecht zur Ableitung.

### Ableitungen

Zur Berechnung von Grauwertänderungen wird der sog. Sobel-Filter, ein Ableitungsfilter 1. Ordung mit einem $3 \times 3$ Filterkern eingesetzt, dessen Koeffizienten für x- und y-Richtung in Gl. 2.2 angegeben sind

$$\mathbf{S}_x = \frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad \mathbf{S}_y = \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{2.2}$$

Der Sobel-Operator $\mathbf{S}_x$ in x-Richtung enthält einen einfachen Gradienten in x-Richtung und eine einfache Gauß-Glättung in y-Richtung. Umgekehrtes gilt für den Sobel-Operator $\mathbf{S}_y$ mit Ableitung in y-Richtung. Generell sind bei Ableitungsfiltern die Koeffizienten der Filtermaske so gewählt, dass deren Summe gleich Null ist, während die bei den Glättungsfiltern gleich eins ist.

**Mögliche Verbesserungen** Verwendet man einen Sobel-Operator mit optimierten Festkommakoeffizienten [42], so erhält man ein Differenzenbild mit niedrigerem Winkelfeh-

ler

$$\mathbf{S}_{\mathbf{opt}_x} = \frac{1}{32}\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}, \quad \mathbf{S}_{\mathbf{opt}_y} = \frac{1}{32}\begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}. \tag{2.3}$$

Eine weitere Minimierung des Winkelfehlers kann erzielt werden indem man noch günstigere ganzzahlige Koeffizienten wählt oder mit größeren Filterkernen, z.B. $5 \times 5$ oder $7 \times 7$ arbeitet [43].

Bei Echtzeitanwendungen werden meistens kleine Filtermasken eingesetzt um den Rechenbedarf so gering wie möglich zu halten. Eine genaue Untersuchung einer FPGA-Implementierung von FIR-Filtern bezüglich Rechenzeit und Ressourcenverbrauch erfolgt in Abschn. 3.3.

**Mögliche Alternativen** Häufig eingesetzt werden Filter die sich der lokalen Struktur des Bildes anpassen. Als Beispiele seien steuerbare Filter und anisotrope Diffusionsfilter genannt [43, 45]. Dabei bestimmen adaptive, datengetriebene Parameter die Stärke der Glättung bzw. Ableitung sowie deren Richtung. Auch die Anwendung mehrerer Filter und anschließender Interpolation der Ergebnisse mittels einer steuerbaren Interpolationsfunktion ist möglich. Vorteile dieser Filter sind, dass Rauschen oder kleinere Strukturen im Bild weggeglättet, Kanten jedoch erhalten und nicht zerstört werden.

Adaptive Filter sind jedoch wesentlich rechenintensiver als einfache Faltungsoperationen. Arbeiten, welche die Implementierung anisotroper Diffusionsfilter auf FPGAs untersuchen, sind dem Autor nicht bekannt und im Rahmen dieser Arbeit nicht erfolgt.

### Binarisierung

Der Schwellwertoperator, auch Binarisierung genannt, ist eine Bild-zu-Bild Transformation, die alle Pixel des Eingangsbildes, dessen Helligkeit unterhalb eines vorgegebenen Grauwertes $\theta$ liegen, auf den Wert 1 und die verbleibenden auf den Wert 0 abbildet [10]. Als Eingangsbild der Binarisierung dient typischerweise ein Gradientenbild oder ein Bild mit höheren Ableitungen. Im hier vorgestellten Algorithmus ist das Eingangsbild die Summe der Beträge der beiden Sobel-Bilder $|G_{\mathbf{S}_x}|+|G_{\mathbf{S}_y}| < \theta$ bzw. die euklidische Summe der Sobel-Bilder $(G^2_{\mathbf{S}_x} + G^2_{\mathbf{S}_y})^{\frac{1}{2}} < \theta$. Insgesamt erhält man ein binäres Kantenbild $I$, wobei der Schwellwert $\theta$ fest vorgegeben wird. Dieser könnte aber auch z.B. in Abhängigkeit der Helligkeitsverteilung für jedes Bild neu berechnet werden. Der Nachteil beim einfachen Schwellwertoperator besteht darin, dass häufig einzelne, isolierte Störpixel entstehen.

**Mögliche Verbesserungen** Oft wird anstelle des globalen Schwellwertoperators, der an lokale Strukturen des Bildes schlecht angepasst ist, auch ein doppelter Schwellwertoperator eingesetzt. Ein binarisiertes Bild mit höherem Schwellwert wird dann als Keim für die Rekonstruktion des binarisierten Bildes mit dem gewünschten niedrigeren Schwellwert eingesetzt. Das resultierende Binärbild enthält weniger Störstellen als das mit einfacher Binarisierung, was sich positiv auf eine anschließende Distanztransformation auswirkt. Ein doppelter Schwellwertoperator wird auch beim sog. Canny-Kantendetektor (Abschn. 1.1.2) zur Binarisierung eingesetzt.

*Cleaning*

Zur Reduzierung von Rauschen oder von Störpixel im Binärbild $I$ werden durch den sog. *Cleaning*-Operator bis zu max. drei zusammenhängende, isolierte Kantenpixel eliminiert. Als Nachbarn eines Pixels gelten seine direkten als auch diagonalen Pixel, also solche aus einer $8$er-Nachbarschaft. Der *Cleaning*-Operator wurde von [1] rekursiv implementiert. Für die Initialisierung der Distanztransformation ist zu beachten, dass hierbei die Kanten- und Hintergrundpixel invertiert werden.

Als mögliche Alternative kommt der in Abschn. 1.1.2 beschriebene morphologische Operator *Closing* in Frage, welcher ebenfalls in der Lage ist Störpixel zu eliminieren. Hierzu wurden in dieser Arbeit jedoch keine Untersuchungen durchgeführt.

Die Implementierungen der Kantenbilder für FPGAs sind in Absch. 3.3 zu finden.

### 2.1.2 Orientierung

Weitere Merkmale der Kanten sind deren lokale Richtungstypen. Man unterscheidet zwischen der Richtung, die über den gesamten Winkelbereich von $0° - 360°(2\pi)$ und der Orientierung, die über einen Winkelbereich von $0° - 180°(\pi)$ definiert ist [6]. Um $180°$ gedrehte Muster können meistens von einem nicht gedrehten Muster nicht unterschieden werden. Aufgrund dessen wird in der Bildverarbeitung sehr oft mit der Orientierung gearbeitet. Die Richtung des Gradienten liefert jedoch zusätzliche Information darüber, ob der Übergang einer Objektkante von hell nach dunkel oder von dunkel nach hell stattfindet, was in dieser Arbeit ausgenutzt wird.

Die näherungsweise Berechnung des Gradienten erfolgt durch die Pixel $(G_{\mathbf{S}_x}, G_{\mathbf{S}_y})$ der Sobel-Bilder, aus denen bereits die Kantenbilder berechnet wurden

$$\nabla g(x,y) = \begin{bmatrix} \frac{\partial g(x,y)}{\partial x} \\ \frac{\partial g(x,y)}{\partial y} \end{bmatrix} \approx \begin{bmatrix} G_{\mathbf{S}_x} \\ G_{\mathbf{S}_y} \end{bmatrix}. \tag{2.4}$$

Die lokale Richtung $\psi$ der Kantenpixel ist dann gegeben durch

$$\psi = arctan \frac{G_{\mathbf{S}_x}}{G_{\mathbf{S}_y}}. \tag{2.5}$$

Die Genauigkeit des Gradienten als auch dessen lokale Richtung kann durch Verwendung des optimierten Sobel-Operators $\mathbf{S}_{opt}$ aus Gl. 2.3 verbessert werden.

Die zusätzliche Richtungsinformation kann man zur Erkennung einer speziellen Klasse von Verkehrszeichen mit dunklem Rand und heller Innenfläche verwenden, wie sie links in Abb. 2.3 gegeben sind.

**Richtungsdiskretisierung** Um die Anzahl der Richtungen als zusätzliche Merkmale zu begrenzen, werden die Richtungen diskretisiert, d.h. einem ganzzahligen Index zugewiesen, der einen bestimmten Winkelbereich repräsentiert. Hierzu unterteilt man den Einheitskreis in $M$ gleich große Teile

$$\{ [ \frac{i}{M} 2\pi, \frac{i+1}{M} 2\pi ] \, | i = 0, ..., M-1 \}. \tag{2.6}$$

**Abbildung 2.3:** Beispiele von drei Verkehrszeichen mit dunklem Rand und heller Innenseite (mitte-links) und einem Verkehrszeichen ohne helle Innenseite (rechts).

Ein Kantenpixel mit lokaler Richtung $\psi$ wird nun folgendem Index

$$\lfloor \frac{\psi}{2\pi} M \rfloor \tag{2.7}$$

zugewiesen. Die Index-Zuweisung ist sowohl für die Kanten des binären Templates $T$ und für die Kanten des binären Bildes $I$ gleichermaßen durchzuführen. Im Rahmen dieser

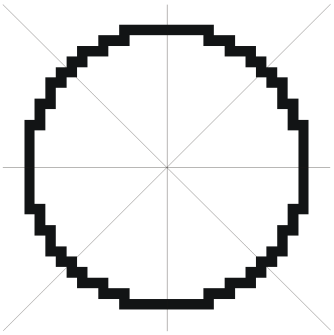


**Abbildung 2.4:** Unterteilung der Templateelemente eines binären Templates bzw. der Pixel eines binären Kantenbildes in acht Richtungsbereiche.

Arbeit wurde mit acht Richtungsbereichen ($M = 8$) gearbeitet, siehe Abb. 2.4 und [1].

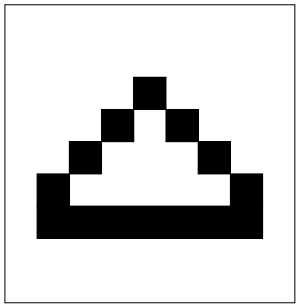
Aufgrund den Fehlern bei der Berechnung der lokalen Richtung, kann zugelassen werden, dass Pixel am Rand zweier benachbarter Richtungsbereiche beiden zugewiesen werden. Genaueres hierzu ist in Abschn. 2.2.3 zu finden. Die Berechnung des $arctan$ erfolgt in Software mittels *Lookup*-Tabelle.

**Mögliche Verbesserungen** Bessere Resultate lassen sich durch Benutzung des Strukturtensors [6] und dessen Eigenwerten erzielen. Diese Methoden sind wesentlich rechenzeitaufwändiger und daher nicht Gegenstand dieser Arbeit.

Die FPGA-Implementierungen hierzu sind ausführlich in Abschn. 3.4 beschrieben.

### 2.1.3 Distanztransformation

Die Distanztransformation (DT) überführt ein binäres Kantenbild $I$, in welchem die Kantenpixel mit 0 und die Hintergrundpixel mit 1 gegeben sind, in ein Bild, in welchem jeder Pixel $d_I$ den Abstand zum nächsten Kantenpixel enthält [49]. In Abb. 2.5 ist die euklidische Distanztransformation (EDT) veranschaulicht. Die euklidische Metrik $||.||_2$



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4.2 | 3.6 | 2.8 | 2.2 | 2.0 | 2.2 | 2.8 | 3.6 | 4.2 |
| 3.6 | 2.8 | 2.2 | 1.4 | 1.0 | 1.4 | 2.2 | 2.8 | 3.6 |
| 2.8 | 2.2 | 1.4 | 1.0 | 0.0 | 1.0 | 1.4 | 2.2 | 2.8 |
| 2.2 | 1.4 | 1.0 | 0.0 | 1.0 | 0.0 | 1.0 | 1.4 | 2.2 |
| 1.4 | 1.0 | 0.0 | 1.0 | 1.4 | 1.0 | 0.0 | 1.0 | 1.4 |
| 1.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 1.0 |
| 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| 1.4 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.4 |
| 2.2 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.2 |

**Abbildung 2.5:** Ein binäres Muster und dessen euklidische Distanztransformation (EDT) (nach [1]).

ist eine exakte Metrik im Kontext eines 2D-Bildes. In der Praxis werden oft Approximationen der euklidischen Metrik verwendet. Als einfache Approximation mit niedrigem Rechenzeitaufwand kommen die sog. $||.||_\infty$ oder die $||.||_1$-Metrik[2] in Frage [51]. Bessere Approximationseigenschaften haben die sog. *chamfer*[3]-Transformationen [49], die im Folgenden besprochen werden.

#### Berechnung der DT

Die Berechnung der globalen Distanztransformation erfolgt durch Maskenoperationen auf kleinen Nachbarschaften des Bildes. Die Idee besteht darin, dass sich lokale Distanzen, die durch die *chamfer*-$a$-$b$ Metrik auf kleinen Umgebungen vorgegeben sind, global ausbreiten. Für ein direktes Nachbarpixel sind der lokale Abstand $a$ und für ein diagonales Nachbarpixel der lokale Abstand $b$ als Approximation für die euklidischen Abstände $1$ und $\sqrt{2}$ zu benutzen, siehe Abb. 2.6. In der Praxis kommen häufig $3 \times 3$ Masken, wie sie in Abb. 2.6 gegeben sind, zum Einsatz. Sie dürfen nicht mit linearen Filtermasken verwechselt werden. Es wird zwischen paralleler und sequentieller Berechnung der DT unterschieden.

**Parallele DT** Bei der parallelen Berechnung der DT wird z.B. eine $3 \times 3$ Maske, wie in Abb. 2.6(a) gegeben, über jeden Pixel des Bildes geschoben. Zu jedem Pixel aus der Nachbarschaft des Zentralpixels $d^k_{h,w}$ werden für ein Bild der Iteration $k$ die Koeffizienten $c_{i,j}$ der Maske hinzuaddiert und $d^k_{h,w}$ wird durch das Minimum ersetzt

$$d^k_{h,w} = \min_{(i,j)\in Maske} (d^{k-1}_{h+i,w+j} + c_{i,j}). \tag{2.8}$$

[2]Die $||.||_\infty$-Metrik wird auch als Maximum- oder Schachbrett-Metrik, die $||.||_1$-Metrik als Stadtblock-Metrik bezeichnet.

[3]Engl. abrunden, abschrägen.

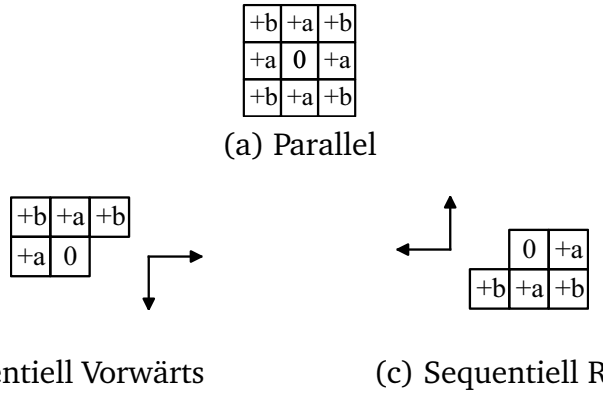



**Abbildung 2.6:** $3 \times 3$ Masken zu Berechnung der parallelen und sequentiellen Distanztransformation: (a) parallel (b) sequentiell Vorwärts (c) sequentiell Rückwärts.

Dieser Prozess wird für jedes Bild so lange wiederholt, bis sich kein Pixel im Bild mehr ändert. Die Anzahl der Iterationen ist datenabhängig und proportional zu dem größten Abstand eines nicht Kantenpixel zum nächsten Kantenpixel im Bild, jedoch begrenzt durch die max. Größe $\max(H, W)$ des Binärbildes $I$.

**Sequentielle DT** Bei der sequentiellen Methode wird die Maske, siehe Abb. 2.6(b), in Vorwärts-Richtung von der linken oberen Ecke des Bildes beginnend über das Bild geschoben und die Maske in Abb. 2.6(c) in Rückwärts-Richtung ausgehend von der rechten unteren Ecke des Bildes, siehe auch Abb. 2.7. Die Ergebnispixel, welche wiederum sofort

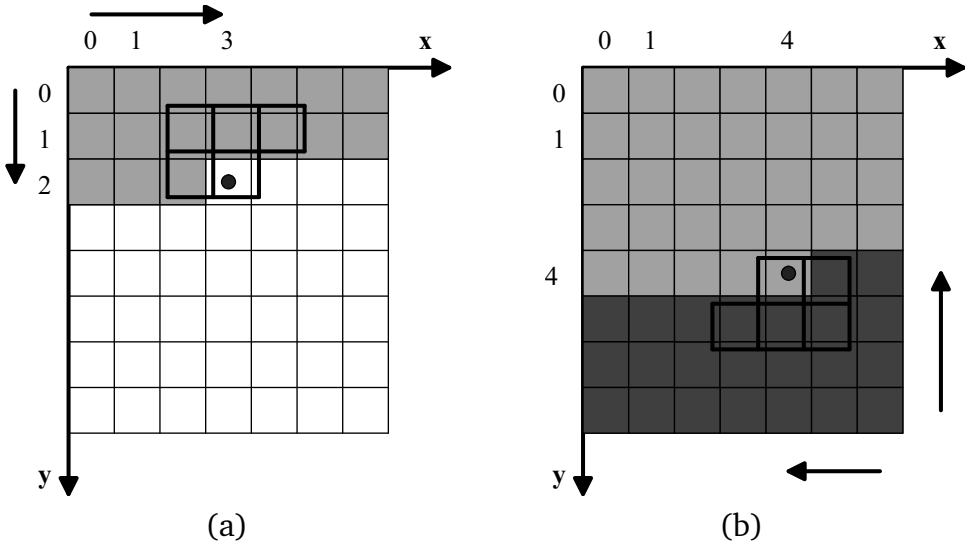



**Abbildung 2.7:** Distanztransformation eines Bildes durch zeilenweises Verschieben der DT-Masken in (a) Vorwärts-Richtung und (b) Rückwärts-Richtung.

in das Bild geschrieben werden, sind die Minima aus den Pixeln der lokalen Umgebung, zu denen die Koeffizienten der Maske hinzuaddiert werden. In den folgenden Gleichun-

gen ist dies für Vorwärts-Richtung

$$d_{h,w} = \min(\, d_{h,w},\ d_{h,w-1} + a,\ d_{h-1,w-1} + b,\ d_{h-1,w} + a,\ d_{h-1,w+1} + b\,) \tag{2.9}$$

und Rückwärts-Richtung

$$d_{h,w} = \min(\, d_{h,w},\ d_{h,w+1} + a,\ d_{h+1,w+1} + b,\ d_{h+1,w} + a,\ d_{h+1,w-1} + b\,) \tag{2.10}$$

veranschaulicht. Beide Methoden, die parallele und sequentielle, liefern dieselben Ergebnisse. Der Rechenaufwand für die sequentielle Methode ist datenunabhängig und i.A. deutlich geringer als für die parallele Methode.

### Approximationseigenschaften

Die Approximationseigenschaften der verschiedenen *chamfer*-$a$-$b$ Metriken zur Berechnung der DT hängen zum einen stark von den Koeffizienten $a$ und $b$ der Masken ab und werden zum anderen i.A. besser, je größer die Masken sind. Die max. Fehler der unterschiedlichen *chamfer*-Metriken für $3 \times 3$ Masken sind in Tabelle 2.1 gegeben. Die $||.||_1$ (Stadtblock) Metrik approximiert die EDT mit dem größten max. Fehler von $-58,8\%$, gefolgt von der am einfachsten zu berechnenden $||.||_\infty$ (Schachbrett) Metrik, bei der nur die orthogonalen Nachbarn relevant sind, mit einem max. Fehler von $+41.1\%$. Bessere Näherungen erhält man mit der *chamfer*-2-3 Metrik, mit einem max. Fehler von $13.4\%$ und der *chamfer*-3-4 Metrik mit einem max. Fehler von $8.1\%$. Letztere wird in [49] für $3 \times 3$ Masken empfohlen. Für höhere und optimale ganzzahlige Koeffizienten können die Fehler auf $7.3\%$ bzw. $4.5\%$ minimiert werden. Die max. Fehler sind entlang der Diagonalen des Bildes am größten. Für die direkten Nachbarn aus der 4er Nachbarschaft ergibt sich ein Fehler von Null[4] für alle in Tab. 2.1 angegebenen Metriken. Insgesamt führt dies zu einem mittleren Fehler, der kleiner als der max. Fehler ist. Zur approximativen Be-

| Metrik | a | b | max. Fehler in % |
|---|---|---|---|
| $\|\|.\|\|_2$ | 1 | $\sqrt{2}$ | 0 |
| *chamfer*-1-$\infty$ (Stadtblock, $\|\|.\|\|_1$) | 1 | $\infty$ | 58,8 |
| *chamfer*-1-1 (Schachbrett, $\|\|.\|\|_\infty$) | 1 | 1 | 41,4 |
| *chamfer*-2-3 | 2 | 3 | 13,4 |
| *chamfer*-3-4 | 3 | 4 | 8,1 |
| *chamfer*-8-11 | 8 | 11 | 7,3 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| *chamfer*-*opta*-*optb* | *opta* | *optb* | 4,5 |

**Tabelle 2.1:** Approximationseigenschaften für unterschiedliche *chamfer*-$a$-$b$ Metriken mit $3 \times 3$ Masken.

rechnung der Distanztransformation mit $5 \times 5$ Masken empfiehlt [49] die *chamfer*-5-7-11 Metrik mit einem max. Fehler von $2,02\%$ zu benutzen. Der max. Fehler bei optimalen ganzzahligen Koeffizienten beträgt hier $1,96\%$ und bei einer $7 \times 7$ Umgebung $0,91\%$. Eine $5 \times 5$ oder $7 \times 7$ Nachbarschaft wird daher nur bei sehr hohen Anforderungen an die Genauigkeit zum Einsatz kommen oder bei rechenzeitunkritischen Anwendungen.

[4] Im Rahmen der Rechengenauigkeit.

**Wertebereich**

Für den Ressourcenbedarf späterer Hardware-Implementierungen ist neben der Größe der Maske von entscheidender Bedeutung, in welchen Wertebereich das Binärbild durch die DT abgebildet wird. Der max. Wertebereich der DT-Pixel hat zum einen Auswirkungen auf die Genauigkeit, mit der die DT auf dem FPGA zu berechnen ist und zum anderen auf das Template-Matching, bei dem für die Berechnung eines Ähnlichkeitsmaßes die DT-Pixel aufzusummieren sind. Die Distanzwerte $d_I$, die sich für einen einzigen Kantenpixel in der linken oberen Ecke eines $8 \times 8$ Binärbildes für unterschiedliche DT-Metriken ergeben, sind in Tab. 2.2 dargestellt.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |
| 3 | 3 | 3 | 3 | 4 | 5 | 6 | 7 |
| 4 | 4 | 4 | 4 | 4 | 5 | 6 | 7 |
| 5 | 5 | 5 | 5 | 5 | 5 | 6 | 7 |
| 6 | 6 | 6 | 6 | 6 | 6 | 6 | 7 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

(a) *chamfer*-1-1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 2 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 4 | 5 | 6 | 8 | 10 | 12 | 14 | 16 |
| 6 | 7 | 8 | 9 | 11 | 13 | 15 | 17 |
| 8 | 9 | 10 | 11 | 12 | 14 | 16 | 18 |
| 10 | 11 | 12 | 13 | 14 | 15 | 17 | 19 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 20 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |

(b) *chamfer*-2-3

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 3 | 6 | 9 | 12 | 15 | 18 | 21 |
| 3 | 4 | 7 | 10 | 13 | 16 | 19 | 22 |
| 6 | 7 | 8 | 11 | 14 | 17 | 20 | 23 |
| 9 | 10 | 11 | 12 | 15 | 18 | 21 | 24 |
| 12 | 13 | 14 | 15 | 16 | 19 | 22 | 25 |
| 15 | 16 | 16 | 17 | 18 | 20 | 23 | 26 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 | 27 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |

(c) *chamfer*-3-4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
| 8 | 11 | 19 | 27 | 35 | 43 | 51 | 59 |
| 16 | 19 | 22 | 30 | 38 | 46 | 54 | 62 |
| 24 | 27 | 30 | 33 | 41 | 49 | 57 | 65 |
| 32 | 35 | 38 | 41 | 44 | 52 | 60 | 68 |
| 40 | 43 | 46 | 49 | 52 | 55 | 63 | 71 |
| 48 | 51 | 54 | 57 | 60 | 63 | 66 | 74 |
| 56 | 59 | 62 | 65 | 68 | 71 | 74 | 77 |

(d) *chamfer*-8-11

**Tabelle 2.2:** Berechnung der DT am Beispiel eines $8 \times 8$ Bildes mit verschiedenen $3 \times 3$ DT-Metriken. (a) *chamfer*-1-1 (Schachbrett), (b) *chamfer*-2-3, (c) *chamfer*-3-4, (d) *chamfer*-8-11 Metrik.

Dabei sind große Abweichungen der max. Wertebereiche der DT-Bilder in Abhängigkeit der verwendeten DT-Metrik festzustellen. Der max. Wertebereich ist abhängig von der der Größe $\max(H, W)$ des Eingangsbildes und dem größeren Koeffizienten $b$ der Metrik (max $d_I \approx \max(H, W) * b$).

### Auswahl der DT-Metrik

Im Hinblick auf den späteren FPGA-Ressourcenbedarf ist eine geeignete DT-Metrik auszuwählen, die zu robusten Ergebnissen beim Template-Matching und einer ressourcengünstigen FPGA-Implementierung führt.

In [48] wird für das Template-Matching die *chamfer*-3-4 Metrik zur Berechnung der DT empfohlen. Für die *chamfer*-2-3 Metrik wird von ähnlich guten Ergebnissen für das Matching berichtet. Allerdings führen mit der *chamfer*-1-$\infty$ (Stadtblock) Metrik erzeugte DT-Bilder zu deutlich mehr Fehlzuweisungen beim Template-Matching, weshalb von derer Benutzung abgeraten wird.

Im Rahmen dieser Arbeit wird die *chamfer*-2-3 bzw. *chamfer*-1-1 Metrik (Schachbrett-Metrik) eingesetzt. Dies wird zum einen damit begründet, dass das Template-Matching nicht zur Klassifikation der Verkehrszeichen, sondern ausschließlich zu deren Detektion (Bestimmung eines ROIs) eingesetzt wird. Die letztendliche Klassifikation übernimmt der in Abschn. 1.1.3 beschriebene RBF-Klassifikator. Das Matching ist derart durchzuführen, d.h. der Schwellwert beim Matching ist derart einzustellen, dass mit einer DT-Metrik mit schlechteren Approximationseigenschaften nach wie vor alle Muster im Bild detektiert werden, weil nur diese anschließend vom Klassifikator in eine der Template-Klassen zugewiesen oder zurückgewiesen werden können. Der Nebeneffekt wird dann aber sein, dass mehr Muster im Bild detektiert (mehr ROIs bestimmt) werden und sich somit die Rechenzeit für die Klassifikation entsprechend erhöht. Verwendet man anstelle der *chamfer*-3-4 die *chamfer*-2-3 Metrik, so erhöht sich die Anzahl der gefundenen Muster im Bild eventl. geringfügig [48]. In dieser Arbeit wird als weiteres Merkmal die Richtungsinformation der Kanten benutzt. Daher ist es gerechtfertigt mit der *chamfer*-2-3 bzw. *chamfer*-1-1 Metrik als Approximation der Euklidischen DT zu arbeiten.

Wird anstelle der *chamfer*-2-3 Metrik die *chamfer*-1-1 Metrik, mit einem ungefähr dreimal so großen max. Approximationsfehler eingesetzt, so wird sich die Anzahl der im Bild gefundenen Muster merklich erhöhen. Empirische Untersuchungen[5] ergaben hierfür ca. 20 % mehr Matching-Ergebnisse (bei entsprechender Anpassung des Schwellwerts). Die Zunahme der Anzahl der Matching-Ergebnisse, die stark abhängig vom Bildinhalt ist, bleibt jedoch in einem akzeptierbaren Rahmen, weil als zusätzliche Merkmale die Orientierungen von Kanten verwendet werden. Genauere Untersuchungen hierzu sind durchzuführen.

### Begrenzung des Wertebereichs

Zur Reduzierung des FPGA-Ressourcenbedarfs wird zusätzlich der max. Wertebereich der DT-Pixel auf einen festen Wert abgeschnitten (*clipping*). Zum einen lässt sich die Berechnung der DT-Pixel auf dem FPGA dann mit einer niedrigeren Genauigkeit durchführen und zum anderen können diese beim Matching mit ressourcengünstigeren Addierern aufsummiert werden. Das Abschneiden der DT-Pixel auf einen festen Wert führt jedoch zu einer höheren Anzahl der Ergebnisse beim Matching, die jedoch vom RBF-Klassifikator als Fehlzuweisungen interpretiert und daher eliminiert werden. Als günstig erweist sich,

[5] Im FPGA-Labor durchgeführt.

dass alle Muster, die mit den nicht abgeschnittenen DT-Pixel erkannt, auch mit den abgeschnittenen DT-Pixel erkannt werden, weil diese in der Nähe eines Musters mit den nicht abgeschnittenen DT-Pixel identisch sind. Des Weiteren wird durch das *Clipping*[6] der Pixel verhindert, dass einzelne, z.B. durch Überdeckungen fehlende Pixel des Musters nicht zu stark gewichtet werden und somit unvollständige Muster im Bild besser gefunden werden.

Für Bilder der Größe $512 \times 512$, wie sie in dieser Arbeit eingesetzt werden, kann mit der *chamfer*-2-3 Metrik im ungünstigsten Fall ein max. DT-Pixel mit einem Wert von ca. 1530 (11 Bit) auftreten. Bei der *chamfer*-2-3 Metrik wird der max. Wert der DT-Pixel auf vier Bit beschränkt mit einem Wertebereich von 0 bis 15 (hexadezimal 0xF). Für ein $8 \times 8$ großes Binärbild mit einem einzigen Kantenpixel in der oberen linken Ecke sind die auf vier Bit abgeschnittenen DT-Pixel $d_I$ in Tab. 2.3(a) dargestellt.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
| 2 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| 4 | 5 | 6 | 8 | 10 | 12 | 14 | 15 |
| 6 | 7 | 8 | 9 | 11 | 13 | 15 | 15 |
| 8 | 9 | 10 | 11 | 12 | 14 | 15 | 15 |
| 10 | 11 | 12 | 13 | 14 | 15 | 15 | 15 |
| 12 | 13 | 14 | 15 | 15 | 15 | 15 | 15 |
| 14 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |

(a) *chamfer*-2-3 (4 Bit)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| 1 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

(b) *chamfer*-1-1 (2 Bit)

**Tabelle 2.3:** Abschneiden der DT-Pixel für die *chamfer*-2-3 Metrik auf vier Bit (a) und für die *chamfer*-1-1 Metrik auf zwei Bit (b) am Beispiel eines $8 \times 8$ Bildes.

Für die abgeschnittenen DT-Pixel lässt sich deren Berechnung auf dem FPGA mit einer Genauigkeit von fünf Bit anstelle von 12 Bit durchführen und beim Matching sind vier Bit DT-Pixel anstelle von 11-Bit Pixel aufzusummieren. Die Anzahl der Ergebnisse beim Matching wird sich durch das Abschneiden leicht erhöhen. Genauere Untersuchungen hierzu sind nicht erfolgt. Im weiteren Verlauf dieser Arbeit sind für die FPGA-Implementiertung der DT (Abschn. 3.5) und des Template-Matchings (Kapitel 4 und 5) mit der *chamfer*-2-3 Metrik erzeugte DT-Bilder, die auf vier Bit abgeschnitten sind, als Referenz zu sehen.

Für DT-Pixel, die mit der *chamfer*-1-1 Metrik erzeugt sind, ist der max. Wert für Bilder der Größe $512 \times 512$ gleich $511$ (neun Bit). Zur Begrenzung des Wertebereichs für die *chamfer*-1-1 Metrik werden die DT-Pixel auf einen max. Wert von sieben (3 Bit) abgeschnitten. Dies entspricht einem Abschneiden der mit der *chamfer*-2-3 Metrik berechneten DT-Pixel auf vier Bit, man vergleiche hierzu Tab. 2.2(a) mit Tab. 2.3(a).

Eine Option zur weiteren Reduzierung des FPGA-Ressourcenbedarfs besteht darin, die mit der *chamfer*-1-1 Metrik berechneten DT-Pixel auf einen max. Wert von drei (zwei Bit) abzuschneiden, siehe Tab. 2.3(b). Die DT-Pixel auf dem FPGA können dann mit drei

[6] In der Software-Implementierung von D. Gavrila werden die mit der *chamfer*-2-3 Metrik erzeugten DT-Pixel auf einen max. Wert von 25 abgeschnitten.

Bit berechnet und beim Matching als zwei Bit Werte aufsummiert werden. Die Anzahl der Matching-Ergebnisse wird sich wahrscheinlich merklich erhöhen.

Genauere Untersuchungen bezüglich den Auswirkungen auf die Ergebnisse beim Template-Matching mit Verwendung der *chamfer*-1-1 Metrik für die DT und dem Abschneiden der DT-Pixel auf 3-Bit bzw. 2-Bit Werte sind durchzuführen.

Die Details der FPGA-Implementierungen der DT sind im nächsten Kapitel in Abschn. 3.5 dargestellt.

## 2.2 Template-Matching

Zunächst werden die in dieser Arbeit verwendeten binären Templates vorgestellt, die Prototypen der Verkehrszeichen sind. Anschließend wird in Abschn. 2.2.2 das Template-Matching auf einem DT-Bild ohne Richtungsinformation und in Abschn. 2.2.3 auf mehreren DT-Bildern beschrieben, die unterschiedlichen Richtungsbereichen zuzuordnen sind. In Abschn. 2.2.4 wird ein hierarchisches Template-Matching diskutiert.

### 2.2.1 Repräsentation der binären Templates

Die Repräsentation der 36 binären Templates $T_j, j = 0, \ldots, 35$, welche abgeleitete Merkmale der Verkehrszeichen unterschiedlicher Größen enthalten, ist in den Abb. 2.8 bis 2.10 gegeben. Die Templates wurden in die drei Klassen Kreise, Dreiecke nach oben und Dreiecke nach unten eingeteilt, mit Radien von 7-18 Pixel.

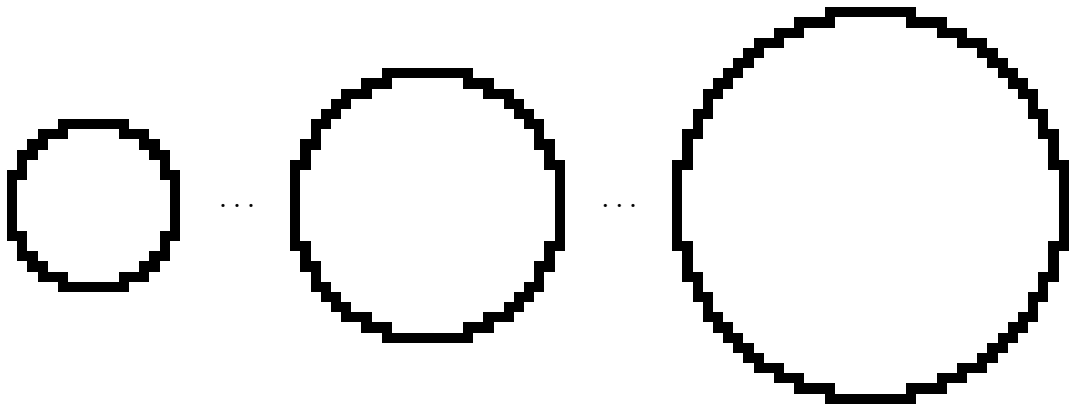


**Abbildung 2.8:** Repräsentation der kreisförmigen Templates mit Radien von 7-18 Pixel.

Ein Ähnlichkeitsmaß wird für alle Punkte des Bildes und für jede Template-Klasse mit den verschieden skalierten Templates berechnet. Die in der Umwelt (im Bild) vorkommenden Verkehrszeichen sind i.A. leicht gedreht und die Verzerrungen, die bei der Bildaufnahme entstehen, sind klein. Daher wird das Matching im Rahmen dieser Arbeit nicht mit rotierten bzw. verzerrten Templates durchgeführt. Die Anzahl der Templates für das Matching

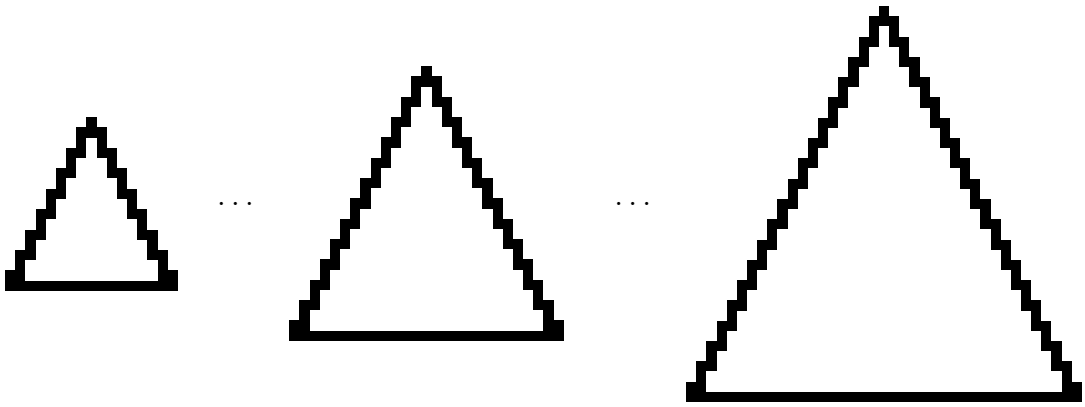


**Abbildung 2.9:** Repräsentation der Dreiecke mit Spitze nach oben mit Radien von 7-18 Pixel.

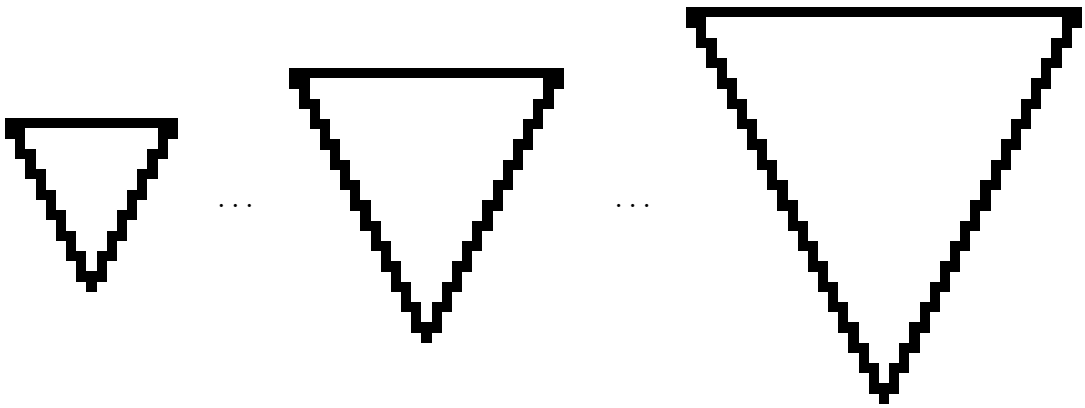


**Abbildung 2.10:** Repräsentation der Dreiecke mit Spitze nach unten mit Radien von 7-18 Pixel.

und damit der Rechenaufwand hätte sich ansonsten vervielfacht. Die Auswahl der von den Verkehrszeichen abgeleiteten Templates ist jedoch von entscheidender Bedeutung. Denn nur an den Punkten im Originalbild, für die ein positives Matching-Ergebniss festgestellt wird, kann anschließend der RBF-Klassifikator entscheiden, ob ein Verkehrszeichen vorhanden ist oder nicht, siehe Abschn. 1.1.3.

### 2.2.2 Template-Matching auf einem DT-Bild

Die Menge $T$ aller $N$ Templates $T_j$ sei mit $T = \{T_0, \ldots, T_{N-1}\}$ bezeichnet. Das $i$-te Templateelement $t_{i,j}$ von Template $T_j$, $j = 0, \ldots, N-1$ sei $t_{i,j} = (t_{i,j,x}, t_{i,j,y}) \in T_j$, $i = 0, \ldots, |T_j| - 1$, mit dessen x- bzw. y-Komponenten. Mit $|T_j|$ wird die Kardinalität der Menge $T_j$ bezeichnet, d.h. die Anzahl der Elemente von Template $T_j$.

Als Ähnlichkeitsmaß, welches an jedem Ort des Bildes für jedes Template $T_j$ zu berechnen

ist, wird der *chamfer*-Abstand benutzt

$$D_{chamfer}(T_j, I) = \frac{1}{|T_j|} \sum_{t_{i,j} \in T_j} d_I(t_{i,j}), \tag{2.11}$$

wobei $d_I(t_{i,j})$ die dem Template entsprechenden Pixel des DT-Bildes sind [1].

Das Distanzmaß $D_{chamfer}(T_j, I)$ ist eine Korrelation zwischen dem Template $T_j$ und dem distanztransformierten Bild des binären Kantenbildes $I$. Ein Template wird als "gematcht" an einem Bildpunkt betrachtet, wenn das Distanzmaß $D_{chamfer}(T_j, I)$ unterhalb einem vom Benutzer vorgegebenen Schwellwert $\Theta_j$ liegt

$$D_{chamfer}(T_j, I) < \Theta_j. \tag{2.12}$$

Die Schwellwerte $\Theta_j$ sind bezüglich der Anzahl der Templatepunkte $|T_j|$ und dem Parameter $a$ der verwendeten *chamfer-a-b* Metrik normiert

$$\Theta_j = \left\lceil \frac{c_\Theta \, |T_j|}{a} \right\rceil, \quad c_\Theta \approx 1. \tag{2.13}$$

Für $c_\Theta$ wird für alle Templates ein fester Wert zwischen $0.8$ und $1.2$ gewählt.

In Abb. 2.11 sind die einzelnen Schritte des Template-Matchings auf DT-Bildern anhand von Beispielbildern verdeutlicht. Die Abb. 2.11(a),(b) zeigt ein Grauwertbild mit künstlichen Objekten und ein hiervon abgeleitetes binäres Template. Das binäre Kantenbild des Grauwertbildes und das zugehörige DT-Bild sind in Abb. 2.11(c),(d) zu sehen. Der Vorteil, das Template-Matching mit einem DT-Bild anstelle mit einem Kantenbild durchzuführen, liegt darin, dass das sich ergebende Ähnlichkeitsmaß glatter ist und die Anzahl der Matching-Ergebnisse geringer ist. Das "Matchen" direkt auf dem Binärbild $I$ liefert bei idealen Mustern im Bild stark punktförmige Antworten die schnell abfallen, bei lückenbehafteten oder verrauschten Mustern jedoch verschmierte Antworten die schwer zuzuordnen sind [1].

### 2.2.3 Template-Matching auf orientierten DT-Bildern

Das Template-Matching erfolgt nun auf den $M$ DT-Bildern, die unterschiedlichen Richtungsbereichen $k$ zuzuordnen sind. Wie in Abschn. 2.1.2 besprochen, wird den Templateelementen $t_{i,j}$ von Template $T_j$ ein zusätzliches Merkmal, die Richtungszugehörigkeit $k$ zugewiesen. Das Template $T_j$ wird nun entsprechend in $M$ Teilmengen $T_j^k$ unterteilt mit $T_j = \{T_j^0, \ldots, T_j^{M-1}\}$, $k = 0, \ldots, M-1$, $T_j^k \subseteq T_j$. Das $i$-te Templateelement $t_{i,j}^k$ von Template $T_j$ aus dem Richtungsbereich $k$ ist gegeben durch $t_{i,j}^k = (t_{i,j,x}^k, t_{i,j,y}^k)$, mit den jeweiligen x- bzw. y-Komponenten. Nicht in jeder Teilmenge $T_j^k$ des Richtungsbereichs $k$ muss ein Templateelement $t_{i,j}^k$ enthalten sein, d.h. $T_j^k = \emptyset$ ist erlaubt. Dies ist z.B. bei den dreieckigen Templates der Fall. Aus der Vereinigung aller Teilmengen $T_j^k$ ergibt sich wieder das ursprüngliche Template $T_j$, d.h. $\cup_k T_j^k = T_j$. Am Beispiel eines kreisförmigen Templates ist in Abb. 2.4 die Unterteilung in acht Richtungsbereiche dargestellt. Für die in dieser Arbeit eingesetzten Templates sind in Abb. 2.12 und den folgenden Abbildungen

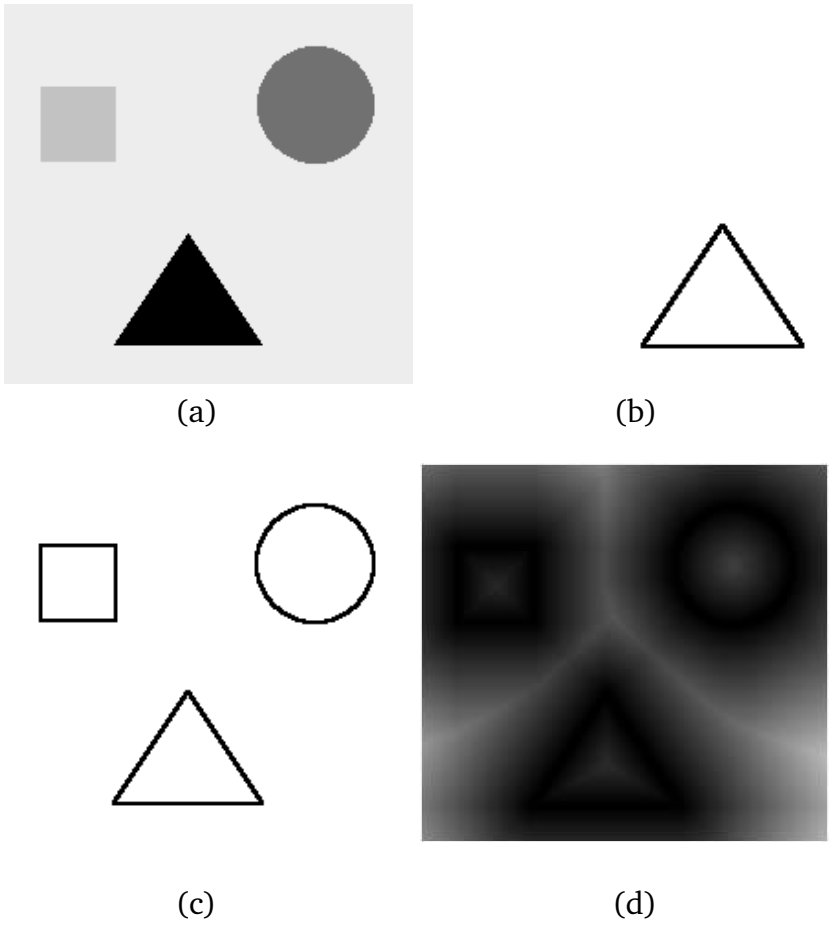



**Abbildung 2.11:** (a) Grauwertbild (b) binäres Template (c) binäres Kantenbild (d) DT-Bild (nach [1]).

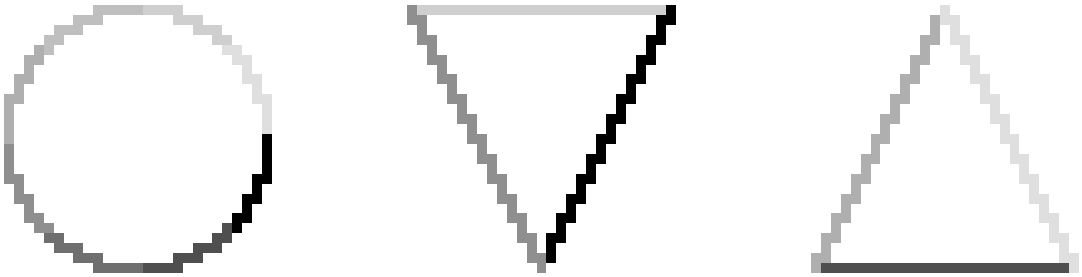


**Abbildung 2.12:** Zuordnung der Templateelemente von kreisförmigen und dreieckigen Templates in einen der $k$ diskreten Richtungsbereiche.

die Templateelemente aus dem Richtungsbereich 0 schwarz und aus dem Richtungsbereich sieben hellgrau illustriert.

Die Berechnung des Ähnlichkeitsmaßes $D_{chamfer}(T_j, I)$ läuft somit über eine zusätzliche Summation

$$D_{chamfer}(T_j, I) = \frac{1}{\sum_{k=0}^{M-1} |T_j^k|} \sum_{k=0}^{M-1} \sum_{t_{i,j}^k \in T_j^k} d_I^k(t_{i,j}^k), \tag{2.14}$$

wobei $|T_j^k|$ die Anzahl der Elemente des Templates $T_j$ aus dem Richtungsbereich $k$ sind und $d_I^k(t_{i,j}^k)$ die dem Template entsprechenden Pixel aus dem zugehörigen DT-Bild. Die Korrelation wird somit zwischen den Elementen des Templates und den DT-Pixeln, jeweils mit Richtungsindex $k$, berechnet.

In Rahmen dieser Arbeit wird jedem Pixel genau einen Richtungsbereich $k$ zugewiesen, d.h. es gilt $\sum_{k=0}^{M-1} |T^k| = |T|$ (disjunkte Zerlegung von $T$). Es kann aber auch erlaubt werden, dass sich die Randbereiche von zwei benachbarten Richtungsbereichen leicht überlappen können, d.h. dass die sich darin befindenden "kritischen" Pixel, die durch systematische Fehler oder Rundungsfehler entstehen, in zwei benachbarte Richtungsbereiche zugewiesen werden. Dann gilt $\sum_{k=0}^{M-1} |T^k| > |T|$ mit $\bigcup_{k=0}^{M-1} T^k = T$ (Überdeckung von $T$).

Die Implementierungsdetails für das Template-Matching für FPGAs sind in Kapitel 4 und dessen Optimierungen in Kapitel 5 zu finden.

### 2.2.4 Hierarchisches Template-Matching

Ein hierarchisches Template-Matching ist in [1] und [3] beschrieben. Die Template-Hierarchie besteht aus insgesamt drei Stufen von Templates, die in Abb. 2.13 angegeben sind. Für jede Klasse von Templates, z.B. für die zwölf kreisförmigen Templates, sind

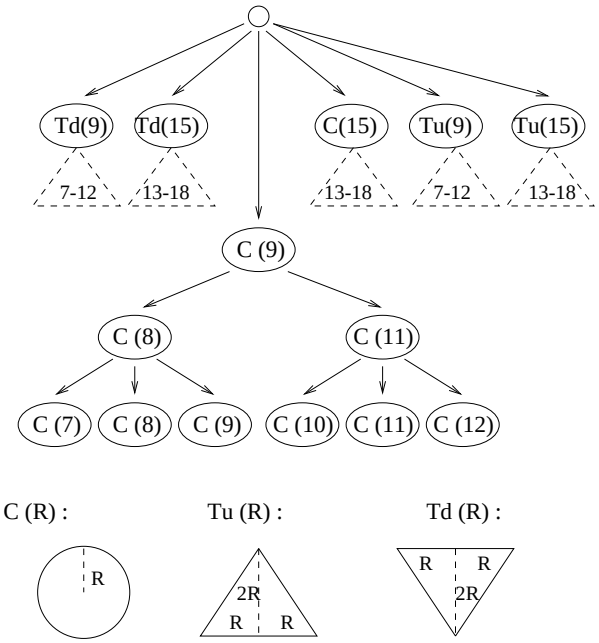



**Abbildung 2.13:** Template-Hierarchie nach [3].

der ersten Stufe der Template-Hierarchie zwei Templates mit Radien von neun bzw. 15 Pixel zugeordnet. Die beiden Templates dienen als Prototypen für die beiden weiteren Template-Stufen, mit jeweils zwei bzw. drei Templates, die in Abb. 2.13 dargestellt sind.

Bei der Berechnung der Ähnlichkeitsmaße werden zunächst die Templates einer tieferen Stufe gegenüber Templates derselben Stufe bevorzugt.

Aus Effizienzgründen wird das Template-Matching zusätzlich auf einem aus drei Ebenen bestehenden Bildraster (Gitter) durchgeführt. Die jeweiligen Gitterebenen besitzen eine Gitterkonstante von acht, vier und einem Pixel. Das Template-Matching wird zunächst auf dem groben Gitter durchgeführt. In ROIs des Bildes wird auf einer der beiden unteren Gitterebenen gearbeitet.

Verwendet man beide Strategien, so kann der Algorithmus um einen Faktor 20 - 200 beschleunigt werden. Der Beschleunigungsfaktor ist jedoch sehr stark vom Bildinhalt abhängig, was bei sicherheitsrelevanten Anwendungen von Nachteil ist.


**Abbildung 2.14:** Eingangsbild (links) und Ergebnisbild für das Matching (rechts), welches mit zwölf kreisförmigen Templates durchgeführt wurde.

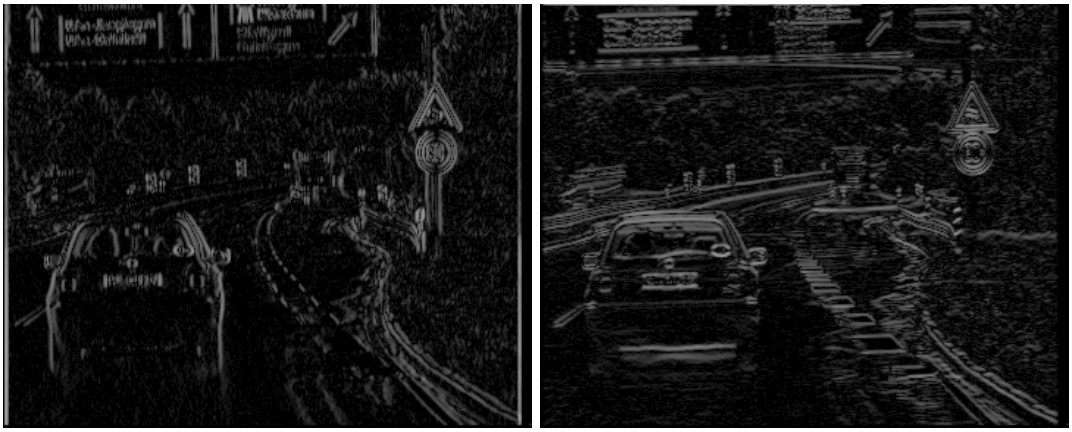


**Abbildung 2.15:** Betrag der Sobelbilder in x-Richtung (links) und y-Richtung (rechts).

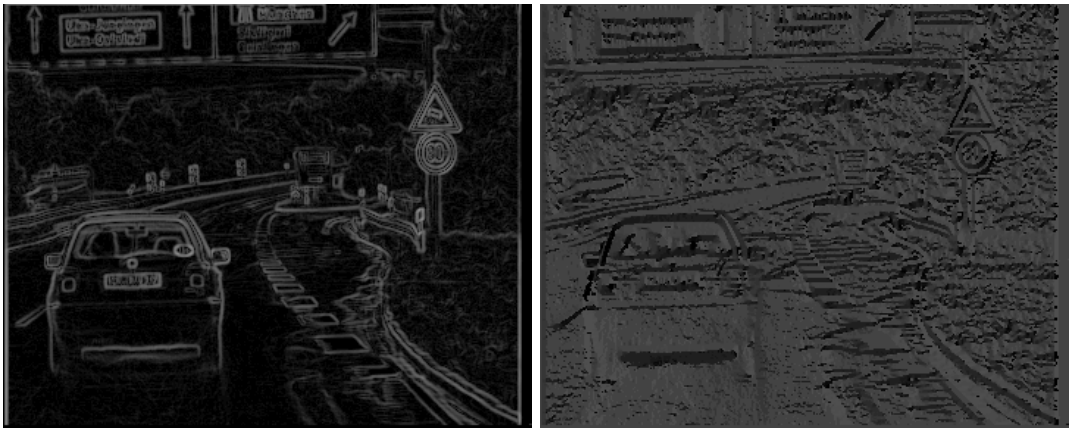


**Abbildung 2.16:** Betrag der beiden Sobelbilder (links) und diskretisiertes Richtungsbild (rechts).

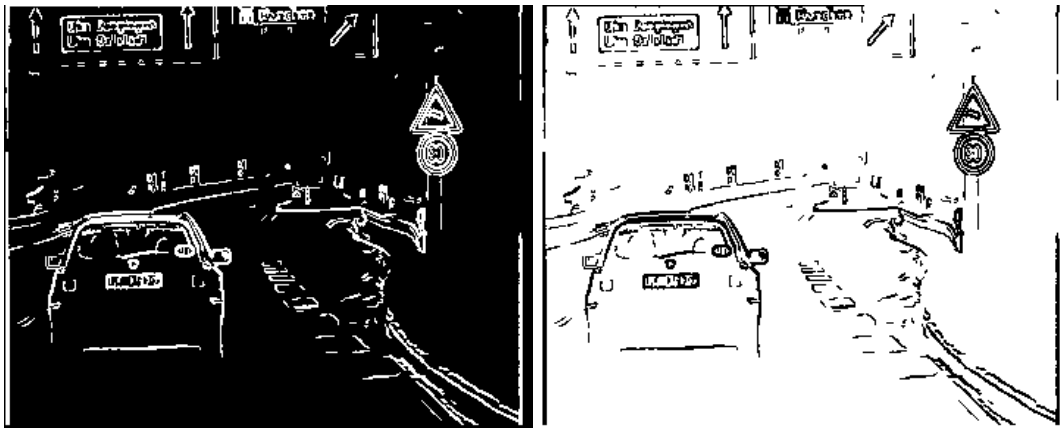


**Abbildung 2.17:** Binäres Kantenbild (links) und Bild nach dem *Cleaning* (rechts).

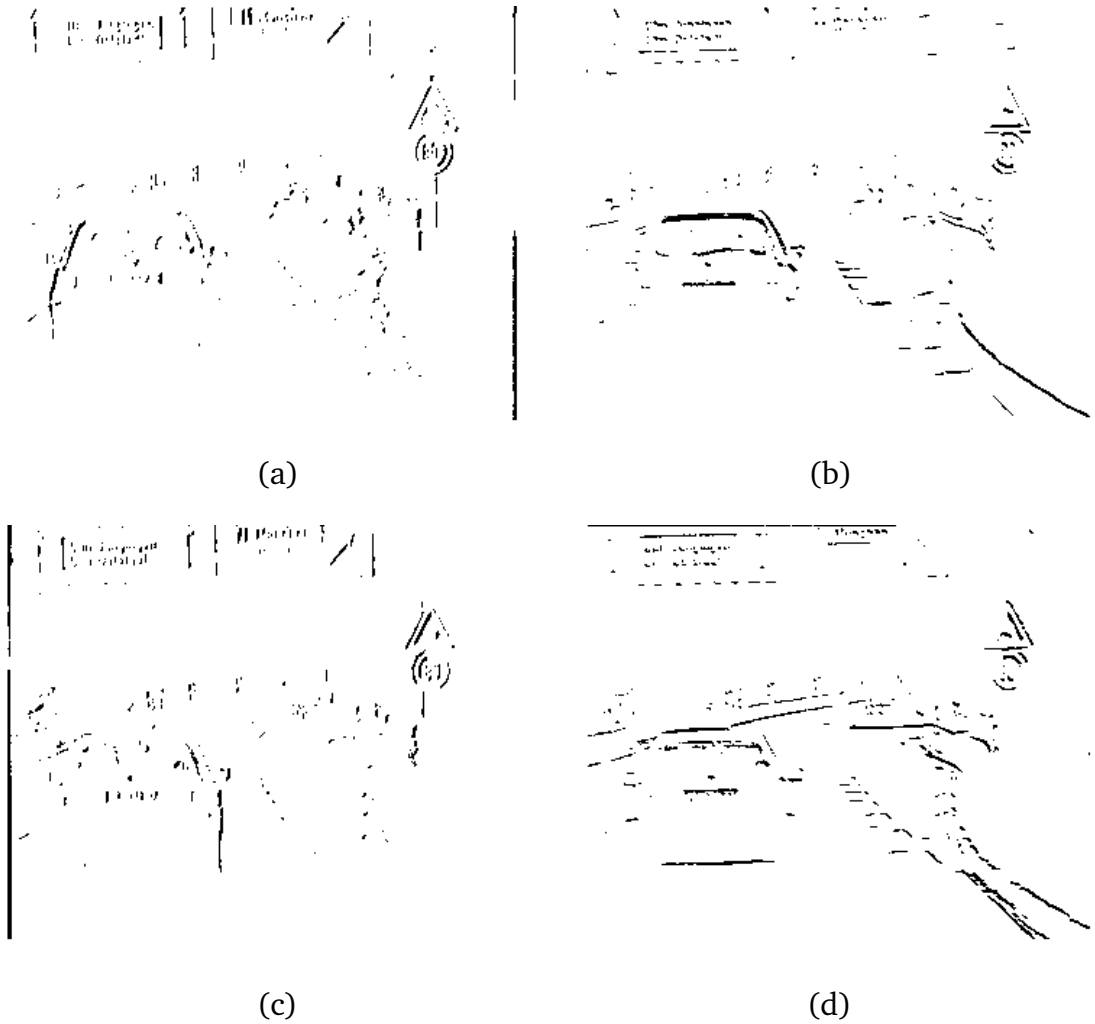
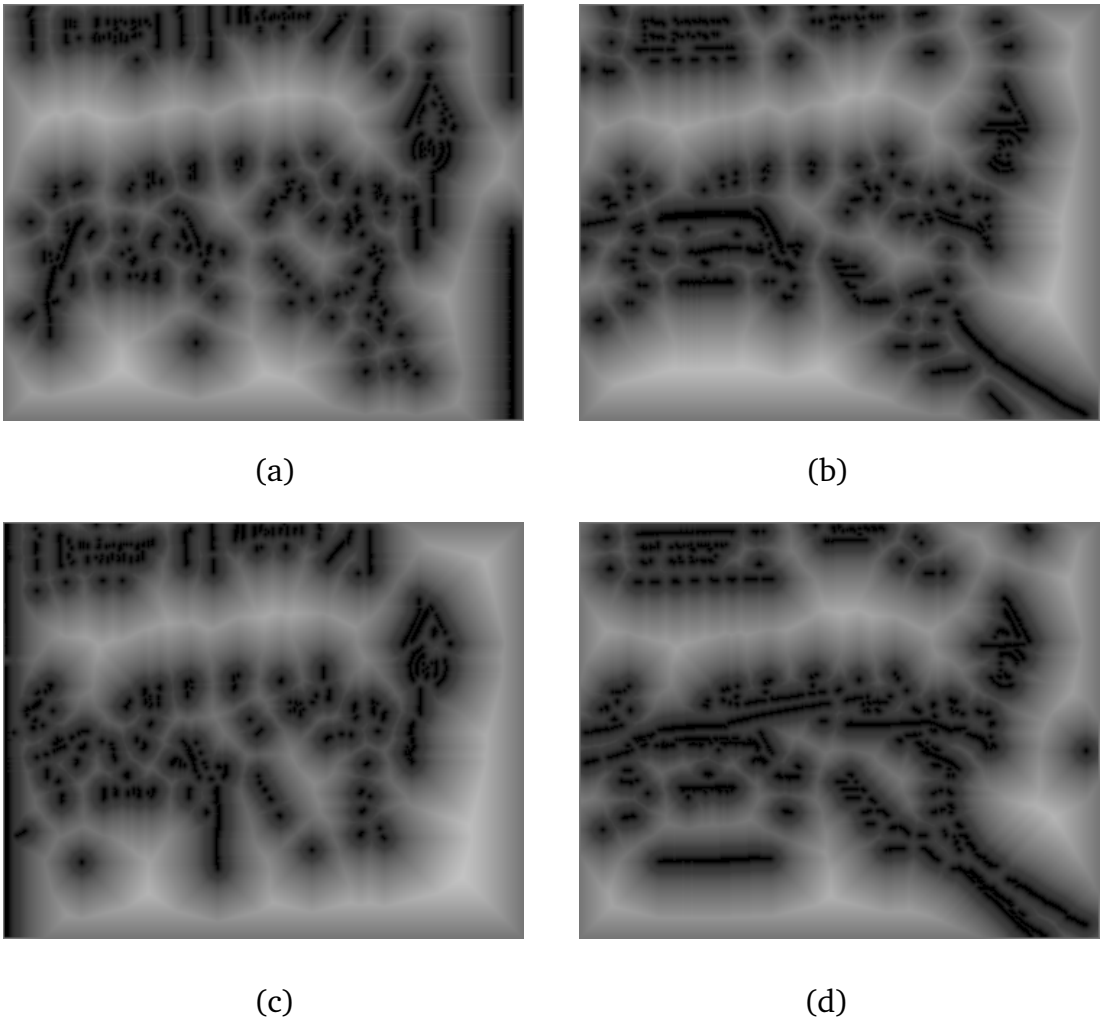



**Abbildung 2.18:** Eingangsbilder für die Distanztransformation für (a) Richtungsbereich 0 (b) Richtungsbereich zwei (c) Richtungsbereich vier (d) Richtungsbereich sechs.



(a) (b)

(c) (d)

**Abbildung 2.19:** DT-Bilder für (a) Richtungsbereich 0 (b) Richtungsbereich zwei (c) Richtungsbereich vier (d) Richtungsbereich sechs.

Kapitel 3

# Erzeugung von DT-Kantenbildern

Nachdem im letzten Kapitel die Algorithmen zu Berechnung der DT-Bilder erläutert wurden, wird in diesem Kapitel die Implementierung der wichtigsten Module für die beiden ersten Stufen der Anwendung, dem *Sehen* und dem *Erkennen*, beschrieben. In Abschn. 3.1 soll geklärt werden, wie der Datenaustausch zwischen den einzelnen Modulen organisiert ist. Die Module zur Bilderfassung sind in Abschn. 3.2 beschrieben und die zur Berechnung der Merkmalsbilder, den DT-Bildern mit Richtungsinformation, in den Abschn. 3.3 - 3.5.

Es zeigt sich, dass die Algorithmen zur Bestimmung der DT-Kantenbilder gut geeignet sind für eine parallele Implementierung nach dem Pipeline-Prinzip. Zur Berechnung der Distanztransformation wird der sequentielle Ansatz benutzt, der in jeweils einen Schritt in Vorwärts- und Rückwärts-Richtung zerfällt. Nach der Transformation in Vorwärts-Richtung werden die Zwischenergebnisse im externen SRAM des FPGA-Koprozessors abgespeichert. Somit lassen sich die vorverarbeitenden Module in zwei Pipelines einteilen. Zum einen in Kantenerkennung, *Cleaning*, Orientierung und DT in Vorwärts-Richtung und zum anderen in DT in Rückwärts-Richtung sowie einer speziellen Sortierung der Daten, die für eine schnelle Berechnung des nachfolgenden Template-Matching notwendig ist. Der Datenfluss der beiden Pipelines und das zentrale Steuermodul wird in Abschn. 3.6 beschrieben. Die Ergebnisse und eine Zusammenfassung sind in Abschn. 3.7 zu finden. Eine Kurzfassung von diesem Kapitel und Kapitel 4 wurde in [63] veröffentlicht.

## 3.1 Datenflussmodell

Um das Design einer digitalen Schaltung überschaubar zu halten wird dieses in Teilprobleme modularisiert. Hierbei wird zwischen datenverarbeitenden Modulen und Steuermodulen unterschieden, die ausschließlich Steuersignale generieren und ggf. mit dem Hostrechner kommunizieren [59].

Ein datenverarbeitendes Modul, wie in Abb. 3.1 dargestellt, empfängt, verarbeitet und sendet Daten in Abhängigkeit der Steuersignale. Außerdem kann ein datenverarbeitendes Modul Steuersignale erzeugen, die dann z.B. direkt an das nachfolgende Modul oder an ein Steuermodul weitergeleitet werden.

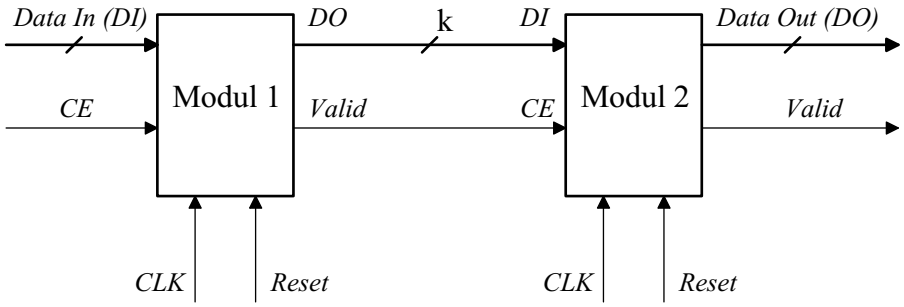



**Abbildung 3.1:** Grundlegender Aufbau und Zusammenschaltung zweier datenverarbeitenden Module.

Die Übertragung der Daten zwischen den Modulen erfolgt über eingehende und ausgehende Datenpfade, die oft zu den Datenleitungen *DI* (*Data In*) und *DO* (*Data Out*) zusammengefasst werden. Da die Module zum Verarbeiten der Daten intern Zeit benötigen, kann es vorkommen, dass die datenverarbeitenden Module nicht in jedem Takt Daten annehmen oder am Ausgang bereitstellen können. Die beiden Kontrollsignale *CE (Clock Enable)* und *Valid* zeigen an, ob die am Eingang bzw. Ausgang des Moduls anliegenden Daten Gültigkeit (beide *high-active*) haben. Das *CE*-Signal des Moduls ist dabei häufig mit den *CE*-Eingängen der im Modul vorhandenen FFs bzw. Register verbunden. Das *Valid*-Signal, welches i.A. mit dem *CE*-Eingang des nachfolgenden Moduls verbunden wird, wird entweder vom datenverarbeitenden Modul selbst generiert, oder von einem zentralen Steuermodul, was oft ressourcengünstiger ist. Fast alle Module besitzen neben der Versorgung mit der Clock noch ein *Reset*-Signal. Das Modul kann mit dem *Reset*-Signal dann wieder in den Initialisierungszustand zurückgesetzt werden.

Es wird darauf hingewiesen, dass der in Abb. 3.1 angegebene typische Aufbau der datenverarbeitenden Module nicht für alle im Rahmen dieser Arbeit implementierten Module Verwendung findet. Beim parallelen Template-Matching werden die datenverarbeitenden Module als nicht anhaltende Datenmodule, also solche mit $CE = 1$ konzipiert. Bei einigen von diesen Modulen wird außerdem auf die *Reset*-Signale verzichtet. Hiermit können viele Routing-Ressourcen auf dem FPGA eingespart werden.

## 3.2 Kameraanbindung

Die physikalische Anbindung der in dieser Arbeit verwendeten Digitalkamera von Pulnix über die *MPRACE-MenableIO* Aufsteckkarte [37] und das CMC-Aufsteckmodul EXT-IDE [38] an den MPRACE-Koprozessor wurde in Abschn. 1.3.5 beschrieben.

In diesem Abschnitt wird ausführlich auf die Design-Aspekte des Kameramoduls eingegangen. An dieser Stelle soll nur erwähnt werden, dass außerdem die Integration der Stecker A und B des MPRACE-Koprozessors, über welche die Kamera angeschlossen wird, in das Design bzw. CHDL realisiert werden musste.

### Vorhandenes Kamera-Modul

Zur Erfassung der Pixeldaten der Pulnix TM-1040 Kamera, wurde das von P. Dillinger [60] entwickelte und in Abb. 3.2 gezeigte Modul `cam` eingesetzt, welches generell für CCD-Kameras mit positiven Synchronisationssignalen geeignet ist. Die beiden po-

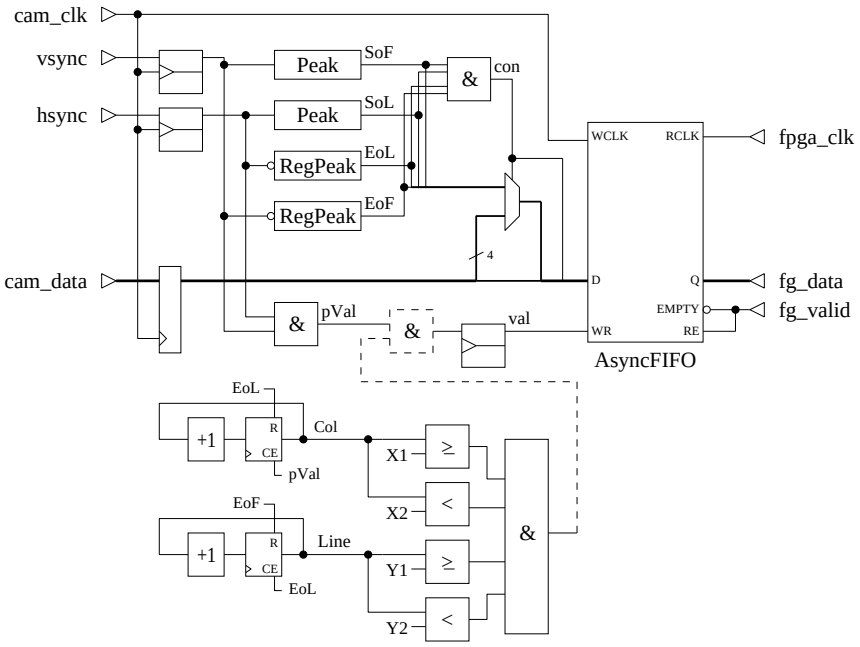



**Abbildung 3.2:** Das Modul `cam` zur Erfassung der Daten einer digitalen Kamera (nach [60]).

sitiven Synchronisationssignale *hsync* und *vsync* sowie die Daten *cam_data* der Kamera werden über Register, die mit der Clock *cam_clk* der Kamera getaktet werden, entkoppelt. Aus den Synchronisationssignalen werden die Statussignale *Start-of-Frame* (SoF), *Start-of-Line* (SoL), *End-of-Line* (EoL) und *End-of-Frame* (EoF) erzeugt, die dann mit den Daten der Kamera in einen gemeinsamen Bereich des asynchronen FIFOs geschrieben werden. Spezielle Verzögerungsglieder sorgen dafür, dass die Statussignale nicht mit den Daten kollidieren. Der Ausschnitt eines gültigen Bildbereiches kann über die Koordinaten $(X_1, Y_1)$ und $(X_2, Y_2)$ ausgewählt werden, siehe unten in Abb. 3.2. Diese Funktionalität wurde i.W. mit Zählern und Komparatoren realisiert.

Am Ausgang des FIFOs werden die Daten *fg_data* und das Gültigkeitssignal *fg_valid* synchron zur FPGA-Clock *fpga_clk* ausgelesen. Ist *fg_valid* high, so handelt es sich um Daten, ansonsten um Statussignale. Eine ausführlichere Beschreibung des Bilderfassungsmoduls ist in [60] zu finden.

### Erweitertes Kamera-Modul

Das oben beschriebene Modul `cam` wird in das neue Bilderfassungsmodul `camctrl` integriert, welches die im Rahmen dieser Arbeit benötigten Funktionalitäten bereitstellt.

Zum einen werden die 10 Bit-Pixel der Pulnix TM-1040 Kamera auf 8 Bit-Pixel reduziert indem die beiden niederwertigsten Bits abgeschnitten werden. Des Weiteren besteht die

Möglichkeit die Pixel seriell zu jeweils vier benachbarten Pixel zusammenzupacken. Die Pixel können dann einzeln oder zu viert über das *DO*-Signal an das nachfolgende Modul weitergegeben werden. Das *Valid*-Signal gibt deren Gültigkeit an. Außerdem kann eine Adressgenerierung aktiviert werden, dessen Signale z.B. mit dem Adresseneingang eines RAM-Bausteins verbunden werden kann. Hierzu werden das Adress-Signal *Addr* und das Signal *AddrWE* erzeugt, wobei letzteres die Gültigkeit der Adressen angibt. Die Adressen werden in Abhängigkeit eines Adress-Offsets erzeugt, der vom Benutzer angegeben werden kann. Von Außen kann das Modul mit dem *CE*-Signal kontrolliert werden. Liegt *CE* $= 1$ an, so wartet das Modul auf das *SoF*-Signal des Submoduls `cam` und beginnt mit der Bildaufnahme und liefert so lange Daten, bis *CE* $= 0$ gesetzt wird. Nach der Ausgabe des Bildes wird das *Done*-Signal erzeugt, welches einen Takt nach dem letzten gültigen Pixel das Ende der Bildaufnahme des ausgewählten Bildausschnitts anzeigt.

Zusätzlich kann das Submodul `downsample2` aktiviert werden, welches den Mittelwert aus den vier Pixel einer $2 \times 2$ Umgebung berechnet und zusätzlich ein reduziertes Bild auf einer kleineren Skala erzeugt, siehe hierzu Abschn. 1.1.2. Die Größe dieses Bildes mit gröberer Auflösung reduziert sich auf $H/2$ und $W/2$. Die Implementierungsdetails des `downsample2`-Moduls werden in Abschn. 3.3.1 angegeben. Die Daten können wiederum einzeln oder zu vier benachbarten Pixel zusammengepackt weitergegeben werden. Außerdem kann für dieses Modul eine Adressgenerierung aktiviert werden. Die Adressen werden dann in Abhängigkeit eines Adress-Offsets generiert, die sich mit dem Adressraum des ersten Bildes nicht überschneiden dürfen. Über einen Daten- und Adress-Multiplexer wird sichergestellt, dass die Daten und Adressen des `downsample2`-Moduls nicht mit denen des ersten Bildes kollidieren. Die Adressgenerierung wurde auf dem FPGA mit ladbaren Zählern und Komparatoren realisiert.

**Parametrisierung** Das Modul `camctrl` ist parametrisierbar bezüglich der Bildbreite $W$ und -höhe $H$. Diese beiden Parameter sind auch während der Ausführung des Designs auf dem FPGA veränderbar. Eine Adressgenerierung mit variablem Adress-Offset kann eingeschaltet werden, der auch während der Ausführung des Designs veränderbar ist. Außerdem kann das `downsample2`-Modul aktiviert werden, welches ein um einen Faktor vier reduziertes Bild und die Adressen für einen vom Benutzer gewünschten Adressbereich generiert.

## 3.3 Kantenbild

Die Kanten im Bild werden mittels Anwendung der Sobel-Operatoren in x- und y-Richtung, einfachem Schwellwert und anschließendem *Cleaning* bestimmt, siehe Abschn. 2.1. Die Implementierung dieser Operationen für FPGAs werden im Folgenden im Detail beschrieben. Außerdem wird in diesem Zusammenhang auf die FPGA-Implementierung von beliebigen FIR-Filtern, separierbaren Filtern und auf weitere Möglichkeiten der Parallelisierung bei FIR-Filtern eingegangen, welche im Rahmen des *OpenEye*-Projekts[1] realisiert wurden.

[1]Das *OpenEye*-Projekt wurde vom Wirtschaftsministerium des Landes Baden-Württemberg gefördert (AZ: 4-4332.62-IMS/4).

### 3.3.1 Sobel-Operator

Zur Bestimmung der Gradientenbilder wird der Sobel-Operator (Gl. 2.2) benutzt, ein Ableitungsfilter mit ganzzahligen, antisymmetrischen Koeffizienten, siehe oben in Abb. 3.3.

Diese Nachbarschaftsoperationen werden - zumindest in Software - oft so berechnet, indem die Filtermaske zeilenweise über das Bild geschoben wird [6], siehe Abschn. 2.1.1. Die im Rahmen dieser Arbeit erfolgte Hardware-Implementierung arbeitet genau umgekehrt. Die Filtermaske wird fest in das FPGA "verdrahtet" und das Bild wird zeilenweise unter der Maske hindurch geschoben [62]. Der Aufbau der Implementierung des parallel arbeitenden Sobel-Moduls ist in Abb. 3.3 dargestellt.

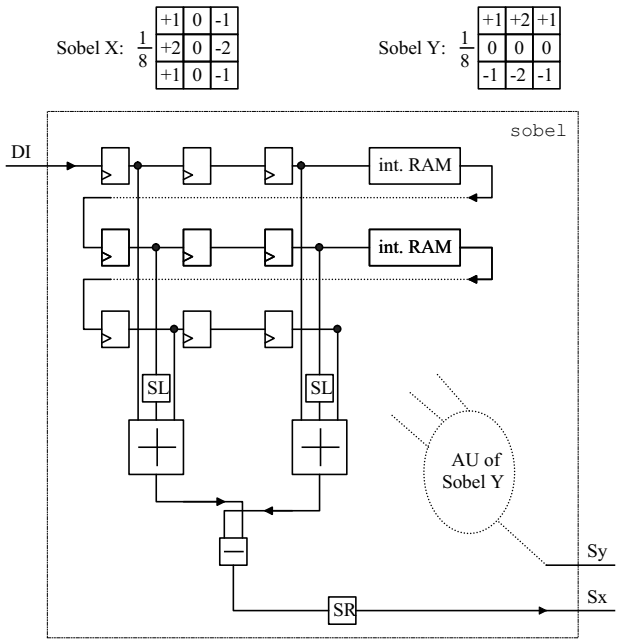



**Abbildung 3.3:** Hardware-Implementierung des Sobel-Operators.

Links oben ist ein $3 \times 3$ Pixel großes *Shift Register Array* (SRA) zu sehen, welches den parallelen Zugriff auf alle Bilddaten, die für das Filter relevant sind, erlaubt. Die Anzahl der Zugriffe auf das externe SRAM des FPGA-Koprozessors kann minimiert werden, indem zwei Zeilen des Bildes im internen RAM des FPGAs gespeichert werden. Dann ist nur eine Zeile aus dem externen SRAM des FPGA-Koprozessors auszulesen. Beim Virtex-II wird zur internen Speicherung auf dessen Block-RAM, bei einem FPGA der 4000er Familie auf dessen LUT-RAM zurückgegriffen. Überschreitet die Tiefe des benötigten internen FPGA-RAMs die Tiefe der einzelnen RAM-Bausteine des FPGAs, so werden diese nacheinander aus den kleineren RAM-Elementen zusammengeschaltet. Für ein Bild mit der Breite $W$ muss das interne FPGA-RAM eine Tiefe von $W - 3$ haben.

Die Register des SRAs und das FPGA-RAM werden von beiden Rechenwerken (AUs[2]) der Sobel-Operatoren in x- und y- Richtung verwendet. Die Koeffizienten der Sobel-Filtermasken sind sehr gut für eine effiziente Hardware-Implementierung geeignet. Die

[2] Arithmetic Units.

Multiplikationen mit den Filterkoeffizienten $1$ und $-1$ werden nicht ausgeführt bzw. durch Subtraktionen realisiert, die mit $2$ und $-2$ durch einen Linksshift (SL)[3] bzw. Subtraktion. Nach Addition bzw. Subtraktion der Daten erfolgt die Normierung mit dem Vorfaktor acht durch einen Rechtsshift (SR[4]) um drei Stellen. Zusätzliche Register, die nicht in Abb. 3.3 zu sehen sind, wurden zwischen den Addierern bzw. Subtrahierern eingefügt, um die Gatterlaufzeiten zwischen zwei synchronen Bauteilen auf dem FPGA zu begrenzen.

**Rechenzeiten und Datendurchsatz** Die beiden AUs sind in der Lage die $3 \times 3$ Pixel in einem Taktzyklus zu berechnen. Dies erlaubt es Takt für Takt ein Pixel in das SRA des Filter-Moduls zu schieben. Somit erhält man einen Datendurchsatz von einem Pixel pro Takt. Die beiden Ergebnispixel liegen nach einer Latenzzeit, die abhängig ist von der Bildgröße, den Filterkoeffizienten und den Registerstufen der AU, am Ausgang des Filter-Moduls an. Die Latenzzeit für das Sobel-Modul berechnet sich für ein Bild mit der Breite $W$ zum einen aus der Latenzzeit von SRA und internem FPGA-RAM zu $W + 2$ und zum anderen aus drei Registerstufen der AU, insgesamt zu $W + 5$.

**Ressourcenverbrauch** Für das SRA, welches von den AUs der beiden Sobel-Operatoren benutzt wird, werden insgesamt neun Register benötigt. Die Koeffizienten der beiden Sobel-Operatoren konnten durch einfache Additionen oder Shift-Operationen ersetzt werden. Insgesamt werden für die AU zehn Addierer bzw. Subtrahierer auf dem FPGA eingesetzt. Zusätzlich sind Register nach den Addierern eingefügt, um die Anzahl der Logikstufen zu begrenzen. Der Ressourcenverbrauch für das Sobel-Modul ist in Tab. 3.3 angegeben.

### 3.3.2 Allgemeine FIR-Filter

Die Daten, die für die Berechnung eines allgemeinen FIR-Filters relevant sind, werden wiederum in einem SRA der Größe $(2r + 1) \times (2r + 1)$, welches der Größe der Filtermaske entspricht, abgespeichert [61]. Die Koeffizienten der Filtermaske $H$, siehe Gl. 2.1, werden durch Multiplizierer auf dem FPGA realisiert. Sind die Koeffizienten der Filtermaske konstant, so werden diese durch Festkomma-Multiplizierer realisiert. Sollen die Koeffizienten des Designs veränderbar sein, so werden Gleitkomma-Multiplizierer eingesetzt. Diese haben jedoch einen höheren FPGA-Ressourcenaufwand als die Festkomma-Multiplizierer. Für eine effiziente Implementierung der verschiedenen Multiplizierer auf dem FPGA werden von den FPGA-Herstellern sog. *Core Tools* zur Verfügung gestellt [33]. Die Ausgänge der Multiplizierer werden durch einen binären Addiererbaum zusammengefasst.

Im Weiteren werden Implementierungsstrategien für separierbare Filter am Beispiel des Sobel-Operators und für Filter, die N-Fach parallele Daten verarbeiten, gegeben. Es folgt eine Beschreibung des `downsample2`-Moduls, mit welchem die Größe der Bilder um einen Faktor zwei reduziert werden kann.

[3] Shift Left.
[4] Shift Right.

### Separierbare Filter

Der Sobel-Operator ist ein separierbares zweidimensionales Filter, welches in eindimensionale vertikale und horizontale Komponenten zerlegt werden kann. Dies ist in Gl. 3.1 für den Sobel-Operator in x-Richtung angegeben

$$\mathbf{S}_x = \frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * \frac{1}{4}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}. \tag{3.1}$$

Im Folgenden wird eine grobe Abschätzung für den Ressourcenverbrauch und die Rechenzeit für eine FPGA-Implementierung des separierbaren Sobel-Operators angegeben. Zur Zwischenspeicherung der Pixel in den SRAs, die mit der Größe der Filtermaske korrespondieren, sind für jede Richtung drei Register, insgesamt also sechs Register, notwendig. Der Ressourcenaufwand an SRAs ist somit für separierbare Filter geringer. Die Recheneinheit (AU) besteht für die Differentiation in x-Richtung aus einer Subtraktion und für die Mittelung in y-Richtung aus zwei Additionen und einem Linksshift. Die Normierung erfolgt mit einem Rechtsshift um eine bzw. zwei Stellen. Für die AU ist der FPGA-Ressourcenaufwand für das separierbare Sobel-Filter ebenfalls geringer.

Die Organisation der Daten ist jedoch wesentlich aufwändiger. Zum einen besteht die Möglichkeit die Daten nach der Faltung in horizontaler Richtung im externen RAM zwischenzuspeichern und anschließend in vertikaler Richtung aus dem externen RAM auszulesen und zu berechnen. Hierzu muss ein zusätzliches Steuerungsmodul implementiert werden, welches die Daten spaltenweise auslesen bzw. abspeichern kann. Der wesentliche Nachteil ist jedoch darin zu sehen, dass sich die Zeit zur Berechnung des Filters ungefähr verdoppelt. Zum anderen besteht die Möglichkeit die Daten nach der zeilenweisen Berechnung im internen RAM des FPGAs zu speichern und dann sofort die spaltenweise Berechnung durchzuführen. Hierzu sind ein zusätzliches Steuerungsmodul und internes FPGA-RAM notwendig.

Bei kleinen Filtermasken ist es empfehlenswert die Filter als nicht separierbare zu implementieren, weil zusätzlich ein Steuerungsmodul notwendig ist. Bei großen Filterkernen lassen sich jedoch viele Ressourcen auf dem FPGA einsparen. Das Sobel-Filter wurde in dieser Arbeit direkt implementiert, siehe Abb. 3.3.

**Separierbarer Binomialfilter** In diesem und dem nächsten Abschnitt werden Arbeiten vorgestellt, die u.A. im Rahmen des *OpenEye*-Projekts vom Autor an der Universität Mannheim erfolgten.

Zunächst wird am Beispiel eines beliebigen separierbaren 1D-Binomialfilters gezeigt, wie dieses effizient auf FPGAs abgebildet werden kann.

Das Binomialfilter $\mathbf{B}^p$ wird aus der einfachen elementaren Glättungsmaske

$$\mathbf{B} = \frac{1}{2}[1\,1] \tag{3.2}$$

durch p-maliges anwenden

$$\mathbf{B}^p = \frac{1}{2^p}\underbrace{[1\,1] * [1\,1] * \ldots * [1\,1]}_{p\text{–mal}} \tag{3.3}$$

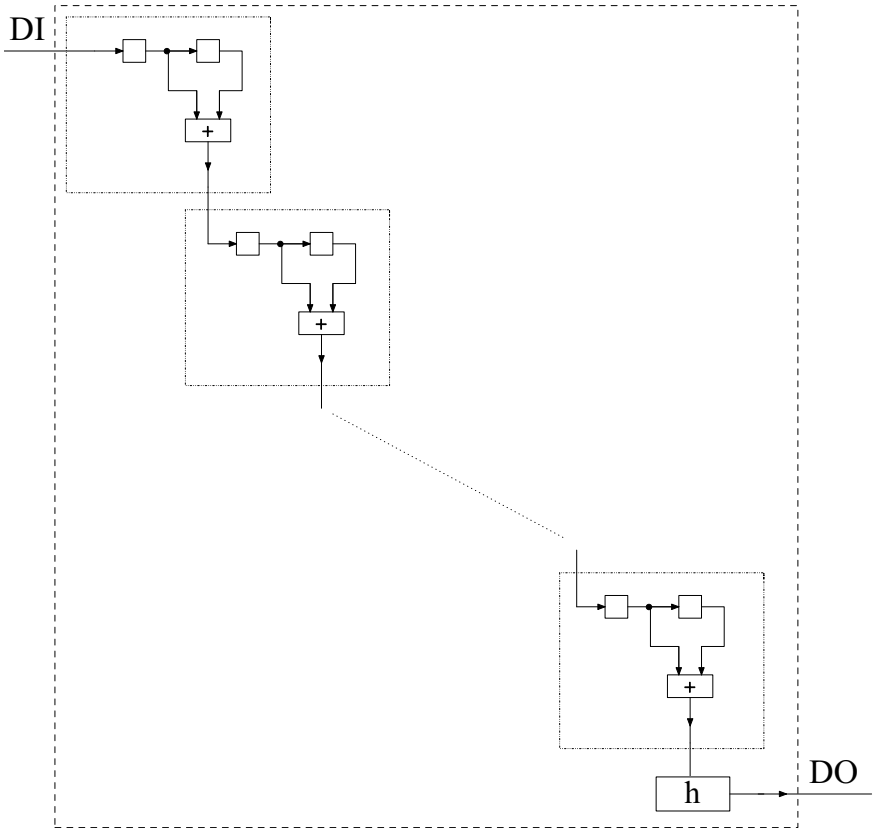



**Abbildung 3.4:** Hardware-Implementierung eines separierbaren 1D-Binomial-Filters.

aufgebaut [6].

Die FPGA-Implementierung der elementaren $[1\,1]$ Maske wird durch ein SRA bestehend aus zwei Registern und einer AU bestehend aus einem Addierer realisiert. Der Ausgang von jedem elementaren Modul wird dabei einfach mit dem Eingang des nachfolgenden Moduls verbunden, für ein Filter $\mathbf{B}^p$ insgesamt $p$-mal, siehe Abb. 3.4.

Die Normierung wird durch einen Rechts-Shift um $p$ Positionen realisiert, da der Normierungsfaktor $h = 2^p$ eine Potenz von $2$ ist. Außerdem kann die Normierung mit dem Faktor $2$ nach der Berechnung von jedem elementaren $[1\,1]$-Filter erfolgen. Dieses Vorgehen führt in jedem Schritt zu Rundungsfehlern, jedoch ist der FPGA-Ressourcenverbrauch minimal. Wird die Normierung mit dem Faktor $2^p$ nur einmal nach dem letzten elementaren Binomialfilter durchgeführt, so kann nur an dieser Stelle ein Rundungsfehler auftreten. Die Bitbreiten der elementaren $[1\,1]$ Masken erhöhen sich dann nach jedem Schritt um eins, was insgesamt zu einem entsprechend höheren Ressourcenbedarf führt. Bei größeren Binomialfilterkernen, wie z.B. dem $\mathbf{B}^8 = 1/256[1\ 8\ 28\ 56\ 70\ 56\ 28\ 8\ 1]$ bewirkt die separierbare Methode merkliche Ressourceneinsparungen, weil die mittleren Koeffizienten des $\mathbf{B}^8$ durch ressourcenaufwändige Festkomma-Multiplizierer zu implementieren sind.

### N-fach Paralleler Filter

Eine weitere Parallelisierung ist für FIR-Filter möglich, falls die Daten in $N$-fach paralleler Form vorliegen [62]. Dabei sind jeweils $N$ benachbarte Pixel in einem Datum abgespeichert. Diese werden in ein entsprechend angepasstes SRA (Shift Register Array) geschoben, so dass $N$ parallel arbeitende Rechenwerke parallelen Zugriff auf die relevanten Daten haben, siehe Abb. 3.5. Dies führt zu einem N-fachen Ressourcenverbrauch

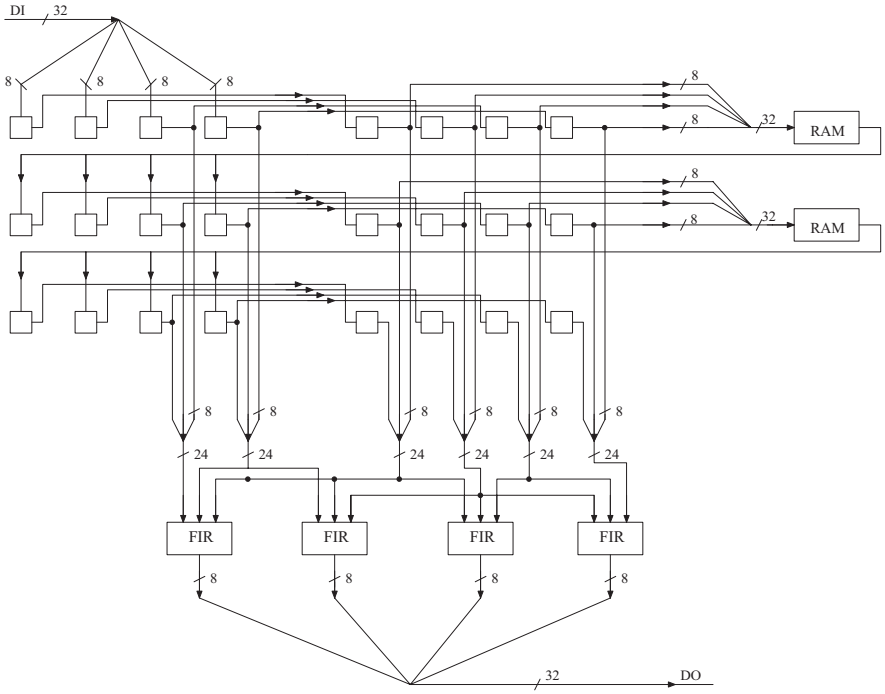



**Abbildung 3.5:** Hardware-Implementierung eines allgemeinen $3 \times 3$ Filters mit vierfach parallelen Daten.

für die AUs und i.A. zu einem etwas weniger als N-fachen für das SRA und sogar einem kleineren für das interne FPGA-RAM, da die Länge der zu speichernden Bildzeilen kürzer wird.

Am Beispiel eines $3 \times 3$ Filters mit vierfach parallelen Daten wird die Funktionsweise in Abb. 3.5 dargestellt. Zur Berechnung der Daten mit vierfacher Geschwindigkeit werden vier FIR-AUs eingesetzt. Der Ressourcenverbrauch für das SRA erhöht sich nur um einen Faktor 2.7 und der für das interne FPGA-RAM verringert sich, da sich die Länge der Bildzeile um fünf verkürzt.

Ein N-fach paralleler Filter wird im Rahmen dieser Arbeit jedoch nicht verwendet, weil dies einerseits einen höheren Ressourcenbedarf zur Folge hätte, auch für die nachfolgenden Module der Pipeline, die dann ebenfalls mit entsprechender N-facher Parallelisierung implementiert würden. Andererseits zeigt sich jedoch, dass die DT nicht ohne weiteres N-fach beschleunigt werden kann.

Das implementierte Filter-Modul zur Berechnung beliebiger FIR-Filter für FPGAs ist parametrisierbar bezüglich der Filtergröße, den Filterkoeffizienten, der Anzahl der parallelen

Daten, der Bildgröße und der Bitbreite der Daten.

#### Downsample

Eine Möglichkeit für das Auffinden von gröberen Strukturen im Bild besteht darin, mit einer niedrigeren Auflösung des Bildes zu arbeiten, siehe Abschn. 1.1.3. Die Reduzierung des Bildes muss allerdings mit einer angemessenen Glättung durchgeführt werden. Die einfachste Form ist die Glättung mit einem $\frac{1}{2}[1,\ 1]$ Filter in x- und y-Richtung, die im Rahmen dieser Arbeit für FPGAs implementiert wurde.

Das Modul `downsample2` berechnet den Mittelwert aus vier Pixel einer $2\times2$ Umgebung. In jedem Takt wird ein Pixel des ursprünglichen Bildes in ein $2 \times 2$ großes SRA geschoben. Dies entspricht der Vorgehensweise beim Sobel-Operator, siehe Abschn. 3.3.1. Die AU besteht aus drei Addierer, die als Addiererbaum aufgebaut sind. Die Normierung mit vier erfolgt über einen Rechts-Shift um zwei Stellen.

Auf die Steuerlogik zur Generierung des *Valid*-Signals, welches die Gültigkeit der Daten am Ausgang des Moduls anzeigt, wird hier nicht näher eingegangen. Im Mittel liegen in jedem vierten Takt die Pixel des reduzierten Bilds am Ausgang des Moduls an.

### 3.3.3 Binarisierung

Die der Summierung der Pixel der Sobel-Bilder nachfolgende Binarisierung wird mittels einfachem Schwellwertoperator $\theta$ realisiert, siehe Abschn. 2.1.1.

Am Eingang des Schwellwertoperator-Moduls `threshold` liegen die Summe der Beträge der Sobel-Bilder in x- und y-Richtung, siehe Abb. 3.12. Die Implementierung des Schwellwertoperators wird mittels eines Komparators realisiert. Das binäre Ergebnispixel wird anschließend in einem FF zwischengespeichert.

**Parametrisierung** Der Schwellwert $\theta$ kann zum einen als Parameter an das Modul `threshold` übegeben werden oder während der Laufzeit des Designs vom PC aus verändert werden. Das hierfür zu benutzende Codewort ist in Tabelle 4.3 zu finden.

### 3.3.4 *Cleaning* Operator

Ziel des *Cleaning*-Operators ist es max. drei zusammenhängende, isolierte Störpixel im Kantenbild zu eliminieren, siehe Abschn. 2.1.1. In Software ist diese morphologische Operation rekursiv implementiert, was einen nicht-systematischen Zugriff auf die Bilddaten zur Folge hat. Einen Überblick über Hardware-Implementierungen von bekannten morphologischen Operatoren ist in [68] zu finden.

Diese Strategie ist nicht gut geeignet für eine FPGA-Implementierung und macht die Integration des *Cleaning*-Operators in die erste Pipeline der Vorverarbeitung (Abb. 3.12) nicht möglich. Aufgrund dessen wurde das Modul `cleaning3` nach dem Pipeline-Prinzip mit einer Logischen Einheit (LU) aufgebaut, die parallelen Zugriff auf alle relevanten

Pixel hat und alle möglichen Kombinationen von max. drei zusammenhängenden Kantenpixel überprüft, siehe Abb. 3.6. Die Pixel des binären Kantenbildes werden hierzu in

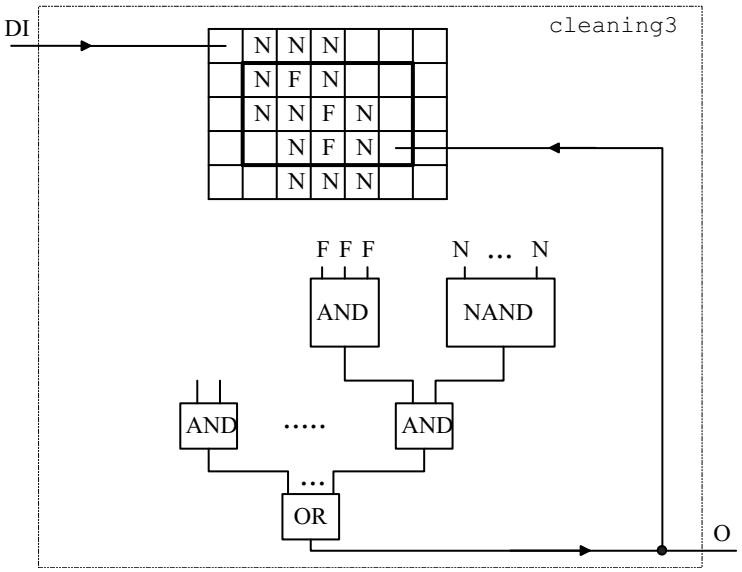



**Abbildung 3.6:** Hardware-Implementierung des Moduls `cleaning3`.

einem SRA der Größe $7 \times 5$ gespeichert. Um den Kontext des Bildes zu erhalten werden die Zeilen des binären Kantenbildes wiederum im internen RAM des FPGA gespeichert, was jedoch nicht in Abb. 3.6 dargestellt ist.

Die LU überprüft alle möglichen Kombinationen von drei oder weniger zusammenhängender Kantenpixel. Für ein isoliertes Pixel ist genau eine Kombination zu überprüfen, für zwei zusammenhängende isolierte Pixel sind sechs Kombinationen, und für drei zusammenhängende Pixel 26 Kombinationen. In der inneren $5 \times 3$ Umgebung werden die zusammenhängenden Pixel betrachtet. Ein isoliertes Kantenpixel wird genau dann eliminiert, wenn max. drei zusammenhängende Kantenpixel (F)[5] auftreten und alle sie umgebenden Pixel bezüglich der 8er-Nachbarschaft keine Kantenpixel (N)[6] sind. Dies wird durch logische UND, NAND und OR Operationen realisiert, siehe Abb. 3.6. Die FFs zur Begrenzung der Anzahl der Logikstufen sind nicht eingezeichnet.

Die eliminierten Kantenpixel werden mit einer Verzögerung von zwei Taktzyklen der LU in das SRA zurück geschrieben, weil sie in der nächsten Zeile benötigt werden.

Der Verbrauch an FPGA-Ressourcen sowie die Latenzzeit des *Cleaning*-Moduls sind in Tab. 3.3 angegeben.

**Parametrisierung** Die Breite $W$ des Bildes und die max. Anzahl der Pixel die isoliert werden sollen ist beim Modul `cleaning3` über einen Parameter des Moduls oder während der Laufzeit über den PCI-Bus vom PC aus einstellbar.

Diese Funktionalität wurde realisiert indem der logische OR-Baustein, welcher alle $1+6+32 = 39$ Kombinationen für die bis max. drei zu isolierenden Pixel zusammenfasst, aus

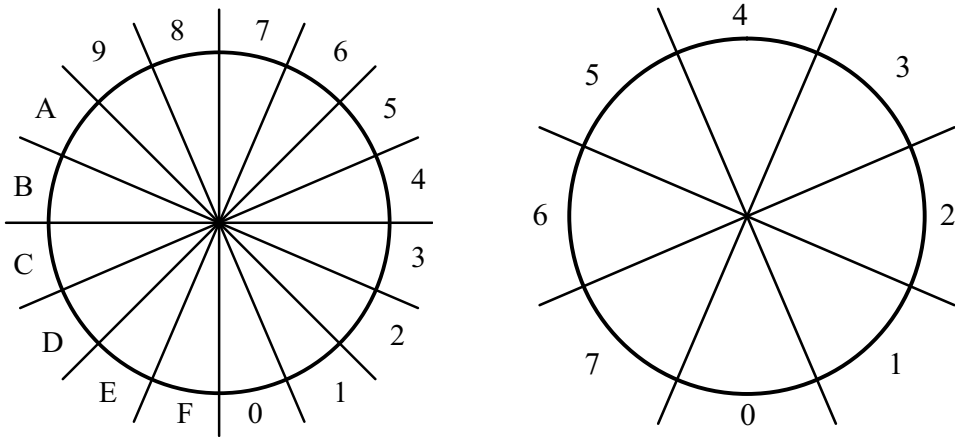
[5] *Feature*-Pixel
[6] *Non-Feature*-Pixel

drei getrennten logischen OR-Bausteinen aufgebaut wird. Die Auswahl der “richtigen” Zusammenschaltung der drei logischen OR-Bausteine erfolgt dann über einen Multiplexer, dessen Steuerung über den PCI-Bus mit einem Codewort erfolg, welches in Tab. 4.3 angegeben ist.

**Optimierungen** Für jede der 37 Kombinationen zur Überprüfung von drei zusammenhängender Pixel werden die *Non-Feature*-Pixel, welche die *Feature*-Pixel umgeben jeweils mit einem eigenen NAND Operator überprüft, siehe Abb. 3.6. Dabei gibt es bei den *Non-Feature*-Pixel der jeweiligen Kombinationen viele Überdeckungen. FPGA-Ressourcen könnten eingespart werden, falls die gemeinsamen Überdeckungen durch gemeinsame NAND-Operatoren realisiert würden.

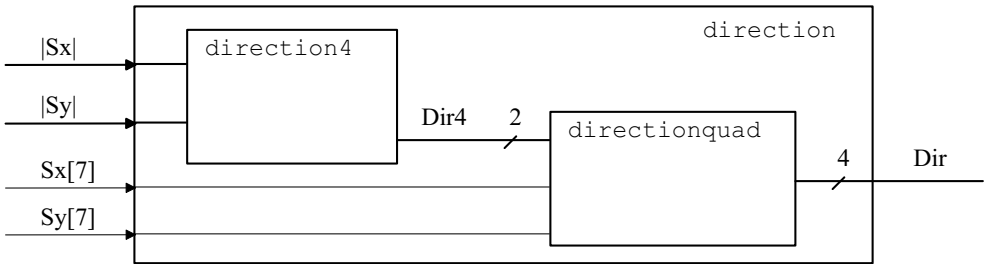
## 3.4 Orientierung

Im Folgenden wird die Berechnung der Orientierung und der daraus folgenden Richtungsquantisierung beschrieben. In Abschn. 2.1.2 wurden bereits die hierzu notwendigen Rechenvorschriften angegeben. Zunächst erfolgt im Modul `direction` eine Zuweisung der Kantenpixel anhand der Sobel-Pixel $(G_{\mathbf{S}_x}, G_{\mathbf{S}_y})$ in x-und y-Richtung in 16 gleich große Richtungsbereiche, siehe Abb. 3.7(a), die später im Modul `demultiplex` zu acht Richtungsbereichen zusammengefasst werden, siehe Abb. 3.7(b). Zunächst wird der Einheitskreis $(2\pi)$ in gleich große Stücke der Größe $\pi/8$ aufgeteilt. In Software wurde diese Operation durch eine *Lookup*-Tabelle, die jedoch einen hohen Bedarf an Speicher hat, realisiert.

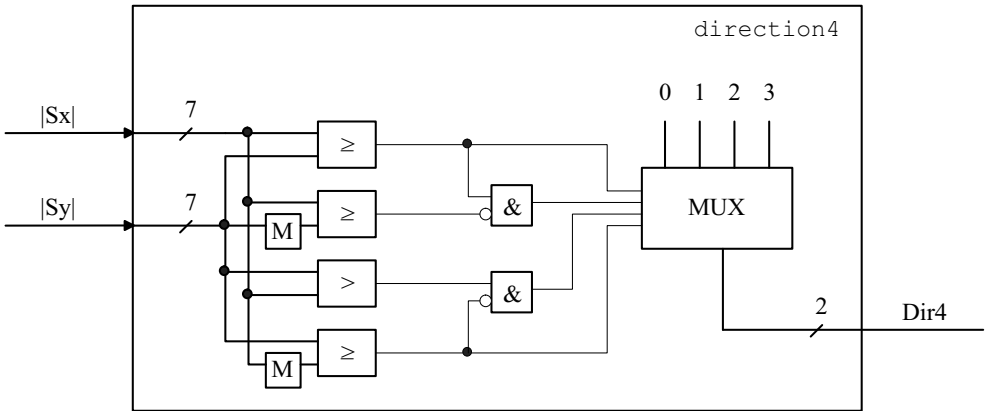



**(a)** 16 Richtungsbereiche. **(b)** 8 Richtungsbereiche.

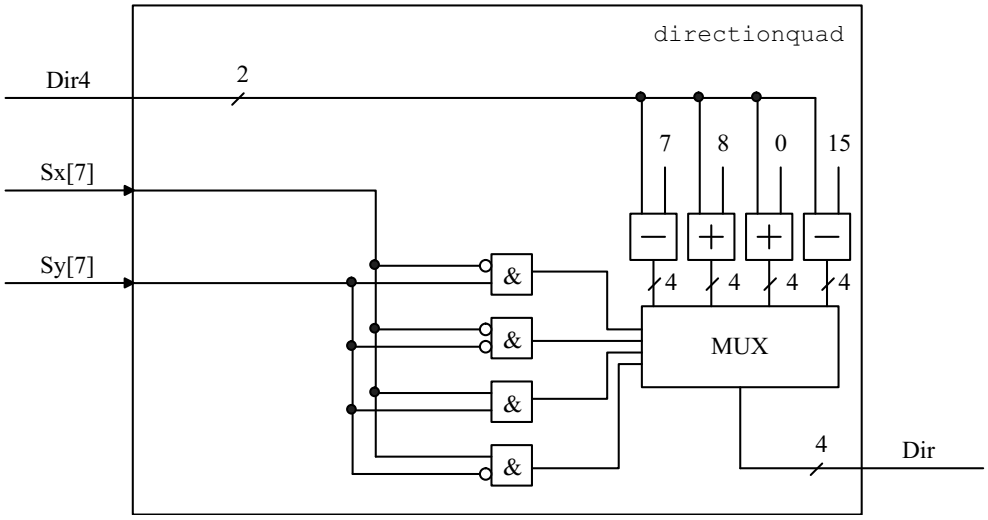
**Abbildung 3.7:** Aufteilung des Einheitskreises in 16 bzw. 8 unterschiedliche Richtungsbereiche.




**(a)** Übersicht über das Modul `direction`.




**(b)** Übersicht über das Submodul `direction4`.




**(c)** Übersicht über das Submodul `directionquad`.

**Abbildung 3.8:** Übersicht über das Modul `direction` und seine Submodule.

### Diskretisierung in 16 Richtungsbereiche

Es wird sich zeigen, dass für FPGAs auch eine direkte Berechnung geeignet ist, die i.W. durch einfache Vergleichsoperationen realisiert wird. Die Zuweisung in einen der 16 Richtungsbereiche erfolgt im Modul `direction` mit den Beträgen der Sobel-Pixel in x- und y-Richtung, sowie deren Vorzeichen. Aus der Vorzeicheninformation wird im Submodul `directionquad` unterschieden, in welchen der vier Quadranten die Zuweisung

erfolgt, siehe Abb. 3.8(c). Eine weitere Unterteilung der jeweiligen Quadranten in vier weitere Richtungsbereiche wird durch Vergleich der Beträge der Sobel-Operatoren im Submodul `direction4` realisiert, siehe Abb. 3.8(b). Hierzu werden für die Pixel der Sobelbilder in x- und y-Richtung folgende Vergleiche durchgeführt: $G_{\mathbf{S}_x} > tan\ \frac{\pi}{4}$[7] $G_{\mathbf{S}_y}$ und $G_{\mathbf{S}_x} > tan\ \frac{\pi}{8}\ G_{\mathbf{S}_y}$. Entsprechende Vergleiche sind für $G_{\mathbf{S}_y} > G_{\mathbf{S}_x}$ durchzuführen. Für den $tan\ \frac{\pi}{8}$ (in Abb.3.8(b) mit $M$ bezeichnet) wird mit einer Näherung

$$tan\ \frac{\pi}{8} = 2,4142.... \approx 2^1 + 2^{-2} + 2^{-3} + 2^{-5} = 2,40625. \tag{3.4}$$

gearbeitet. Je nach Zugehörigkeit zu einem der vier Richtungsbereiche erfolgt die Zuweisung eines entsprechenden diskreten Wertes von $0-3$ über einen Multiplexer, und wird an das Submodul `directionquad` weitergeleitet.

Mit den Vorzeichen der Sobel-Pixel wird im Submodul `directionquad` bestimmt, in welchen der vier Quadranten die Zuweisung erfolgt. Diese eindeutige Zuordnung wird mit logischen UND-Bausteinen und Invertern realisiert. Wiederum über einen Multiplexer erfolgt nun die endgültige Zuweisung in einen der 16 Richtungsbereiche. An den Eingängen des Multiplexers liegt das Ausgangssignal des Submoduls `direction4` zu welchem je nach der Zugehörigkeit zum jeweiligen Quadranten eine feste Zahl addiert bzw. subtrahiert wird. Man mache sich dies nochmals Anhand den Abb. 3.8(b), 3.8(c) und 3.7(a) klar.

**Parametrisierung** Der $tan\frac{\pi}{8}$ kann im Modul `direction4` entweder durch die Zahl $2$ oder $2,40625$ genähert werden, siehe Gl. 3.4. Die näherungsweise Multiplikation für $tan\ \frac{\pi}{8}$ ist hierbei aus Addierern einem Links-Shift- und mehreren Rechts-Shift-Operationen zusammengesetzt. Die Anzahl von 16 Richtungen ist fest vorgegeben.

### Diskretisierung in acht Richtungsbereiche

Jeweils zwei der 16 Richtungsbereiche werden gemäß Abb. 3.7(a) zusammengefasst. Jedes Kantenpixel aus einem dieser acht Richtungsbereichen wird immer genau zwei Richtungsbereichen zugewiesen, siehe Tab. 3.1. Das Modul `demultiplex` übernimmt diese Zuordnung, welches in Abb. 3.9 dargestellt ist.

Die Zuordnung wird durch 3-Bit Funktionsgeneratoren realisiert, die eine beliebige logische Funktion repräsentieren. Diese wird durch den *FValue*-Wert des FGs festgelegt. Die *FValue*-Werte für die einzelnen Richtungsbereiche sind in Tab. 3.1 angegeben. Die 1-Bit Output-Signale der FGen werden nur dann weitergeleitet, wenn ein Kantenpixel am Eingang des Moduls anliegt. Um die Gatterlaufzeiten zwischen zwei Registern gering zu halten wurde nach jeden der acht UND-Bausteinen ein FF geschaltet. Jeweils einer der acht Ausgänge $O1, \ldots, O8$ wird mit dem entsprechenden Initialisierungs-Modul `init_DT` der nachfolgenden Distanztransformation verbunden.

**Parametrisierung** Die Anzahl der 16 Richtungen und Art der Zuweisung in die acht Richtungsbereiche sind nicht veränderbar.

[7] $tan\ \frac{\pi}{4} = 1.$

| Richtungsbereich | hexadezimal | binär |
|---|---|---|
| 0 | 0x81 | 1000 0001 |
| 1 | 0x03 | 0000 0011 |
| 2 | 0x06 | 0000 0110 |
| 3 | 0x0C | 0000 1100 |
| 4 | 0x18 | 0001 1000 |
| 5 | 0x30 | 0011 0000 |
| 6 | 0x60 | 0110 0000 |
| 7 | 0xC0 | 1100 0000 |

**Tabelle 3.1:** Zuweisung eines Kantenpixels über die *FValue*-Werte der Funktionsgeneratoren in zwei Richtungsbereiche.

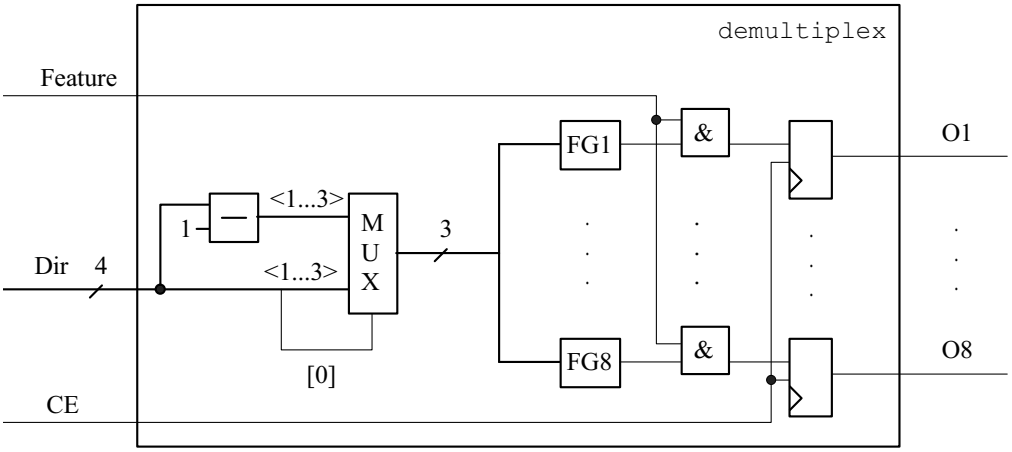



**Abbildung 3.9:** Übersicht über das Modul `demultiplex`.

## 3.5 Distanztransformation

Zur Approximation der Euklidischen Distanztransformation (EDT) werden die *chamfer-2-3* Metrik und die sequentielle Berechnungsmethode benutzt, die in Abschnitt 2.1.3 und [49] beschrieben wurden. Am Ende dieses Abschnitts wird eine Ressourcenabschätzung für eine parallele Implementierungs-Strategie angegeben.

### Sequentielle Implementierung

Eine nichtsymmetrische Maske in Vorwärts- und Rückwärts-Richtung, wie in Abb. 3.10 oben dargestellt, wird fest im FPGA "verdrahtet" und das Bild wird dann zeilenweise zunächst in Vorwärts- und dann in Rückwärts-Richtung unter der Maske hindurch geschoben. Alle DTs der Kantenbilder der acht Richtungsbereiche werden parallel berechnet. Hierfür werden acht identische DT-Module `disttransf3` auf dem FPGA implementiert. Diese können dann sowohl für die Berechnung in Vorwärts- als auch Rückwärts-Richtung verwendet werden, da beide Masken spiegelverkehrt sind. Für beide Masken wird das zentrale Pixel der Maske (links) mit dem Minimum der drei Pixel der vorausge-

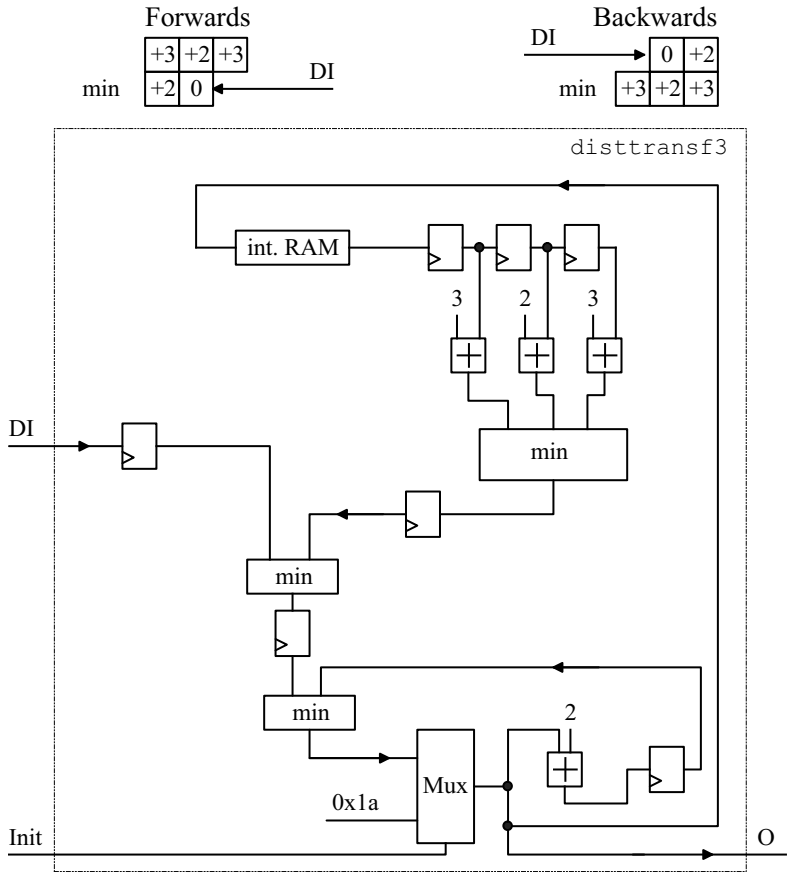



**Abbildung 3.10:** Pipeline der Distanztransformation `disttransf3`.

henden Zeile, zu denen die Distanzwerte $3, 2, 3$ hinzuaddiert wurden (oben), verglichen. Das Minimum beider wird dann mit dem benachbarten Pixel verglichen, welches sich im Register in Abb. 3.10 rechts unten befindet, nachdem dieses um 2 erhöht wurde. Das Minimum hiervon, welches das Ergebnispixel der DT ist, wird dann erstens für die Berechnung des Ergebnispixels der DT im nächsten Takt wieder im Register rechts unten gespeichert nachdem es um zwei inkrementiert wurde, zweitens ins interne FPGA-RAM geschrieben, da es zur Berechnung in der nächsten Zeile Verwendung findet und drittens an das nachfolgende Modul weitergegeben.

Außerdem werden die Randpixel einer speziellen Randbehandlung unterworfen. Um zu vermeiden, dass sich die Distanzen über den Rand hinweg fortpflanzen, werden die Randpixel auf einen hinreichend großen Wert gesetzt. Dies geschieht durch den in Abb. 3.10 eingezeichneten Multiplexer, der diesen sicheren Wert sowohl für die Berechnung im nächsten Takt, der nächsten Zeile und an den Ausgang des DT-Moduls liefert. Insgesamt geht bei dieser Realisierung an jedem Rand die Information von genau einem Pixel verloren.

Die interne Berechnung der DT-Pixel erfolgt mit 5-Bit *Integer*-Genauigkeit. Die Ergebnisse nach der Berechnung in Vorwärts- und Rückwärts-Richtung werden auf vier Bit abgeschnitten. Dies wird durch das Modul `clip_DT` realisiert, welches in Abb. 3.11(b) dargestellt ist und direkt dem DT-Modul nachgeschaltet wird. Somit können die acht

DT-Pixel, die den acht Richtungsbereichen entsprechen, in einem 32-Bit Datum in *eine* externe SRAM-Bank des FPGA-Koprozessors geschrieben werden. Als sicherer Clip-Wert wurde die Zahl 26 (hexadezimal 0x1a) gewählt, weil somit sichergestellt wird, dass der max. Wertebereich von 31 durch die auszuführenden Additionen nicht überschritten wird.

Außerdem werden bei der Berechnung der DT in Vorwärts-Richtung das binäre Ausgangssignal des `demultiplex`-Moduls zur Initialisierung des DT-Moduls in 5-Bit Werte umgewandelt. Dies wird im Modul `init_DT` realisiert, welches in Abb. 3.11(a) dargestellt ist. Die Kantenpixel werden mit 0 (als 5-Bit Zahl) initialisiert und die Nicht-Kantenpixel werden auf die Zahl 26 (hexadezimal 0x1a) gesetzt. Bei der Berechnung der DT in Rückwärts-Richtung werden die DT-Pixel direkt aus dem SRAM an das DT-Modul geleitet.

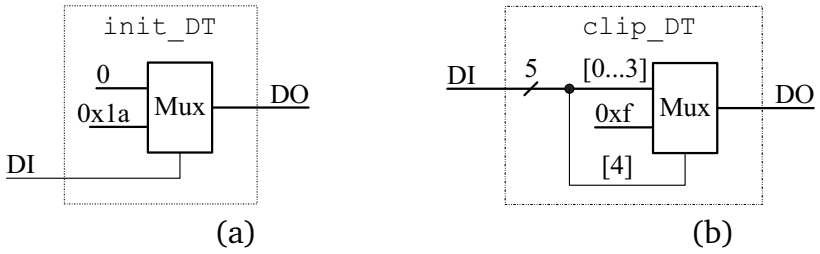



**Abbildung 3.11:** Übersicht über die Submodule (a) `init_DT` und (b) `clip_DT`.

Das DT-Modul wurde nach dem Pipeline-Prinzip aufgebaut und kann pro Takt mit einem Pixel des `demultiplex` Moduls initialisiert werden. Nach einer Latenzzeit von $W+2$ Takten liegt ein DT-Pixel am Ausgang an. Die Berechnung der DT führt sowohl in Vorwärts- als auch Rückwärts-Richtung zu einem Datendurchsatz von einem Pixel pro Takt und für beide Richtungen zu einem Datendurchsatz von $\frac{1}{2}$ Pixel pro Takt.

**Ressourcenverbrauch** Zur Realisierung eines DT-Moduls auf dem FPGA werden insgesamt fünf Register zum Speichern der Daten benötigt, was der Maskengröße entspricht. Die beiden anderen Register haben die Aufgabe die Anzahl der Logikstufen in der Pipeline zu begrenzen. Des Weiteren kommen vier Addierer und vier Minimierer zum Einsatz. Dies entspricht der Anzahl der Additionen bzw. Vergleichsoperationen die bei der Berechnung der Maske durchzuführen sind. Der Bedarf an FPGA-RAM zum Zwischenspeichern der DT-Pixel beträgt eine Zeile der Länge $W-6$. Dieser ist etwas weniger als eine Zeilenlänge, weil einige Pixel schon in der Pipeline zwischengespeichert bzw. weiterverarbeitet werden. Zusätzlich wird für die Randbehandlung ein Multiplexer eingesetzt. An den Eingag von jedem DT-Modul wird ein `init_DT`-Modul und an dessen Ausgang ein `clip_DT`-Modul geschaltet.

### Parallele Implementierung

Der alternative parallele Ansatz zur Berechnung der DT wurde im Rahmen dieser Arbeit nicht implementiert, da die Anforderungen an die Ressourcen des FPGAs wesentlich

höher sind. Dies soll im Folgenden kurz begründet werden. Der Ressourcenbedarf für ein paralleles DT-Modul gegenüber dem sequentiellen DT-Modul würde sich etwas weniger als verdoppeln. Entsprechend der Maskengröße, siehe Abb. 2.6, würden die Pixel in neun anstelle von fünf Registern gespeichert werden. Der Bedarf an internem FPGA-RAM würde sich von einer Zeile auf zwei Zeilen und der an Minimierern (mit zwei Eingängen) von vier auf acht verdoppeln.

Allerdings sind zur Berechnung eines DT-Bildes mehr als ein paralleles DT-Modul notwendig. Um sicherzugehen, dass bei einem max. Wertebereich der 4-Bit Zahlen, d.h. einem Wertebereich von 0 bis 15, keine Veränderungen im DT-Bild mehr auftreten, müssen bei der *chamfer*-2-3 Metrik die parallelen DTs acht Mal hintereinander ausgeführt werden. Bei zwei Durchläufen, wie bei der sequentiellen Methode, wären dann vier parallele hintereinander geschaltete DT-Module notwendig. Möchte man das Bild in einem Durchlauf berechnen, müssen acht parallele DT-Module hintereinander geschaltet werden. Hierzu sind für die acht Richtungsbereiche insgesamt 64 parallele DT-Module, notwendig. Um eine doppelt so schnelle Ausführungsgeschwindigkeit gegenüber der sequentiellen Methode zu erreichen, ist also insgesamt etwas weniger als ein 16-facher Ressourcenbedarf notwendig.

**Parametrisierung** Beim Modul `disttransf3` besteht die Möglichkeit, während der Ausführung des Designs auf dem FPGA die Bildbreite $W$ zu ändern und zwischen der *chamfer-2-3* und *chamfer-1-1* Metrik auszuwählen. Hierzu können vom Anwender über ein Codewort, siehe Tab. 4.3, Register beschrieben werden, in denen die Koeffizienten der Maske gespeichert sind und die direkt mit den Eingängen der entsprechenden Addierern verbunden sind. Die Wahl von größeren Koeffizienten für die DT-Metrik macht keinen Sinn, weil die ALU des DT-Moduls hierfür nicht dimensioniert ist. Das Modul `disttrans3` ist ebenfalls nach der Bit-Breite der Eingangsdaten parametrisierbar. Die Größe der sequentiellen DT-Maske ist jedoch nicht veränderbar.

**Bisherige Arbeiten** Ein in [69] beschriebener Ansatz für die Berechnung der sequentiellen DT in Vorwärts- und Rückwärts-Richtung für FPGAs arbeitet mit sog. *Prozessing*-Elementen, welche in jedem vierten Takt ein DT-Pixel liefern. Die Maskenkoeffizienten für jedes PE sind in LUTs gespeichert und jedes PE arbeitet (wie in der in diesem Abschnitt beschriebenen Implementierung) auf fünf Registern. Mehrere *Prozessing*-Elemente können gleichzeitig ausgeführt werden wobei sich die Verarbeitungsgeschwindigkeit entsprechend erhöht. Der FPGA-Ressourcenbedarf für die in [69] beschriebene Implementierungsstrategie lässt sich schwierig mit dem Ressourcenbedarf für die in diesem Abschnitt vorgestellten Implementierung vergleichen, weil in den PEs zusätzlich die Sobel-Pixel in x- und y-Richtung und eine Binarisierung berechnet wird.

## 3.6 Datenfluss und Steuerung

Die Beschreibung des Datenflusses und des Steuerungsmoduls `controlDT` erfolgt für das MPRACE-Board, siehe Abschn. 1.3.2, welches über mehrere unabhängige externe SRAM-Bausteine verfügt.

### Datenfluss

Zunächst werden die Bilddaten der Kamera mit dem `camctrl`-Modul, siehe Abb. 3.2 in das SRAM 0 des MPRACE-Boards geschrieben. Jeweils vier benachbarte Pixel sind in einem Langwort abgespeichert. Parallel hierzu kann ein mit dem `downsample2`-Modul reduziertes Bild in einen anderen Adressbereich gespeichert werden, wobei ebenfalls vier Pixel in einem Datum zusammengefasst sind. Die Berechnungen der DT-Bilder können dann zunächst auf dem ursprünglichen Bild und anschließend auf dem reduzierten Bild jeweils unabhängig voneinander erfolgen.

Die Module der Vorverarbeitungs-Schritte (*Erkennen*) können in zwei Pipelines, siehe Abb. 3.12 und 3.13, zusammengefasst werden. In der ersten Phase werden die Bild-

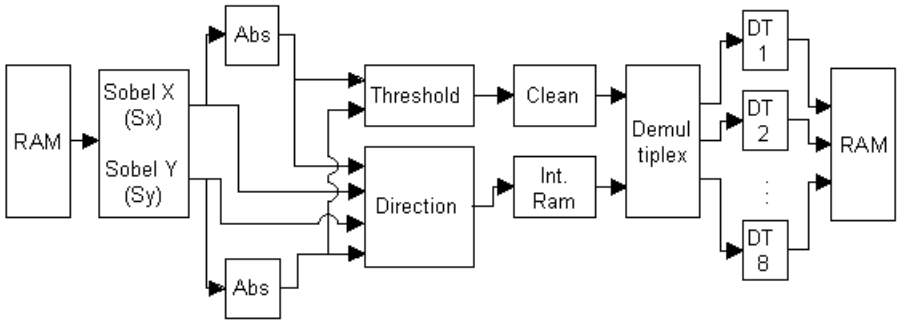



**Abbildung 3.12:** Pipeline 1 der Merkmalsextraktion mit DT in Vorwärts-Richtung.

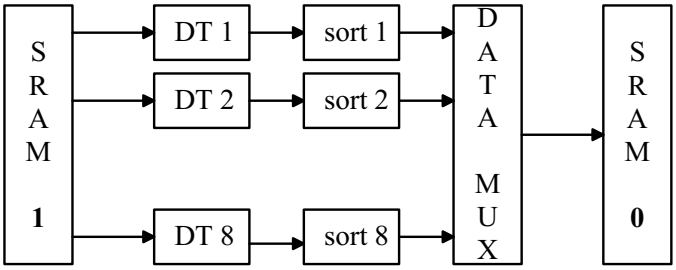



**Abbildung 3.13:** Pipeline 2 der Merkmalsextraktion mit DT in Rückwärts-Richtung.

Daten aus SRAM 0 ausgelesen - in jedem vierten Takt ein 32-Bit Datum. Die vierfach parallelen Daten werden von einem *Parallel-zu-Seriell*-Modul entpackt (nicht in Abb. 3.12 eingezeichnet), so dass der ersten Pipeline pro Takt genau ein Pixel zugeführt wird. Einerseits wird nun ein binäres Kantenbild berechnet (Sobel in x- und y-Richtung, Binarisierung, *Cleaning*) und andererseits jedem Pixel einen Richtungsbereich zugewiesen. Da der erste Zweig eine längere Verarbeitungszeit hat als der zweite, werden die Daten des zweiten Zweigs im internen BlockRAM des FGPAs zwischengespeichert. Jeder Kantenpixel initialisiert genau zwei der acht DT-Module. Die Ergebnisse der Hintransformation

der acht DT-Module werden parallel in einem Datenwort im SRAM 1 abgespeichert. In der zweiten Phase werden diese Zwischenergebnisse in Rückwärts-Richtung aus dem SRAM 1 ausgelesen und die Berechnung der DT abgeschlossen. Die DT-Ergebnisse werden in einer speziellen Anordnung im SRAM 0 des MPRACE-Boards gespeichert. Die `sort`-Module, welche *Seriell-zu-Parallel*-Module sind, sorgen dafür, dass jeweils acht benachbarte DT-Pixel aus einem Richtungsbereich in einem Datenwort gespeichert werden. Dies ist für eine effiziente Berechnung des nachfolgenden Template-Matching notwendig. Die Latenzzeiten und der Ressourcenbedarf der einzelnen Module bzw. der Pipelines sind in Tab. 3.3 angegeben.

### Steuerung

Das Steuerungsmodul, siehe Abb. 3.14, ist für zwei unabhängige SRAM-Bausteine des MPRACE-Boards konzipiert[8]. Das Modul `ControlDT` übernimmt alle Aufgaben zur Steuerung der Teilmodule, d.h. es werden alle eingehenden Steuersignale für alle datenverarbeitenden Teilmodule (`sobel`, `cleaning`, `direction`, `demultiplex`, `disttransf3`, `sort`) generiert. Außerdem übernimmt es die Kommunikation über den PCI-Bus mit dem Host-Rechner. Es empfängt die Signale *Start*, *CE* und *Reset* vom Host-Rechner und generiert das Flag-Signal *End*, welches einen Takt nach dem Abspeichern des letzten Ergebnispixels der DT-Bilder im SRAM 0 aktiviert wird.

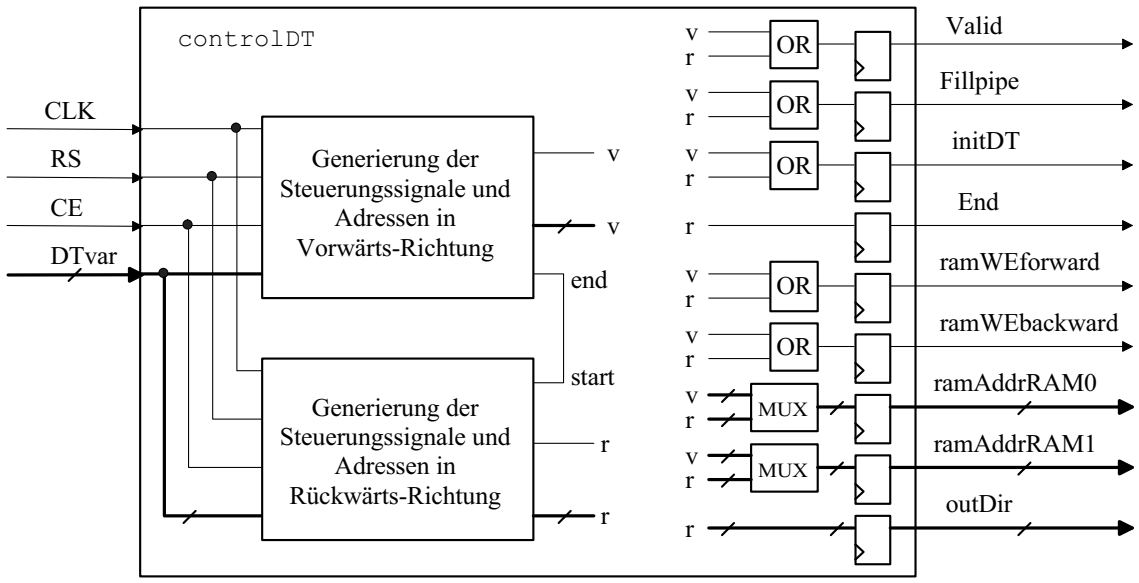



**Abbildung 3.14:** Steuerungsmodul zur Berechnung der DT-Bilder (Merkmalsextraktion).

Das *Valid*-Signal des `controlDT`-Moduls, welches die Gültigkeit der Daten anzeigt, wird mit den *CE*-Eingängen aller datenverarbeitenden Module verbunden. Somit arbeiten diese Module synchron zu einem globalen *Valid*-Signal. Dies hat zur Folge, dass die dem

[8]Es wird darauf hingewiesen, dass ein Steuerungs-Modul basierend auf einer *State-Machine* existiert, welches für *einen* externen SRAM-Baustein ausgelegt ist und für den *microEnable*-Koprozessor konzipiert wurde. Die *State-Machine* wird jedoch von der aktuellen CHDL-Version (1.106) nicht unterstützt.

Sobel-Modul nachfolgenden Module schon Daten verarbeiten, bevor diese Gültigkeit haben. Dies stellt jedoch kein Problem dar, weil nur die gültigen Ergebnisse nach der DT gespeichert werden, die zuvor berechneten ungültigen jedoch nicht. Alternativ hätte für jedes datenverarbeitende Modul ein eigenes *Valid*-Signal erzeugt werden müssen, was zu einem höheren Ressourcenaufwand geführt hätte.

Bei der Berechnung der Distanztransformation steuert das *InitDT*-Signal den Multiplexer im DT-Modul `disttransf3`, sowie das am Dateneingang befindliche Initialisierungs-Modul, siehe Abb. 3.10.

Die Kontrolle der beiden externen SRAM-Bausteine erfolgt über die folgenden *RAM*-Signale. Die Lese-Adressen für SRAM 0, die beim Füllen der Pipeline 1 benötigt werden, liegen am *ramAddrRam0*-Signal, die für SRAM 1 der Pipeline 2 am *ramAddrRam1*-Signal. Für die Schreib-Adressen gilt genau umgekehrtes. Das *WE*-Signal *ramWEforward* ist mit SRAM 1, das *ramWEbackward*-Signal mit SRAM 0 verbunden. Mit dem *outDir*-Signal werden die `sort`-Module, die den Datenstrom konvertieren, und den nachfolgenden Multiplexer, siehe Abb. 3.13, gesteuert. Das Ende der Berechnung der DT-Bilder wird durch das *End*-Signal angezeigt. Es dient als Flag-Signal und ist über einen Multiplexer direkt mit dem PCI-Slave-Modul verbunden. Einen Überblick über die Gesamtschaltung incl. Template-Matching wird in Abb. 4.9 gegeben.

Auf die interne Generierung der Signale und RAM-Adressen soll hier nur kurz eingegangen werden. Diese werden für Vorwärts- und Rückwärts-Richtung in unterschiedlichen Submodulen generiert. Hierzu wurde ein sog. verallgemeinertes Zähler-Modul `CHDL_for` implementiert, das ähnlich einer `for(...)` Schleife in C oder C++ zu parametrisieren und das aus ladbaren Zählern, Komparatoren und Steuer-Logik aufgebaut ist. Die Parameter der Submodule können auch während der Ausführung des Designs verändert werden.

Am Ausgang des Steuerungsmoduls `controlDT` werden die unterschiedlichen intern generierten Steuersignale für die Vorwärts- und Rückwärts-Richtung über logische OR-Bausteine bzw. Multiplexer zusammengeschaltet. Vor alle ausgehenden Signale werden Register geschaltet um die Anzahl der Logikstufen gering zu halten.

**Parametrisierung** Das Steuerungsmodul `controlDT` ist parametrisierbar bezüglich der Bildbreite $W$ und -höhe $H$. Diese Parameter sind auch während der Laufzeit des Designs veränderbar. Die Zähler und Komparatoren in den Submodulen des `controlDT`-Moduls werden dann mit den max. Bildhöhen und -breiten, mit denen man arbeiten möchte, aufgebaut. Über das Signal `DTVar` können diese Größen während der Laufzeit angepasst werden. Hierzu sind die Codewörter, die in Tab. 4.3 angegeben sind, zu benutzen. Des Weiteren kann der Adressbereich aus dem das Bild bzw. das reduzierte Bild ausgelesen wird während der Laufzeit verändert werden.

## 3.7 Ergebnisse und Zusammenfassung

Alle datenverarbeitenden Module zur Berechnung der DT-Bilder konnten nach dem Pipeline-Prinzip aufgebaut werden. Den Modulen können pro Takt genau ein Pixel bzw. Zwi-

schenergebnis zugeführt werden. Insgesamt konnten aus den Modulen zwei Pipelines aufgebaut werden, die in den Abb. 3.12 und 3.13 dargestellt sind. Für die Berechnung der DT in Vorwärts- und Rückwärts-Richtung konnten dieselben Module verwendet werden. Der Datendurchsatz liegt für jede der beiden Pipelines bei einem Pixel pro Takt und für beide zusammen bei $\frac{1}{2}$ Pixel pro Takt. Somit konnte erreicht werden, dass sich für beide Pipelines eine konstante Verarbeitungsgeschwindigkeit ergibt, die unabhängig vom Bildinhalt ist und linear mit der Größe des Bildes anwächst. Dies ist für sicherheitsrelevante Anwendungen wie z.B. im Automobilbereich von großem Vorteil.

Sowohl für die datenverarbeitenden Module als auch für das Steuerungsmodul `controlDT` konnte ein hohes Maß an Flexibilität erreicht werden. Während der Laufzeit des Designs können die Bildbreite $W$ und -höhe $H$ variiert werden, der jedoch auf einen max. zulässigen Wert von 512 begrenzt ist. Diese Funktionalität konnte bei allen betreffenden Modulen der beiden Pipelines realisiert werden. Es wurde des Weiteren sichergestellt, dass die Kamerabilder aus beliebigen Adressbereichen ausgelesen werden können und diese auch während der Ausführung des FPGA-Designs variierbar sind. Somit wird die Ausführung des Algorithmus auf dem FPGA für Bilder möglich, die unterschiedlichen Skalenräumen angehören, siehe Abschn. 1.1.2. Außerdem ist während der Ausführung des Designs der globale Schwellwert $\theta$ im Modul `threshold`, sowie die Anzahl der zu isolierenden Pixel im Modul `cleaning` und die Metrik im `disttransf3`-Modul veränderbar. Hierbei kann zwischen der *chamfer*-1-1 und *chamfer*-2-3 Metrik gewählt werden. Die Codewörter zum Schreiben der jeweiligen Parameter sind in Tab. 4.3 zu finden.

Das Design der Gesamtschaltung wurde mit der von K. Kornmesser [70] an der Universität Mannheim entwickelten Hardwarebeschreibungssprache CHDL (Version 1.106) erstellt, siehe Abschn. 1.3.3. Neben den einzelnen Modulen wurde die Gesamtschaltung sowie deren Anbindung an den Host-PC simuliert. Für das Kameramodul stand jedoch kein Simulationsmodell zur Verfügung. Die Erzeugung der Netzliste (Synthese) erfolgte ebenfalls mit CHDL. Der Bitstream wurde mit P&R-Software (Version 3.1) von Xilinx für den Virtex-II XC2V3000 FPGA generiert.

### 3.7.1 Geschwindigkeit und Datendurchsatz

Ziel der Implementierung war die Echtzeitfähigkeit des Systems, d.h. eine Bildwiederholrate von ungefähr 30 Hz zu gewährleisten. Dies sollte zunächst für Bilder der Größe $288 \times 360$ und später für Bilder bis zu einer Größe von $1024 \times 1024$ möglich sein. Sozusagen als Mittelwert sind in dieser Arbeit Ergebnisse für Bilder der Größe $512 \times 512$ angegeben. Der FPGA-Ressourcenbedarf wird im Folgenden ausschließlich für den Virtex-II XC2V3000 FPGA diskutiert.

Die max. Frequenz mit der das Gesamtdesign gespeist werden kann ist durch den längsten Datenpfad begrenzt, der beim Steuerungsmodul `controlDT` liegt. Das Design der Gesamtschaltung bestehend aus Kameramodul, den beiden Pipelines der Vorverarbeitung und der Steuerung kann mit einer Frequenz von 98 MHz getaktet werden. Bei einer Verarbeitungsgeschwindigkeit von ungefähr einem $\frac{1}{2}$ Pixel/Takt ergibt sich für die Berechnung eines Bildes der Größe $512 \times 512$ eine theoretische Rechenzeit von $5.4$ ms. Das Design wird mit der *Local-Bus*-Clock des MPRACE-Boards gespeist. Auf die Verwendung eines DCMs (Digital Clock Managers) im Design wurde verzichtet. Daher kann das

Design nur mit einer max. Taktfrequenz der *Local-Bus*-Clock von 64 MHz betrieben werden[9]. Somit ergibt sich eine Rechenzeit von $8.0$ ms die mit der gemessenen von $8.1$ ms nahezu übereinstimmt, siehe auch Tabelle 3.2. Zu beachten ist, dass in der theoretischen Rechenzeit die Latenzzeiten der Pipelines nicht berücksichtigt sind. Die Berechnung der acht DT-Bilder führt bei einer Bildgröße von $512 \times 512$ zu einer Bildwiederholrate von $122$ Hz.

| | MPRACE [ms] | PC [ms] |
|---|---|---|
| Kamera | min. 17 | min. 17 |
| Schreiben der Codewörter | <0.1 | - |
| DT-Bilder | 8.1 | 350 |
| $\sum$ DT-Bilder | 8.2 | 350 |
| $\sum$ Kamera + DT-Bilder | min. 25.2 | min. 367 |

**Tabelle 3.2:** Ausführungszeiten für Bildaufnahme (*Sehen*) und Erzeugung der DT-Bilder (*Erkennen*) der Größe $512 \times 512$ für MPRACE-Koprozessor und Standard-PC.

Für die Berechnung der reduzierten Bilder der Größe $256 \times 256$, siehe Abschn. 1.1.2, werden $2.1$ ms benötigt. Werden die DT-Bilder für Bilder in Originalgröße und zusätzlich für reduzierte Bilder erzeugt, so wird hierfür insgesamt eine Ausführungszeit von $10.3$ ms benötigt mit einer Bildwiederholrate von $97$ Hz. Nimmt man die Bildaufnahme mit der in Abschn. 1.3.5 beschriebenen Kamera TM-1040 von Pulnix hinzu, so ist eine max. Bildwiederholrate[10] von $40$ Hz und eine minimale von $17$ Hz möglich.

Die Ergebnisse für die Rechenzeit werden mit denen verglichen, die mit einem C-Code von D. Gavrila erzeugt wurden, der jedoch keine SIMD-Optimierungen enthält. Die Kompilierung des Quellcodes wurde mit dem Visual C++ Studio (Version 6.0) durchgeführt. Ein PC mit Pentium-III 833 MHz und 1 GByte SDRAM-133 benötigt für die Berechnung der Kantenbilder einschließlich Orientierungsinformation ungefähr $84$ ms und für die acht DTs ungefähr $266$ ms, insgesamt $350$ ms. Somit führt die FPGA-Implementierung für die Berechnung der acht DT-Bilder gegenüber einer Software-Implementierung zu einer Beschleunigung um einen Faktor $43$.

### 3.7.2 Ressourcenverbrauch

Der Ressourcenverbrauch für die einzelnen Module zur Berechnung der acht DT-Bilder und das Kameramodul sind in Tabelle 3.3 für den Virtex-II FPGA XC2V3000 des MPRACE-Koprozessors angegeben.

Von den datenverarbeitenden Modulen sind das Sobel- und *Cleaning*-Modul sowie das DT-Modul ressourcenintensiv, welches acht Mal im FPGA reproduziert wird. Das Steuer-

[9] Nach der Integration des Designs für das Template-Matching in ein Gesamtdesign wird sich zeigen, dass der Verzicht auf das DCM gerechtfertigt ist, weil dann die max. Taktfrequenz bei 68 MHz liegt, also knapp oberhalb der möglichen 64 MHz der *Local-Bus*-Clock des MPRACE-Koprozessors.

[10] Im Rahmen dieser Arbeit wird nicht bereitgestellt, dass der Aufnahmevorgang über das asynchrone Reset der TM-1040 Kamera jederzeit neu gestartet werden kann. Die max. Zeit für die Aufnahme von einem Bild der Größe $512 \times 512$ kann daher ca. $50$ ms betragen.

| Modul | Slices | (in %) | Block-RAM | Latenzzeit |
|---|---|---|---|---|
| `camctrl` | $\approx 170$ | (1) | - | $\approx W\,H$ |
| `sobel` | $\approx$140 | (1) | 1 | $2W+6$ |
| `abs` | $\leq$ 2*8 | (0) | - | 1 |
| `threshold` | $\leq$6 | (0) | - | 1 |
| `cleaning3` | $\approx$155 | (1) | 1 | $4W+7$ |
| `direction` | $\leq$60 | (0) | - | 2 |
| `demultiplex` | $\leq$35 | (0) | - | 2 |
| `disttrans3` | $\approx$8 * 65 | (4) | 2 | $1W+2$ |
| `sort` | $\approx$8 * 33 | (2) | - | $\approx 4$ |
| `ControlDT` | $\approx$340 | (3) | - | - |
| $\sum$ DT-Bilder | $\approx$1710 | (12) | 4 | $7W+25$ |
| $\sum$ `camctrl` + DT-Bilder | $\approx$1880 | (13) | 4 | $\approx W\,H\,+\,7W+25$ |

**Tabelle 3.3:** Ressourcenbedarf und Latenzzeiten für die Module der Merkmalsextraktion für den XC2V3000-FPGA von Xilinx, für Bilder der Breite $W$ und Höhe $H$ und 8-Bit Eingangsdaten.

ungsmodul `controlDT` ist ebenfalls relevant, besonders dann, wenn die Parameter während der Laufzeit veränderbar sind. Dies liegt an den vielen Registern, in denen die Parameter abgespeichert werden müssen, sowie an der hierfür notwendigen Logik. Außerdem sind alle relevanten Submodule während der Laufzeit veränderbar, was sich auf den FPGA-Ressourcenverbrauch durchschlägt.

Der Ressourcenverbrauch der datenverarbeitenden Module ist weitgehend unabhängig von der Parametrisierung der Module. Jedoch ist der Bedarf an internem FPGA-RAM, in welchem insgesamt 15 Bildzeilen zwischengespeichert werden von der Bildbreite $W$ abhängig. Dies sind zwei Bildzeilen mit jeweils acht Bit für das Sobel-Modul, vier Bildzeilen mit jeweils einem Bit für den *Cleaning*-Modul und parallel hierzu vier Bit für die Ergebnisse des `direction`-Moduls, sowie acht Zeilen mit jeweils fünf Bit für die acht DT-Module.

Insgesamt führen alle Module zur Berechnung der acht orientierten DT-Bilder zu einem Ressourcenbedarf für den XC2V3000 von ungefähr 12 %. Auf dem Virtex-II bleibt somit ausreichend Platz für die Berechnung des Template-Matchings.

### 3.7.3 Ausblick

Die Ausführungszeiten für die Bildaufnahme und Berechnung der DT-Bilder können verringert werden, falls das Kameramodul `camctrl` und die erste Pipeline zur Berechnung der acht DT-Bilder zu einer größeren Pipeline zusammenfügt werden. Die Berechnung der DT-Bilder würde dann schon während der Bildaufnahme stattfinden. Hierzu muss das Steuerungsmodul `controlDT` so angepasst werden, dass es in Abhängigkeit der Steuersignale des Kameramoduls arbeiten kann. Zusätzlich muss dafür gesorgt werden, dass die Pipeline, nach dem letzten gültigen Pixel aus dem Kameramodul so lange weiterarbeitet bis sie leer ist. Parallel hierzu wird das Bild im externen SRAM des MPRACE-Koprozessors zwischengespeichert.

Eine weitere Verringerung der Ausführungszeit auf dem FPGA kann erreicht werden, falls man die sequentielle Methode zur Berechnung der DT durch die parallele ersetzt. Insgesamt könnte das Kameramodul und die Pipeline mit der parallelen DT zu einer großen Pipeline integriert werden und somit in einem Durchlauf berechnet werden. Dies würde jedoch für die DT einen ca. 16-fachen FPGA-Ressourcenaufwand bedeuten.

Das Design wird momentan mit der *Local-Bus*-Clock des MPRACE-Prozessors gespeist, die mit max. 64 MHz betrieben werden kann. Durch Integration eines DCMs (Digital Clock Managers) in das Design könnte dieses mit einer höheren Frequenz von bis zu 125 MHz getaktet werden.

Kapitel 4

# Implementierung des Template-Matchings

Nachdem im Kapitel 3 die FPGA-Implementierungsdetails für die ersten beiden Stufen des Algorithmus, dem *Sehen* und dem *Erkennen* beschrieben wurden, folgt in diesem Kapitel eine Beschreibung der Implementierung für das Template-Matching, welches Teil der dritten Stufe, dem *Entscheiden* ist.

Zunächst wird in Abschn. 4.1 ein kurzer Überblick über bisherige Arbeiten auf dem Gebiet Template-Matching für FPGAs gegeben. Anschließend werden in Abschn. 4.2 zwei elementare Implementierungsstrategien vorgestellt, die jedoch im Rahmen dieser Arbeit nicht realisiert wurden. In den folgenden Abschnitten wird die vom Autor vorgeschlagene FPGA-Implementierungsstrategie des Template-Matchings beschrieben. In Abschn. 4.3 wird ein Überblick über die datenverarbeitenden Module gegeben, die i.W. aus SRAs (Shift Register Arrays) und Addiererbäumen bestehen. Das Steuerungsmodul und der Datenfluss wird anschließend in Abschn. 4.3.3 vorgestellt. Eine optimale Handhabung der Daten der acht DT-Bilder konnte realisiert werden, so dass sich keines der datenverarbeitenden Module im Wartezustand befindet. Außerdem wird beschrieben, wie der gesamte Algorithmus des Template-Matchings auf dem in Abschn. 1.3.5 vorgestellten hybriden Bildverarbeitungssystem, bestehend aus FPGA und Host-PC, aufgeteilt wird. Eine Zusammenfassung und Diskussion der Ergebnisse ist in Abschn. 4.4 zu finden.

Der in Abschn. 4.3 vorgestellte parallele Ansatz führt zu einem hohen FPGA-Ressourcenbedarf. Dieser kann reduziert werden, falls die Templates so verschoben werden, dass sich möglichst viele der Templateelemente überdecken und viele der gemeinsam auszuführenden Teilsummen nur einmal berechnet werden. Optimierungsstrategien zum Aufbau von optimierten SRAs und optimierten Addiererbäumen und zum Verschieben der Templateelemente werden in Kapitel 5 diskutiert.

## 4.1 Bisherige Arbeiten

In diesem Abschnitt wird ein Überblick über bisherige FPGA-Implementierungen für das Template-Matching gegeben. Dabei ist festzustellen, dass die Implementierungsstra-

tegien sehr stark von dem Einsatzgebiet und den Anforderungen an die Anwendung abhängen. Unterscheidungskriterien der Ansätze sind, ob die Templates während der Ausführung des Designs veränderbar oder fest sein sollen oder ob ein nach einem Muster oder mach mehreren Mustern im Bild gleichzeitig gesucht wird. Ein weiteres Unterscheidungsmerkmal ist die Art der Daten, auf denen das Matching durchgeführt wird.

### 4.1.1 Transformationsbasiertes Template-Matching

Ein Ansatz eines Matchings für FPGAs basierend auf der Hough-Transformation [13] wurde u.A. von Klefenz et al. [73] durchgeführt. Die Anwendung besteht darin, dass die Parameter von sehr vielen möglicherweise auftretenden Pion Spuren, die von einem TR[1]-Detektor geliefert werden, in sehr kurzer Zeit bestimmt werden müssen. Die Pion Spuren, die sich über das gesamte Bild ausdehnen, werden mittels Hough-Transformation, siehe Abschn. 1.1.2, lokalisiert.

Der erste Schritt bei der Durchführung der Hough-Transformation auf der FPGA-Hardware *Enable* [73] sieht vor, die Bild-Daten einem *systolic array* zuzuführen, welches nach dem Pipeline-Prinzip aufgebaut ist und aus identischen *Prozessing*-Elementen (PEs) besteht. Die lokalen Beziehungen der binären Daten werden ausgenutzt und es wird der Pipeline in jedem Takt eine neue Spalte von Pixeln des Bildes hinzugefügt. Die für die Spurerkennung relevanten Daten werden in Registern gespeichert. Es werden Spuren mit einer festen Steigung in allen Spalten mit unterschiedlichen Offsets untersucht. Jedes PE entspricht hierbei genau einem Muster. Eine Historgammierung der Muster wird durchgeführt, d.h. die Anzahl der Pixel, die zum binären Pattern gehören, wird gezählt. Falls Matches gefunden werden, werden die Zähler erhöht, einen für jeden Offset. Das Muster für jede Spur muss nur einmal im internen FPGA-RAM gespeichert werden. Durch Beschreiben des internen FPGA-RAMs können die Muster jederzeit geändert werden. Anschließend analysiert der sog. Trigger den Inhalt der Histogrammierungs-Kanäle und führt die Klassifizierung durch.

### 4.1.2 Direktes Template-Matching

In [74] wird ein rekonfigurierbarer Ansatz beschrieben bei dem jeweils ein max. $16 \times 32$ Pixel großes binäres Template gegen ein binäres Kantenbild “gematcht” wird. Über 16 Input-Pixel werden pro Takt die Daten in eine 32 stufige Pipeline geladen, die aus insgesamt 512 Registern und einer festverdrahteten, jedoch veränderbaren Korrelator-Logic besteht. Die den Templates entsprechenden Pixel der Pipeline werden mit Addiererbäumen aufsummiert. Die Entscheidung, ob ein Matching an einem Bildpunkt erfolgreich war, wird durch einen Schwellwert festgelegt.

Eine neuere Arbeit über *Shape-Adaptives*-Template-Matching (SA-TM) stammt von J. Gause [75]. Das Anwendungsgebiet ist die Analye von Video-Bildern. Die binären Templates (max. $100 \times 100$), die im FPGA-RAM gespeichert sind, sind statisch oder dynamisch veränderbar. Die Bild-Daten werden wiederum *systolic arrays* zugeführt und von *Processing*-Elementen (PEs) verarbeitet. Eine *bounding box* der Maske passt sich dabei

[1] Transition radiation.

dynamisch der Größe der Templates an während die Form der Templates in dynamisch veränderbaren PEs eingestellt wird.

Beide Ansätze haben gemeinsam, dass die Form der Templates variierbar ist, jedoch in jedem Durchlauf des Bildes nur ein Template "gematcht" werden kann.

Ein paralleler Ansatz für die Korrelationen multipler binärer Templates der festen Größe von $16 \times 16$ Pixel auf $8$-Bit Bilddaten ist in [76] zu finden. Die nichtveränderbare Templatemaske ist hierbei fest im FPGA verdrahtet und die Bild-Daten werden unter der Maske hindurch geschoben. Die den Templates entsprechenden Bilddaten, die in Registern gespeichert sind werden durch Addiererbäume, die das Herz der Verarbeitung darstellen, aufsummiert. Die Effizienz kann gesteigert werden, indem mehrere Templates auf dem FPGA kombiniert und Teile der Addiererbäume gemeinsam verwendet werden. Näheres hierzu ist in Abschn. 5.2.3 zu finden.

### 4.1.3 Analyse

Bei dem auf der Hough-Transformation basierenden Ansatz, siehe Abschn. 4.1.1, können mehrere Templates parallel berechnet werden. Zusätzlich kann die Form der Muster, die im internen RAM des FPGAs gespeichert sind, während der Ausführung des Designs geändert werden. Während bei Anwendungen in der Hochenergiephysik die Auflösung meistens nicht sehr hoch ist, liegt diese beim Template-Matching für die im Rahmen dieser Arbeit untersuchten Anwendung im Pixelbereich. Für jeden Bildpunkt im ursprünglichen Bild würde dann ein Punkt im Parameterraum benötigt, welcher z.B. den Mittelpunkt (x- und y- Koordinate) eines Templates beschreibt. Falls die verschiedenen Templates gleichzeitig berechnet werden, so wird der Parameterraum um eine dritte Dimension (Radius bzw. die Größe der Templates) erweitert. Für jeden Punkt im Transformationsraum würde genau ein Zähler benötigt, dessen Inhalt im BlockRAM des FPGA gespeichert würde. Dieser Anzatz wurde im Rahmen dieser Arbeit zum einen nicht weiter berücksichtigt, weil dessen Realisierung auf FPGAs einen sehr hohen FPGA-Ressourcenbedarf zur Folge gehabt hätte. Die Transformation in den Parameterraum erfolgt während dem Matching üblicherweise nicht über eine Transformationsgleichung, sondern über eine *Lookup*-Tabelle, in welcher die vorbestimmten Werte der Transformation gespeichert sind. Die Transformationsgleichung für Dreiecke ist komplizierter als diejenige für Kreise, weil neben der Steigung und dem Offset der Liniensegmente des Dreiecks auch deren einzelnen Längen zu bestimmen sind. Ansätze für die Hough-Transformation für Dreiecke sind dem Autor nicht bekannt. Eine weiter Nachteil bei der Hough-Transformation besteht darin, dass dessen Ausführungszeit vom Bildinhalt, genauer, von der Anzahl der Kantenpixel im Binärbild, welche stark variieren kann, abhängig ist. Dies ist bei sicherheitsrelevanten Anwendungen von großem Nachteil.

Bei den beiden in Abschn. 4.1.2 vorgestellten rekonfigurierbaren Ansätzen kann die Form der Templates während der Ausführung des Designs geändert werden. Hierzu wird zusätzliche Logik benötigt was einen zusätzlichen Ressourcenbedarf auf dem FPGA erfordert. Der wesentliche Nachteil besteht jedoch darin, dass die beiden Ansätze so konzipiert sind, dass nur ein Template pro Durchlauf gefunden werden kann. Für die im Rahmen dieser Arbeit verwendeten 36 Templates wären demnach 36 Durchläufe notwendig. Die Echtzeitbedingung für das Matching von 36 Templates auf Bildern der Größe

$512 \times 512$ mit einer Bildwiederholrate von 30 Hz wäre dann nur sehr schwierig einzuhalten gewesen.

Der parallele Ansatz aus Abschn. 4.1.2 erlaubt es, mehrere Templates, deren Form nicht variiert werden kann, gleichzeitig zu berechnen. Der Ressourcenaufwand steigt mit der Anzahl der Templates, weil sich die Ressourcen für die Berechnung des Matchings entsprechend multiplizieren. Im Rahmen dieser Arbeit wird ein paralleler Ansatz mit festverdrahteten Templates eingesetzt, weil hiermit die Anforderung an die Echtzeitbedingung erfüllt werden kann. Der parallele Ansatz ist in Abschn. 4.3 ausführlich beschrieben.

Bei der Umsetzung des in Abschn. 2.2.4 beschriebenen hierarchischen Template-Matchings [3] für FPGAs ist zu beachten, dass die Distanzmaße für die unterschiedlichen Templates nacheinander zu berechnen sind. Eine Parallelisierung, welche die gleichzeitige Berechnung mehrerer Distanzmaße vorsieht, ist möglich, falls das Matching auf mehreren Teilbildern durchgeführt wird. Dieses Vorgehen führt jedoch zu einem deutlich erhöhten Ressourcenbedarf an internem FPGA-RAM oder externen SRAM des FPGA-Koprozessors. Die oben beschriebenen Implementierungsstrategien für das Matching auf FPGAs, die alle einen geordneten Zugriff auf die Daten erlauben, sind nicht für das hierarchische Template-Matching geeignet, welches einen ungeordneten Zugriff auf die den Templates entsprechenden Pixeln zur Folge hat. Arbeiten über ein hierarchisches Matching für FPGAs sind dem Autor nicht bekannt. In Abschn. 4.2.2 wird eine elementare FPGA-Implementierungsstrategie beschrieben, welche für das hierarchische Matching geeignet ist.

## 4.2 Elementare Implementierungsstrategie

In diesem Abschnitt werden zwei elementare Implementierungsstrategien beschrieben. Die erste Implementierungsstrategie ist eine sequentielle, die einer PC-basierten Implementierung ähnlich ist. Anhand dieser werden die zeitlichen Anforderungen an das Matching mit den im Rahmen dieser Arbeit verwendeten Templates, die in Abschn. 2.2.1 angegeben sind, aufgezeigt. Bei der zweiten Implementierungsstrategie wird eine Beschleunigung des sequentiellen Matchings durch eine zeilenweise Parallelisierung erzielt. Diese ist auch für ein hierarchisches Matching geeignet. Die beiden Implementierungsstrategien wurden jedoch im Rahmen dieser Arbeit nicht realisiert.

### 4.2.1 Elementares Matching

Die im Folgenden vorgestellte Implementierungsstrategie für ein sequentielles Matching auf FPGAs soll die Anforderungen an das Matching für die im Rahmen dieser Arbeit verwendeten 36 Templates verdeutlichen und die Problematik von möglichen Parallelisierungen beim Matching aufzeigen.

Die Idee des sequentiellen Matchings besteht darin, die verschiedenen Templates an allen Punkten des Bildes nacheinander zu berechnen. Die dem Template entsprechenden Pixel werden nacheinander einer einfachen Korrelationseinheit zugeführt, in welcher das Distanzmaß, siehe Abschn. 2.2, berechnet wird.

Eine FPGA-Implementierung für eine sequentielle Korrelationseinheit ist in Abb. 4.1 dargestellt. Ein Steuerungsmodul wird dafür sorgen, dass einer AU, die aus einem Akkumulator und einem Schwellwertmodul besteht, in jedem Takt ein dem Template entsprechendes DT-Pixel, welches im externen SRAM des FPGA-Koprozessors gespeichert ist, zugeführt wird. Die hierfür notwendigen Templatedaten[2] werden im internen RAM des FPGAs gespeichert.

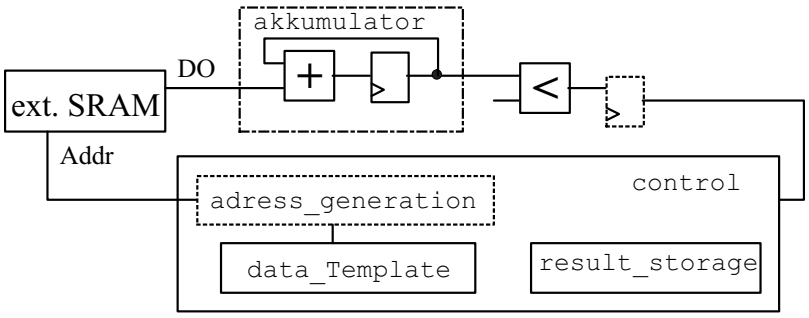



**Abbildung 4.1:** Korrelationseinheit für sequentielles Template-Matching.

Für alle der in den Abb. 2.8 bis 2.10 angegebenen 36 Templates $T_j$, bei denen jedes eine mittlere Anzahl von ungefähr 108 Templatepunkten hat, sind für jeden Bildpunkt ca. 3800 Additionen durchzuführen. Für ein Bild der Größe $512 \times 512$ Pixel abzüglich einem Randbereich von 20 Pixel, in dem keine Korrelationen berechnet werden, sind dies ca. 920 Millionen Additionen. Bei einer angenommenen Taktfrequenz von 100 MHz wird ein FPGA-Koprozessor mit dem oben beschriebenen Design für die Berechnung der Korrelation der 36 Templates ca. 9.2 Sekunden benötigen. Zu berücksichtigen ist jedoch, dass zusätzlich im FPGA in jedem Takt überprüft wird, ob das aktuelle Zwischenergebnis am Ausgang des Akkumulators den zum Template $T_j$ gehörenden Schwellwert $\Theta_j$ überschreitet. Sollte dies zutreffen, so werden die Additionen für die verbleibenden Templateelemente nicht mehr durchgeführt. Angenommen, diese Vorgehensweise führt zu einer Beschleunigung des sequentiellen Ansatzes um einen Faktor drei, so reduziert sich die Rechenzeit für alle 36 Templates auf etwas mehr als drei Sekunden. Dabei ist festzuhalten, dass die Größe dieses Faktors stark abhängig vom Bildinhalt ist. Bei sicherheitsrelevanten Anwendungen wie z.B. der Erkennung von Verkehrszeichen im Straßenverkehr ist diese Abhängigkeit von Nachteil. Der Vorteil dieses Ansatzes besteht darin, dass der FPGA-Ressourcenbedarf gering ist. Für jede Korrelationseinheit sind lediglich ein Akkumulator, der aus einem Addierer und einem Register besteht, ein Schwellwertmodul sowie ein Steuerungsmodul notwendig. Letzteres besitzt einerseits Zugang zu den Templatedaten und andererseits zu den DT-Bildern deren Pixel dem Akkumulator zuführt werden, siehe Abb. 4.1. Die Templatedaten werden im internen FPGA-RAM oder in einer externen SRAM-Bank auf dem FPGA-Koprozessor gespeichert.

Das oben beschriebene sequentielle Verfahren kann beschleunigt werden, indem mehrere Templates gleichzeitig mit mehreren sequentiellen Korrelationseinheiten berechnet werden. Dies kann erreicht werden, indem die Bilder in mehreren externen SRAM-Bausteinen abgespeichert und die Korrelationseinheiten auf dem FPGA entsprechend

[2]Für die in Abschn. 2.2.1 angegebenen 36 Templates werden ca. 11,5 kByte RAM benötigt.

vervielfacht werden. Diese Vorgehensweise stellt bei hoher Parallelisierung, d.h. sehr vielen Korrelationseinheiten jedoch hohe Anforderungen an den FPGA-Koprozessor. Dieser müsste mit vielen externen SRAMs ausgestattet sein und auch mit mehreren FPGAs, um die vielen physikalischen Verbindungen mit den SRAM Bausteinen zu ermöglichen. Um die Echtzeitbedingung für das Template-Matching von 30 Bilder/sec zu erfüllen, müsste der FPGA-Koprozessor mit ca. 100 unabhängigen externen SRAM-Bausteinen bestückt sein. Dieser FPGA-Prozessor wäre sehr kostenintensiv und hätte sehr große Ausmaße.

Das Problem beim sequentiellen Matching besteht weniger darin viele Korrelationseinheiten auf einem FPGA unterzubringen, sondern die den Templates entsprechenden DT-Pixel, die in einer nichtregelmäßigen Abfolge aus dem RAM auszulesen sind, den Korrelationseinheiten zuzuführen.

### 4.2.2 Sequentielles Matching mit zeilenweiser Parallelisierung

Im Folgenden wird eine Strategie beschrieben, die geeignet ist zur Beschleunigung der Ausführung des Matchings von *einem* Template an einem Bildpunkt. Dieser Ansatz kann auch zur Beschleunigung eines hierarchischen Matchings eingesetzt werden, bei welchem die Templates nacheinander zu berechnen sind. Einen Überblick über die datenverarbeitenden Submodule und Steuermodule ist in Abb. 4.2 zu sehen.

Die Idee dieser Implementierungsstrategie besteht darin, einen für das Template-Matching ausreichend großen Bildausschnitt im internen FPGA-RAM zeilenweise abzuspeichern. Dies erlaubt einen parallelen Zugriff auf alle für die Templates relevanten Bildzeilen und führt zu einer höheren Verarbeitungsgeschwindigkeit.

#### Datenspeicher

Im FPGA-Datenspeicher `FPGA_RAM`, siehe links oben in Abb. 4.2, wird jede Bildzeile getrennt in einem Zeilenspeicher, dem `line_RAM`-Modul gespeichert. Hierfür wird das Block-RAM des Virtex-II FPGAs benutzt. Die Anzahl der im FPGA zu speichernden Bildzeilen ist gleich der max. Ausdehnung der Templates in y-Richtung. Bei einer max. Ausdehnung $D_y$ der Templates in y-Richtung von 39 Pixel sind 39 `line_RAM`-Module notwendig. Wird außerdem ein hierarchisches Matching nach [1] durchgeführt, bei dem die Verfeinerung des Gitters auch in positiver und negativer y-Richtung erfolgen kann, werden weitere Zeilenspeicher benötigt, deren Anzahl von der Gitterkonstante des gröbsten Gitters abhängig ist. Arbeitet man auf einem groben Gitter mit einer Gitterkonstante von $k_G$ Pixel werden insgesamt $K = D_y + k_G$ Zeilenspeicher benötigt.

#### ALU

Die ALU, die rechts oben in Abb. 4.2 dargestellt ist, besteht aus $K$ `akkumulator`-Modulen, einem Addiererbaum, der dessen Ausgänge zusammenfasst und einem Schwellwertmodul.

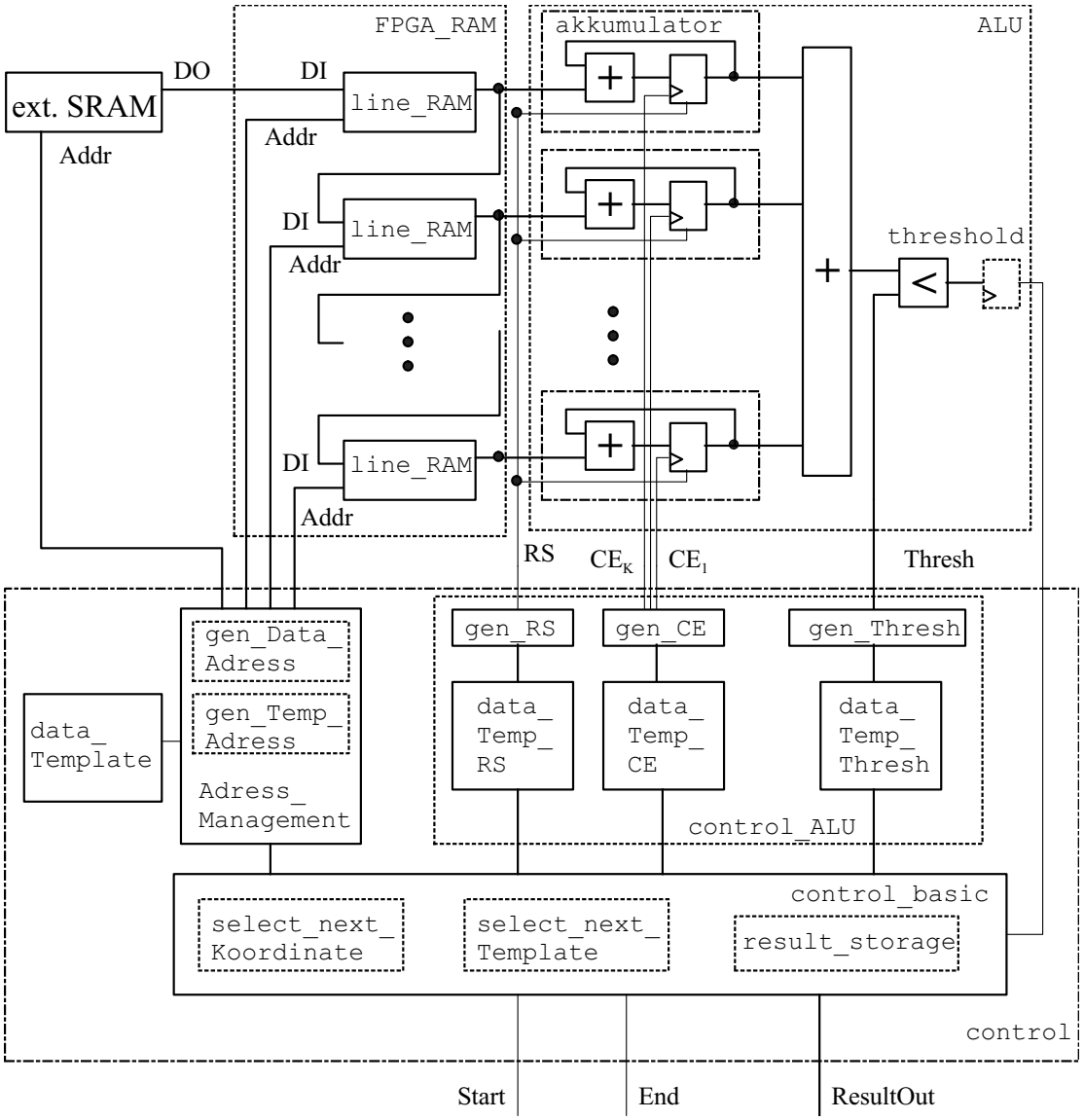



**Abbildung 4.2:** Implementierung des sequentiellen Template-Matchings mit zeilenweiser Parallelisierung.

**Akkumulatoren und Addiererbaum** Für jede der $K$ Bildzeilen, die sich in den Zeilenspeichern befinden, wird eine Akkumulatoreinheit eingesetzt, die jeweils aus einem Addierer und einem Register besteht. Die Dimensionierung der Bitbreite ist dabei abhängig von der Anzahl der maximal auszuführenden Additionen, d.h. von der max. Anzahl der Templateelemente aller Zeilen in x-Richtung.

Nach der Berechnung des Distanzmaßes für ein Template werden die Register der Akkumulatoreinheiten von dem im nächsten Abschnitt beschriebenen Steuerungsmodul zurückgesetzt. Je nach Anzahl der Templateemenete in jeder Zeile, dürfen die Register der Akkumulatoreinheiten Daten annehmen und deren *CE*-Eingänge werden gleich eins gesetzt. Hierbei muss für jedes Register ein eigenes *CE*-Signal erzeugt werden, dessen Länge von der Anzahl der Templateelemente des Templates in der entsprechenden

Zeile abhängig ist.

Die Zwischensummen der $K$ Akkumulatoren werden dann von einem Addiererbaum zusammengefasst, der nach dem Pipeline-Prinzip aufgebaut ist.

**Schwellwert** Die Summe der Akkumulatoreinheiten, die am Ausgang des Addiererbaums anliegen, werden in jedem Takt von dem Schwellwertmodul `threshold` überprüft. Wird das Ähnlichkeitsmaß von Template $T_j$ berechnet, so wird dem `threshold`-Modul der entsprechende Schwellwert $\Theta_j$ zugeführt, siehe Gl. 2.13.

Die Schwellwerte $\Theta_j$ werden im internen FPGA-RAM `data_Temp_Thresh` gespeichert und über das `gen_Thresh`-Modul dem ladbaren `threshold`-Modul zugeführt.

### Steuerung

Das Steuerungsmodul `control`, welches unten in Abb. 4.2 dargestellt ist, setzt sich zum einen zusammen aus dem `control_basic`-Modul, dem Modul `control_Address`, welches für das Adressmanagement verantwortlich ist und dem Modul `control_ALU` mit der Aufgabe die Steuersignale für die `akkumulator`-Module und das `threshold`-Modul zu generieren.

**Basis-Steuerung** Das Steuerungsmodul `control_basic` hat als wesentliche Aufgabe in Abhängigkeit des Ausgangssignals des `threshold`-Moduls den nächsten Bildpunkt zu bestimmen, an dem das nächste Template berechnet wird.

Im Submodul `select_next_Template` wird das Template, für welches als nächstes das Matching durchzuführen ist, ausgewählt. Falls keine hierarchische Matching-Strategie benutzt wird, wird als nächstes Template das in der Template-Liste folgende ausgewählt. Erreicht man das letzte Template in der Template-Liste, so wird das erste Template aus der Liste beim nächsten Bildpunkt berechnet. Die Bestimmung des nächsten Bildpunkts wird vom Submodul `select_next_Point` übernommen. Die Verfeinerungsstrategie für das Bildraster und der Template-Hierarchie beim hierarchischen Matching wird ebenfalls in diesen Submodulen realisiert.

Des Weiteren kontrolliert das Steuerungsmodul `control_basic` die beiden weiteren Steuerungsmodule `control_Adress` und `control_ALU`, welche die Adressen zum Speichern und Auslesen der Daten und die Steuerungssignale für die ALU erzeugen. Das übergeordnete Steuerungsmodul hat die Aufgabe, nach dem Setzen des Flag-Signals *Start* mit der Ausführung des Matchings zu beginnen, nach Beendigung des Matchings das *End*-Signal zu generieren und die Übergabe der Matching-Ergebnisse nach außen zu gewährleisten.

**Adressmanagement** Die erste Aufgabe des Adressmanagement-Moduls `control_Adress` besteht darin, die DT-Bilder aus dem externen SRAM des MPRACE-Boards in die Zeilenspeicher `line_RAM` des FPGAs zu laden. Bevor die Berechnung der Distanzmaße beginnt, wird zunächst ein Bildausschnitt in alle Zeilenspeicher des internen RAMs geladen. Während dem Matching werden die Zeilen dann einzeln in den Ringspeicher nachgeladen und die nicht mehr benötigten Bildzeilen werden überschrieben.

Die zweite Aufgabe besteht in der Generierung der $K$ Adressen für die $K$ Zeilenspeicher für die Berechnung der Korrelation. Die Adressen der Templateelemente der jeweiligen Zeilen des Templates werden aus den `data_Template`-Modulen ausgelesen und zu einem globalem Adresszähler, welcher die momentane Lage im Bild bestimmt, hinzuaddiert. Für jeden der $K$ Zeilenspeicher wird dann genau ein Addierer benötigt. Die Adressen für jedes Template werden im `data_Template`-Modul zeilenweise abgespeichert.

Die Auswahl zwischen der Adresse für das Nachladen der Pixel und der Adresse zur Berechnung der Ähnlichkeitsmaße erfolgt für jedes `RAM_line`-Modul über einen Adressmultiplexer, die nicht in Abb. 4.2 eingezeichnet sind.

**Steuerung der ALU** In den `gen_RS`- und `gen_CE`-Submodulen des `control_ALU`-Moduls werden die Singale *RS* und $CE_1$ - $CE_K$ zur Steuerung der Register in den Akkumulatoren erzeugt. Die Art der Signale wurden oben beschrieben. Des Weiteren wird im `gen_Threshold`-Submodul das *Threshold*-Signal für das `threshold`-Modul erzeugt. Die drei Submodule haben dabei Zugriff auf die für jede Templatezeile relevanten und für die jeweilige Aufgabe entsprechenden Templatedaten, die in den `data_Temp_RS`-, `data_Temp_CE`- und `data_Temp_Thresh`-Modulen abgespeichert sind.

### Geschwindigkeit

Die Rechenzeit für die jeweiligen Templates ist von der max. Anzahl der Templateelemente in einer Zeile abhängig. Bei den in Abb. 2.8 angegebenen kreisförmigen Templates befinden sich max. sechs bis sieben Templateelemente in einer Zeile. Der Schwellwert kann überschritten werden, bevor alle Templateelemente eingelesen sind. Das Verfahren bricht dann sofort ab und das Distanzmaß wird für das nächste Template berechnet. Es wird angenommen, dass sich aufgrund dessen die Ausführung des Algorithmus um einen Faktor drei beschleunigt. Diesbezüglich sind jedoch weitere Untersuchungen durchzuführen. Die reine Verarbeitungszeit in den Akkumulatoreinheiten reduziert sich dann von sechs Takten auf ungefähr zwei Takte. Bei einer Latenzzeit für den Addiererbaum und das Schwellwertmodul von einem Takt ergibt sich für die gesamte ALU eine Latenzzeit von drei Takten. Des Weiteren wird angenommen, dass für die Bestimmung des nächsten Templates und des nächsten Bildpunktes ein Takt bei der einfachen und zwei Takte bei der hierarchischen Strategie benötigt werden. Für die Adressgenerierung kommt noch ein weiterer Takt hinzu. Jedoch ist zu beachten, dass Teile dieser Steuerungsaufgaben schon während der Berechnung des Distanzmaßes ausgeführt werden können. Insgesamt werden ungefähr vier Takte Verarbeitungszeit pro Template veranschlagt. Außerdem muss die Zeit für das Laden bzw. Nachladen des Bildes in die Zeilenspeicher `lineRAM` berücksichtigt werden, die linear mit der Bildgröße $H \times W$ wächst. Pro Takt kann ein Pixel in das FPGA-RAM geladen werden.

Insgesamt ist die benötigte Rechenzeit stark von den Eigenschaften der Templates, genauer von der max. Anzahl von Templateelementen in einer Zeile abhängig. Bei den Dreiecken sind diese ungünstig auf die einzelnen Zeilen verteilt. In den horizontalen Seiten der Dreiecke liegen zwischen 17 und 39 Templateelemente, was zu entsprechend langen Rechenzeiten führen kann. Hier ist es günstig, die Berechnung für die Templateelemente der horizontalen Seiten spaltenweise durchzuführen, gleichzeitig mit der zeilenwei-

sen Berechnung der diagonalen Seiten der dreieckigen Templates. Diese Vorgehensweise führt zu einem entsprechend höheren Ressourcenaufwand.

Bei der elementaren sequentiellen Methode wurde angenommen, dass für die Berechnung eines Templates im Mittel ungefähr 36 Takte benötigt werden und bei der sequentiell-parallelen Methode ungefähr vier Takte. Somit ist gegenüber dem sequentiellen Ansatz eine Beschleunigung von ca. einem Faktor neun zu erwarten. Eine hierarchische Erweiterung der beiden in diesem Abschnitt vorgestellten Implementierungsstrategien führt bei diesen zu einer weiteren Beschleunigung um mindestens einen Faktor zehn, siehe Abschn. 2.2.4.

#### Ressourcenverbrauch

Der relevante Bildausschnitt wird in $K$ Zeilenspeicher auf dem FPGA abgebildet. Anstelle von einem Akkumulator beim elementaren sequentiellen Matching kommen nun $K$ Akkumulatoren zum Einsatz. Hinzu kommen ein Addiererbaum bestehend aus $K-1$ Addierern sowie ein Schwellwertmodul. Letzteres hat jedoch einen relativ geringen Ressourcenbedarf. Die Templatedaten, die zur Adressgenerierung benötigt werden, sind im internen FPGA-RAM abgespeichert. Diese sind, wie beim elementaren Matching, nur einmal zu speichern. Außerdem kommen ein Addierer sowie ein Adressmultiplexer für die Berechnung der Adressen für jeden Zeilenspeicher zum Einsatz. Zusätzlich müssen die Templatedaten zur Generierung der *RS*-und *CE*-Signale für jeden Akkumulator gespeichert werden. Hinzu kommt die benötigte Logik für die Steuermodule insbesondere die für ein hierarchisches Matching.

Der Ressourcenbedarf erhöht sich gegenüber dem sequentiellen Matching um etwas weniger als einen Faktor $K$, für die im Rahmen dieser Arbeit eingesetzten Templates um ca. einen Faktor 30. Hierbei handelt es sich allerdins um eine grobe Abschätzung.

Die Effizienz dieser Implementierungsstrategie hängt stark von der Form der Templates ab, da für Zeilen mit nur einem Templateelement die Akkumulatoreinheiten nur einen Takt und bei Zeilen mit mehreren Templateelementen entsprechend viele Takte arbeiten. Ideal sind Templates, die in jeder Zeile gleich viele Templateelemente besitzen.

#### Ausblick

Zunächst bleibt zu klären, ob eine effiziente FPGA-Implementierung für ein hierarchisches Template-Matching erreicht werden kann und somit tatsächlich eine weitere Beschleunigung um mindestens einen Faktor zehn möglich ist. Außerdem muss die Implementierungsstrategie auf die DT-Bilder der $k$ Richtungsbereiche erweitert werden. Es wird angenommen, dass sich der Ressourcenaufwand bei den Kreisen ungefähr verdoppelt und der für alle Templates um einen Faktor vier erhöhen wird.

Die Ausführungszeiten für die beiden sequentiellen Ansätze und insbesondere für den hierarchischen Ansatz sind vom Bildinhalt abhängig. Der im Folgenden beschriebene parallele Ansatz, der im Rahmen dieser Arbeit implementiert wurde, garantiert die Ausführung des Template-Matchings in einer Zeit, die mit der Bildgröße $H \times W$ skaliert und nur minimal abhängig ist vom Bildinhalt.

## 4.3 Paralleles Template-Matching

In den folgenden Abschnitten wird der vom Autor entwickelte parallele Entwurf zur Ausführung des Template-Matchings auf FPGAs beschrieben. Er kann als Erweiterung des in [76] vorgeschlagenen Ansatzes gesehen werden. Zunächst wird ein kurzer Überblick über das Verfahren gegeben.

Zur Berechnung des Distanzmaßes eines Templates $T_j$ an einem Bildpunkt sind die zum entsprechenden Template gehörenden Pixel der $M$ DT-Bilder gemäß Gl. 2.14 aufzusummieren. Dieser Vorgang ist für alle $N$ Templates und für alle Bildpunkte durchzuführen, siehe Abb. 4.3.

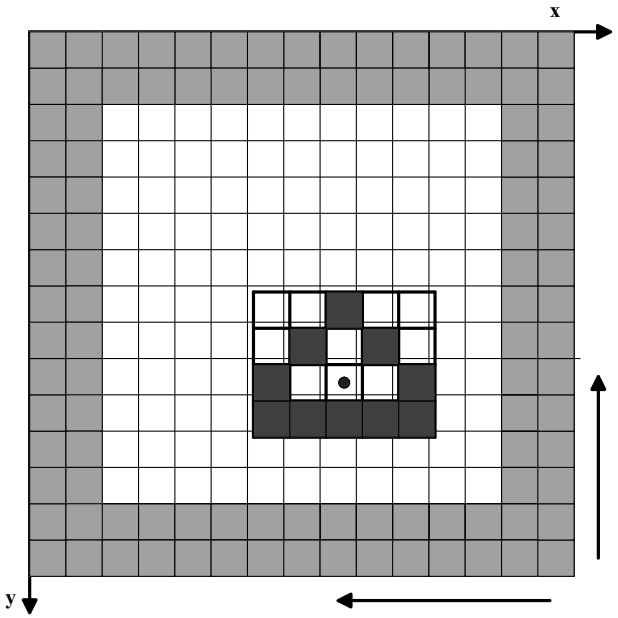



**Abbildung 4.3:** Template-Matching durch zeilenweises Verschieben einer Templatemaske über ein DT-Bild in Rückwärts-Richtung. Mit dunkelgrau gekennzeichnet sind die Templateelemente, mit hellgrau die Pixel des Bildrandes.

Die FPGA-Implementierung des Template-Matchings hat Ähnlichkeiten mit deren des Sobel-Operators, die in Abschn. 3.3.1 beschrieben wurde. Die für alle Templates relevanten Daten werden in SRAs (Shift Register Arrays) gespeichert, die Templatemasken sind fest im FPGA verdrahtet und die Berechnung der Distanzmaße wird für alle Templates parallel mit Addiererbäumen, durchgeführt, siehe Abb. 4.5 und 4.6. Den Ausgängen der Addiererbäume sind Schwellwertmodule nachgeschaltet. Somit kann für alle Templates in jedem Takt entschieden werden, ob alle $N$ Distanzmaße unterhalb den vom Benutzer vorgegebenen Schwellwerten $\Theta_i$ liegen, siehe Gl. 2.13. Dieser Entwurf hat den wesentlichen Vorteil, dass keine Zwischenergebnisse anfallen, die dann z.B. im externen RAM des FPGA-Koprozessors zwischengespeichert werden müssten und die weitere Berechnung verzögern würden.

### 4.3.1 Shift Register Arrays

Um die $N$ Distanzmaße der $N$ Templates gleichzeitig berechnen zu können, wird für jeden Richtungsbereich $k$ ein ausreichend großes SRA (Shift Register Array) generiert, welches den parallelen Zugriff auf die für alle Templates relevanten DT-Pixel, die in den Registern gespeichert sind, ermöglicht. Der Aufbau der SRAs ist in Abb. 4.4 am Beispiel zweier linienförmigen Templates dargestellt, dessen entsprechende Register fett gekennzeichnet sind. Für jedes Template $T_j$ wird genau ein Addiererbaum mit $|T_j|$ Eingängen erzeugt. Die Eingänge der Addiererbäume werden mit den Registern, die zu den entsprechenden Templates gehören, verbunden. Die DT-Pixel der jeweiligen restlichen Bildzeilen werden im internen Block-RAM des FPGAs zwischengespeichert. Insgesamt muss jeder DT-Pixel zur Berechnung der Distanzmaße aller Templates an allen Bildpunkten genau einmal aus dem externen RAM ausgelesen werden.

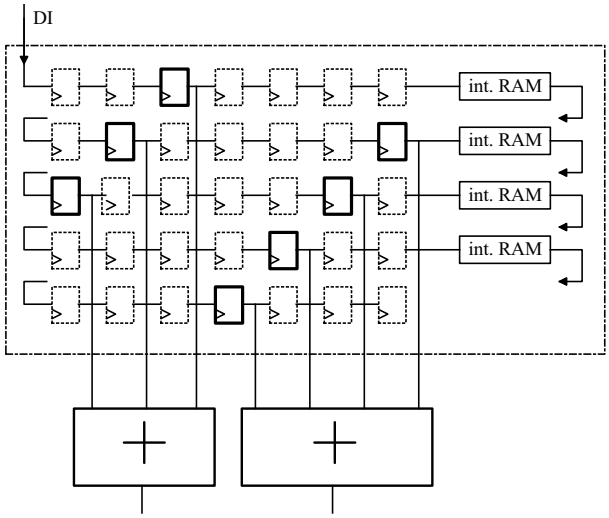



**Abbildung 4.4:** Generierung eines SRAs abhängig von der Form der Templates. Die den Templateelementen entsprechenden Register sind fett eingefärbt.

Die acht SRAs werden abhängig von der Form der Templates generiert und unterscheiden sich daher i.A. in ihrer Ausdehnung in x- und y-Richtung. Dies bedeutet, dass sich somit zusätzlich deren Lage verändert, wie dies in Abb. 4.5 am Beispiel zweier kreisförmigen Templates dargestellt ist. Dies hat zur Folge, dass die DT-Pixel für jedes SRA aus unterschiedlichen Adressbereichen aus den DT-Bildern ausgelesen werden müssen. Dies ist der Grund, weshalb die acht DT-Bilder nach der Berechnung der zweiten Pipeline der Vorverarbeitung in getrennten RAM-Bereichen abgespeichert werden, siehe Abschn. 3.6. Hierbei wurden jeweils acht benachbarte Pixel eines DT-Bildes in einem Adresswort zusammengefasst. Diese werden, bevor sie den SRAs zugeführt werden, in den sog. `resort`-Modulen, die in Abschn. 4.3.3 ausführlich beschrieben sind, in eine serielle Form umgewandelt. Die Auswirkungen auf die Adressgenerierung werden ebenfalls in Abschn. 4.3.3 besprochen.

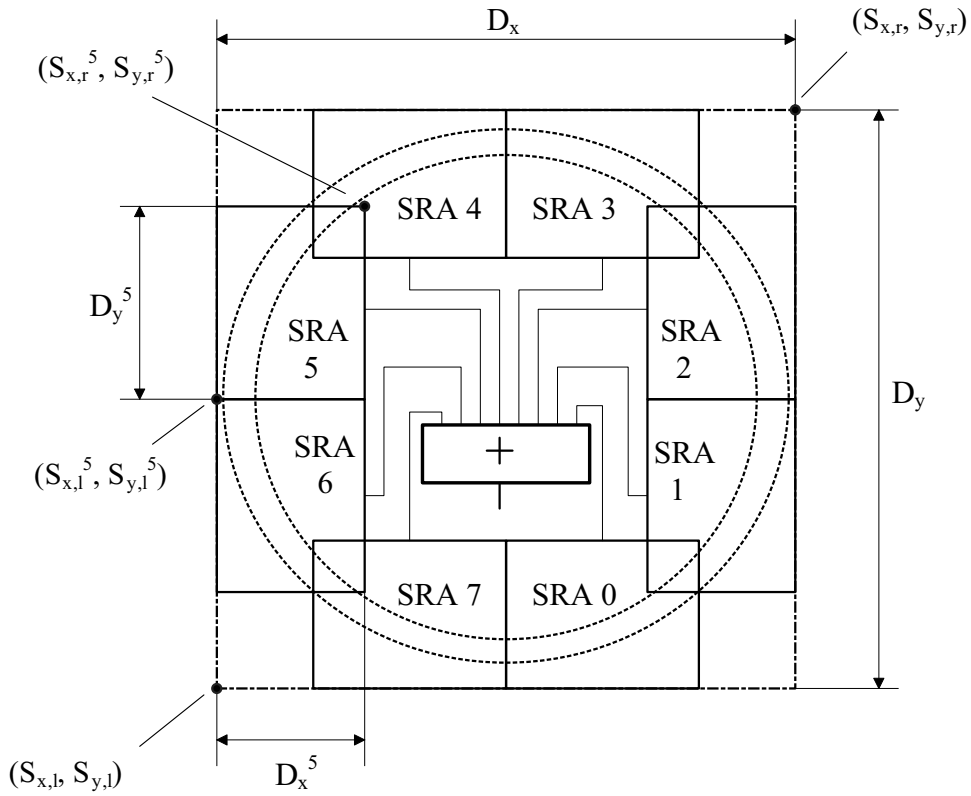



**Abbildung 4.5:** Aufbau von acht Shift Register Arrays (SRAs) am Beispiel zweier kreisförmiger Templates. Die acht SRAs unterscheiden sich in ihren Ausdehnungen und Lagen.

### Größe der SRAs

Im Folgenden werden die quantitativen Eigenschaften der SRAs beschrieben. Außerdem wird für die spätere Adressgenerierung ein die SRAs umliegendes Rechteck $S$ definiert, welches für die Berechnung der unterschiedlichen Adress-Offsets in Abschn. 4.3.3 benötigt wird.

Die Größe eines SRAs $S^k$ aus Richtungsbereich $k$, $k = 0, ..., M-1$ berechnet sich aus *allen* Templateelementen $t^k_{i,j} = (t^k_{i,j,x}, t^k_{i,j,y}) \in T^k_j$ der $N$ Templates, die dem Richtungsbereich $k$ zuzuordnen sind (Abschn. 2.2). Das Rechteck, welches das SRA $S^k$ begrenzt, siehe Abb. 4.5, ist durch dessen linken unteren Punkt $(S^k_{x,l},\ S^k_{y,l})$ und oberen rechten Punkt $(S^k_{x,r},\ S^k_{y,r})$ eindeutig bestimmt

$$S^k_{x,l} = \min(t^k_{i,j,x}) \quad \forall\, t^k_{i,j,x} \in T^k_j, \quad j = 0, \ldots, N-1 \tag{4.1}$$
$$S^k_{y,l} = \min(t^k_{i,j,y}) \quad \forall\, t^k_{i,j,y} \in T^k_j, \quad j = 0, \ldots, N-1 \tag{4.2}$$
$$S^k_{x,r} = \max(t^k_{i,j,x}) \quad \forall\, t^k_{i,j,x} \in T^k_j, \quad j = 0, \ldots, N-1 \tag{4.3}$$
$$S^k_{y,r} = \max(t^k_{i,j,y}) \quad \forall\, t^k_{i,j,y} \in T^k_j, \quad j = 0, \ldots, N-1. \tag{4.4}$$

Die Punkte $(S_{x,l},\ S_{y,l})$ und $(S_{x,r},\ S_{y,r})$ eines Rechtecks $S$, welches alle $M$ SRAs begrenzt,

ist gegeben durch

$$(S_{x,l},\, S_{y,l}) = (\min(S^k_{x,l}),\, \min(S^k_{y,l})) \quad k = 0, \ldots, M-1 \tag{4.5}$$
$$(S_{x,r},\, S_{y,r}) = (\max(S^k_{x,r}),\, \max(S^k_{y,r})) \quad k = 0, \ldots, M-1. \tag{4.6}$$

Die Anzahl der Register $D^k_x$ und $D^k_y$ des jeweiligen SRAs $S^k$ ist gegeben durch

$$D^k_x = S^k_{x,r} - S^k_{x,l} + 1 \tag{4.7}$$
$$D^k_y = S^k_{y,r} - S^k_{y,l} + 1 \tag{4.8}$$

und die Anzahl der Register $D_x$ und $D_y$ des umschließenden Rechtecks $S$ durch

$$D_x = S_{x,r} - S_{x,l} + 1 \tag{4.9}$$
$$D_y = S_{y,r} - S_{y,l} + 1. \tag{4.10}$$

**Register** Mit $R^k$ wird die Menge der Register $r^k_l$ von SRA $S^k$ aus dem Richtungsbereich $k$ bezeichnet und mit $R$ die Menge der Register der $M$ Richtungsbereiche mit $R = \cup_k R^k$. Die Anzahl der Register von SRA $S^k$ ist $|R^k| = D^k_x * D^k_y$, die aller Register $|R| = \sum_k |R^k|$. Die Register $r^k_l \in R^k,\ l = 0, \ldots, |R^k| - 1$ von SRA $S^k$ seien von links oben nach rechts unten durchnummeriert. Jedem Register $r^k_l$ wird somit ein Index $\{l^k\}$ zugeordnet. Die Register werden im Weiteren, insbesondere in Kapitel 5 als als Konten eines Graphen betrachtet.

**Parametrisierung** Die SRA-Module werden abhängig von der Form der Templates aufgebaut und sind bezüglich der Ausdehnung in x- bzw. y-Richtung parametrisierbar. Deren Werte werden nach den Gl. 4.7 und 4.8 bestimmt. Die internen Speichermodule zum Zwischenspeichern der relevanten Bildzeilen sind nach der Länge $W - D_x$ parametrisierbar oder können während der Ausführung des Designs über die in Tab. 4.3 angegebenen Codewörter verändert werden. Allerdings sind die internen Speichermodule momentan auf eine max. Zeilenlänge von 512 begrenzt. Die Erweiterung auf größere Zeilenlängen ist jedoch leicht durchzuführen.

### 4.3.2 Addiererbäume

#### Verbindungen zwischen Addiererbäumen und Registern der SRAs

Ein Addiererbaum, der mit $A_j$ bezeichnet wird, ist zunächst als ein Baustein mit $|A_j|$ gleichberechtigten Eingängen aufzufassen. Der innere Aufbau soll zunächst nicht von Interesse sein. Die Berechnung der Korrelation wird für jedes Template $T_j$ durch einen separaten Addiererbaum $A_j$ ausgeführt. Jeder Addiererbaum $A_j$ hat genau $|T_j|$ Eingänge, d.h. $|A_j| = |T_j|,\ j = 0, \ldots, N-1$. Die Eingänge von Addiererbaum $A_j$ werden mit $a_{i,j},\ i = 0, \ldots, |A_j| - 1$ bezeichnet.

Die $|T_j|$ Templateelemente $t^k_{i,j}$ von Template $T_j$ bzw. die $|A_j|$ Eingänge $a_{i,j}$ eines Addiererbaums $A_j$ sind den Registern $r^k_l$ eindeutig zuzuordnen. Es besteht eine Abbildung $z$

zwischen jedem Templateelement $t^k_{i,j}$ und dem Index $l^k$ des Registers im Richtungsbereich $k$, mit dem es verbunden wird. Die Abbildung $z : T^k \rightarrow R^k,\ t^k_{i,j} \rightarrow r^k_l$ ist gegeben durch

$$l^k = r^k_l = t^k_{i,j,x} - S^k_x + (t^k_{i,j,y} - S^k_y) * D^k_y \qquad \forall i,j. \tag{4.11}$$

Die Eingänge von unterschiedlichen Addiererbäumen können mit demselben Register verbunden werden, d.h. $z(t^k_{i,j'}) = z(t^k_{i,j})$ für $j' \neq j$ ist erlaubt. Jeder Eingang von einem Addiererbaum wird mit jeweils genau einem unterschiedlichen Register verbunden. Dies ergibt sich aus der Voraussetzung an die Templates, dass jedes Templateelement nur einmal in $T_j$ enthalten sein darf.

Mit $\tilde{R}^k$ sei die Menge aller Register von SRA $S^k$ definiert, die mit mindestens einem der Eingänge $a_{i,j}$ der $N$ Addiererbäume $A_j$ verbunden sind. Mit $\tilde{R}$ sei die Menge aller Register bezeichnet, die mit mindestens einem der $N$ Addiererbäume verbunden ist

$$\tilde{R} = \bigcup_{k=0}^{M-1} \tilde{R}^k. \tag{4.12}$$

$\tilde{R}$ ist eine Teilmenge der Menge aller Register $R$, d.h. $\tilde{R} \subseteq R$. Außerdem gilt $\tilde{R}^k \subseteq R^k$.

**Verbindungen** Die Kantenmenge $E_j$, d.h. die Menge der aus Gl. 4.11 berechneten Verbindungen $E_j$ zwischen den Registern $\tilde{R}$ und dem Addierern $A_j$ sei mit $E = \tilde{R} \times A_j$ gegeben. Die Elemente der Menge $E_j$ sind die Tupel $(\tilde{r}_l, a_{i',j})$, wobei $i'$ eine beliebige Permutation $P(i)$ von $i = \{0, \ldots, |A_j| - 1\}$ ist, d.h. jedes Register $\tilde{r}_l$ kann genau einem beliebigen Addierereingang $a_{i',j}$ von $A_j$ zugeordnet werden.

Die Zuordnung der Addierereingänge zu den Registern ist bis auf die Permutation $P(i)$ eindeutig. Zu beachten ist ein möglicher negativer Einfluss der Wahl von $P(i)$ auf das anschließende *Routing* im FPGA. Es zeigt sich, dass die Register der SRAs im FPGA dicht zusammen liegen. Daher ist es von Vorteil, wenn möglichst viele (2-elementige) Addierer mit Registern aus demselben Richtungsbereich $k$ verbunden werden. Somit können die Signal-Wege auf dem FPGA möglichst gering gehalten werden.

### Binäre Addiererbäume

Ein binärer Addiererbaum führt die Addition von $n$ Operanden der Bitbreite $k$ mit zweielementigen Addierern gleichzeitig aus [80]. In einer ersten Stufe werden die $n$ Eingänge durch (zweielementige) Addierer zusammengefasst. In der nächsten Stufe werden die $\lfloor \frac{n}{2} \rfloor$ Ausgänge der Addierer der ersten Stufe wiederum durch Addierer zusammengefasst. Die letzte Stufe besteht aus einem Addierer. In jedem Takt liegt die Summe aller Operanden mit einer Verzögerung am Ausgang des Addiererbaums an. Der FPGA-Ressourcenbedarf eines Addiererbaums ist ungefähr durch $k * \lfloor \frac{n}{2} \rfloor + (k+1) * \lfloor \frac{n}{4} \rfloor + (k+2) * \lfloor \frac{n}{8} \rfloor \ldots$ LUTs gegeben.

Die Begrenzung der max. Gatterlaufzeit im Addiererbaum wird reduziert, indem Register in den Addiererbaum eingefügt werden. Wird an allen Ausgängen der Addierer ein Register geschaltet, so liegt die Summe am Ausgang des Addiererbaums mit einer Verzögerung von $\lceil \frac{\log n}{\log 2} \rceil$ Takten an.

**Wallace Tree** Bei der Implementierung von binären Addiererbäumen in ASICs werden häufig sog. *Wallace Trees* verwendet [80]. Die Additionen können schneller ausgeführt werden bei einem geringeren Ressourcenbedarf. Der Einsatz eines Wallace-Baums ist jedoch für FPGAs nicht geeignet, weil die Logik auf die LUTs abgebildet wird und die Verbindungen der Logikelemente über normale Signalleitungen erfolgen, und nicht über die schnellen, hochoptimierten Fast-Carry-Verbindungen [81].

**Effizienz der SRAs** Nicht auf alle Register der SRAs wird von den Addiererbäumen zugegriffen. Einige werden ausschließlich zum Weiterleiten der Daten benötigt. Der Ressourcenverbrauch eines SRAs $S^k$ ist von der Größenordnung $O(D_x^k \times D_y^k)$. Wünschenswert wäre eine Auslastung von der Größenordnung $O(\sum_j |T_j|)$, der Summe aller Templateelemente. Ein Algorithmus, der diesbezüglich optimierte SRA generiert, wird in Abschn. 5.1 beschrieben.

**Parametrisierung** Die Addiererbäume sind parametrisierbar bezüglich der Anzahl der Eingänge und der Bitbreite der Eingänge. Über einen weiteren Parameter kann angegeben werden, nach welchen Addiererstufen eine Registerstufe eingefügt wird. Somit wird gewährleistet, dass die Anzahl der Logikstufen nicht zu groß wird und das Design mit einer hohen Taktfrequenz gespeist werden kann. Die Schwellwerte $\Theta_i$ sind bei den `threshold`-Modulen parametrisierbar oder während der Laufzeit des Designs veränderbar. Alle Größen können über die in Tab. 4.3 angegebenen Codewörter eingestellt werden.

### 4.3.3 Datenfluss und Steuerung

#### Datenfluss

Wie oben beschrieben, wird für die Berechnung der Korrelation eine große Pipeline aufgebaut aus acht `resort`-Modulen, den acht SRAs und einer ALU, bestehend aus $N$ binären Addiererbäumen und Schwellwertmodulen, siehe Abb. 4.6. Die Berechnung der Pipeline erfolgt in Rückwärts-Richtung. Sind die Daten der acht DT-Bilder in einem externen SRAM-Baustein des FPGA-Koprozessors gespeichert, kann jedem der acht SRAs in jedem Takt genau ein Pixel zugeführt werden, vorausgesetzt, die acht DT-Bilder sind in unterschiedlichen Bereichen des externen SRAMs gespeichert und acht benachbarte DT-Pixel eines Richtungsbereichs in einem Adresswort. Entsprechend der Speicherung, werden die DT-Pixel eines Richtungsbereichs in jedem achten Takt ausgelesen, jeweils um einen Takt verzögert, und den `resort`-Modulen, siehe Abb. 4.7 zugeführt. In den `resort`-Modulen werden die Daten von einer parallelen in eine serielle Form umgewandelt und in den SRLT-Bausteinen des Virtex-II FPGAs verzögert. Mit den Verzögerungsgliedern (SRLTs) wird das um einen Takt verzögerte Auslesen der Daten kompensiert. Insgesamt ist ein effizientes Arbeiten der Pipeline gewährleistet.

**Adressgenerierung** Beim Füllen der einzelnen SRA-Module und der Zeilenspeicher müssen sowohl deren unterschiedlichen Größen als auch unterschiedlichen Lagen berück-

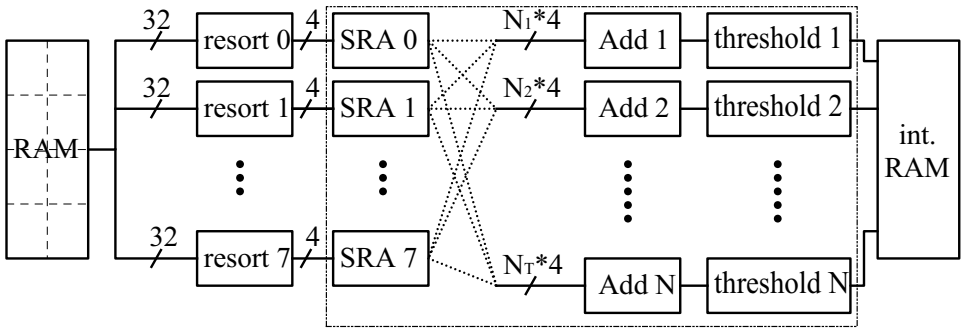



**Abbildung 4.6:** Datenfluss beim Template-Matching. Die Verbindungen zwischen den SRAs and den Addiererbäumen ist abhängig von den Eigenschaften Templates.

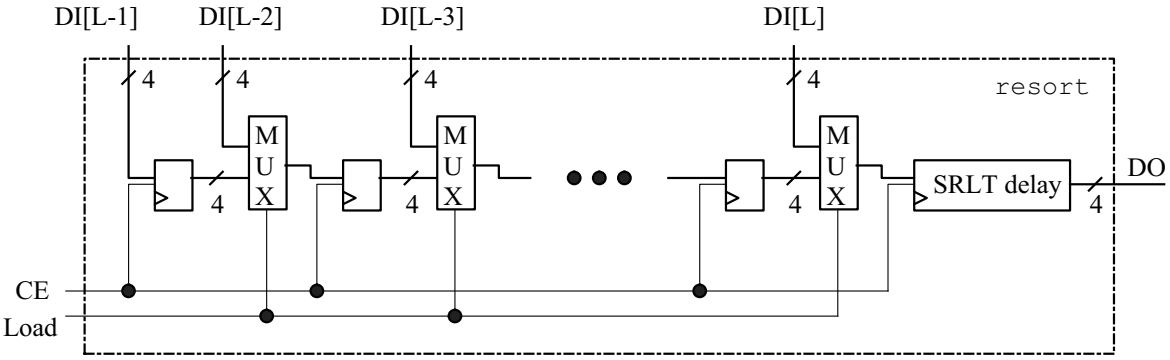



**Abbildung 4.7:** `Resort`-Modul: L-Bit parallel-zu-seriell Wandler mit Verzögerungsglied für vier Bit Daten.

sichtigt werden, siehe Abb. 4.4 und 4.5.

Die Größe von SRA $S^k$ entspricht dabei der Anzahl der Takte $T^k$, die für sein Füllen benötigt werden. Diese ist für ein Bild der Breite $W$ gegeben durch

$$T^k = D_x^k * D_y^k + (W - D_x^k) * (D_y^k - 1). \tag{4.13}$$

Der erste Term gibt dabei die Anzahl der Takte für das Füllen der Register des SRAs aus Richtungsbereich $k$ an und der zweite die Anzahl der Takte für das Füllen der internen FPGA-Zeilenspeicher, siehe Abb. 4.4.

Die unterschiedlichen Anordnungen der SRAs, die in Abb. 4.5 exemplarisch dargestellt sind, haben einen Adress-Offset zur Folge, der beim Auslesen der DT-Pixel aus den unterschiedlichen Adressbereichen berücksichtigt werden muss. Der Adress-Offset $O^k$ für den Richtungsbereich $k$ ist für ein Bild der Größe $H \times W$ gegeben durch

$$O^k = \underbrace{S_{x,r}^k - S_{x,r}}_{\text{x-Richtung}} + \underbrace{(S_{y,r}^k - S_{y,r}) * W}_{\text{y-Richtung}} + \underbrace{k\frac{W * H}{M}}_{\text{DT-Bilder}}, \quad k = 0, \ldots, M-1. \tag{4.14}$$

Der ersten beiden Terme geben den Adress-Offset in x- bzw. y-Richtung an. Der dritte Term für den Adress-Offset folgt aus den unterschiedlichen Speicherbereichen in denen die $M$ DT-Bilder, die den jeweiligen Richtungsbereichen $k$ entsprechen, liegen.

Im Folgenden werden drei Strategien beschrieben, die zum Füllen der SRAs geeignet sind. Die erste Strategie besteht darin, zuerst die acht SRAs nacheinander zu füllen, zunächst das erste SRA, nach dessen Abschluss, das zweite und zuletzt das achte. Die zweite Strategie sieht vor, zunächst das größte SRA zu füllen, und währenddessen mit dem zweitgrößten, usw. zu beginnen. Die dritte Variante sieht vor, bei allen SRAs gleichzeitig mit dem Zuführen der DT-Daten zu beginnen. Dann müssen jedoch die DT-Pixel, der sieben kleineren SRAs aus einem anderen Adressbereich ausgelesen werden. Hierbei muss die Differenz zwischen dem größten SRA und den anderen SRAs für die Anzahl der Takte zum Füllen der SRAs berücksichtigt werden. Für die neuen Adress-Offsets gilt dann

$$O^k = O^k + \max_k T^k - T^k. \tag{4.15}$$

Im Rahmen dieser Arbeit wurden alle drei Strategien zum Füllen der Pipeline implementiert. Letztendlich eingesetzt wird jedoch die dritte, weil diese fast acht Mal so schnell ist als die erste und weniger Logik in Form von Zählern und Komparatoren benötigt als die beiden anderen Strategien.

Wie bereits erwähnt, sind die DT-Pixel nicht einzeln sondern jeweils als acht benachbarte DT-Pixel in einem Adresswort im SRAM des FPGA-Koprozessors gespeichert und werden in dieser Form den `resort`-Modulen zugeführt. Dies ist bei der Berechnung der Adress-Offsets zu berücksichtigen.

Für die weiteren Berechnungen werden die Anzahl der Takte $T^k$ zum Füllen von SRA $S^k$, siehe Gl. 4.13 durch die Anzahl $M$ der DT-Pixel, die in einem Adresswort liegen, geteilt. Der Quotient, der mit $T^k_Q$, und dessen Rest, der mit $T^k_R$ bezeichnet wird, sind dann gegeben durch

$$T^k_Q = \lfloor \frac{T^k}{M} \rfloor \tag{4.16}$$

$$T^k_R = T^k \bmod M. \tag{4.17}$$

Für den Adress-Offset $O^k$, siehe Gl. 4.14 bzw. 4.15, sind ebenfalls dessen Quotienten $O^k_Q$ und dessen Rest $O^k_R$ zu berechnen

$$O^k_Q = \lfloor \frac{O^k}{M} \rfloor \tag{4.18}$$

$$O^k_R = O^k \bmod M. \tag{4.19}$$

Der zusätzliche Adress-Offset $O^k_{delay}$, der durch die Restterme entsteht und durch die Verzögerungsglieder in den `resort`-Modulen, siehe Abb. 4.7, ausgeglichen wird ist ist gegeben durch

$$O^k_{delay} = \underbrace{M - k}_{\text{Richtungen}} + \underbrace{M - T^k_R}_{\text{Füllen}} + \underbrace{M - O^k_R.}_{\text{Adress-Offset}} \tag{4.20}$$

Der erste Term ist notwendig, weil die DT-Pixel aus den unterschiedlichen Richtungsbereichen jeweils um einen Takt verzögert ausgelesen werden, beginnend mit Richtungsbereich Null. Die beiden anderen Terme entstehen zum einen durch die unterschiedliche

Anzahl der Takte, die zum Füllen der SRAs benötigt werden und zum anderen durch deren unterschiedlichen Anordnung, d.h. deren Adress-Offset.

Insgesamt kann sichergestellt werden, dass jedes SRAs in jedem Takt mit genau einem Pixel gefüllt wird. Dies bedeutet, dass die Pipeline nach deren Start nicht mehr angehalten werden muss und somit alle Register immer auf CE = *high* geschaltet sein können. Somit kann ein großes Auffächern[3] des *CE*-Signals des SRA-Moduls vermieden und das *Routing*-Netzwerk des FPGAs erheblich entlastet werden.

### Steuerung

Das in Abb. 4.8 dargestellte Steuerungsmodul `controlTM` besteht im Wesentlichen aus Elementen zur Adressgenerierung und zur Speicherung der Ergebnisse des Template-Matchings.

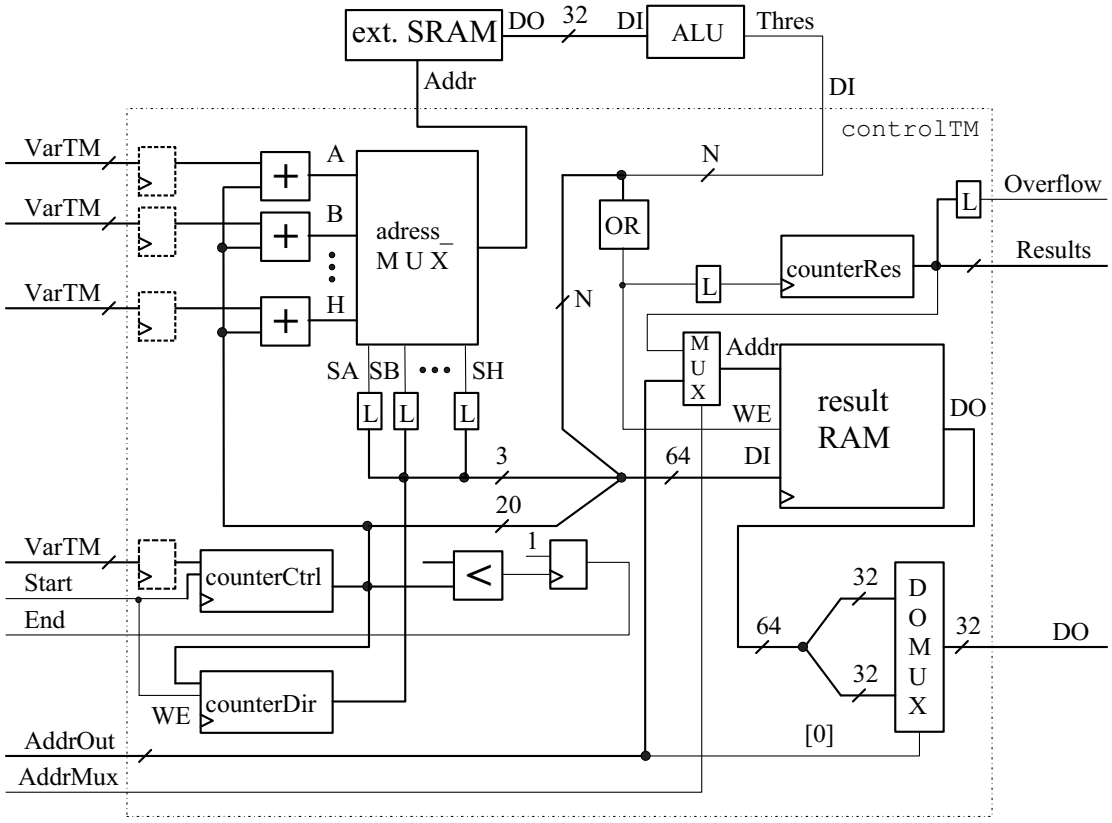



**Abbildung 4.8:** Steuerungsmodul für das parallele Template-Matching.

**Adressgenerierung** Auf der linken Seite in Abb. 4.8 sind die Hardware-Bausteine dargestellt, welche die Generierung der Adressen für das Auslesen der Pixel der acht DT-Bilder übernehmen. Die Pixel sind in unterschiedlichen Adressbereichen gespeichert und

[3] Engl. *fanout*.

müssen den acht `resort`-Modulen der ALU zugeführt werden. Das Template-Matching wird in Rückwärts-Richtung durchgeführt.

Die Adressgenerierung kann über das *Start*-Signal aktiviert werden. Der ladbare Rückwärts-Zähler `counterCtrl` deckt den für das Template-Matching relevanten Bildbereich ab. Dessen Startadresse $A_S$ ist gegeben durch

$$A_S = \frac{W\,H}{M} - 1 \tag{4.21}$$

und kann über einen Parameter des Steuerungsmoduls festgelegt werden oder ist über das *VarTM*-Signal während der Ausführung des Designs veränderbar. Der Rückwärts-Zähler wird in jedem achten Takt dekrementiert, entsprechend der acht Richtungsbereiche. Das Ende des Bildausschnitts wird über einen Komparator festgestellt, der dem Rückwärts-Zähler nachgeschaltet ist. Der Wert $A_E$ des Komparators, ist gegeben durch

$$A_E = \lfloor \frac{D_y\,W - max_k\,T^k}{M} \rfloor \tag{4.22}$$

und ist wiederum über einen Parameter des Steuerungsmoduls oder über das *VarTM*-Signal während der Ausführung des Designs einstellbar. Bei der Berechnung der Endadresse $A_E$ ist zu beachten, dass das Template-Matching im Randbereich des Bildes nicht durchgeführt wird.

Dem Komparator ist ein Flip-Flop (FF) nachgeschaltet, dessen Ausgang nach dem Erreichen der Endadresse auf eins gesetzt wird. Das Ende der Verarbeitung des Template-Matchings für ein Bild wird durch das *End*-Signal angezeigt. Des Weiteren wird über einen 3-Bit Zähler `counterDir`, der von 0 bis 7 läuft und in jedem Takt inkrementiert wird, eine feinere Unterteilung der Adresse realisiert. Dieser steuert die *Select*-Eingänge des Adressmultiplexers `adress_MUX`. An dessen Eingänen liegen die Summe aus dem globalen Zähler `counterCtrl` und den Adress-Offsets $O^k$, die in Gl. 4.14 bzw. 4.15 angegeben sind. Die Adress-Offsets können als Parameter des Steuerungsmoduls fest vorgegeben oder während der Laufzeit des Designs über die in Tab. 4.3 angegebenen Codewörter geändert werden.

**Speichermanagement der Ergebnisse** Die Hardware-Bausteine des FPGAs, die für das Speichern der Ergebnisse des Template-Matchings Verwendung finden, sind rechts in Abb. 4.8 eingezeichnet. Alle Signale der Ausgänge der $N$ Schwellwertmodule der ALU werden an das `resultRAM`-Modul geführt. Ein logischer OR-Baustein überprüft, ob mindestens eines der $N$ Schwellwertmodule aktiv ist. Sollte dies zutreffen, so werden die Daten der beiden Zähler `counterCtrl` und `counterDir` und die *Thres*-Signale der Ausgänge der $N$ Schwellwertmodule im `resultRAM` gespeichert. Außerdem wird der Adresszähler `counterRes` inkrementiert. Der Datenbus des Speichermoduls ist für 64 Datenbits und einer Datentiefe von 512 Worten ausgelegt. Bei 23 Adressbits können dann max. 41 *Thres*-Signale gespeichert werden. In dieser Arbeit werden maximal 36 *Thres*-Signale für die 36 Templates benötigt.

Außerdem werden keine Ergebnisse während dem Füllen der Pipeline und im Randbereich des Bildes gespeichert. Die hierfür notwendige Logik wurde mit Zählern und Komparatoren realisiert, die nicht im Steuerungsmodul eingezeichnet sind und generell mit `L`

für nicht näher beschriebene Logik gekennzeichnet sind. Des Weiteren wird ein *Overflow*-Signal erzeugt, falls die max. zu speichernde Anzahl von Ergebnissen überschritten wird.

Die im `ResultRAM` gespeicherten Ergebnisse werden über einen Multiplexer ausgelesen, über welchen der Datenbus von 64 Datenbits auf 32 Bits reduziert wird. Zum Auslesen eines Ergebnisses werden dann zwei Takte benötigt. Diese Adressen werden von einem übergeordneten Steuerungsmodul generiert, siehe Abb. 4.9, und über die Signale `AddrOut` und `AddrMux` an den *Select*-Eingang des Multiplexer vor dem `resultRAM` weitergeleitet. Das `ResultRAM`-Modul kann für das Schreiben von max. 512 Matching-Ergebnissen aus zwei BlockRAMs des VirtexII FPGAs oder aus internem CLB-RAM aufgebaut werden.

**Parametrisierung** Das Steuerungsmodul ist bezüglich der Bildgröße $H \times W$ parametrisierbar bzw. während der Laufzeit des Designs veränderbar. Dem ladbarern Zähler `conterCtrl` und dessen nachgeschalteten Komparator werden der Anfang des Bildbereichs bzw. das Ende des Bildbereichs übergeben. Diese beiden Werte sind nach den Gl. 4.21 und 4.22 zu bestimmen. Außerdem können die Adress-Offsets für die DT-Bilder, die in Gl. 4.14 bzw. 4.15 angegeben sind, über das *VarTM*-Signal auf die jeweilige Bildgröße eingestellt werden. Die Verzögerungsspeicher in den `resort`-Modulen (Abb. 4.7) sind ebenfalls gemäß Gl. 4.20 einzustellen. Alle Parameter können während der Ausführung des Designs über die Signale *VarTM* verändert werden. Die jeweils zu benutzenden Codewörter sind in Tab. 4.3 angegeben.

### Auswertung der Ergebnisse des Template-Matching auf dem PC

Die weitere Auswertung der bisher erzielten Ergebnisse des Template-Matchings auf dem FPGA, die im internen FPGA-RAM gespeichert wurden, wird vom PC durchgeführt. Über den PCI-Bus wird zunächst der Ergebniszähler `counterRes` ausgelesen und anschließend die entsprechende Anzahl an Ergebnissen aus dem `resultRAM`-Modul, siehe Abb. 4.8, via PCI-Bus zum PC transferiert.

Zum einen müssen die Adressen $A_{Ctrl}$ und $A_{Dir}$ der beiden Zähler `counterCtrl` und `counterDir` ausgelesen und auf den Ergebnispunkt $(G_h, G_w)$ des Matchings zurückgerechnet werden. Die Latenzzeit der Pipeline, die abhängig von der Größe und Lage der SRAs und der Anzahl der Registerstufen in den Addiererbäumen ist, wird in der folgenden Gleichung nicht berücksichtigt. Der Ergebnispunkt $(G_h, G_w)$ des Template-Matchings ist für ein Bild der Breite $W$ gegeben durch

$$A_{Res} = A_{Ctrl} - A_{Dir} \tag{4.23}$$

$$G_w = A_{Res} \; mod \; W \tag{4.24}$$

$$G_h = \lfloor \frac{A_{Res}}{W} \rfloor. \tag{4.25}$$

Aus den $N$-Bit Signalen der $N$ Schwellwerte wird der Template-Typ des Templates ermittelt. An einem Bildpunkt kann mehr als ein Template gefunden werden.

## 4.4 Ergebnisse und Zusammenfassung

Im Rahmen dieser Arbeit wurde der im vorigen Abschnitt beschriebene massiv parallele Ansatz für das Matching multipler Templates realisiert. Dabei konnten alle datenverarbeitenden Module der ALU nach dem Pipeline-Prinzip aufgebaut und zu einer großen Pipeline zusammengefasst werden. Die $N$ Distanzmaße für alle Templates $T_j, j = 0, \ldots, N-1$ an *einem* Bildpunkt werden gleichzeitig berechnet.

Die `resort`-Module haben die Aufgabe, die zu jeweils acht benachbarten zusammengefassten DT-Pixel der acht DT-Bilder aufzutrennen und den SRAs zuzuführen, in denen alle für die Templates relevanten DT-Pixel abgespeichert sind. Die Berechnung der Korrelation erfolgt mit Addiererbäumen, die das Herz der ALU darstellen, nach dem Pipeline-Prinzip aufgebaut sind und Zugriff auf die Register der SRAs haben. Alle anfallenden Additionen werden somit gleichzeitig durchgeführt. Die Distanzmaße werden von den Schwellwertmodulen gleichzeitig überprüft und die Ergebnisse des Template-Matchings zunächst im internen BlockRAM des FPGAs gespeichert. Die erfolgte Implementierung ist als eine Erweiterung von [76], siehe Abschn. 4.1 zu sehen.

Der Datenfluss konnte derart organisiert werden, dass die Pipeline nicht angehalten werden muss und alle datenverarbeitende Module die DT-Pixel aus den acht DT-Bildern parallel verarbeiten können, siehe Abb. 4.6. Aufgrund der Dimensionierung des Steuerungsmoduls folgt, dass die Anzahl der max. zu berechnenden Templates auf 41 und die max. Bildgröße auf $512 \times 512$ beschränkt ist. Diese ist jedoch leicht erweiterbar auf größere Bilder.

Insgesamt konnte eine hohe Parametrisierbarkeit der Module erreicht werden. Die $M$ SRAs und die $N$ Addiererbäume werden dabei abhängig von der Form der Templates $T_j$ generiert. Alle Module sind direkt oder indirekt bezüglich der Bildgröße $H \times W$ parametrisierbar. Die entsprechenden Parameter für die einzelnen Module werden dann in Abhängigkeit der Bildgröße und den Eigenschaften der Templates bestimmt. Diese Parameter können auch während der Ausführung des Designs über die in Tab. 4.3 gegebenen Codewörter auf dem FPGA geändert werden.

### Geschwindigkeit und Datendurchsatz

Der Vorteil der hier vorgestellten Implementierung ist, dass die Ausführungszeit des Algorithmus linear mit der Bildgröße $H \times W$ zunimmt und nicht von der Anzahl der Templates oder dem Bildinhalt abhängig ist.

Die Erstellung des Designs und dessen Synthese erfolgte mit CHDL (Version 1.106) [70]. Nähere Angaben hierzu sind in Abschn. 3.7 zu finden. Die max. Taktfrequenz mit der sich das FPGA-Design mit der *Local-Bus*-Clock speisen lässt beträgt 64 MHz. Für die Pipeline ergibt sich ein Datendurchsatz von einem DT-Pixel pro Takt, für jedes der acht DT-Bilder. Dies führt bei einer Größe des Ausgangsbildes von $512 \times 512$ zu einer Rechenzeit von $4.1$ ms und bei einer reduzierten Bildgröße von $256 \times 256$ von $1.1$ ms. Die Berechnung des Template-Matchings kann in Echtzeit durchgeführt werden, weil die Zeit für die Berechnung der DT-Bilder und des Matchings $12.4$ ms beträgt mit einer Bildwiederholrate von $80$ fps. Die für den MPRACE-Koprozessor gemessenen Ausführungszeiten für das Template-Matching sind in Tab. 4.1 angegeben.

| | MPRACE [ms] | PC [ms] |
|---|---|---|
| Schreiben der Parameter für das Matching | 0.1 | - |
| Matching, 12 (36) Templates | 4.1 (4.1) | 890 (2650) |
| DMA Ergebnisse Matching | 0.1 | - |
| Auswertung der Matching-Ergebnisse, 12 (36) Templates | 0.1 (0.2) | - |
| $\sum$ Matching, 12 (36) Templates | 4.4 (4.5) | 890 (2650) |

**Tabelle 4.1:** Rechenzeiten für das Template-Matching von 12 (36) Templates für das MPRACE-Board und Host-PC für DT-Bilder der Größe $512 \times 512$.

Wird zusätzlich das Matching auf den reduzierten Bildern mit geringerer Auflösung durchgeführt, so erfolgt zunächst die Berechnung der DT-Bilder und des Matchings mit Originalgröße. Anschließend werden die Parameter für eine reduzierte Bildgröße übertragen. Danach folgt die Ausführung des gesamten Algorithmus auf dem Bild mit geringerer Auflösung. Für ein Bild der reduzierten Größe $256 \times 256$ werden insgesamt $3.2$ ms benötigt, für beide Bilder insgesamt $15.5$ ms einer Bildwiederholrate von $64$ fps.

Die Rechenzeiten, die sich für das Template-Matching auf dem MPRACE-Board ergeben, werden mit den Rechenzeiten für das Matching auf dem Host-PC, der in Abschn. 1.3.5 beschrieben wurde, für zwölf bzw. 36 Templates verglichen, siehe Tab. 4.1. Insgesamt führt dies zu einer Beschleunigung bei der Ausführung des Algorithmus um ca. einen Faktor $200$ für zwölf Templates und einen Faktor $575$ für 36 Templates.

### Ressourcenverbrauch

In diesem Abschnitt wird der Ressourcenverbrauch für das implementierte FPGA-Design für das Matching der kreisförmigen und dreieckigen Templates, die in den Abb. 2.8 bis 2.10 dargestellt sind, untersucht. Die benötigten Logikblöcke und internes BlockRAM für den Virtex-II XC2V3000 FPGA sind in Tab. 4.2 angegeben.

Insgesamt zeigt sich, dass der FPGA-Ressourcenbedarf für die parallele Implementierung des Template-Matchings für die SRAs und die Addiererbäume sehr hoch ist und auf dem Virtex-II 3000 FPGA das Matching nur für 24 Templates, durchgeführt werden kann. Die Module, welche mit Abstand die meisten FPGA-Ressourcen benötigen, sind die SRAs und die Addiererbäume. Der FPGA-Ressourcenverbrauch der SRAs ist gegeben mit $\sum_{k=0}^{M-1} D_x^k D_y^k$ und der für die Addiererbäume ist von der Größenordnung $O(\sum_{j=0}^{N-1} T_j - 1)$. Für die Addiererbäume sind zusätzlich die Register, die in den jeweiligen Registerstufen eingefügt werden, zu beachten. Hier wurde nach jeder dritten Addiererstufe ein Register eingefügt. Hinzu kommen die $M$ `resort`-Module, $N$ `threshold`-Module sowie das Steuermodul `controlTM`. Die letzteren Module benötigen wenige FPGA-Ressourcen mit einer Auslastung des Virtex-II 3000 FPGAs von $4\%$.

Bei den Shift Register Arrays werden viele Register ausschließlich zum Weiterleiten der Daten benötigt. Für jedes Template wird ein eigener binärer Addiererbaum generiert. Vorhandene Überdeckungen einzelner Templateelemente verschiedener Templates zwischen den Addiererbäumen werden nicht ausgenutzt. Optimierungen hierfür sind ausführlich im nächsten Kapitel beschrieben. Auf den Datenfluss und die Rechenzeiten werden

| Modul | Slices | (in %) | Block-RAM |
|---|---|---|---|
| `resort` | 8*34 | (2) | 0 |
| `sra` 12 | 3136 | (22) | 16 |
| `sra` 24 | 4948 | (35) | 22 |
| `sra` 36 | 6760 | (47) | 28 |
| `addertree` 12 | 3517 | (25) | 0 |
| `addertree` 24 | 7026 | (49) | 0 |
| `addertree` 36 | 10547 | (74) | 0 |
| `threshold` | 12(24)*6 | (1) | 0 |
| `controlTM` | $\approx$ 270 | (2) | 2 |
| $\sum$ Template-Matching 12 | 7267 | (51) | 18 |
| $\sum$ Template-Matching 24 | 12660 | (88) | 24 |
| $\sum$ DT-Bilder + Template-Matching 12 | 10443 | (73) | 22 |
| $\sum$ DT-Bilder + Template-Matching 24 | 14334 | (100) | 28 |

**Tabelle 4.2:** Ressourcenbedarf der Module für das parallele Template-Matching für den Virtex-II XC2V3000 FPGA von Xilinx. Die Module sind ausgelegt für DT-Bilder der Größe $512 \times 512$ und vier Bit Pixel.

die Optimierungsstrategien, die ausschließlich den FPGA-Ressourcenbedarf minimieren, keine Auswirkungen haben.

### 4.4.1 Gesamtschaltung

Einen Überblick über die Gesamtschaltung des FPGA-Designs wird in Abb. 4.9 gegeben. Zusammenfassend wird nochmals der Datenfluss beschrieben. Die Kameradaten vom `camctrl`-Modul, siehe Abschn. 3.2, werden zunächst in das SRAM 0 des MPRACE-Boards geschrieben. Das Bild wird ausgelesen und durch die Pipeline 1 (Sobel, Cleaning, Orientierung, DT-Vorwärts) zur Berechnung der DT-Bilder geschoben und im SRAM 1 des FPGA-Koprozessors gespeichert, siehe Abschn. 3.6. Die Zwischenergebnisse werden in Rückwärts-Richtung ausgelesen und der Pipeline 1 (DT-Rückwärts, `sort`) zugeführt und die DT-Bilder in unterschiedlichen Bereichen von SRAM 0 gespeichert. Anschließend werden die DT-Bilder der Pipeline des Template-Matchings (`resort`, SRAs, Addiererbäume, `threshold`) zugeführt und dessen Ergebnisse im interen BlockRAM des FPGAs gespeichert, siehe Abschn. 4.3. Diese werden nach Beendigung des Matchings via PCI-Bus zum Host-PC übertragen und auf diesem weiter ausgewertet.

**Codewörter und *Flag*-Signale** Die Codewörter zum Beschreiben der datenverarbeitenden Module und der Steuermodule sind in Tab. 4.3 angegeben. Im oberen Teil der Tabelle sind die globalen Codewörter zu sehen, mit denen die einzelnen Teilaufgaben gesteuert werden. Außerdem kann ein globaler Reset und die Übertragung der Daten mit dem DMA-Modus über den PCI-Bus aktiviert werden. Es folgen die Codewörter für die Module zur Berechnung der DT-Bilder und des Template-Matchings.

Die von den Steuermodulen generierten *Flag*-Signale werden über einen PCI-Multiplexer

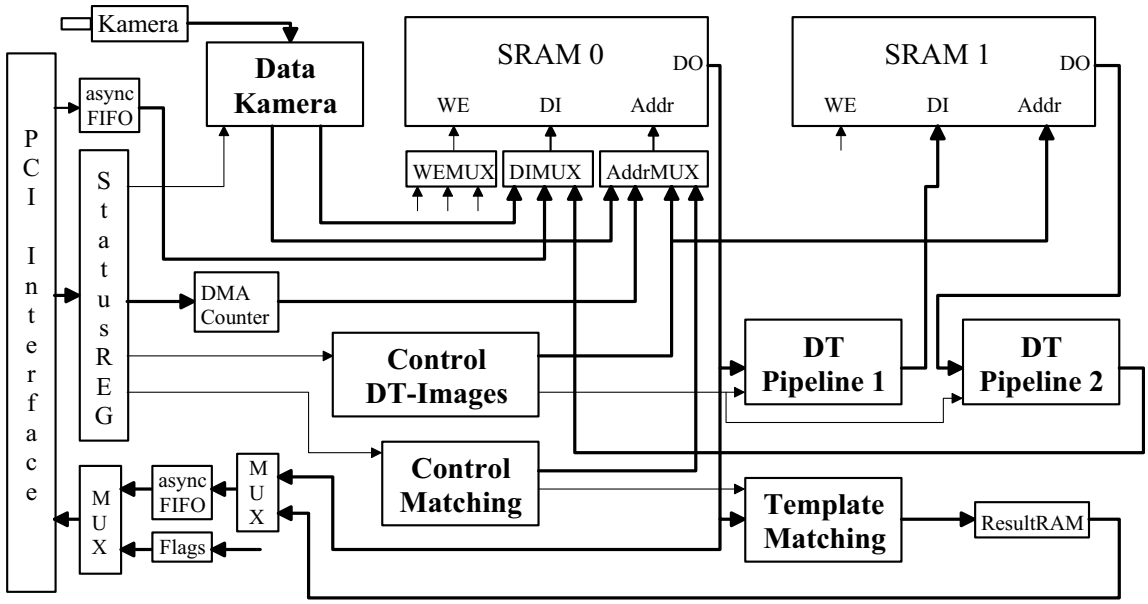



**Abbildung 4.9:** Überblick über die Gesamtschaltung zur Bildaufnahme, Berechnung der DT-Bilder und Ausführung des parallelen Template-Matchings.

an den Host-Rechner weitergeleitet. Über diesen kann jederzeit der Status der Ausführung des Designs abgefragt werden. Die Belegung der Pins ist in Tab. 4.4 angegeben.

### 4.4.2 Ausblick

Bei der sequentiellen Berechnung der DT besteht die Möglichkeit, die Pipeline in Rückwärts-Richtung, siehe Abschn. 3.6, und die des Template-Matchings zu einer Pipeline zusammenzufassen. Die DT-Bilder sind dann nach deren Berechnung nicht mehr im externen SRAM des FPGA-Prozessors zwischenzuspeichern. Die unterschiedlichen Adress-Offests der SRAs können dann ausgeglichen werden, indem die Pixel der DT-Bilder im internen BlockRAM des FPGAs verzögert werden. Die Ausführung dieser beiden Pipelines könnte dann ungefähr um einen Faktor zwei beschleunigt werden.

Außerdem besteht die Möglichkeit das Kameramodul, die Pipeline zur Berechnung der DT-Bilder und die des Template-Matchings zu einer großen Pipeline zusammenzufassen, vorausgesetzt die DT-Bilder werden mit der parallelen Methode berechnet, siehe Abschn. 3.5. Dann erhält man die Ergebnisse des Template-Matchings nach der Latenzzeit der Pipeline, welche unter 1 ms liegt. Allerdings würde sich der Ressourcenbedarf für die Berechnung der DT-Bilder auf dem FPGA um etwas weniger als einen Faktor 16 erhöhen.

Sind Bilder mit einer Bildbreite $W$ größer als 512 Pixel auf dem Virtex-II FPGA zu berechnen, so kann internes BlockRAM des FPGAs eingespart werden, falls die Berechnung des Matchings nicht auf den DT-Bildern als Ganzes erfolgt, sondern auf Streifen von diesen. Die maximale Breite eines Bildstreifens könnte 512 Pixel plus der Ausdehnung $D_x$ des kleinsten SRAs in x-Richtung betragen. Zu beachten ist, dass sich zwei benachbarte Bildstreifen überlappen müssen, weil am Rand der Bildstreifen keine Ergebnisse des

| Status-Register [Bits] | Aktion |
|---|---|
| 4 . . . 0 | globale Steuersignale |
| 0 | Reset/Set |
| 1 | Schreibe Daten über DMA |
| !3 & 2 | Start DT-Bilder |
| 3 & 2 | Start Bildaufnahme |
| 3 & !2 | Start Template-Matching |
| 4 | Lesen Daten über DMA |
| 15 . . . 5 | Schreibe Parameter (Steuersignale) |
| 5 & 4 | Schreibe Parameter DT-Bilder |
| !5 & !4 | Schreibe Parameter Template-Matching |
| !15 & !14 &13 & 12 & 5& 4 | Schwellwert (Binarisierung) |
| !15 & 14 &13 & 12 & 5& 4 | *Cleaning* |
| 15 & !14 &!13 & 12 & 5& 4 | DT-Metrik |
| 11 . . . 6 & 5 & !4 | Steuerung DT-Bilder |
| !11 . . . !6 & !5 | Schwellwert Template-Matching |
| ⋮ | ⋮ |
| 11 . . . 6 & !5 | SRA + Steuerung Template-Matching |
| 31 . . . 16 | Daten/Adressen |

**Tabelle 4.3:** Codewörter für das Statusregister zur Steuerung des FPGA-Designs vom PC.

| *Flag*-Signale [Bits] | Bedeutung |
|---|---|
| 0 | Ende Vorverarbeitung |
| 1 | Ende Bildaufnahme |
| 2 | Ende Template-Matching |
| 12 . . . 3 | Anzahl Ergebnisse TM |
| 13 | Overflow |
| 31 . . . 14 | Optional, z.B. Debug |

**Tabelle 4.4:** Belegung der *Flag*-Signale am PCI-Interface.

Template-Matchings gefunden werden können. Die Rechenzeit würde sich dadurch leicht erhöhen.

Führt man das Template-Matching auf DT-Bildern aus, die mit der *chamfer*-1-1 DT-Metrik berechnet werden, siehe Abschn. 2.1.3, könnten die DT-Pixel auf den max. Wert sieben (drei) abgeschnitten und das Template-Matching auf Pixel mit drei (zwei) anstelle von vier Bit durchgeführt werden. Dies würde bei den SRAs des Template-Matchings zu einer Ressourcenersparnis von 25% (50%) führen und bei den Addiererbäumen zu einer etwas geringeren Ressourcenersparnis als 25% (50%). Die DT-Bilder könnten zunächst mit der *chamfer*-2-3 Metrik berechnet werden. Als Approximation für die *chamfer*-1-1 DT-Pixel würde das niederwertigste Bit der mit der *chamfer*-2-3 Metrik berechneten DT-Pixel nicht benutzt werden. Der Nachteil beim Matching mit der *chamfer*-1-1 Metrik besteht jedoch darin, dass sich die Anzahl der Matching-Ergebnisse leicht erhöhen würde. In

einem weiteren Schritt könnten dann auf dem PC die Matching-Ergebnisse nochmals für DT-Bilder, die mit der *chamfer*-2-3 Metrik berechnet wurden, überprüft werden. Als Nachteil ist jedoch zu sehen, dass die DT-Bilder via PCI-Bus ins RAM des PCs übertragen werden müssten. Alternativ könnte das nachfolgende Matching auf dem FPGA mit einer sequentiellen Korrelationseinheit (aus Abschn. 4.2.1) für die mit der *chamfer*-2-3 Metrik erzeugten DT-Pixel durchgeführt werden.

Der Algorithmus [1] ist nicht invariant gegenüber Rotationen und Verzerrungen der Templates. Gedrehte bzw. verzerrte Templates können nur dann im Bild gefunden werden, falls diese explizit berechnet, d.h. zu den bisherigen Templates hinzugenommen werden. Die Anzahl der zu berechnenden Templates und damit der Ressourcenbedarf auf dem FPGA würde sich dann um ein Vielfaches erhöhen. Eine andere Strategie für das Auffinden der gedrehten bzw. verzerrten Templates im Bild besteht darin, zunächst rotierte bzw. verzerrte Bilder zu berechnen und auf diesen das Matching durchzuführen. Diese würden dann nacheinander durch die drei Pipelines zur Berechnung der DT-Bilder und des Template-Matchings auf dem FPGA geschoben werden. Die Rechenzeiten würde sich dann entsprechend vervielfachen. Zusätzliche Ressourcen müssten jedoch nur für die Module zur Berechnung der gedrehten bzw. verzerrten Bilder bereitgestellt werden. Eine FPGA-Implementierung für rotierte Bilder ist in [79] zu finden.

Kapitel 5

# Optimierungen beim parallelen Template-Matching

Nachdem im vorigen Kapitel 4 eine Basisimplementierung für das parallele Template-Matching beschrieben wurde, werden in diesem Kapitel vier Strategien zur Reduktion der FPGA-Ressourcen vorgestellt.

Zunächst wird in Abschn. 5.1 angegeben, wie der Ressourcenverbrauch der Shift Register Arrays (SRAs) durch das Einfügen von Verzögerungsgliedern (SRLTs) für das FPGA reduziert werden kann.

Für die Optimierungen des FPGA-Ressourcenbedarfs der binären Addiererbäume für das Template-Matching, werden zunächst in Abschn. 5.2 Grundlagen über gerichtete Graphen und Addierergraphen beschrieben und es wird auf bisherige Arbeiten für Hardware-Optimierungen für gemeinsame Addiererbäume eingegangen. In Abschn. 5.3 wird ein iterativer Algorithmus eingeführt, der den Aufbau von optimierten Addierergraphen ermöglicht. Die Teilsummen, die für mehrere Templates zu berechnen sind, werden nicht mehrmals, sondern mit gemeinsamen Addiererknoten berechnet. Die Auswahl eines Addiererknotens in jedem Iterationsschritt erfolgt mit einem lokalen Entscheidungskriterium. Die Erweiterungen des Algorithmus für Templates, die als Ganzes verschoben sind und für Templates mit einzeln verschobenen Templateelementen werden in den Abschn. 5.4 und 5.5 gegeben. Die Besonderheiten beim Aufbau der Addiererbäume werden in jedem der Abschnitte zunächst anhand von Beispielen erklärt und motiviert. Es folgt eine Beschreibung des iterativen Algorithmus. Zum Schluss der jeweiligen Abschnitte werden die einzelnen Iterationsschritte des Algorithmus anhand von Beispielen nochmals ausführlich erklärt.

Eine weitere Möglichkeit zur Reduzierung des Ressourcenbedarfs auf dem FPGA besteht darin, Templates zu verwenden, die bezüglich der Anzahl der Templateelemente reduziert sind, siehe Abschn. 5.6. Auf dem FPGA wird dann zunächst ein “gröberes” Template-Matching durchgeführt.

Strategien für das Verschieben der Templates, die zu vielen Überlappungen zwischen den einzelnen Templateelementen führen, sind in Abschn. 5.7 beschrieben. Die im Rahmen dieser Arbeit benutzten Templates, die als Ganzes oder deren einzelne Templateelemente verschoben werden, sind außerdem graphisch dargestellt.

Der Bedarf an FPGA-Ressourcen für die SRAs bzw. optimierten SRAs und die binären bzw. optimierten Addiererbäume ist für die unverschobenen und verschobenen Templates in Abschn. 5.8 angegeben.

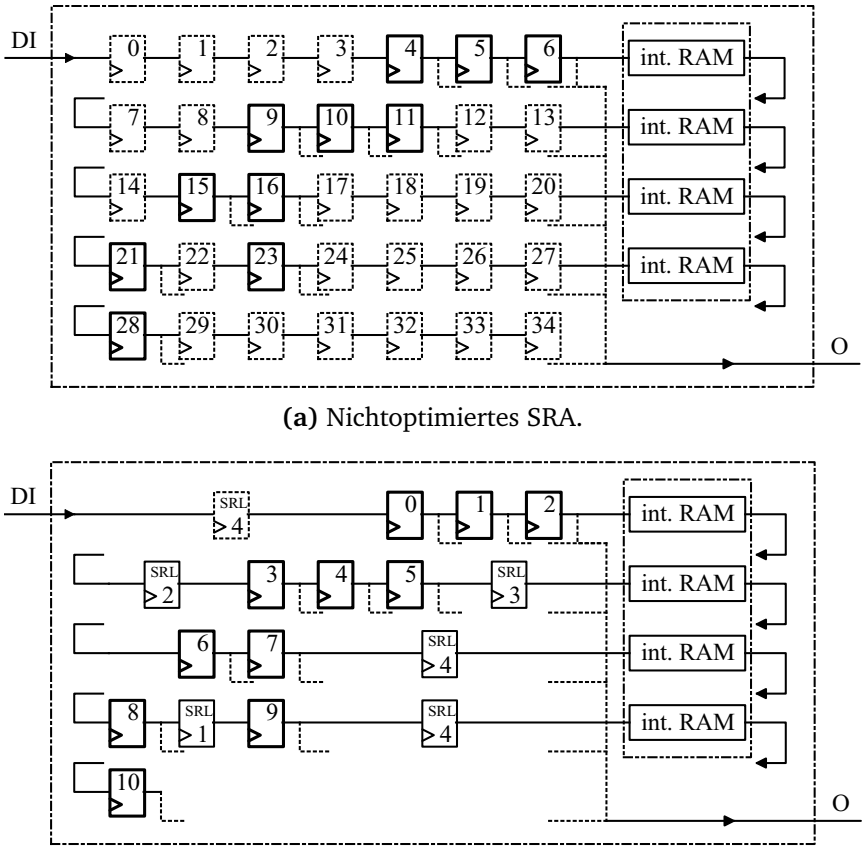
## 5.1 Optimierte SRAs

Beim Aufbau von optimierten Shift Register Arrays (SRAs) werden die Shift-Register, auf die nicht von den Addiererbäumen zugegriffen wird, durch die ressourcensparenden *Shift-Register-Look-Up Tables* (SRLTs) ersetzt. Die SRLT-Bausteine wurden erstmals bei den Virtex FPGAs von Xilinx, siehe Abschn. 1.3.1, eingeführt. Sie werden im weiteren Verlauf dieses Kapitels auch als Verzögerungsglieder oder Verzögerungsknoten bezeichnet.

Innerhalb der Slices des Virtex-II FPGAs können alle LUTs als 16-Bit *Shift*-Register (SRL16) konfiguriert werden, ohne dabei die FFs der Slices zu benutzen. Deren Länge kann fest vorgegeben werden oder ist dynamisch änderbar. Im Rahmen dieser Arbeit wurden ausschließlich statische SRLTs mit fester Länge eingesetzt. Die Verwendung der SRLTs anstellte von Registern führt zu einer max. Ressourcenersparnis um den Faktor 16.

Anhand von Abb. 5.1 wird erläutert, welche Besonderheiten beim Aufbau der optimierten SRAs (oSRAs) auftreten können. Die Register eines SRAs, siehe Abb. 5.1(a), seien von links oben nach rechts unten zeilenweise durchnummeriert. Alle zusammenhängenden Register, auf die nicht von den Addierern zugegriffen wird, und die in Abb. 5.1(b) gestrichelt gezeichnet sind, werden zusammengefasst und durch SRLTs mit der entsprechenden Länge ersetzt. Dies geschieht auch über den Rand hinweg, was anhand des Beispiels am Ende der zweiten und am Anfang der dritten Zeile illustriert ist und die Register 12, 13 und 14 betrifft. Die Register werden ebenfalls über mehrere aufeinander folgende Zeilen zusammengefasst, falls deren Register nicht mit den Addiererbäumen zu verbinden sind. Die zusammenhängenden Register, die der letzten Zeile angehören und sich nach dem letzten von den Addiererbäumen benutzten Register befinden, werden komplett gestrichen, wie in Abb. 5.1 anhand den Registern 29 bis 34 zu sehen ist. Die ersten zusammenhängenden Register der ersten Zeile, die nicht mit den Addiererbäumen zu verbinden sind, können ebenfalls durch ein SRLT entsprechender Länge ersetzt werden, siehe Register null bis drei in Abb. 5.1. Außerdem besteht die Möglichkeit dieses SRLT zu streichen und stattdessen zu den Adress-Offsets des Adressgenerators, die in Gl. 4.20 gegeben sind, die entsprechende Länge des SRLTs zu addieren. Ein einzelnes, nicht von den Addiererbäumen benötigtes Register, siehe Register 22 in Abb. 5.1, wird durch ein SRLT der Länge eins ersetzt, obwohl es einen gleichwertigen Ressourcenbedarf wie ein Register hat. Der Vorteil besteht jedoch darin, dass es mit einem der benachbarten Register effizient in ein Slice gepackt werden kann und externe *Routing*-Ressourcen eingespart werden.

Durch die Veränderung der Anzahl der Register und deren Verbindungsstruktur ändert sich auch deren Nummerierung, siehe Abb. 5.1. Zwischen den Registern $\tilde{r}_l$ des optimierten SRAs und den Registern $r_{l'}$ des nicht optimierten SRAs existiert eine eindeutige Zuordnung. Diese ist auf die in Abschn. 4.3.2 beschriebenen Verbindungen mit den Addiererbäumen anzuwenden.




**(a)** Nichtoptimiertes SRA.

**(b)** Optimiertes SRA (oSRA).

**Abbildung 5.1:** Ressourcen-Optimierung eines SRAs (Shift Register Arrays).

**Ressourcenabschätzung** Der Ressourcenbedarf für die Register eines nichtoptimierten SRAs $S^k$ ist $D_x^k * D_y^k$, also abhängig von der max. Ausdehnung der SRAs in x- und y-Richtung, siehe Abschn. 4.3.1. Beim optimierten SRA sind die Ressourcen einerseits abhängig von der Anzahl der Register, auf die von den Addiererbäumen zugegriffen wird und andererseits von der Anzahl der SRLTs. Die Anzahl der Register ist stark von den Überlappungen der einzelnen Templateelemente der verschiedenen Templates abhängig. Falls von den Addiererbäumen auf keines der Register mehrmals zugegriffen wird, ist die Anzahl der Register in den SRAs $|\tilde{R}| = \sum_j |T_j|$. Bei vielen Überlappungen der $N$ Templates gilt $|\tilde{R}| << \sum_j |T_j|$. Die Anzahl der Verzögerungsglieder bei den oSRAs ist i.A. schwer abzuschätzen und stark abhängig von den Eigenschaften der Templates.

## 5.2 Allgemeines zum Aufbau von Addierergraphen

Zunächst werden in diesem Abschnitt (gerichtete) Graphen und deren Eigenschaften beschrieben. Die Einführung der Notation für gerichtete Graphen ist sehr nahe an [82] und [83] gehalten. Die bisher verwendeten binären Addiererbäume zur Berechnung der Ähnlichkeitsmaße beim Template-Matching werden genauer betrachtet und formal mit der Graphentheorie beschrieben. Außerdem wird auf bisherige Arbeiten für ressourcen-

sparende Addiererbäume für FPGAs eingegangen.

### 5.2.1 Gerichtete Graphen

Ein (gerichteter) Graph oder Digraph[1] $G(V, E)$ ist ein geordnetes Paar $(V, E)$, wobei $V$ eine endliche Menge von Knoten[2] und $E$ eine Menge von (gerichteten) Kanten[3] zwischen den Knoten ist. Die Kantenmenge $E$ ist eine Teilmenge der zweielementigen Teilmengen von $V$, also $E = \{(u, v)\}|u, v \in V, u \neq v\}$, und definiert eine Relation auf der Menge der Knoten. Eine gerichtete Kante verläuft genau dann vom Konten $u$ zum Knoten $v$, wenn $(u, v) \in E$. Sie wird graphisch als Pfeil von $u$ nach $v$ dargestellt. Bei einem gerichteten Graphen ist die Richtung des Datenflusses fest vorgegeben.

Ein (gerichteter) Pfad der Länge $k$ ist eine Folge $W = (v_0, \ldots, v_k)$ von paarweise verschiedenen Knoten aus $V$, so dass $(v_i, v_{i+1}) \in E$ für alle $i = 0, \ldots, k-1$, d.h. jeweils zwei aufeinander folgende Knoten sind jeweils durch eine (gerichtete) Kante miteinander verbunden. Ein Pfad heißt zyklisch, falls der Anfangsknoten $v_0$ und der Endknoten $v_k$ identisch sind. Ein gerichteter Graph $G$ ist azyklisch oder kreisfrei, falls er keinen zyklischen Pfad enthält.

Ist $(u, v)$ eine gerichtete Kante, so wird $u$ Vorgänger von $v$ und $v$ Nachfolger von $u$ genannt. Für einen Knoten $v$ wird die Anzahl der gerichteten Kanten, die zu $v$ führen, als Eingangsgrad bezeichnet. Entsprechend ist der Ausgangsgrad eines Knotens $v$ die Zahl der Kanten, die in $v$ beginnen. Ein gerichteter Graph $G(E, V)$ heißt stark zusammenhängend, wenn für jedes Paar von Knoten $u, v \in V$ ein gerichteter $u-v$-Pfad existiert und (schwach) zusammenhängend, falls für jedes Paar von Knoten ein $u - v$-Pfad in $G$ existiert.

Ein Baum ist ein zusammenhängender, azyklischer Graph. Ein Knoten $v$ eines Baumes mit Eingangsgrad 0 heißt Blatt. Ein Baum heißt gewurzelt oder Wurzelbaum, wenn es genau einen Knoten mit Ausgangsgrad 0 gibt. Dieser Knoten wird Wurzel genannt. Ein Knoten $v$ des Graphen, der weder Wurzel noch Blatt ist, wird als innerer Knoten bezeichnet. Die Länge des (eindeutigen) Pfades von der Wurzel zum Knoten $v$ wird Tiefe von $v$ genannt. Ein Binärbaum ist ein Wurzelbaum, in dem jeder Knoten höchstens zwei unmittelbare Vorgänger hat.

Ein Graph $H(V_H, E_H)$ heißt (schwacher) Teilgraph oder Subgraph eines Graphen $G(V_G, E_G)$, falls $V_H \subseteq V_G$ und $E_H \subseteq E_G$ gilt.

### 5.2.2 Addierergraphen

Die parallele Berechnung der Distanzmaße für das Template-Matching von $N$ Templates wurde bisher, siehe Abschn. 4.3, mit $N$ unabhängigen binären Addiererbäumen $A_j, j = 0, \ldots, N-1$ durchgeführt. Diese wurden bisher für jedes Template $T_j$ unabhängig voneinander aufgebaut. Gemeinsame Partialsummen, die von zwei oder mehreren Addiererbäumen gemeinsam auszuführen sind, werden mehrmals berechnet.

[1] Engl. Directed Graph.
[2] Engl. vertices.
[3] Engl. edges.

In den folgenden Abschnitten werden Methoden vorgestellt, wie für alle Templates $T_j$ ein gemeinsamer Addierergraph, der mit $A$ bezeichnet wird, aufgebaut werden kann. Das Ziel beim Aufbau des Addierergraphen $A$ besteht darin, dass gemeinsame Summen, die von mehreren Addiererbäumen $A_j$ zu berechnen sind, gemeinsam von einem Addierer ausgeführt werden. Insgesamt werden dabei weniger Ressourcen benötigt, als bei den voneinander getrennt aufgebauten binären Addiererbäumen. Der Addierergraph $A$, der i.A. kein Baum ist, soll aus $N$ Teilgraphen bestehen. Diese sollen die $N$ Addiererbäume $A_j$ für die $N$ Templates $T_j$ sein. Die Vereinigung der Addiererbäume $A_j$ ist dann der Addierergraph mit $A = \cup A_j$.

In den Abb. 5.2(a) bis Abb. 5.2(c) sind für ein einfaches Beispiel der Aufbau von binären Addierbäumen und gemeinsamen Addierergraphen dargestellt.

Die Knoten im Addierergraphen $A$ können nicht nur Addierer, sondern auch Register oder Verzögerungsglieder (SRLTs) sein. Wie bereits bei den binären Addiererbäumen aus Abschn. 4.3.2 beschrieben, werden auch hier Register in den Addiererbaum eingefügt, um die Gatterlaufzeit zwischen zwei Register der später auf dem FPGA realisierten Schaltung zu begrenzen. Zusätzlich wird zugelassen, Verzögerungsglieder in den Addierergraphen einzufügen. Der Grund liegt darin, weil es in dem späteren Abschn. 5.5 erlaubt wird, die Templateelemente der Templates einzeln zu verschieben. Diese Verschiebungen sind durch Verzögerungsglieder auszugleichen. Die unterschiedlichen Knoten werden als Addiererknoten $\tilde{a}_l$, Registerknoten $\tilde{r}_l$ und Verzögerungsknoten $\tilde{v}_l$ bezeichnet. Wie bereits erwähnt, kann jeder Addiererknoten $\tilde{a}_l$, Registerknoten $\tilde{r}_l$ oder Verzögerungsknoten $\tilde{v}_l$ im Addierergraphen $A$ Teil mehrerer Addiererbäume $A_j$ sein, die Teilgraphen des Addierergraphen $A$ sind.

Die Definition des im vorigen Abschnitt angegebenen Begriffs Tiefe wird geändert, weil zu Beginn des Aufbaus des Addierergraphen die Wurzeln der einzelnen Addiererbäume $A_j$ noch nicht bekannt sind. Den Registerknoten $\tilde{r}_l$ der SRAs, siehe z.B. Abschn. 4.3.1, welche dem Addierergraphen die Eingangsdaten liefern und im Weiteren als Blätter des Addierergraphen aufgefasst werden, wird die Tiefe 0 zugewiesen. Die Tiefe eines Addiererknotens $\tilde{a}_l$ ist die max. Länge der Pfade von $\tilde{a}_l$ zu den Blättern und nicht wie bisher, zur Wurzel. Der Addiererknoten $\tilde{a}_l$, der Element des Addiererbaums $A_j$ ist und von allen zum Addiererbaum $A_j$ gehörenden Addiererknoten die größte Tiefe besitzt, ist die Wurzel des Addiererbaumes $A_j$. Der Datenfluss ist beginnend bei den Blättern hin zur Wurzel der jeweiligen Addiererbäume.

Der Eingangsgrad eines Addiererknotens $\tilde{a}_l$ ist zwei und der der Register- oder Verzögerungsknoten jeweils eins. Der Ausgangsgrad eines Addiererknotens ist bei einem binären Addiererbaum gleich eins. Bei dem Addierergraphen $A$ können die Ausgänge der Addiererknoten jedoch mit mehreren nachfolgenden Addiererknoten verbunden werden. Der Ausgangsgrad ist demnach größer gleich eins. Dies gilt ebenfalls für die Register- und Verzögerungsknoten.

Für weitere Eigenschaften der Knoten, die für den Aufbau des Addierergraphen relevant sind, wird die folgende Notation eingeführt. Des Weiteren wird die Notation für Schwellwertknoten, die mit $\tilde{s}_j$ bezeichnet und den Wurzeln der jeweiligen Addiererbäumen $A_j$ nachgeschaltet sind, eingeführt.

- **Addiererknoten:** Für den Addiereknoten $\tilde{a}_l$ wird dessen Tiefe mit $t(\tilde{a}_l)$ und dessen

Anzahl der Addiererknoten, die Vorgänger von $\tilde{a}_l$ sind, mit $v(\tilde{a}_l)$, sowie dessen Bitbreite mit $b(\tilde{a}_l)$ bezeichnet. Die Bitbreite $b(\tilde{a}_l)$ von $\tilde{a}_l$ ist abhängig von der Anzahl der vorausgehenden Knoten $v(\tilde{a}_l)$ und wird bestimmt durch

$$b(\tilde{a}_l) = b + \lceil \frac{log(v(\tilde{a}_l))}{log(2)} \rceil. \tag{5.1}$$

Die Menge aller Addiererknoten wird mit $\tilde{A}$ bezeichnet.

- **Registerknoten:** Für den Registerknoten $\tilde{r}_l$ wird dessen Tiefe mit $t(\tilde{r}_l)$ und dessen Bitbreite mit $b(\tilde{r}_l)$ bezeichnet. Die Menge aller Registerknoten sei $\tilde{R}$.
- **Verzögerungsknoten:** Für den Verzögerungsknoten $\tilde{v}_l$ wird dessen Tiefe mit $t(\tilde{v}_l)$ und dessen Bitbreite mit $b(\tilde{v}_l)$ bezeichnet. Die Menge aller Verzögerungsknoten sei $\tilde{V}$.
- **Schwellwertknoten:** Für den Schwellwertknoten $\tilde{s}_j$ wird dessen Tiefe mit $t(\tilde{s}_j)$ und dessen Bitbreite mit $b(\tilde{s}_j)$ bezeichnet. Die Menge der Schwellwertknoten sei $\tilde{S}$.

Die unterschiedlichen Typen von Knoten für den Addierergraphen sind in der Menge der Knoten $V = \{\tilde{R} \cup \tilde{A} \cup \tilde{V}\}$ zusammengefasst. Die jeweiligen Typen von Knoten werden durch einen unterschiedlichen Wert gekennzeichnet. Den Registerknoten wird der Wert 0 zugewiesen, den Addiererknoten der Wert eins und den Verzögerungsknoten der Wert zwei.

### 5.2.3 Bisherige Arbeiten

#### Zerlegung in Basistemplates

In dem in Abschn. 4.1 und [76] beschriebenen parallelen Ansatz zur Berechnung der Korrelation multipler Templates ist eine Vorgehensweise für den Aufbau von ressourcensparenden Addiererbäumen skizziert. Hierbei werden die Templateelemente mehrerer Templates, welche überlappen, durch gemeinsame Addierer aufgebaut, so dass FPGA-Ressourcen eingespart werden.

In [76] wird anhand eines Beispiels die iterative Vorgehensweise grob umrissen. Die Menge der Templates wird zunächst in Untermengen, den sog. Basistemplates aufgeteilt. Überlappende Templateelemente, die einer festen Kombination von Templates angehören, bilden ein Basistemplate. Für jede der Untermengen werden Addiererbäume berechnet. Die beiden Basistemplates, aus welchen die Templates am häufigsten aufgebaut sind, werden durch ein neues Basistemplate ersetzt und die Addiererbäume für die Templateelemente der beiden Basistemplates werden zusammengefasst. Dieser Teil der Addiererbäume wird dann von mehreren Templates gemeinsam benutzt. In einem letzten Iterationsschritt wird ein Basistemplate erzeugt, welches alle Templateelemente umfasst.

In [76] sind keine Details und keine Analyse des Algorithmus gegeben. Dieser wird jedoch anhand eines Beispiels anschaulich erläutert.

## 5.3 Optimierte Addiererbäume

In diesem und den beiden folgenden Abschnitten werden Methoden für den gemeinsamen Aufbau des Addierergraphen $A$ bzw. der $N$ Addiererbäumen $A_j$ der $N$ Templates $T_j$ angegeben. Ziel des im Folgenden dargestellten Algorithmus ist es, gemeinsame Zwischensummen, die von unterschiedlichen Addiererbäumen auszuführen sind, nicht mehrmals, sondern nach Möglichkeit nur einmal zu berechnen. Die gemeinsamen Zwischensummen treten genau dann auf, wenn sich mindestens zwei Templateelemente unterschiedlicher Templates überdecken, d.h. es gilt$(t_{i,j}, t_{i',j}) = (t_{i'',j'}, t_{i''',j'})$ mit $j \neq j'$, $0 \leq i, i' < |T_j| - 1, 0 \leq i'', i''' < |T_{j'}| - 1$.

Aspekte, die beim Aufbau der Addiererbäume zu berücksichtigen sind, werden zunächst in Abschn. 5.3.1 anhand eines Beispiels näher betrachtet. Der im Rahmen dieser Arbeit entwickelte und implementierte Algorithmus zum Aufbau des Addierergraphen $A$ bzw. der Addiererbäume $A_j$ arbeitet mit einem lokalen Optimalitätskriterium und wird in Abschn. 5.3.2 angegeben. In Abschn. 5.3.3 wird ein Algorithmus vorgestellt, mit dem in beliebig vorgegebenen Stufen Register effizient eingefügt werden können.

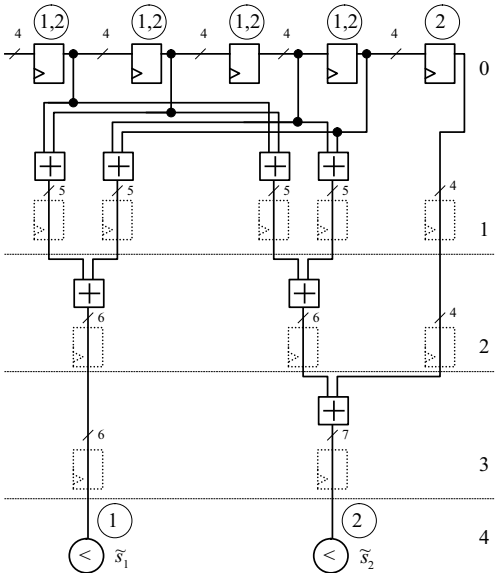
### 5.3.1 Beispiel

In den Abb. 5.2(a) - 5.2(c) ist jeweils ein eindimensionales SRA bestehend aus fünf Registern zu sehen. Das für das Template-Matching zu berechnende Distanzmaß für das erste Template besteht aus der Summe der Pixel der ersten vier Register, und das für das zweite Template aus den Pixeln aller Register, d.h. die Eingänge des ersten Addiererbaums sind mit den ersten vier Register zu verbinden, die des zweiten Addiererbaums mit allen Registern.

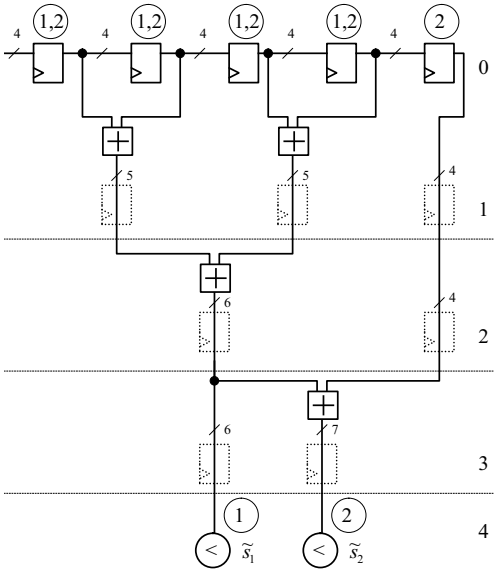
Die Registerknoten, die optional nach jeder Addiererstufe eingefügt werden können, sind in allen nachfolgenden Abbildungen gestrichelt gezeichnet. Durch horizontale gestrichelte Linien wird die Stufenzugehörigkeit bzw. Tiefe der einzelnen Addierer- bzw. Registerknoten, die in den Abbildungen rechts angegeben ist, getrennt. Außerdem sind die Ausgänge der jeweiligen Addiererbäume $A_j$ in der letzten Stufe gekennzeichnet und mit den nachfolgenden Schwellwertmodulen verbunden.

In Abb. 5.2(a) wird der Aufbau von zwei getrennten binären Addiererbäumen $A_1$ und $A_2$ gezeigt. Wie bereits erwähnt, besitzt jeder Addiererknoten $\tilde{a}_l$ genau zwei Vorgängerknoten und einen nachfolgenden Addierknoten. Insgesamt bestehen die beiden binären Addiererbäume aus sieben Addiererknoten, die sich in ihrer Bitbreite $b(\tilde{a}_l)$ unterscheiden. Der Ressourcenbedarf für die binären Addiererbäume beträgt für die in Abschn. 1.3.1 beschriebenen FPGAs von Xilinx $4 * 4 + 2 * 5 + 1 * 6 = 32$ LUTs. Die Tiefe des Wurzelnkotens des ersten Addierbaums ist zwei, die des zweiten Addiererbaums drei. Soll in jeder Stufe ein Registerknoten eingefügt werden, so sind hierfür sieben Registerknoten notwendig.

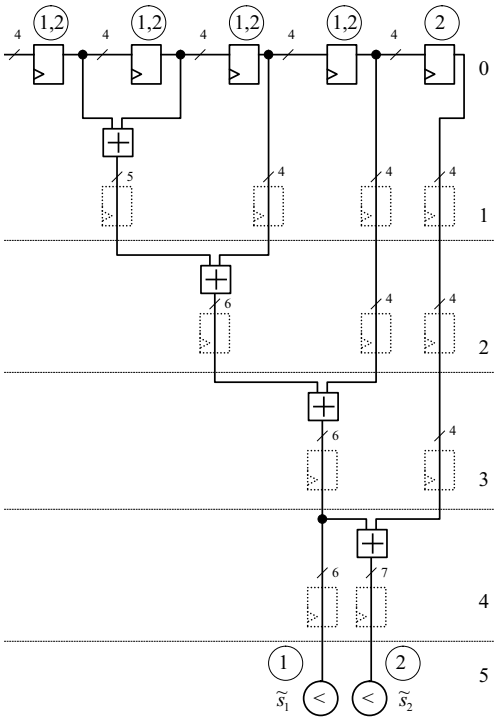
Eine optimale Generierung eines Addierergraphen $A$, der als Teilgraphen die beiden Addiererbäume $A_1$ und $A_2$ enthält, ist in Abb. 5.2(b) dargestellt. Hier werden gemeinsame Partialsummen zwischen den beiden Addiererbäumen von nur jeweils einem Addiererknoten berechnet. Insgesamt werden vier Addiererknoten benötigt, was der erforderlichen Mindestanzahl an Addiererknoten für den zweiten Addiererbaum entspricht. Der




**(a)** Getrennte binäre Addiererbäume.




**(b)** Optimierter Addierergraph.




**(c)** Tiefer Addierergraph.

**Abbildung 5.2:** Unterschiedliche Vorgehensweisen beim Aufbau von Addiererbäumen.

Ressourcenbedarf beträgt $2 * 4 + 1 * 5 + 1 * 6 = 19$ LUTs und die Tiefen der beiden Addiererbäume sind zwei bzw. drei. Die Tiefen der jeweiligen Addiererbäume sind identisch mit denen der binären Addiererbäumen. Soll auf jeden Addiererknoten eine Stufe von Registern folgen, so sind insgesamt vier Register einzufügen. Insgesamt ist der Ressourcenbedarf beim optimierten Addierergraphen deutlich geringer als bei den beiden getrennten binären Addiererbäumen aus Abb. 5.2(a).

In Abb. 5.2(c) ist der Aufbau eines Addierergraphen $A$ zu sehen, der bezüglich der Anzahl der Addiererknoten ebenfalls optimal ist, jedoch nicht bezüglich des Ressourcenbedarfs. Wegen der größeren Tiefe des Addierergraphen und der mit der Tiefe zunehmenden Breite der Addierer führt dies zu einem leicht erhöhten Ressourcenverbrauch von $1 * 4 + 2 * 5 + 1 * 6 = 20$ LUTs. Für die beiden letzten Addiererknoten der dritten bzw. vierten Stufe ist eine Breite von sechs bzw. sieben Bit ausreichend, siehe Gl. 5.1. Der erste Addierbaum hat eine Tiefe von drei, der zweite eine von vier, jeweils eins höher als die der binären bzw. optimierten Addiererbäume. Das Einfügen zusätzlicher Registerstufen macht sich bezüglich des Ressourcenbedarfs ebenfalls negativ bemerkbar. Maximal müssen acht Register eingefügt werden, d.h. ein Register mehr als bei den beiden binären Addiererbäumen und vier Register mehr als bei den optimierten Addiererbäumen.

Zum einen zeigt sich, dass durch den Aufbau eines Addierergraphen $A$, bei dem einzelne Addiererknoten von mehreren Addiererbäumen $A_j$ benutzt werden, der Ressourcenbedarf auf dem FPGA reduziert werden kann. Zum anderen wird deutlich, dass flache Addiererbäume tiefen Bäumen vorzuziehen sind. Hieraus wird im nächsten Abschnitt die sog. *Tiefenminimierungs*-Regel abgeleitet, die bei den in den folgenden Abschnitten beschriebenen Algorithmen zum Aufbau von optimierten Addiererbäumen Verwendung findet und für Verzögerungsknoten erweitert wird.

Der Algorithmus zum Aufbau von optimierten Addiererbäumen lässt sich folgendermaßen aufteilen

- Optimierter Aufbau von gemeinsamen Addiererbäumen $A_j$, Abschn. 5.3.2.
- Optionales Einfügen von Registerstufen, Abschn. 5.3.3.

### 5.3.2 Algorithmus

Im Folgenden wird beschrieben, wie der Addierergraph $A$ bzw. die Addiererbäume $A_j$, $j = 0, \ldots, N-1$ sukzessiv aus den Addiererknoten $\tilde{a}_l$ aufgebaut werden. Es wird auf die Notation für die Register und die Addiererbäume aus Abschn. 4.3.1 und 4.3.2 verwiesen.

**Grundlegende Idee** Der Grundgedanke des iterativen Algorithmus besteht darin, in jedem Iterationsschritt zu bestimmen, mit welchen Registern des SRAs bzw. mit welchen der inzwischen vom Algorithmus generierten Addiererknoten der als nächstens einzufügende Addiererknoten verbunden wird. Dabei werden diejenigen Addierer- bzw. Registerknoten ausgewählt, zwischen denen die meisten gemeinsamen Partialsummen auszuführen sind und somit der Ressourcenbedarf an Addiererknoten zumindest lokal gesehen minimal wird. Es handelt sich demnach um einen Algorithmus mit einem lokalen Entscheidungskriterium. Die Register- und Addiererknoten werden in gemeinsamen Datenstrukturen gehandhabt.

**Voraussetzung an die Templates** Voraussetzung für den im Folgenden beschriebenen Algorithmus ist, dass alle im Template $T_j$ enthaltenen Templateelemente $t_{i,j}$ unterschiedlich sind und somit jeder Addiererbaum $A_j$ mit höchstens *einem* Register $\tilde{r}_l$ des SRAs verbunden wird.

**Variablen** Die Variablen des Algorithmus sind die Registerknoten $\tilde{r}_l$, $l = 0, \ldots, |\tilde{R}| - 1$ der SRAs, mit denen die Eingänge $a_{i,j}$ der Addiererbäume $A_j$ zu verbinden sind.

Die Register $\tilde{R}^k$ der einzelnen Richtungsbereiche $k$ werden nicht gesondert betrachtet, weil alle Registerknoten gleichberechtigt sind. Der Aufbau des Addierergraphen $A$ bzw. der Addiererbäume $A_j$ wird daher auf der Menge aller Register $\tilde{R}$ gelöst.

Der Algorithmus zum Aufbau von optimierten Addiererbäumen gliedert sich in die beiden Teile

- Initialisierung der Datenstrukturen
- Änderungen in den Datenstrukturen und iterativer Aufbau der Addiererbäume.

In jedem Iterationsschritt wird einerseits genau ein Addiererknoten in den Addierergraphen $A$ eingefügt, der gemeinsam von einem oder mehreren Addiererbäumen benutzt werden kann. Andererseits werden die Matrizen und Vektoren der Datenstrukturen des Algorithmus um genau eine Zeile bzw. Spalte erweitert.

### Initialisierung der Datenstrukturen

Zunächst werden die zur Durchführung des Algorithmus erforderlichen Datenstrukturen eingeführt und deren Initialisierung beschrieben.

***Connection*-Matrix $C$** Zur Darstellung der Verbindungen zwischen den Eingängen der $N$ Addiererbäumen $A_j$ und $|\tilde{R}|$ Registerknoten $\tilde{r}_l$ der SRAs wird die *Connection*-Matrix $C$ ($|\tilde{R}| \times N$) eingeführt. In dieser wird die gesamte Information über die Verbindungen der Register mit den Addiererbäumen $A_j$ eingetragen, die für die weiteren Schritte des Algorithmus benötigt wird. Die Komponenten der *Connection*-Matrix $C$ werden zunächst mit Null initialisiert, d.h. $c_{l,j} = 0, \forall l, j$ gesetzt. Gibt es eine Verbindung zwischen Addiererbaum $A_j$ und Register $\tilde{r}_l$, d.h. $(\tilde{r}_l, a_{i,j}) \in E_j$, bzw. ist Register $\tilde{r}_l$ Variable von Addiererbaum $A_j$, so wird das Element $c_{l,j} = 1$ gesetzt, mit $j = 0, \ldots, N-1$, $i = 0, \ldots, |A_j|-1$. Die Anzahl der Einträge der *Connection*-Matrix $C$ für Spalte $j$, die nach deren Initialisierung gleich eins sind, entspricht der Summe der Templateelemente $|T_j|$ von Template $T_j$ und die Anzahl aller Einträge von $C$ entspricht der Summe aller Templateelemente $\sum |T_j|$.

Beim Aufbau des Addierergraphen $A$ bzw. der Addiererbäume $\tilde{A}_j$ wird die *Connection*-Matrix $C$ in jedem Iterationsschritt um genau eine Zeile erweitert. Die Zugehörigkeit des in jedem Iterationsschritt neu generierten Addiererknotens $\tilde{a}_L$ zu den Addiererbäumen $A_j$ wird in die neue Zeile $\tilde{L}$ der *Connection*-Matrix $C$ eingetragen. Die Änderungen von $C$, die in jedem Iterationsschritt durchzuführen sind, werden im weiteren Verlauf dieses Abschnitts beschrieben, nachdem die weiteren Datenstrukturen eingeführt wurden. Die Struktur der *Connection*-Matrix ist in Abb. 5.3 dargestellt. Der obere Teil von $C$ beschreibt die Verbindungsstruktur der Registerknoten $\tilde{r}_l$ der SRAs mit den Addiererbäumen $A_j$ und ist nach der Initialisierung vorhanden. Im unteren Teil von $C$ wird die Zugehörigkeit der Addiererknoten zu den Addiererbäumen, die im weiteren Verlauf des Algorithmus hinzugefügt werden, beschrieben.

Als nächstes wird ein Maß für die Partialsummen, die für mehrere Templates gemeinsam zu berechnen sind, definiert.

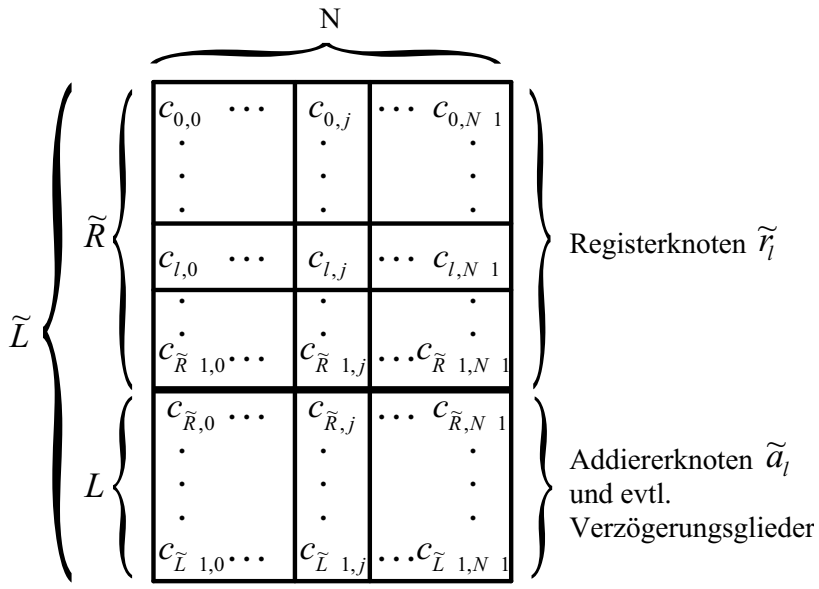



**Abbildung 5.3:** Struktur der *Connection*-Matrix $C$. Im oberern Teil von $C$ sind die Verbindungen der $\tilde{R}$ Registerknoten nach deren Initialisierung enthalten, im unterern Teil die Addiererknoten, die beim iterativen Aufbau des Addierergraphen eingefügt werden.

***Partialsummen*-Matrix $P$** Zur Darstellung der Anzahl der gemeinsamen Partialsummen, die für die verschiedenen Templates zwischen jeweils zwei Registerknoten $\tilde{r}_l$ und $\tilde{r}_m$ auszuführen sind, wird die *Partialsummen*-Matrix $P$ ($|\tilde{R}| \times |\tilde{R}|$) eingeführt, die aus der *Connection*-Matrix $C$ erzeugt wird. Sie wird generiert durch Vergleich der Komponenten jeder Zeile $l$ der *Connection*-Matrix $C$ mit jeder anderen Zeile $m$ von $C$. Zunächst werden alle Elemente $p_{l,m}$ der *Partialsummen*-Matrix $P$ mit Null initialisiert. Das Element $p_{l,m}$ wird inkrementiert, falls für beide Einträge von $C$ gilt: $(c_{l,j} = 1) \wedge (c_{m,j} = 1)$, für $j = 0, \ldots, N-1$ und $l \neq m$. Im Element $p_{l,m}$ ist dann die Anzahl der Partialsummen, die gemeinsam für Register $l$ und Register $m$ der $N$ Addiererbäume auszuführen sind, enthalten. Es gilt $p_{l,m} \leq N$. Die Elemente von $P$ können auch als Korrelation zwischen jeweils zwei Zeilen der *Connection*-Matrix $C$ betrachtet werden. Die Korrelation $p_{l,m}$ zwischen Zeile $l$ und $m$ von $C$ ist gegeben durch $p_{l,m} = \sum_{j=0}^{N-1} c_{l,j} * c_{m,j}$.

Die *Partialsummen*-Matrix $P$ ($\tilde{R} \times \tilde{R}$) ist eine symmetrische Matrix, d.h. $p_{l,m} = p_{m,l}$, weil das Ergebnis des Vergleichs von Zeile $l$ mit $m$ unabhängig von deren Reihenfolge ist. Dies liegt an der Kommutativität des $\wedge$ Operators bzw. der Multiplikation bei der Korrelation. Von den beiden Elementen wird nur das Element links unterhalb der Diagonalen berechnet. Die Diagonalelemente $p_{l,l}$ werden gleich Null gesetzt. Somit hat die *Partialsummen*-Matrix $P$ die Gestalt einer unteren Dreiecksmatrix mit insgesamt $\frac{1}{2}|\tilde{R}| * (|\tilde{R}| - 1)$ Komponenten.

Beim Aufbau des Addierergraphen $A$ wird neben der *Connection*- auch die *Partialsummen*-Matrix in jedem Iterationsschritt erweitert. Aus der erweiterten Matrix $C$ wird dann die erweiterte *Partialsummen*-Matrix $P$ berechnet, die sich in jedem Iterationsschritt um ei-

ne Zeile und Spalte vergrößert. Die Struktur der *Partialsummen*-Matrix ist in Abb. 5.4 dargestellt. Oben links sind die Elemente von $P$ für die Registerknoten nach deren Initialisierung gegeben. Unten sind die Erweiterungen von $P$ für die Addiererknoten zu sehen, die in den einzelnen Iterationsschritten eingefügt werden.

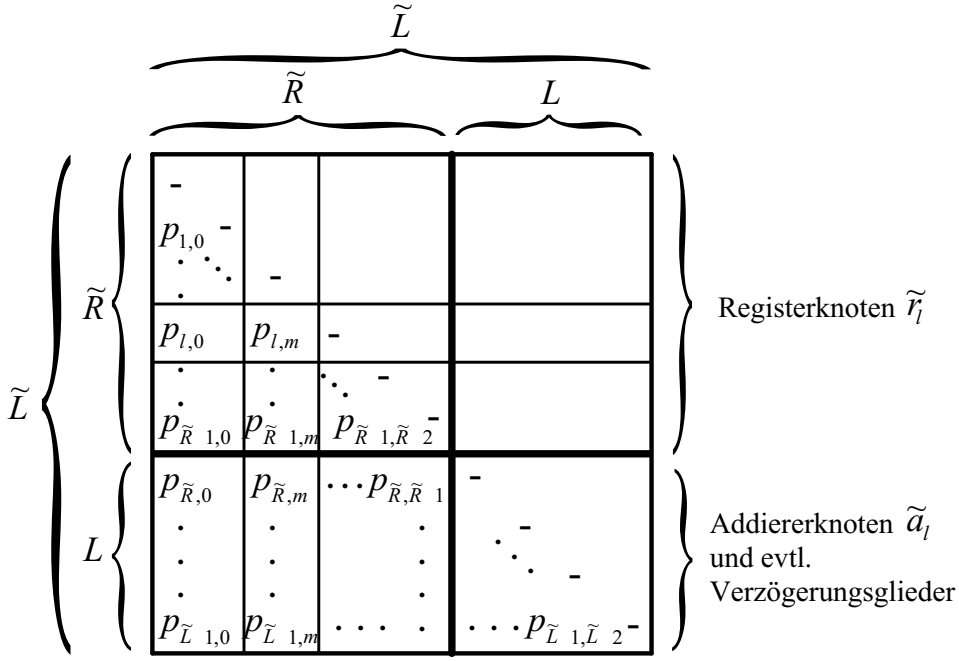



**Abbildung 5.4:** Struktur der *Partialsummen*-Matrix $P$.

**Weitere Datenstrukturen** Für den iterativen Aufbau der Addiererbäume werden neben der *Connection*- und *Partialsummen*-Matrix die folgenden Datenstrukturen definiert.

- Der **Iterationszähler** $i$ wird zunächst auf $i = 0$ gesetzt und in jedem Iterationsschritt inkrementiert, d.h. $i = i + 1$. Die Anzahl der eingefügten Addierer wird mit $L$ bezeichnet, wobei in diesem Abschnitt gilt $L = i$. Mit $\tilde{L} = |\tilde{R}| - 1 + i$ wird die Anzahl der Register- und Addiererknoten bezeichnet. Die Matrizen $C$ und $P$ werden nach deren Initialisierung als nullte Iteration $C^{(0)}$ bzw. $P^{(0)}$ betrachtet.

- Außerdem wird die **Indexliste** $I = \{0, \ldots, \tilde{L}\}$ eingeführt. Sie umfasst alle Register- und Addiererknoten, die in den nachfolgenden Iterationsschritten weiter zu verbinden sind.

- Die **Summe** $s_l$ der Zeile $l$ der *Connection*-Matrix $C$ sei mit $s_l = \sum_{j=0}^{N-1} c_{l,j}$ gegeben. Die Elemente $s_l, l = 0, \ldots, \tilde{L}$ sind im Summenvektor $s$ zusammengefasst. Die Summe $s_j = \sum_{l=0}^{|\tilde{R}|-1} c_{l,j}$ der Spalten von $C$ ist im nullten Iterationsschritt gleich $|T_j|$.

- Jedem Registerknoten $\tilde{r}_l$ bzw. Addiererknoten $\tilde{a}_l$ wird dessen **Tiefe** $t_l$ im Addierergraphen zugeordnet. Den Registerknoten $\tilde{r}_l$, $l = 0, \ldots, |\tilde{R}| - 1$ wird, wie bereits in

Abschn. 5.2.2 erwähnt, eine Tiefe $t_l = 0$ zugewiesen. Die Elemente $t_l, l = 0, \ldots, \tilde{L}$ sind im Tiefenvektor $t$ zusammengefasst.

### Iterativer Aufbau der Addiererbäume

Der Algorithmus zum iterativen Aufbau der optimierten Addiererbäume gliedert sich in die drei Teilschritte

- Gewinn-Analyse mit *Tiefenminimierungs*-Regel
- Änderungen in den Datenstrukturen
- Aufbau des Addierergraphen.

Die Idee des iterativen Verfahrens besteht darin, in jedem Iterationsschritt auszuwählen, mit welchen der Register- bzw. bereits generierten Addiererknoten der neue Addiererknoten $\tilde{a}_L$ zu verbinden ist. In jedem Iterationsschritt wird die im Folgenden beschriebene Gewinn-Analyse durchgeführt, die mit einem lokalen Entscheidungskriterium arbeitet. Anschließend werden die gemeinsamen Datenstrukturen geändert und die Struktur des bisherigen Addierergraphen mit dem *neuen* Addiererknoten erweitert. Die einzelnen Teilschritte für den iterativen Aufbau des Addierergraphen sind zusammenfassend in Abb. 5.5 illustriert.

**Gewinn-Analyse und *Tiefenminimierungs*-Regel** Zunächst wird die max. Anzahl an Überdeckungen zwischen jeweils zwei Register- bzw. Addiererknoten bestimmt, welche identisch mit dem maximalen Eintrag $p_{l,m}$ der *Partialsummen*-Matrix $P$ ist. Gibt es nur eine maximale Komponente $p_{\bar{l},\bar{m}}$, so wird ein neuer Addiererknoten $\tilde{a}_L$ generiert und dessen Eingänge mit den Register- bzw. Addiererknoten $\tilde{r}_{\bar{l}}$ und $\tilde{r}_{\bar{m}}$ verbunden.

Existieren mehrere max. Komponenten $p_{l,m}$, so wird als weiteres Kriterium die Summe der Tiefen der Register- bzw. Addiererknoten $T_{l,m} = t_l + t_m$ herangezogen. Dabei wird die Komponente $p_{\bar{l},\bar{m}}$ mit minimalem $T_{\bar{l},\bar{m}}$ gewählt, weil dieses Vorgehen zu einem flacheren Addiererbaum führt. Diese Regel, die *Tiefenminimierungs*-Regel genannt wird, wurde anhand des günstigeren FPGA-Ressourcenbedarfs für das in Abschn. 5.3.1 angegebene Beispiel abgeleitet.

Gibt es weiterhin mehrere maximale Komponenten $p_{l,m}$ mit identischem $T_{l,m}$, so wird das Element $p_{\bar{l},\bar{m}}$ mit kleinstem Index $\bar{l}$ bzw. Index $\bar{m}$ gewählt.

**Änderungen in den Datenstrukturen** Nachdem der neue Addiererknoten $\tilde{a}_L$ ausgewählt wurde, der mit den Register- bzw. Addiererknoten $(\bar{l}, \bar{m})$ zu verbinden ist, sind die *Connection*- und *Partialsummen*-Matrizen und die weiteren Datenstrukturen neu zu berechnen. Die folgenden Änderungen sind in jedem Iterationsschritt durchzuführen.

- **Iterationszähler:** Der Iterationszähler $i$ wird inkrementiert, d.h. $i = i + 1$ und somit $L = L + 1$ und $\tilde{L} = \tilde{L} + 1$ gesetzt. Außerdem wird die Indexliste $I$ um $\{\tilde{L}\}$ erweitert, d.h. $I^{(i+1)} = I^{(i)} \cup \{\tilde{L}\}$.

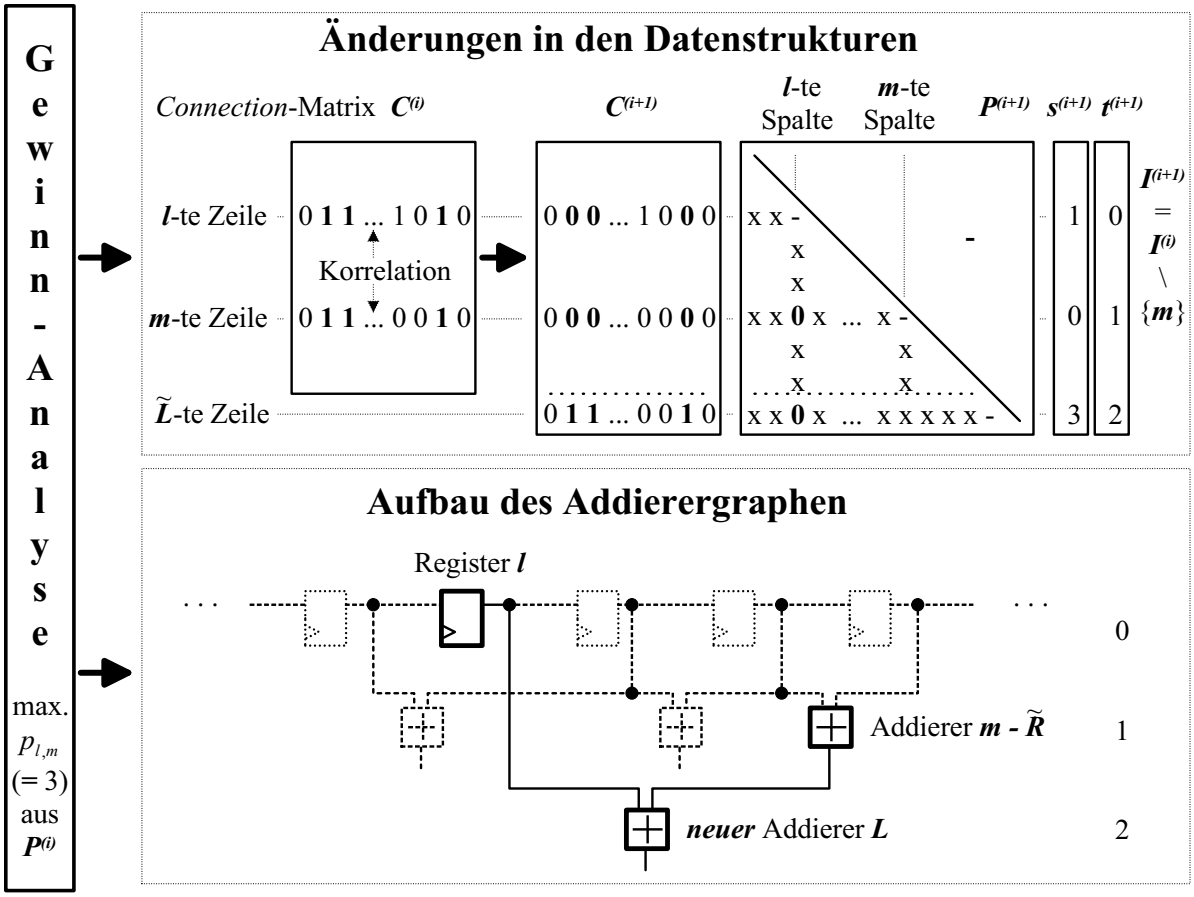



**Abbildung 5.5:** Ein Iterationsschritt für den Algorithmus zum Aufbau von optimierten Addiererbäumen. Aus der Gewinn-Analyse mit *Tiefenminimierungs*-Regel (links) wird der neue Addierer bestimmt. Für diesen werden die Datenstrukturen $C, P, s, t, I$ geändert (oben). Der neue Addiererknoten wird in den bisherigen Addierergraphen eingefügt (unten).

- **Änderungen der** *Connection*-**Matrix** $C^{(i+1)}$**:** Die Eigenschaften des neuen Addiererknoten $\tilde{a}_L$, der auf die Register- bzw. Addiererknoten $\bar{l}$ und $\bar{m}$ zugreift, werden in die neue *Connection*-Matrix $C^{(i+1)}$ eingetragen. Hierzu werden die alten Zeilen $\bar{l}$, $\bar{m}$ der *Connection*-Matrix $C^{(i)}$ und die neue Zeile $\tilde{L}$ folgendermaßen geändert. Die Elemente ungleich 0, die übereinstimmen, d.h. $(c^{(i)}_{\bar{l},j} = 1) \wedge (c^{(i)}_{\bar{m},j} = 1)$ für $j = 0, \ldots, N-1$, werden aus den Zeilen $\bar{l}$ und $\bar{m}$ der *Connection*-Matrix $C^{(i)}$ gestrichen, d.h. $c^{(i+1)}_{l,j} = 0$, $c^{(i+1)}_{m,j} = 0$ gesetzt und in die neue Zeile $\tilde{L}$ übertragen, d.h. $c^{(i+1)}_{\tilde{L},j} = 1$ gesetzt. Die Elemente der anderen Zeilen ändern sich nicht.

- **Änderungen der** *Partialsummen* **Matrix** $P^{(i+1)}$**:** Die Komponenten $p_{\bar{l},\bar{m}}$ der Zeilen und Spalten $\bar{l}$ und $\bar{m}$ werden wiederum durch Vergleich der entsprechenden Zeilen neu berechnet. Für die Zeile $\bar{l}$ sind dies die Elemente $p_{\bar{l},m}$, $m = 0, \ldots, \bar{l}-1$ und für die Spalte $\bar{m}$ die Elemente $p_{l,\bar{m}}$, $l = \bar{m}+1, \ldots, \tilde{L}$. Zusätzlich werden die Elemente $p_{\tilde{L},m}$ der neuen Zeile $\tilde{L}$, $m = 0, \ldots, \tilde{L}-1$ berechnet, siehe auch Abb. 5.5 (oben).

- **Summe:** Es wird die Summe der Elemente der alten Zeilen $\bar{l}$, $\bar{m}$ und der neuen Zeile $\tilde{L}$ der *Connection*-Matrix mit $s_{\bar{l}} = \sum_{j=0}^{N-1} c_{\bar{l},j}$ berechnet.
- **Tiefe:** Des Weiteren erfolgt die Berechnung der Tiefe $t_{\tilde{L}}$ für den neuen Addiererknoten $\tilde{a}_L$. Dieser berechnet sich aus dem Maximum der Tiefen seiner Eingangsknoten zu $t_{\tilde{L}} = \max(t_{\bar{l}}, t_{\bar{m}}) + 1$.
- **Indexliste:** Falls für die Summen der Zeilen $s_{\bar{l}} = 0$ oder $s_{\bar{m}} = 0$ gilt, werden diese aus der Indexliste $I$ gestrichen, d.h. $I^{(i+1)} = I^{(i)} \setminus \{\bar{l}\}$ bzw. $I^{(i+1)} = I^{(i)} \setminus \{\bar{m}\}$. Somit werden die entsprechenden Zeilen $\bar{l}$ bzw. $\bar{m}$ der *Connection*-Matrix und die Zeilen und Spalten der *Partialsummen*-Matrix, deren Einträge gleich Null sind, im weiteren Verlauf des Algorithmus nicht mehr berücksichtigt.
- **Aufbau des Addierergraphen:** Parallel zu den bisher beschriebenen Änderungen in den Datenstrukturen, wird die Verbindungsstruktur des Addierergraphen in jedem Iterationsschritt $i$ aufgebaut. Dem Addierergraphen wird der Addiererknoten $\tilde{a}_L$ hinzugefügt. Dieser wird ebenfalls dem Addiererbaum $A_j$ zugewiesen, falls der neue Eintrag $c_{\tilde{L},j}$ von $C$ gleich eins ist. Den beiden Eingängen des Addiererknotens $\tilde{a}_L$ werden die beiden vorausgehenden Register- bzw. Addiererknoten zugewiesen. Die Tiefe $t(\tilde{a}_L)$ des Addiererknotens ist gleich $t_{\tilde{L}}$. Die Anzahl $v(\tilde{a}_L)$ der vorausgehenden Addiererknoten von $\tilde{a}_L$ wird aus den beiden eingehenden Addiererknoten bestimmt und dessen Bitbreite $b(\tilde{a}_L)$ nach Gl. 5.1 berechnet.
- **Überprüfen der Endbedingung:** Als Abbruchkriterium des Algorithmus wird die Summe $S$ aller Zeilen $s_l$ der *Connection*-Matrix, d.h. $S = \sum_{l=0}^{\tilde{L}} s_l$ ermittelt. Falls $S$ gleich der Anzahl der Addiererbäume, d.h. $S = N$, wird der Algorithmus beendet.

Nach dem Ausführen des iterativen Teils des Algorithmus ist der Aufbau des Addierergraphen $A$, in welchem die $N$ Addiererbäume $A_j$ enthalten sind, abgeschlossen.

**Zuweisung der Wurzel- und Schwellwertknoten** In einem letzten Schritt ist jedem Addiererbaum $A_j$ seinen Wurzelknoten zuzuweisen. Dies geschieht anhand der $N$ Koeffizienten $c_{l,j} \neq 0$ der *Connection*-Matrix. In jeder Spalte $j$ von $C$ existiert genau ein Eintrag $c_{l,j} \neq 0$. Demnach wird der Ausgang von Addiererknoten $\tilde{a}_{l-\tilde{R}}$ dem Schwellwertknoten $\tilde{s}_j$ zugewiesen, siehe auch folgendes Beispiel und Abb. 5.6(e).

### Allgemeine Bemerkungen

Die Anzahl der Iterationen ist kleiner gleich der Anzahl der Addierer der binären Addiererbäume $\sum_{j=0}^{N-1}(|T_j| - 1)$, weil sich in jedem Iterationsschritt die Anzahl der Komponenten von $C$ mit $c_{l,j} = 1$ mindestens um eins verringert. Wird der eingefügte Addiererknoten von $n$ Addiererbäumen gemeinsam benutzt, so verringert sich die Anzahl der Komponenten von $C$ mit $c_{l,j} = 1$ um $n$.

Die maximale Anzahl an Nachfolgern eines Addiererknotens ist begrenzt durch $N$, der Anzahl der Addiererbäume $A_j$ bzw. Templates $T_j$. Die $N$ Addiererbäume sind die Teilgraphen des Addierergraphen $A$. Von den Blättern zu den Wurzelknoten der Addiererbäume gibt es immer genau einen eindeutigen Pfad.

Die *Connection*-Matrix ist die grundlegende Datenstruktur des Algorithmus. Ebenfalls von grundlegender Bedeutung ist der Tiefenvektor $t$, weil dessen Elemente für die *Tiefenminimierungs*-Regel von Relevanz sind. Die *Partialsummen*-Matrix $P$, der Summenvektor $s$ sowie die Indexmenge $I$ wurden letztendlich aus Effizienzgründen für den Algorithmus eingeführt. Wird auf $P$ verzichtet, so müssten für die Gewinn-Analyse in jedem Iterationsschritt alle Kombinationen der Zeilen von $C$ miteinander verglichen werden, wie dies bereits bei der Initialisierung von $P$ durchgeführt wurde. Wird ebenfalls auf den Summenvektor $s$ und die Indexmenge $I$ verzichtet, so würden die Berechnungen immer für alle Zeilen von $C$ ausgeführt, auch wenn diese nicht mehr relevant sind weil deren Einträge gleich 0 sind.

Bei der Durchführung der Gewinn-Analyse, bei der ein neuer Addiererknoten $\tilde{a}_l$ ausgewählt wird, ist eine weitere Verbesserung möglich, falls die Nachbarschaftsrelationen zwischen den alten und dem neuen einzufügenden Addiererknoten berücksichtigt werden. Die Nachbarschaftsbeziehungen können Auswirkungen auf das FPGA-*Routing* haben. Diese wurden jedoch im Rahmen dieser Arbeit nicht untersucht.

Die Integration der optimierten Addiererbäume anstelle der binären Addiererbäume (aus Abschn. 4.3.2) ist in das Design für das parallele Template-Matching durchzuführen.

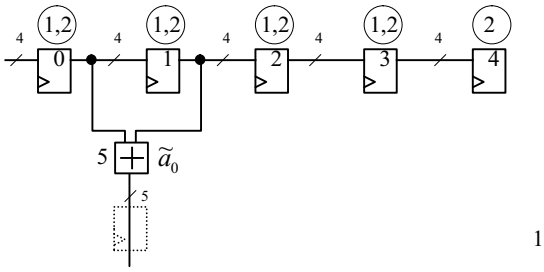
### Beispiel

Der Aufbau eines Addierergraphen $A$ wird im Folgenden am Beispiel der beiden Addiererbäume $A_j$ aus Abb. 5.2 erklärt. In jedem Iterationsschritt wird die Berechnung der *Connection*-Matrix $C$, der *Partialsummen*-Matrix $P$, der Indexmenge $I$, des Summenvektors $s$ und des Tiefenvektors $t$ angegeben. Die den Register- und Addiererknoten entsprechenden Zeilen in den Matrizen bzw. Vektoren sind für alle nachfolgenden Beispiele durch eine horizontale Linie getrennt. Außerdem werden ausschließlich diejenigen Zeilen in den Matrizen bzw. Vektoren dargestellt, die noch in der Indexliste enthalten sind. Die Matrizen und Vektoren sind demnach im Folgenden in einer reduzierten Form dargestellt. Die Indexliste ist ebenfalls in Vektorform dargestellt und erleichtert die einfache Zuordnung der Zeilen der reduzierten Matrix zu den Zeilen der nicht reduzierten Matrix. Die einzelnen Iterationsschritte sind außerdem in den Abb. 5.6(a) bis Abb. 5.6(e) graphisch dargestellt.

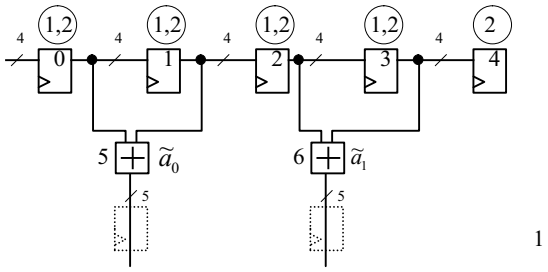
Zunächst werden die Indexzähler mit $i = 0$ bzw. $L = 0$ und $\tilde{L} = 4$ und die weiteren Datenstrukturen initialisiert

$$\mathbf{C^{(0)}} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \mathbf{P^{(0)}} = \begin{bmatrix} - & - & - & - & - \\ \bar{2} & - & - & - & - \\ 2 & 2 & - & - & - \\ 2 & 2 & 2 & - & - \\ 1 & 1 & 1 & 1 & - \end{bmatrix}, [\mathbf{I}, \mathbf{s}, \mathbf{t}]^{(\mathbf{0})} = \left[\begin{array}{c|c|c} 0 & 2 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & 0 \\ 3 & 2 & 0 \\ 4 & 1 & 0 \end{array}\right], \tilde{a}_0 = (0, 1). \tag{5.2}$$

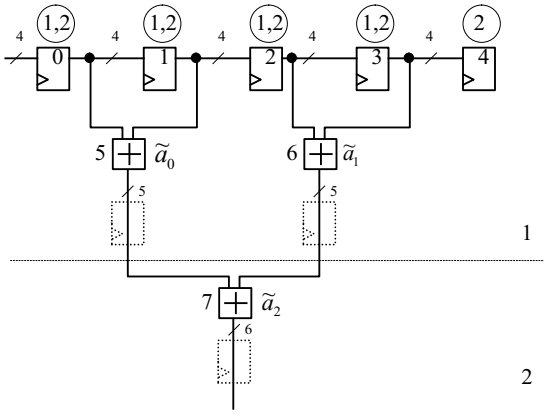
Als erstes wird die Gewinn-Analyse durchgeführt. Als max. Einträge stehen die Koeffizienten $p_{1,0} = p_{2,0} = p_{3,0} = p_{2,1} = p_{3,1} = p_{3,2} = 2$ zur Auswahl. Die Summe der Tiefen $T_{l,m}$ dieser sechs Koeffizienten sind gleich Null. Daher wird das erste Element $p_{1,0}$ aus der Liste ausgewählt, welches mit einem Überstrich gekennzeichnet ist. Der neue Addiererknoten $\tilde{a}_0$ ist demnach mit den Registern $\tilde{r}_0$ und $\tilde{r}_1$ zu verbinden.



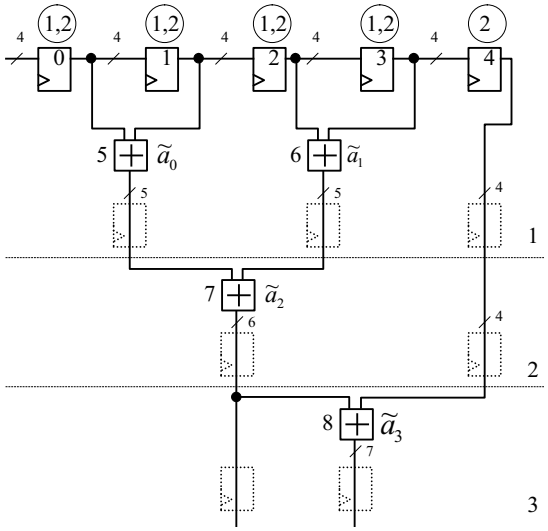

**(a)** Iterationsschritt 1.



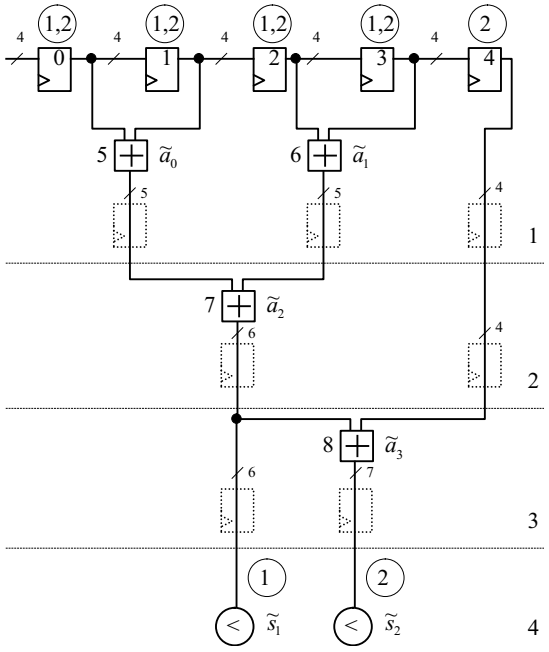

**(b)** Iterationsschritt 2.




**(c)** Iterationsschritt 3.




**(d)** Iterationsschritt 4.




**(e)** Zuweisung der Schwellwertknoten.

**Abbildung 5.6:** Iterativer Aufbau der Addiererbäume für das Beispiel aus Abb. 5.2.

Zunächst wird der Indexzähler $i$ erhöht, d.h. $i = 1$, $L = 1$ und $\tilde{L} = 5$ gesetzt und die Indexmenge $I^{(1)} = I^{(0)} \cup \{5\}$ erweitert. Anschließend werden die erweiterten Matrizen $C^{(1)}$ und $P^{(1)}$ und die Vektoren $s$ und $t$ berechnet und die Indexmenge wird nochmals angepasst mit $I^{(1)} = I^{(1)} \setminus \{0, 1\}$. Zuletzt wird das Abbruchkriterium des Algorithmus überprüft.

$$\mathbf{C^{(1)}} = \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ \hline 1 & 1 \end{array}\right], \mathbf{P^{(1)}} = \left[\begin{array}{cccccc} - & - & - & - & - & - \\ 0 & - & - & - & - & - \\ 0 & 0 & - & - & - & - \\ 0 & 0 & \bar{2} & - & - & - \\ 0 & 0 & 1 & 1 & - & - \\ \hline 0 & 0 & 2 & 2 & 1 & - \end{array}\right], [\mathbf{I}, \mathbf{s}, \mathbf{t}]^{(1)} = \left[\begin{array}{c|c|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 2 & 0 \\ 3 & 2 & 0 \\ 4 & 1 & 0 \\ \hline 5 & 2 & 1 \end{array}\right], \tilde{a}_1 = (2, 3). \tag{5.3}$$

Zu Beginn des nächsten Iterationsschritts erfolgt die Gewinn-Analyse. Die max. Elemente der *Partialsummen*-Matrix der fünften Zeile besitzen jeweils eine Tiefe von eins, das der dritten Zeile eine Tiefe von null. Als Ergebnis der Gewinn-Analyse wird $p_{3,2}$ gewählt, d.h. der neue Addiererknoten $\tilde{a}_1$ wird mit den Registern $\tilde{r}_2$ und $\tilde{r}_3$ verbunden. Nach der Durchführung der Gewinn-Analyse werden die Indexzähler erhöht, d.h. $i = 2$, $L = 2$ und $\tilde{L} = 6$. Die Elemente der alten Zeilen der *Connection*-Matrix $C^{(1)}$ werden in deren neue Zeile sechs übertragen und hieraus die *Partialsummen*-Matrix $P^{(2)}$ generiert. Des Weiteren werden die Zeilen zwei, drei und sechs von $s$ und von $t$ angepasst. Aus der Indexliste werden die Zeilen zwei und drei gestrichen, da deren Summen der *Connection*-Matrix gleich Null und daher für weitere Betrachtungen nicht mehr relevant sind. Die Indexliste wird entsprechend angepasst, d.h. $I^{(2)} = I^{(1)} \setminus \{2, 3\} \cup \{6\}$.

$$\mathbf{C^{(2)}} = \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ \hline 1 & 1 \\ 1 & 1 \end{array}\right], \mathbf{P^{(2)}} = \left[\begin{array}{ccccc} - & - & - & - & - \\ 0 & - & - & - & - \\ 0 & 0 & - & - & - \\ \hline 0 & 0 & 1 & - & - \\ 0 & 0 & 1 & \bar{2} & - \end{array}\right], [\mathbf{I}, \mathbf{s}, \mathbf{t}]^{(2)} = \left[\begin{array}{c|c|c} 2 & 0 & 0 \\ 3 & 0 & 0 \\ 4 & 1 & 0 \\ \hline 5 & 2 & 1 \\ 6 & 2 & 1 \end{array}\right], \tilde{a}_2 = (5, 6). \tag{5.4}$$

Das max. Element der *Partialsummen*-Matrix ist $p_{5,6} = 2$. Die Eingänge von Addiererknoten $\tilde{a}_2$ werden mit den Ausgängen der Addiererknoten $\tilde{a}_0$ und $\tilde{a}_1$ verbunden. Die Iterationszähler werden inkrementiert zu $i = 3$, $L = 3$ und $\tilde{L} = 7$, die weiteren Datenstrukturen berechnet und die Indexmenge $I^{(3)} = I^{(2)} \setminus \{5, 6\} \cup \{7\}$ angepasst.

$$\mathbf{C^{(3)}} = \left[\begin{array}{cc} 0 & 1 \\ \hline 0 & 0 \\ 0 & 0 \\ 1 & 1 \end{array}\right], \mathbf{P^{(3)}} = \left[\begin{array}{cccc} - & - & - & - \\ 0 & - & - & - \\ \hline 0 & 0 & - & - \\ \bar{1} & 0 & 0 & - \end{array}\right], [\mathbf{I}, \mathbf{s}, \mathbf{t}]^{(3)} = \left[\begin{array}{c|c|c} 4 & 1 & 0 \\ \hline 5 & 0 & 1 \\ 6 & 0 & 1 \\ 7 & 2 & 2 \end{array}\right], \tilde{a}_3 = (4, 7). \tag{5.5}$$

Das Ergebnis der Gewinn-Analyse ist das max. Element $p_{4,7} = 1$. Es wird der Addiererknoten $\tilde{a}_3$ eingefügt und mit Addiererknoten $\tilde{a}_2$ und Registerknoten $\tilde{r}_4$ verbunden. Der

Iterationszähler wird inkrementiert, d.h. $i = 4$, $L = 4$ und $\tilde{L} = 8$ gesetzt und die Indexmenge $I^{(4)} = I^{(3)} \setminus \{4\} \cup \{8\}$ angepasst.

$$\mathbf{C^{(4)}} = \left[\begin{array}{cc} 0 & 0 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right], \mathbf{P^{(4)}} = \left[\begin{array}{ccc} - & - & - \\ \hline 0 & - & - \\ 0 & 0 & - \end{array}\right], [\mathbf{I}, \mathbf{s}, \mathbf{t}]^{(4)} = \left[\begin{array}{c|c|c} 4 & 0 & 0 \\ \hline 7 & 1 & 2 \\ 8 & 1 & 3 \end{array}\right], END. \tag{5.6}$$

Beim Überprüfen des Abbruchkriteriums, d.h. $S = N$, wird festgestellt, dass dieses erfüllt ist.

Der iterative Teil des Algorithmus wird daher beendet und anhand der *Connection*-Matrix $C^{(4)}$ und der Indexliste $I^{(4)}$ wird jedem Addiererbaum $A_j$ seinen Wurzelknoten zugewiesen. Die Elemente $c_{7,0}$ und $c_{8,1}$ der *Connection*-Matrix sind gleich eins. Demnach wird der Ausgang von Addiererknoten $\tilde{a}_2$ dem ersten Addiererbaum und der Addiererknoten $\tilde{a}_3$ dem zweiten Addiererbaum zugewiesen. Die Tiefe der Addiererbäume kann direkt aus dem Tiefenvektor $t$ abgelesen werden und ist für den ersten Addiererbaum gleich zwei und für den zweiten Addiererbaum gleich drei.

Anschließend werden die Ausgänge der Addiererbäume mit den Schwellwertknoten verbunden. Der Wurzelnkoten $\tilde{a}_2$ von Addiererbaum $A_1$ wird dem Schwellwertknoten $\tilde{s}_1$ und der Wurzelknoten $\tilde{a}_3$ vom zweiten Addiererbaum dem Schwellwertknoten $\tilde{s}_2$ zugewiesen.

### 5.3.3 Einfügen von Registerstufen

Um die Anzahl der Logikstufen in den gemeinsamen Addiererbäumen zu begrenzen, werden Registerstufen zwischen den Addiererknoten eingefügt. Die Stufen, bei denen die Register eingefügt werden, sollen dabei fest vorgegeben werden können. Die $|P|$ Registerstufen seien in der Liste $P = \{p_0, \ldots, p_{|P|-1}\}$ angegeben. Es soll nicht erlaubt sein Register bei der nullten Stufe einzufügen, d.h. es gilt $p_0 \geq 1$. Das Einfügen von Registerstufen bei der nullten Stufe hätte zur Folge, dass direkt nach den Registern $\tilde{r}_l$, $l = 0, \ldots, \tilde{r}_{|\tilde{R}|-1}$ der SRAs wiederum Register eingefügt würden, was jedoch zu keiner Reduzierung der Anzahl der Logikstufen in den Addiererbäumen führen würde.

Im Folgenden wird ein Algorithmus beschrieben, der für die vorgegebenen Registerstufen $P$ eine minimale Anzahl an Registerknoten $\tilde{r}_l$ in die gemeinsamen Addiererbäume einfügt. Die Idee des Algorithmus besteht darin, für jeden Register- bzw. Addiererknoten zunächst die Anzahl der einzufügenden Register zwischen seinem Ausgang und den Eingängen von dem bzw. den nachfolgenden Addiererknoten zu bestimmen. Die einzufügenden Registerknoten sollen dann von möglichst vielen nachfolgenden Addiererknoten benutzt werden. Es soll außerdem erlaubt sein, $h$ hintereinander folgende Register durch ein Verzögerungsglied (SRLT) der Länge $h - 1$ und ein Register zu ersetzen.

Für jeden Register- bzw. Addiererknoten $\tilde{a}_l$ seien deren $|V|$ nachfolgende Addiererknoten $\tilde{a}_n$ mit $V = \{\tilde{a}_{n_0}, \ldots \tilde{a}_{n_{|V|-1}}\}, n = n_0, \ldots, n_{|V|-1}$ bezeichnet. Die Tiefe des Register- bzw. Addiererknotens $\tilde{a}_l$ sei mit $t(\tilde{a}_l)$ gegeben, die der nachfolgenden Addiererknoten mit $t(\tilde{a}_{n_0})$ bis $t(\tilde{a}_{n_{|V|-1}})$. Zunächst wird für den Register- bzw. Addiererknoten $\tilde{a}_l$ und jeden seiner $|V|$ nachfolgenden Addiererknoten $\tilde{a}_n$ die Anzahl der einzufügenden Registerknoten festgestellt, welche mit $K_{l,n}$ bezeichnet wird. Hierzu werden die $|V|$ Intervalle

$I_{l,n_0} = [t(\tilde{a}_l),\, t(\tilde{a}_{n_0}) - 1]$ bis $I_{l,n_{|V|-1}} = [t(\tilde{a}_l),\, t(\tilde{a}_{n_{|V|-1}}) - 1]$ bestimmt. Die Anzahl $K_{l,n}$ der einzufügenden Register zwischen dem Register- bzw. Addiererknoten $\tilde{a}_l$ und den nachfolgenden Addiererknoten $\tilde{a}_n, n = n_0, \ldots, n_{|V|-1}$ ist gleich der Anzahl der Elemente der Liste $P$, die in dem Intervall $I_{l,n}$ liegen.

Insgesamt soll erreicht werden, dass die Anzahl der einzufügenden Registerknoten so gering wie möglich ist. Die einzufügenden Register werden daher nicht für jeden nachfolgenden Addiererknoten getrennt betrachtet. Das Ziel besteht darin, dass die Registerknoten von möglichst vielen nachfolgenden Addiereknoten benutzt werden.

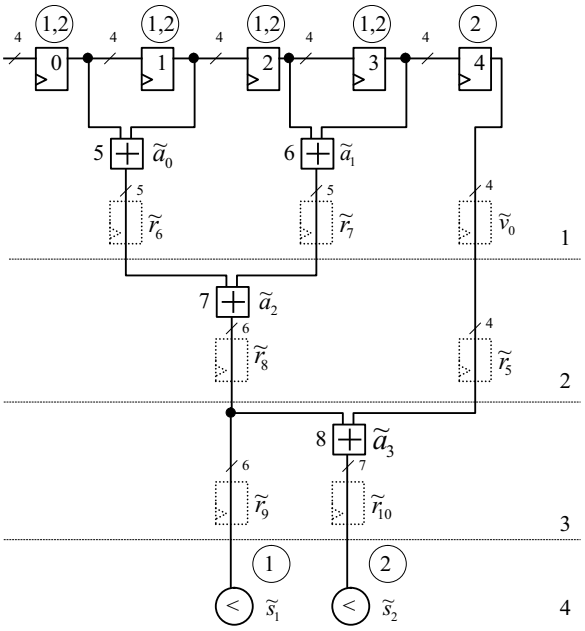
Zunächst werden die $|V|$ nachfolgenden Addiererknoten $\tilde{a}_n$ nach der Anzahl $K_{l,n}$ der einzufügenden Register geordnet, beginnend mit der niedrigsten Anzahl an einzufügenden Registern. Anschließend wird mit dem Einfügen der Registerknoten begonnen. Für den bzw. die Nachfolgerknoten mit der niedrigsten Anzahl $\min(K_{l,n})$ werden nun die $\min(K_{l,n})$ Register genau einmal eingefügt und von den Nachfolgerknoten gemeinsam benutzt. Die Nachfolgerknoten mit minimalem $K_{l,n}$ werden aus der Liste $P$ gestrichen. Den verbleibenden nachfolgenden Addiererknoten wird $\min(K_{l,n})$ von den von den verbleibenden Längen $K_{l,n}$ abgezogen, d.h $K'_{l,n} = K_{l,n} - \min(K_{l,n})$ . Für die nun verbleibenden Addiererknoten mit den niedrigsten Längen $\min(K_{l,n})$ werden wiederum $\min(K_{l,n})$ gemeinsam benutzte Registerknoten eingefügt und anschließend aus der Liste $V$ gestrichen. Das Verfahren wird fortgesetzt, bis keine Addiererknoten mehr der Liste $V$ angehören, d.h. $V = \emptyset$.

Ist die Anzahl $K_{l,n}$ der einzufügenden Register gleich eins, so wird genau ein Register eingefügt. Sind mehrere Register einzufügen, so kann der Ressourcenbedarf an Registern reduziert werden, falls für $K_{l,n} > 1$ nicht $K_{l,n}$ Shift-Register, sondern ein Verzögerungsknoten der Länge $K_{l,n}-1$ und nachfolgend ein Register eingefügt werden, wie in Abb. 5.7 rechts oben zu sehen ist. Der Nachteil bei dieser Vorgehensweise ist jedoch, dass sich die Anzahl der Logikstufen erhöht, falls der Verzögerungsknoten nicht auf einen Register-, sondern auf einen Addiererknoten folgt.

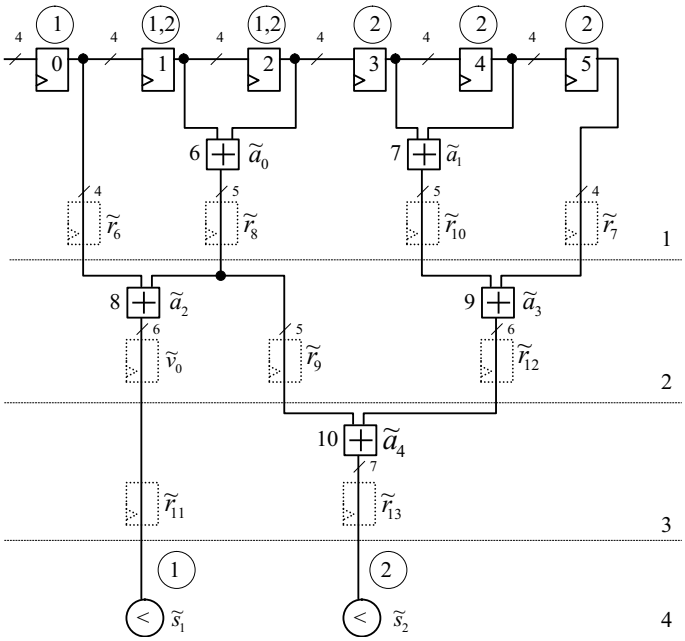
**Beispiel**

Der Algorithmus für das Einfügen der Registerstufen $P$ wird im Folgenden anhand der beiden in Abb. 5.7 angegebenen Addiererbäumen veranschaulicht.

In jeder Stufe des in Abb. 5.7(a) angegebenen Beispiels, soll ein Register eingefügt werden, d.h. $P = \{1, 2, 3\}$. Zunächst werden die fünf Registerknoten des SRAs und deren nachfolgenden Addiererknoten untersucht. Der Registerknoten $\tilde{r}_0$ besitzt als Nachfolger den Addiererknoten $\tilde{a}_0$. Es wird das Intervall $I_{0,0} = [0, 0]$ und $K_{0,0} = 0$ berechnet. Hieraus folgt, dass nach $\tilde{r}_0$ keine Registerknoten eingefügt werden. Dies gilt ebenfalls für die drei weiteren Registerknoten des SRAs bis einschließlich Registerknoten $\tilde{r}_3$. Der Ausgang von Registerknoten $\tilde{r}_4$ wird mit dem Eingang von Addiererknoten $\tilde{a}_3$ verbunden. Es wird das Intervall $I_{3,8} = [0, 2]$ und $K_{3,8} = 2$ berechnet. Allerdings werden nicht $K_{3,8} = 2$ Register eingefügt, sondern das Verzögerungsglied $\tilde{v}_0$ der Länge eins und nachfolgend das Register $\tilde{r}_5$, siehe Abb. 5.7(a). Nun erfolgt die Bestimmung der nachfolgenden Registerknoten für die Addiererknoten. Für den Addiererknoten $\tilde{a}_0$ mit einer Tiefe $t(\tilde{a}_0) = 1$ und dessen Nachfolger $\tilde{a}_2$ mit einer Tiefe $t(\tilde{a}_0) = 2$ ergibt sich das Intervall $I_{0,2} = [1, 1]$. Die Anzahl der einzufügenden Register ist $K_{0,2} = 1$ und es wird Registerknoten $\tilde{r}_6$ eingefügt. Nach




(a)




(b)

**Abbildung 5.7:** Einfügen von Registerstufen in den Addierergraphen.

dem Addiererknoten $\tilde{a}_1$ mit $I = [1, 1]$ wird das Register $\tilde{r}_7$ eingefügt. Der Addiereknoten $\tilde{a}_2$ besitzt als Nachfolgerknoten zum einen den Schwellwertknoten $\tilde{s}_1$ und den Addiereknoten $\tilde{a}_3$. Sowohl für das Schwellwertmodul als auch für den Addiererknoten $\tilde{a}_3$ wird ein gemeisames Register eingefügt.

In Abb. 5.7(b) wird nochmals die Besonderheit des Auftretens mehrerer Nachfolgerkno-

ten am Beispiel des Addiererknotens $\tilde{a}_0$ beschrieben. Hier soll ebenfalls in jeder Stufe ein Registerknoten eingefügt werden. Auf Addiererknoten $\tilde{a}_0$ folgen die beiden Addiererknoten $\tilde{a}_2$ und $\tilde{a}_4$. Es werden die beiden Intervalle $I_{0,2} = [1, 1]$ und $I_{0,4} = [1, 2]$ und die Anzahl der einzufügenden Register $K_{0,2} = 1$ und $K_{0,4} = 2$ berechnet. Zunächst wird für die beiden nachfolgenden Addiereknoten das Register $\tilde{r}_8$ eingefügt. Anschließend wird die Anzahl der einzufügenden Register für $K_{0,2}$ und $K_{0,4}$ um jeweils eins reduziert. Der Addiererknoten $\tilde{a}_2$ mit $K_{0,2} = 0$ wird aus der Liste $V$ gestrichen und für den Addiererknoten $\tilde{a}_4$ mit $K_{0,4} = 1$ wird das Register $\tilde{r}_9$ eingefügt. Anschließend wird ebenfalls Addiererknoten $\tilde{a}_4$ mit $K_{0,4} = 0$ aus der Liste $V$ gestrichen und es erfolgt die Bestimmung der einzufügenden Registerknoten nach Addiererknoten $\tilde{a}_1$.

## 5.4 Optimierte Addiererbäume mit Verschieben der Templates

Eine weitere Strategie zur Reduzierung des FPGA-Ressourcenbedarfs besteht darin, die Templates $T_j$ als Ganzes zu verschieben, mit dem Ziel, möglichst viele Überlappungen zwischen den einzelnen Templateelementen $t_{i,j}$ der unterschiedlichen Templates zu erzielen. Dieses Vorgehen hat zum einen Auswirkungen auf die Anzahl der Register $\tilde{r}_l$ der SRAs und zum anderen auf die Anzahl der Addiererknoten $\tilde{a}_l$ in den Addiererbäumen. Es wird erlaubt, die Templates $T_j$ sowohl in x- als auch y-Richtung um $(v_{j,x}, v_{j,y})$ Pixel zu verschieben. Die Anzahl der Freiheitsgrade der Verschiebungen ist $2N$. Die Verschiebungen werden zu den einzelnen Templateelementen $t_{i,j} = (t_{i,j,x}, t_{i,j,y})$, $j = 0, \ldots, N-1, i = 0, \ldots, |T_j| - 1$ hinzuaddiert

$$t'_{i,j,x} = t_{i,j,x} + v_{j,x} \tag{5.7}$$

$$t'_{i,j,y} = t_{i,j,y} + v_{j,y}. \tag{5.8}$$

Mit $t'_{i,j,x}$ und $t'_{i,j,y}$ sind die Templateelemente nach deren Verschieben bezeichnet.

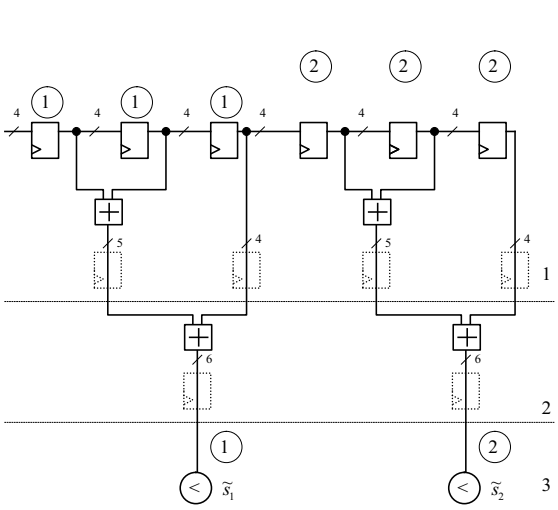
Das Verschieben der Templates $T_j$ kann durch zwei Methoden rückgängig gemacht werden. Zum einen kann nach dem Wurzelknoten des zum verschobenen Template gehörenden Addiererbaums bzw. nach dessen Schwellwertknoten ein Verzögerungsknoten eingefügt werden. Die Templates können dann jedoch nur in negativer x- und y-Richtung verschoben werden, weil die Verzögerungen der Daten in den Verzögerungsknoten (SRLTs) positiv sind. Zum anderen kann das Verschieben der Templates bei der Berechnung der Ergebnisse des Template-Matchings berücksichtigt werden.

In Abschn. 5.6 werden Strategien für das Verschieben der in den Abb. 2.8 bis 2.10 dargestellten kreisförmigen und dreickigen Templates angegeben. Die hierfür benötigten Ressourcen für die SRAs und den Addierergraphen sind in Abschn. 5.8 angegeben.

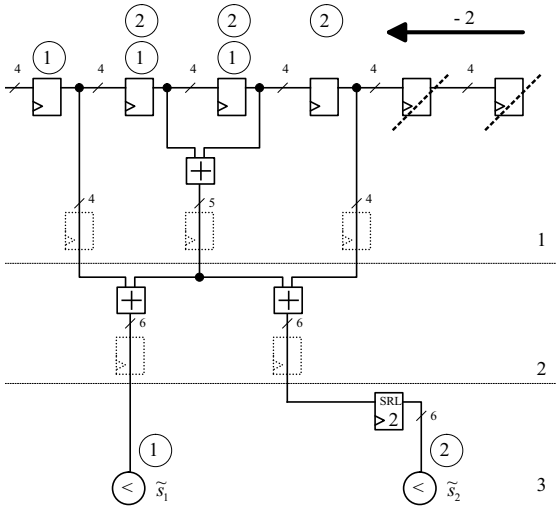
### 5.4.1 Beispiel

Am Beispiel eines eindimensionalen SRAs bestehend aus sechs Registern und zwei Addiererbäumen, siehe Abb. 5.8, von denen der erste Addiererbaum $A_1$ mit den ersten drei Registern und der zweite Addiererbaum $A_2$ mit den letzten drei Registern verbunden wird,

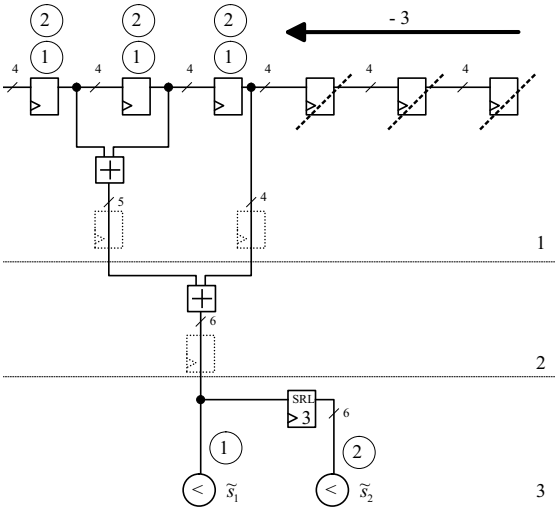
werden die Auswirkungen der Verschiebungen des zweiten Templates um $v_{2,x} = -2$ und $v_{2,x} = -3$ auf den FPGA-Ressourcenbedarf untersucht. Die jeweils um einen festen Wert verschobenen Templateelemete können wie bisher beim Aufbau des Addierergraphen durch Addiererknoten zusammengefasst werden.



**(a)** Binäre Addiererbäume ohne Verschieben der Templates.




**(b)** Addiererbäume mit nicht optimalem Verschieben der Templates.




**(c)** Addiererbäume mit optimalem Verschieben der Templates.

**Abbildung 5.8:** Addiererbäume mit Verschieben der Templates als Ganzes.

**Ressourcenbedarf** Für das in Abb. 5.8(a) angegebene Ausgangsproblem werden für das SRA sechs Register (24 FFs) und für die Addiererbäume $2 * 4 + 2 * 5 = 18$ LUTs auf dem FPGA benötigt. Beim Verschieben des zweiten Templates um zwei Pixel in negativer x-Richtung, siehe Abb. 5.8(b), sind für das SRA vier anstatt sechs Register und für die Addiererbäume $1 * 4 + 2 * 5 = 14$ LUTs notwendig. Falls das Verschieben des zweiten Templates durch einen Verzögerungsknoten der Länge zwei kompensiert wird, so werden

hierfür zusätzlich sechs LUTs benötigt, insgesamt 20 LUTs. Wie in Abb. 5.8(c) dargestellt, ist das zweite Template um drei Pixel nach links verschoben. Für das SRA werden nur noch drei Register (12 FFs) und für die Addiererbäume $1 * 4 + 1 * 5 = 9$ LUTs benötigt. Zusätzlich erhöht sich der Ressourcenbedarf um sechs LUTs, falls das Verschieben durch ein Verzögerungsglied der Länge drei ausgeglichen wird. Nimmt man den Ressourcenbedarf von FFs und LUTs zusammen, bewirkt das Verschieben des zweiten Templates als Ganzes eine deutliche Reduzierung des Ressourcenbedarfs.

### 5.4.2 Algorithmus

Der in Abschn. 5.3.2 angegebene iterative Teil des Algorithmus zum Aufbau der Addiererbäume ist bei den Templates, die als Ganzes verschoben wurden, wie bisher durchzuführen.

#### Ausgleich durch Verzögerungsglieder

Wie bereits erwähnt, können die als Ganzes verschobenen Templates durch Verzögerungsglieder ausgeglichen werden. Bevor die Wurzelknoten $\tilde{a}_l$ den Addiererbäumen $A_j$ zugewiesen und die Schwellwertknoten $\tilde{s}_j$ nachgeschaltet werden, sind für jeden Addiererbaum $A_j$ ein Verzögerungsglied mit der Länge

$$v_{j,x} + v_{j,y} * W \tag{5.9}$$

einzufügen. Die Verschiebungen in y-Richtung, die von der Breite des Bildes $W$ abhängig sind, sind teuer und müssen durch sehr lange Verzögerungsglieder ausgeglichen werden. Der Tiefe des Verzögerungsglieds wird die Tiefe des nachfolgenden Schwellwertknotens zugewiesen. Wird ein Verzögerungsglied erst nach dem Schwellwertknoten eingefügt, so reduziert sich der Ressourcenbedarf hierfür auf dem FPGA, weil nur 1-Bit Daten zu verzögern sind. Im Rahmen dieser Arbeit werden die Verschiebungen nicht durch Verzögerungsglieder ausgeglichen.

#### Änderungen bei der Berechnung der Ergebnisse des Template-Matchings

Das Verschieben der Templates als Ganzes kann bei der Berechnung der Ergebnisse des Template-Matchings berücksichtigt werden, indem der Verschiebungsvektor $(v_{j,x}, v_{j,y})$ von allen Ergebnispunkten $(G_h, G_w)$, siehe Gl. 4.23 abgezogen wird

$$G'_w = G_w - v_{j,x} \tag{5.10}$$
$$G'_h = G_h - v_{j,y}. \tag{5.11}$$

Mit $(G'_w, G'_h)$ ist der neue Ergebnispunkt bezeichnet.

Die Berechnung wird auf dem Host-Rechner durchgeführt und die hierfür benötigte Rechenzeit ist für wenige Ergebnispunkte vernachlässigbar. Diese Methode, die keine zusätzlichen Ressourcen auf dem FPGA benötigt, wird in dieser Arbeit eingesetzt.

## 5.5 Optimierte Addierergraphen mit Verschieben einzelner Templateelemente

Als weitere Optimierungsstrategie wird nun erlaubt, dass nicht nur die Templates $T_j$ als Ganzes, sondern auch deren einzelne Templateelemente $t_{i,j}$ gegeneinander verschoben werden dürfen. Das Ziel besteht darin, möglichst viele Überdeckungen zwischen den Templateelementen der jeweiligen Templates zu erreichen, die einen ressourcengünstigen Aufbau des Addierergraphen ermöglichen. Diese einzelnen Verschiebungen müssen beim Aufbau des Addierergraphen $A$ bzw. der Addiererbäume $A_j$ durch Verzögerungsglieder ausgeglichen werden. Für die einzelnen Templateelemente $t_{i,j}$ wird ein Verschieben ausschließlich in negativer x-Richtung um $v_{i,j,x}$ Pixel zugelassen, weil die Verschiebungen in y-Richtung nur durch sehr lange Verzögerungsglieder auszugleichen und daher sehr teuer wären, siehe Gl. 5.9. Die Verschiebungen $v_{i,j,x}$ sollen dabei begrenzt sein durch den linken Rand des SRAs $S^k$ des zugehörigen Richtungsbereichs $k$, siehe z.B. Abb. 4.4.

Die Verschiebungen $v_{i,j,x}$ werden in dem Verschiebevektor $v_x$ zusammengefasst, der von der Dimension $\sum_j |T_j|$ ist. Die Anzahl der Freiheitsgrade der Verschiebungen $v_{i,j,x}$ der einzelnen Templateelemente erhöht sich i.A. gegenüber dem Verschieben der $N$ Templates als Ganzes von $2N$ auf $\sum_{j=0}^{N-1} |T_j|$.

Die Struktur der Templatedaten eines Templateelements $t_{i,j}$ von Template $T_j$ wird durch die Komponente $t_{i,j,v}$ erweitert zu $t_{i,j} = (t_{i,j,x}, t_{i,j,y}, t_{i,j,v})$ mit $t_{i,j,v} = v_{i,j,x}$. Die Verschiebungen sind ebenfalls auf die ersten Komponente $t_{i,j,x} + v_{i,j,x}$ zu addieren. Zunächst soll jedoch nicht erlaubt werden, mehrere Templateelemente $t_{i,j}$ eines Templates $T_j$ derart übereinander zu schieben, so dass diese nach dem Verschieben die gleiche Position besitzen. Für eine gültige Verschiebung muss gelten $t_{i,j,x} - v_{i,j,x} \neq t_{i',j,x} - v_{i',j,x}, i \neq i', 0 \leq i, i' < |T_j| - 1$. Der Sonderfall, der es erlaubt, mehrere Templateelemente eines Templates übereinanderzuschieben, wird in Abschn. 5.5.3 behandelt.

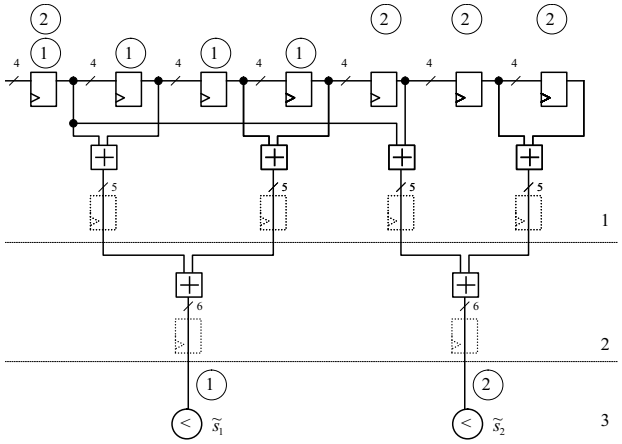
In diesem Abschnitt wird ausgeführt, welche Auswirkungen sich aus den Verschiebungen der Templateelemente für den Aufbau des Addierergraphen $A$ bzw. der Addiererbäume $A_j$ ergeben und inwiefern der in Abschn. 5.3.2 beschriebene Algorithmus zum Aufbau von optimierten Addiererbäumen erweitert werden muss. Wesentliche Erweiterungen ergeben sich bei der Gewinn-Analyse, weil nun zunächst zu entscheiden ist, ob ein Addierer- oder Verzögerungsknoten einzufügen ist. Die Länge der Verzögerung des einzufügenden Verzögerungsknotens ist bei der Gewinn-Analyse ein weiterer Parameter. Für die Gewinn-Analyse wird eine Basis-Methode und eine Methode mit *virtuellen* Verzögerungsgliedern beschrieben.

### 5.5.1 Beispiele

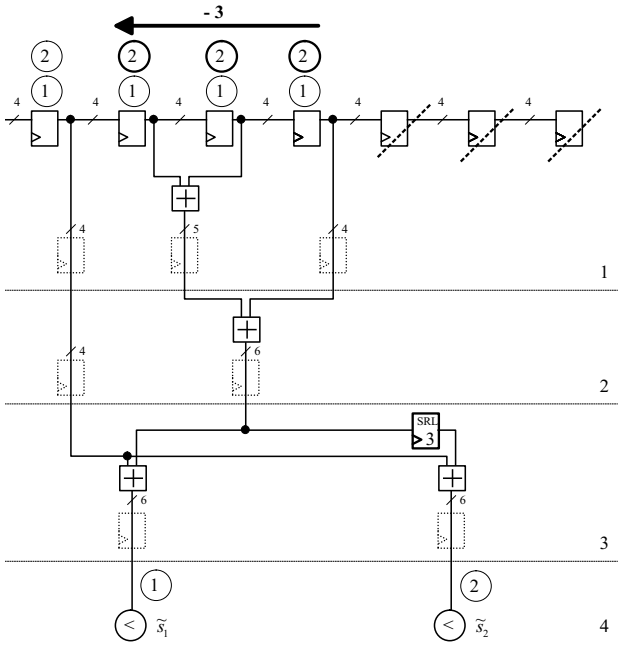
#### Beispiel für Basis-Methode der Gewinn-Analyse

Zunächst wird anhand eines einfachen Beispiels, welches in Abb. 5.9 dargestellt ist, der Aufbau der Addiererbäume ohne bzw. mit einzeln verschobenen Templateelementen er-

klärt und eine Abschätzung für den FPGA-Ressourcenbedarf für die SRAs und die Addiererbäume gegeben.



**(a)** Addiererbäume ohne Verschieben einzelner Templateelemente.




**(b)** Addiererbäume mit Verschieben einzelner Templateelemente.

**Abbildung 5.9:** Aufbau der Addiererbäume mit Verschieben einzelner Templateelemente.

Die Eingänge des ersten Addiererbaums sind mit den ersten vier Registern, die des zweiten Addiererbaums mit dem ersten und mit den drei letzten Registern zu verbinden, siehe Abb.5.9(a). Das Ziel beim Aufbau der beiden Addiererbäume besteht darin, sowohl für die unverschobenen Templateelemente des ersten Templates und die verschobenen Templateelemente des zweiten Templates gemeinsame Addiererressourcen zu benutzen. Die Register eins bis drei der verschobenen und nicht verschobenen Templateelemente können zunächst gemeinsam zusammengefasst werden, wie in Abb.5.9(b) illustriert ist.

Anschließend werden für den zweiten Addiererbaum die Verschiebungen der entsprechenden Templateelemente durch ein Verzögerungsglied der Länge drei ausgeglichen. Der weitere Aufbau der beiden Addiererbäume erfolgt getrennt.

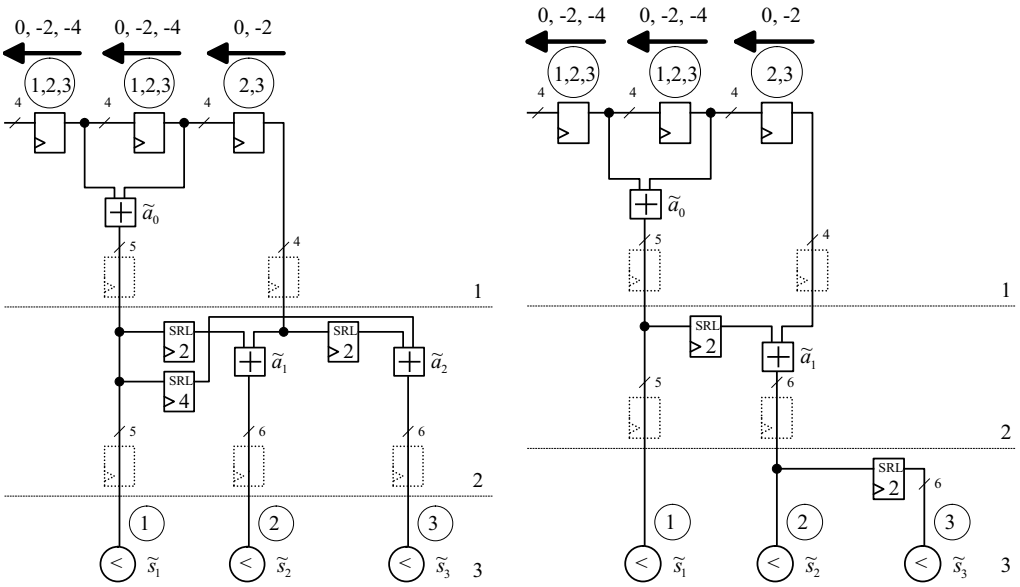
**Ressourcenverbrauch** In Abb. 5.9(a) ist der Aufbau des Addierergraphen zu sehen, bei dem die einzelnen Templateelemente nicht verschoben sind. Für das SRA werden sieben Register (28 FFs) und für den Addiererbaum sechs Addierer mit einem Ressourcenbedarf von $4 * 4 + 2 * 5 = 26$ LUTs auf dem FPGA benötigt.

Werden die drei Templateelemente des zweiten Addiererbaums um drei Pixel nach links verschoben, siehe Abb. 5.9(b), so werden für die SRAs nur noch vier Register (16 FFs) und vier Addiererknoten mit einem Ressourcenbedarf von $1 * 4 + 3 * 5 = 19$ LUTs sowie einem Verzögerungsglied mit sechs LUTs benötigt. Der Ressourcenbedarf an LUTs verringert sich leicht auf $25$ LUTs. Nimmt man den Ressourcenbedarf an Registern und LUTs zusammen, so ergibt sich für die Addiererbäume mit den verschobenen Templateelementen eine deutliche Ressourcenersparnis für das FPGA.

### Beispiel für Methode mit *virtuellen* Verzögerungsknoten der Gewinn-Analyse

In Abb. 5.10 sind die Verschiebungen der Templateelemente von drei Templates sowie der Aufbau der entsprechenden drei Addiererbäume dargestellt. Sowohl für die Basis-Methode als auch der Methode mit *virtuellen* Verzögerungsknoten werden die ersten beiden Registerknoten durch einen gemeinsamen Addiererknoten zusammengefasst, weil die Verschiebungen der Templateelemente der drei Templates identisch sind. Wird nach der Basis-Methode vorgegangen, werden die Verschiebungen anschließend durch getrennte Verzögerungsglieder ausgeglichen und dessen Ausgänge durch weitere Addiererknoten zusammengefasst, siehe Abb. 5.10(a). Wird an dieser Stelle nach der Methode mit *virtuellen* Verzögerungsgliedern vorgegangen, so werden zunächst nach allen sich im Addierergraphen befindlichen Addierer- oder Registerknoten *virtuelle* Verzögerungsglieder mit Längen von eins bis zu dem Betrag der max. Verschiebung eingefügt. Für diese wird anschließend der Gewinn ermittelt, der im nächsten Schritt durch das Einfügen eines Addiererknotens erzielt werden kann. Am günstigsten ist es, die Verschiebungen des zweiten und dritten Addiererbaums nach dem Addiererknoten $\tilde{a}_0$ teilweise durch ein Verzögerungsglied der Länge zwei auszugleichen. Anschließend kann der Ausgang des Verzögerungsknoten mit den verschobenen Templateelementen von Template zwei und drei, die im rechten Register gespeichert sind, gewinnbringend zusammengesfasst werden, siehe Abb. 5.10(b).

**Ressourcenverbrauch** Für den Addierergraphen, der in Abb. 5.10(a) dargestellt ist und nach der Basis-Methode aufgebaut wurde, sind drei Addierer mit einem Ressourcenbedarf von $1*4+2*5 = 14$ LUTs und drei Verzögerungsglieder mit $1*4+2*5 = 14$ LUTs, insgesamt $28$ LUTs notwendig. Wird der Addierergraph nach der Methode mit *virtuellen* Verzögerungsgliedern aufgebaut, siehe Abb. 5.10(b), werden nur zwei Addiererknoten mit einem Ressourcenbedarf von $1*4+1*5 = 9$ LUTs und zwei Verzögerungsknoten mit $1*5+1*6 = 11$ LUTs, insgesamt 20 LUTs benötigt. Die Methode mit *virtuellen* Verzögerungsgliedern führt zu einer deutlichen Reduzierung des FPGA-Ressourcenbedarfs.




**(a)** Addiererbäume mit Verschieben einzelner Templateelemente mit der Basis-Methode der Gewinn-Analyse.

**(b)** Addiererbäume mit Verschieben einzelner Templateelemente mit der Methode mit *virtuellen* Verzögerungsgliedern der Gewinn-Analyse.

**Abbildung 5.10:** Addiererbäume mit Verschieben einzelner Templateelemente und unterschiedlichen Methoden der Gewinn-Analyse.

## 5.5.2 Algorithmus

In diesem Abschnitt werden die Erweiterungen des in Abschn. 5.3.2 vorgestellten iterativen Algorithmus für Templates mit einzeln verschobenen Templateelementen angegeben.

**Grundlegende Idee** Beim Aufbau der Addiererbäume sind neben den Addiererknoten zusätzlich Verzögerungsglieder in den Addiererbaum einzufügen, mit denen die Verschiebungen der Templateelemente ausgeglichen werden. Die Idee des erweiterten Algorithmus besteht nun darin, in jedem Iterationsschritt entweder einen Addierer- oder einen Verzögerungsknoten einzufügen und diese in gemeinsamen Datenstrukturen zu handhaben. Im Folgenden wird beschrieben, wie die Datenstrukturen gegenüber dem bisherigen Algorithmus erweitert werden, und welche Änderungen sich bei der Initialisierung der Datenstrukturen und in den einzelnen Iterationsschritten ergeben. Außerdem wird eine Basis-Methode und eine mit *virtuellen* Verzögerungsgliedern beschrieben, mit denen die Auswahl des einzufügenden Addierer- oder Verzögerungsknotens erfolgt.

### Initialisierung der Datenstrukturen

***Translation*-Matrix $D$** Als wesentliche neue Datenstruktur wird die *Translation*-Matrix $D$ eingeführt, in der die Verschiebungen der einzelnen Templateelemente eingetragen werden. Sie besitzt dieselbe Struktur wie die *Connection*-Matrix $C$ und ist ebenfalls eine

Matrix der Größe $(|\tilde{R}| \times N)$. Die Elemente $d_{l,j}$ der *Translation*-Matrix $D$ werden zunächst mit Null initialisiert. Die weitere Initialisierung der *Translation*-Matrix $D$ wird gleichzeitig mit der Initialisierung der *Connection*-Matrix $C$ durchgeführt. Bei der *Connection*-Matrix $C$ wird das Element $c_{l,j} = 1$ gesetzt, falls eine Verbindung zwischen Addiererbaum $A_j$ und Register $\tilde{r}_l$ existiert, d.h. $(\tilde{r}_l, a_{i,j}) \in E_j$, für $j = 0, \ldots, N-1$, $i = 0, \ldots, |A_j| - 1$. Existiert eine Verbindung zwischen Register $\tilde{r}_l$ und Addiererbaum $A_j$, so wird das Element $d_{l,j} = t_{i,j,v}$ gesetzt. Für die nicht verschobenen Templateelemente gilt daher $d_{l,j} = 0$ und für die verschobenen $d_{l,j} < 0$.

*Partialsummen*-**Matrix** $P$ Bei der Initialisierung der *Partialsummen*-Matrix $P$ ist folgende Änderung zu berücksichtigen. Das Element $p_{l,m}$ wird genau dann inkrementiert, falls für die beiden Einträge von $C$ *und* für die beiden Einträge von $D$ gilt $(c_{l,j} = 1) \wedge (c_{m,j} = 1) \wedge (d_{l,j} = d_{m,j})$ für $j = 0, \ldots, N-1$ und $l \neq m$. Sind die beiden Einträge $d_{l,j}$ und $d_{m,j}$ unterschiedlich, so können diese erst dann zusammengefasst werden, nachdem die Verschiebungen durch das Einfügen von einem oder zwei Verzögerungsknoten mit den entsprechenden Längen ausgeglichen wurden.

**Iterationszähler und Indexliste** Neben dem Iterationszähler $i$ und dem Zähler $L$ für die Addierer wird nun der Zähler $F$ für die Verzögerungsglieder eingeführt. $\tilde{L}$ umfasst nun neben den Registerknoten der SRAs und den Addiererknoten auch die Verzögerungsknoten des Addierergraphen $A$. Für $\tilde{L}$ gilt wie bisher $\tilde{L} = |\tilde{R}| - 1 + i$. In der Indexliste $I$ werden neben den Register- und Addiererknoten zusätzlich die eingefügten Verzögerungsknoten eingetragen.

**Verzögerungsvektor** $h$ Eine Unterscheidung zwischen den Addiererknoten und den Verzögerungsknoten in den gemeinsamen Datenstrukturen erfolgt durch den Verzögerungsvektor $h$. Dessen Elemente $h_{\tilde{L}}$ werden gleich eins gesetzt, falls der in der $i$-ten Iteration eingefügte Knoten ein Verzögerungsknoten ist.

### Iterativer Aufbau der Addiererbäume

**Gewinn-Analyse** Bei dem in Abschn. 5.3.2 beschriebenen Algorithmus wurde zu Beginn eines jeden Iterationsschritts entschieden, mit welchen Register- bzw. Addiereknoten der einzufügende Addierknoten verbunden wird. Nun ist zunächst auszuwählen, ob ein Addiererknoten oder ein Verzögerungsknoten einzufügen ist.

- **Basis-Methode:** In jedem Iterationsschritt wird zunächst überprüft, ob mindestens ein Eintrag der *Partialsummen*-Matrix $P$ größer 0 ist. Trifft dies zu, so wird dem Addierergraphen ein Addiererknoten hinzugefügt. Sind alle Einträge der *Partialsummen*-Matrix $p_{l,m} = 0$, so wird zunächst nach einem Eintrag in der *Translation*-Matrix $D$ gesucht mit $d_{l,\bar{j}} \neq 0$. Die weiteren in dieser Zeile $l$ vorkommenden Elemente von $D$ werden auf Gleichheit überprüft, $d_{l,j} = d_{l,\bar{j}}, j = 0, \ldots, N-1, j \neq \bar{j}$. Für all diese wird genau ein Verzögerungsglied mit der Länge $-d_{l,j}$ eingefügt. Die entsprechenden, in der Zeile $l$ vorkommenden Elemente $c_{l,j}$ der *Connection*-Matrix $C$ werden

gleich Null gesetzt und in die neue Zeile $\tilde{L}$ übertragen, d.h. $c_{\tilde{L},j} = 1$. Die entsprechenden Elemente $d_{l,j}$ der alten und neuen Zeile der *Translation*-Matix $D$ werden gleich Null gesetzt, weil für diese ein Verzögerungsglied der Länge $-d_{l,\tilde{j}}$ eingefügt wurde. Außerdem wird das Element $h_{\tilde{L}}$ im Verzögerungsvektor $h$ gleich eins gesetzt.

Für das in Abb. 5.9 angegebene Beispiel führt die Basis-Methode zu einem minimalen Ressourcenbedarf. In den Gl. 5.12 bis Gl. 5.22 sind für dieses Beispiel die Initialisierung und Änderungen der Datenstrukturen in den einzelnen Iterationsschritten im Detail angegeben.

- **Methode mit *virtuellen* Verzögerungsgliedern:** Die Idee der Methode mit *virtuellen* Verzögerungsgliedern besteht darin, in jedem Iterationsschritt zunächst die Gewinne zu bestimmen, die sich durch das Einfügen von *virtuellen* Verzögerungsglieder für alle möglichen Variationen an Verzögerungen $vd$ ergeben. Erst nach dem Einfügen des Verzögerungsknotens lassen sich die virtuellen Gewinne im nächsten Iterationsschritt durch das Einfügen eines Addiererknotens in einen realen Gewinn umwandeln.

  Die Bestimmung der virtuellen Gewinne für das Einfügen der Verzögerungsglieder erfolgt für alle Kombinationen zweier Zeilen $l$ und $m$, die in der Indexliste $I$ enthalten sind. Das Verschieben einer Zeile $l$ wird für alle Einträge mit $c_{l,j} = 1$ und $d_{l,j} + vd \leq 0$ mit $vd = 1, \ldots, \max|d_l|$ durchgeführt, wobei $\max|d_l|$ der maximale Betrag der Einträge der *Translations*-Matrix $D$ der Zeile $l$ ist. Der Gewinn für das virtuelle Verzögerungsglied mit einer Verschiebung um $vd$ wird inkrementiert, falls gilt $(c_{l,j} = 1) \wedge (c_{m,j} = 1) \wedge (d_{l,j} + vd = d_{m,j})$ für $j = 0, \ldots, N-1$.

  Ist der max. virtuelle Gewinn, der durch Einfügen eines *virtuellen* Verzögerungsglieds mit einer Verzögerung von $vd$ erzielt wird größer als der max. Eintrag der *Partialsummen*-Matrix $P$, so wird ein *realer* Verzögerungsknoten mit der Verzögerung $vd$ eingefügt, ansonsten ein Addierknoten. Als weiteres Kriterium dafür, welcher Verzögerungsknoten einzufügen ist, wird die in Abschn. 5.3.2 eingeführte *Tiefenminimierungs*-Regel für Verzögerungsknoten erweitert.

  - Ist der virtuelle Gewinn zweier einzufügender Verzögerungsknoten identisch, so ist derjenige Verzögerungsknoten mit der kleineren Tiefe einzufügen. Hiermit wird sichergestellt, dass der Addierergraph möglichst flach gehalten wird.
  - Wird für unterschiedliche Verschiebungen $vd$ der beiden Zeilen $l$ und $m$ derselbe virtuelle Gewinn erzielt, so wird das Verzögerungsglied mit der größten Länge eingefügt. Diese Regel verhindert, dass ein längeres Verzögerungsglied durch mehrere kürzere ersetzt wird, siehe hierzu auch das Beispiel am Ende dieses Abschnitts.

### Änderungen in den Datenstrukturen

- **Änderungen der *Connection*-Matrix $C^{(i+1)}$ und der *Translation*-Matrix $D^{(i+1)}$:** Wird ein neuer Addiererknoten $\tilde{a}_L$ eingefügt, so werden die übereinstimmenden

Elemente $(c_{\bar{l},j}^{(i)} = 1) \wedge (c_{\bar{m},j}^{(i)} = 1) \wedge (d_{\bar{l},j} = d_{\bar{m},j})$ für $j = 0, \ldots, N-1$ aus den Zeilen $\bar{l}$ und $\bar{m}$ der *Connection*-Matrix $C^{(i)}$ und der *Translation*-Matrix $D^{(i)}$ gestrichen, d.h. $c_{\bar{l},j}^{(i+1)} = 0$, $c_{\bar{m},j}^{(i+1)} = 0$ und $d_{\bar{l},j}^{(i+1)} = 0$, $d_{\bar{m},j}^{(i+1)} = 0$ gesetzt und in die neue Zeile $\tilde{L}$ übertragen, d.h. $c_{\tilde{L},j}^{(i+1)} = 1$ und $d_{\tilde{L},j}^{(i+1)} = d_{\bar{l},j}^{(i)} = d_{\bar{m},j}^{(i)}$ gesetzt.

Wird ein Verzögerungsglied der Länge $vd$ nach der Zeile $\bar{l}$ eingefügt, so gilt für die Einträge der neuen Zeile $c_{\tilde{L},j}^{(i+1)} = 1$ und $d_{\tilde{L},j}^{(i+1)} = d_{\bar{l},j}^{(i)} + vd$ und für die alte Zeile $d_{\bar{l},j}^{(i+1)} = d_{\bar{l},j}^{(i)} + vd$, vorausgesetzt $c_{\bar{l},j}^{(i)} = 1$ und $d_{\bar{l},j}^{(i)} + vd \leq 0$.

- **Änderungen der *Partialsummen*-Matrix $P^{(i+1)}$:** Die *Partialsummen*-Matrix $P^{(i+1)}$ ist wie bisher in jedem Iterationsschritt für die alten Zeilen und Spalten $\bar{l}$ und $\bar{m}$ sowie für die neue Zeile $\tilde{L}$ zu berechnen, siehe Abschn. 5.3.2. Das Element $p_{l,m}$ wird genau dann inkrementiert, falls für die beiden Einträge von $C$ *und* von $D$ gilt $(c_{\bar{l},j} = 1) \wedge (c_{\bar{m},j} = 1) \wedge (d_{\bar{l},j} = d_{\bar{m},j})$.

- **Tiefe $t$:** Die Verzögerungsglieder (SRLTs) werden mit den Addiererknoten zu einer Stufe zusammengefasst. Bei der Berechnung der Tiefe $t_{\tilde{L}}$ eines neuen Addierer- bzw. Verzögerungsknoten ist zu unterscheiden, ob dieser mit einem bzw. zwei Verzögerungsglieder oder mit einem bzw. zwei Addiererknoten zu verbinden ist. Sind die beiden Eingänge des neuen Addiererknotens $\tilde{a}_{\tilde{L}}$ mit zwei Addiererknoten zu verbinden, so ist weiterhin die in Abschn. 5.3.2 beschriebene Berechnung der Tiefe $t_{\tilde{L}} = \max(t_l, t_m)+1$ anzuwenden. Ist ein Eingang des neuen Addierers $\tilde{a}_L$ mit einem Verzögerungsknoten zu verbinden, so gilt für die Tiefe des neuen Addiererknotens $t_{\tilde{L}} = \max(t_l, t_m + 1)$. Falls beide Eingänge mit Verzögerungsgliedern zu verbinden sind, so gilt $t_{\tilde{L}} = \max(t_l, t_m)$.

  Die Berechnung der Tiefe $t_{\tilde{L}}$ für einen neu eingefügten Verzögerungsknoten, der Nachfolger des Addiererknotens $\tilde{a}_l$ ist, ergibt $t_{\tilde{L}} = t_l+1$. Ist der Verzögerungsknoten mit einem weiteren Verzögerungsknoten zu verbinden, so gilt für dessen Tiefe $t_{\tilde{L}} = t_l$. Falls der Verzögerungsknoten auf einen Registerknoten des SRAs folgt, so ist $t_{\tilde{L}} = 1$.

- **Aufbau des Addierergraphen:** Parallel zu den Änderungen in den Datenstrukturen in jedem Iterationsschritt wird der Addierergraph aufgebaut. Die Vorgehensweise beim iterativen Aufbau des Addierergraphen für Addiererknoten wurde bereits in Abschn. 5.3.2 beschrieben. Wird dem Addierergraphen ein Verzögerungsknoten $\tilde{v}_F$ hinzugefügt, so wird dem Eingang von $\tilde{v}_F$ der vorausgehende Register-, Addierer-, bzw. Verzögerungsknoten zugewiesen. Die Tiefe $t(\tilde{v}_F)$ des Verzögerungsknotens ist gleich dem Element $t_{\tilde{L}}$ des Tiefenvektors $t$, der nach obiger Regel bestimmt wird. Die Bitbreite $b(\tilde{v}_F)$ von $\tilde{v}_F$ entspricht derjenigen des eingehenden Knotens.

- **Überprüfen der Endbedingung:** Neben der bisherigen Endbedingung, dass die Summe der Einträge des Summenvektors $s$ gleich der Anzahl der Templates $N$ sein müssen, wird als zusätzliches Abbruchkriterium eingeführt, dass alle Einträge der *Translation*-Matrix $D$ gleich Null sein müssen. Für Einträge $d_{l,j} \neq 0$ werden Verzögerungsglieder mit der entsprechenden Länge $-d_{l,j}$ eingefügt.

Nach Beendigung des iterativen Teils des Algorithmus werden, wie im vorigen Abschnitt beschrieben, den Addiererbäumen ihre jeweiligen Wurzel- bzw. Schwellwertknoten zugewiesen.

### Beispiel für die Basis-Methode der Gewinn-Analyse

Für das in Abb. 5.9(b) dargestellte Beispiel wird der iterative Aufbau der *Connection*-Matrix $C$, *Translation*-Matrix $D$, *Partialsummen*-Matrix $P$ sowie der Indexliste $I$, Summevektor $s$, Tiefenvektor $t$ und Verzögerungsvektor $h$ angegeben.

Zunächst erfolgt die Initialisierung der Datenstrukturen

$$\mathbf{C^{(0)}} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{D^{(0)}} = \begin{bmatrix} 0 & 0 \\ 0 & -3 \\ 0 & -3 \\ 0 & -3 \end{bmatrix}, \mathbf{P^{(0)}} = \begin{bmatrix} - & - & - & - \\ 1 & - & - & - \\ 1 & \bar{2} & - & - \\ 1 & 2 & 2 & - \end{bmatrix}, \tag{5.12}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(\mathbf{0})} = \left[\begin{array}{c|c|c|c} 0 & 2 & 0 & - \\ 1 & 2 & 0 & - \\ 2 & 2 & 0 & - \\ 3 & 2 & 0 & - \end{array}\right], \ \tilde{a}_0 = (1, 2). \tag{5.13}$$

Das Ergebnis der Gewinn-Analyse ist es, zunächst den Addiererknoten $\tilde{a}_0$ einzufügen und mit den Registerknoten $\tilde{r}_1$ und $\tilde{r}_2$ zu verbinden.

$$\mathbf{C^{(1)}} = \left[\begin{array}{cc} 1 & 1 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ \hline 1 & 1 \end{array}\right], \mathbf{D^{(1)}} = \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & -3 \\ \hline 0 & -3 \end{array}\right], \mathbf{P^{(1)}} = \left[\begin{array}{ccccc} - & - & - & - & - \\ 0 & - & - & - & - \\ 0 & 0 & - & - & - \\ 1 & 0 & 0 & - & - \\ \hline 1 & 0 & 0 & \bar{2} & - \end{array}\right], \tag{5.14}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(\mathbf{1})} = \left[\begin{array}{c|c|c|c} 0 & 2 & 0 & - \\ 1 & 0 & 0 & - \\ 2 & 0 & 0 & - \\ 3 & 2 & 0 & - \\ \hline 4 & 2 & 1 & 0 \end{array}\right], \ \tilde{a}_1 = (3, 4). \tag{5.15}$$

In diesem Iterationsschritt wird der Addiererknoten $\tilde{a}_1$ bei der Gewinn-Analyse ausgewählt und mit dem Registerknoten $\tilde{r}_3$ und dem Addiererknoten $\tilde{a}_0$ verbunden.

$$\mathbf{C^{(2)}} = \left[\begin{array}{cc} 1 & 1 \\ 0 & 0 \\ \hline 0 & 0 \\ 1 & 1 \end{array}\right], \mathbf{D^{(2)}} = \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ \hline 0 & 0 \\ 0 & -3 \end{array}\right], \mathbf{P^{(2)}} = \left[\begin{array}{cccc} - & - & - & - \\ 0 & - & - & - \\ \hline 0 & 0 & - & - \\ \bar{1} & 0 & 0 & - \end{array}\right], \tag{5.16}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(\mathbf{2})} = \left[\begin{array}{c|c|c|c} 0 & 2 & 0 & - \\ 3 & 0 & 0 & - \\ \hline 4 & 0 & 1 & 0 \\ 5 & 2 & 2 & 0 \end{array}\right], \ \tilde{a}_2 = (0, 5). \tag{5.17}$$

Das Ergebnis der Gewinn-Analyse ist, für den einzig verbliebenen Eintrag der Partialsummenmatrix $p_{0,5} \neq 0$ den Addiererknoten $\tilde{a}_2$ einzufügen und mit Registerknoten $\tilde{r}_0$ und Addiererknoten $\tilde{a}_1$ zu verbinden.

$$\mathbf{C^{(3)}} = \left[\begin{array}{cc} 0 & 1 \\ \hline 0 & 1 \\ 1 & 0 \end{array}\right], \mathbf{D^{(3)}} = \left[\begin{array}{cc} 0 & 0 \\ \hline 0 & -3 \\ 0 & 0 \end{array}\right], \mathbf{P^{(3)}} = \left[\begin{array}{ccc} - & - & - \\ \hline 0 & - & - \\ 0 & 0 & - \end{array}\right] \tag{5.18}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(3)} = \left[\begin{array}{c|c|c|c} 0 & 1 & 0 & - \\ \hline 5 & 1 & 2 & 0 \\ 6 & 1 & 3 & 0 \end{array}\right], \ \tilde{v}_0 = (5). \tag{5.19}$$

Es wird festgestellt, dass alle Elemente der *Partialsummen*-Matrix gleich Null sind und das Element $d_{5,1} = -3$ der *Translation*-Matrix $D$ ungleich Null ist. Daher wird nach der Basis-Methode der Gewinn-Analyse das Verzögerungsglied $\tilde{v}_0$ mit der Länge $-d_{5,1} = 3$ eingefügt und mit dem Addiererknoten $\tilde{a}_1$ verbunden. Für die neuen Elemente der *Connection*-Matrix und *Partialsummen*-Matrix der alten Zeile gilt $c_{5,1} = 0$ und $d_{5,1} = 0$ und für die neue Zeile $c_{7,1} = 1$.

$$\mathbf{C^{(4)}} = \left[\begin{array}{cc} 0 & 1 \\ \hline 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{array}\right], \mathbf{D^{(4)}} = \left[\begin{array}{cc} 0 & 0 \\ \hline 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{array}\right], \mathbf{P^{(4)}} = \left[\begin{array}{ccc} - & - & - \\ \hline 0 & - & - \\ 0 & 0 & - \\ \bar{1} & 0 & 0 \end{array}\right], \tag{5.20}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(4)} = \left[\begin{array}{c|c|c|c} 0 & 1 & 0 & - \\ \hline 5 & 0 & 2 & 0 \\ 6 & 1 & 3 & 0 \\ 7 & 1 & 3 & 1 \end{array}\right], \ \tilde{a}_3 = (0, 7). \tag{5.21}$$

Durch den im letzten Iterationsschritt eingefügten Verzögerungsknoten $\tilde{v}_0$ ist es nun möglich einen weiteren Addiererknoten $\tilde{a}_3$ einzufügen. Aus der Gewinn-Analyse ergibt sich, dass dieser mit Registerknoten $\tilde{r}_0$ und $\tilde{v}_0$ zu verbinden ist. Die Tiefe $t(\tilde{a}_3)$ für den Addiererknoten $\tilde{a}_3$ ist nach der im vorigen Abschnitt angegebenen Regel zur Tiefenbestimmung gleich der Tiefe von $\tilde{v}_0$.

$$\mathbf{C^{(5)}} = \left[\begin{array}{cc} 0 & 0 \\ \hline 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{array}\right], \mathbf{D^{(5)}} = \left[\begin{array}{cc} 0 & 0 \\ \hline 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{array}\right], \mathbf{P^{(5)}} = \left[\begin{array}{ccc} - & - & - \\ \hline 0 & - & - \\ 0 & 0 & - \\ 0 & 0 & 0 \end{array}\right], \tag{5.22}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(5)} = \left[\begin{array}{c|c|c|c} 0 & 0 & 0 & - \\ \hline 6 & 1 & 3 & 0 \\ 7 & 0 & 3 & 1 \\ 8 & 1 & 3 & 0 \end{array}\right], \ END. \tag{5.23}$$

Das Abbruchkriterium des iterativen Teils des Algorithmus ist erfüllt und es folgt die Zuweisung der Schwellwertknoten $\tilde{s}_1$ und $\tilde{s}_2$.

**Beispiel für Methode mit *virtuellen* Verzögerungsgliedern der Gewinn-Analyse**

Für das in Abb. 5.10 dargestellte Beispiel werden die Ergebnisse der Gewinn-Analyse mit der Methode der *virtuellen* Verzögerungsgliedern in den einzelnen Iterationsschritten beschrieben. Zunächst wird die Initialisierung der Datenstrukturen angegeben

$$\mathbf{C}^{(\mathbf{0})} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \mathbf{D}^{(\mathbf{0})} = \begin{bmatrix} 0 & -2 & -4 \\ 0 & -2 & -4 \\ 0 & 0 & -2 \end{bmatrix}, \mathbf{P}^{(\mathbf{0})} = \begin{bmatrix} - & - & - \\ \bar{3} & - & - \\ 0 & 0 & - \end{bmatrix}, \tag{5.24}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(\mathbf{0})} = \left[\begin{array}{c|c|c|c} 0 & 3 & 0 & - \\ 1 & 3 & 0 & - \\ 2 & 2 & 0 & - \end{array}\right], \ \tilde{a}_0 = (0, 1). \tag{5.25}$$

Das max. Element der *Partialsummen*-Matrix ist $p_{1,0} = 3$. Der max. Gewinn, der durch das Einfügen eines *virtuellen* Verzögerungsknotens erzielt werden kann, ist gleich zwei. Das Ergebnis der Gewinn-Analyse ist es, den Addiererknoten $\tilde{a}_0$ einzufügen und mit den Registerknoten $\tilde{r}_1$ und $\tilde{r}_0$ zu verbinden.

$$\mathbf{C}^{(\mathbf{1})} = \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 1 & 1 \end{array}\right], \mathbf{D}^{(\mathbf{1})} = \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \\ \hline 0 & -2 & -4 \end{array}\right], \mathbf{P}^{(\mathbf{1})} = \left[\begin{array}{ccc} - & - & - & - \\ 0 & - & - & - \\ 0 & 0 & - & - \\ \hline 0 & 0 & 0 & - \end{array}\right], \tag{5.26}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(\mathbf{1})} = \left[\begin{array}{c|c|c|c} 0 & 0 & 0 & - \\ 1 & 0 & 0 & - \\ 2 & 2 & 0 & - \\ \hline 3 & 3 & 1 & 0 \end{array}\right]. \tag{5.27}$$

Alle Einträge der *Partialsummen*-Matrix sind gleich Null. Werden die Elemente der dritten Zeile um zwei verschoben mit $d_{3,j} \neq 0$ und $d_{3,j} + 2 \leq 0$, ergibt sich ein virtueller Gewinn von zwei. Daher wird ein *reales* Verzögerungsglied $\tilde{v}_0$ der Länge zwei eingefügt und mit dem Addiererknoten $\tilde{a}_0$ verbunden. Bei der Berechnung der neuen Matrizen $C$ und $D$ ist zu beachten, dass nur die beiden Einträge der dritten Zeile, die für den Verzögerungsknoten relevant sind, in die neue Zeile übertragen werden.

$$\mathbf{C}^{(\mathbf{2})} = \left[\begin{array}{ccc} 0 & 1 & 1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right], \mathbf{D}^{(\mathbf{2})} = \left[\begin{array}{ccc} 0 & 0 & -2 \\ \hline 0 & 0 & 0 \\ 0 & 0 & -2 \end{array}\right], \mathbf{P}^{(\mathbf{2})} = \left[\begin{array}{ccc} - & - & - \\ \hline 0 & - & - \\ \bar{2} & 0 & - \end{array}\right], \tag{5.28}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(\mathbf{2})} = \left[\begin{array}{c|c|c|c} 2 & 2 & 0 & - \\ \hline 3 & 1 & 1 & 0 \\ 4 & 2 & 0 & 1 \end{array}\right], \ \tilde{a}_1 = (2, 4). \tag{5.29}$$

Die Gewinn-Analyse ergibt anhand des max. Eintrags $p_{4,2} = 2$ der *Partialsummen*-Matrix, dass der Addiererknoten $\tilde{a}_1$ eingefügt wird und mit dem Registerknoten $\tilde{r}_2$ und dem Verzögerungsglied $\tilde{v}_0$ verbunden wird.

$$\mathbf{C^{(3)}} = \left[\begin{array}{ccc} 0 & 0 & 0 \\ \hline 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right], \mathbf{D^{(3)}} = \left[\begin{array}{ccc} 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{array}\right], \mathbf{P^{(3)}} = \left[\begin{array}{cccc} - & - & - & - \\ \hline 0 & - & - & - \\ 0 & 0 & - & - \\ 0 & 0 & 0 & - \end{array}\right], \quad (5.30)$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(3)} = \left[\begin{array}{c|c|c|c} 2 & 2 & 0 & - \\ \hline 3 & 1 & 1 & 0 \\ 4 & 2 & 0 & 1 \\ 5 & 2 & 0 & 0 \end{array}\right], \tilde{v}_1 = (5). \quad (5.31)$$

Alle Einträge der *Partialsummen*-Matrix sind gleich Null. Für das einzig verbliebene Element der *Translation*-Matrix $d_{5,5} \neq 0$ wird der Verzögerungsknoten $\tilde{v}_1$ mit der Länge zwei eingefügt.

$$\mathbf{C^{(4)}} = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right], \mathbf{D^{(4)}} = \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right], \mathbf{P^{(4)}} = \left[\begin{array}{ccc} - & - & - \\ 0 & - & - \\ 0 & 0 & - \end{array}\right], \quad (5.32)$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}]^{(4)} = \left[\begin{array}{c|c|c|c} 3 & 1 & 1 & 0 \\ 5 & 1 & 2 & 0 \\ 6 & 1 & 3 & 1 \end{array}\right], END. \quad (5.33)$$

Alle Komponenten der *Partialsummen*-Matrix $P$ sowie der *Translation*-Matrix $D$ sind gleich Null und das Abbruchkriterium aus Abschn. 5.3.2, welches anhand der *Connection*-Matrix bestimmt wird, ist erfüllt. Der iterative Teil des Algorithmus ist beendet und den Wurzelknoten der beiden Addiererbäumen werden deren Schwellwertknoten zugewiesen.

### 5.5.3 Verschieben mehrerer Templateelemente eines Addierers auf einen Registerknoten

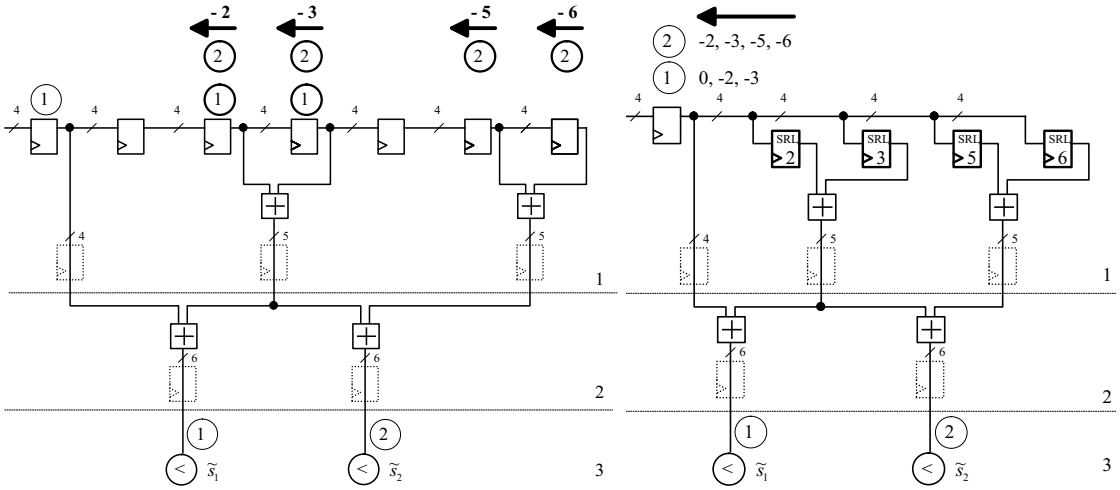
Eine weitere Strategie zur Reduzierung des FPGA-Ressourcenbedarfs ist es, die Templateelemente $t_{i,j}$ eines Templates $T_j$ so zu verschieben, dass sich mehrere Templateelemente des Templates $T_j$ nach dem Verschieben überdecken, d.h. $t_{i,j,x} - v_{i,j,x} = t_{i',j,x} - v_{i',j,x}$ erlaubt ist mit $i \neq i', 0 \leq i, i' < |T_j| - 1$. Diese Bedingung wurde im vorigen Abschnitt nicht zugelassen. Bisher war erlaubt, höchstens einen Eingang eines Addiererbaumes $A_j$ mit einem Register $\tilde{r}_l$ zu verbinden.

Daher muss die Datenstruktur der *Translation*-Matrix $D$ erweitert werden und die Einträge der *Connection*-Matrix $C$ können nun größer eins sein. Die erweiterte Form von $D$ wird im Laufe des Algorithmus auf die bisherige Datenstruktur von $D$ zurückgeführt, die Normalform genannt wird. Die Methoden zum Aufbau der Addiererbäume, die im vorigen Abschn. 5.5.2 entwickelt wurden, sind an die neuen Datenstrukturen anzupassen.

Die in diesem Abschnitt beschriebene Strategie zum Verschieben der einzelnen Templateelemente und der entsprechende Aufbau der Addiererbäume wird bei den in Abb. 2.8

angegebenen kreisförmigen Templates, bei denen die einzelnen Templateelemente durch Verschieben der Templates als Ganzes nur schlecht zur Überdeckung gebracht werden können, eingesetzt. Strategien zum Verschieben der einzelnen Templateelemente sind in Abschn. 5.7.2 und der Ressourcenbedarf für die SRAs und Addiererbäume in Abschn. 5.8.1 angegeben.

**Beispiel für einfaches Zurückführen auf Normalform (Basis-Methode)**




**(a)** Addiererbäume ohne Verschieben einzelner Templateelemente.

**(b)** Addiererbäume mit einzeln verschobenen Templateelementen mit einfachem Zurückführen auf Normalform (Basis-Methode).

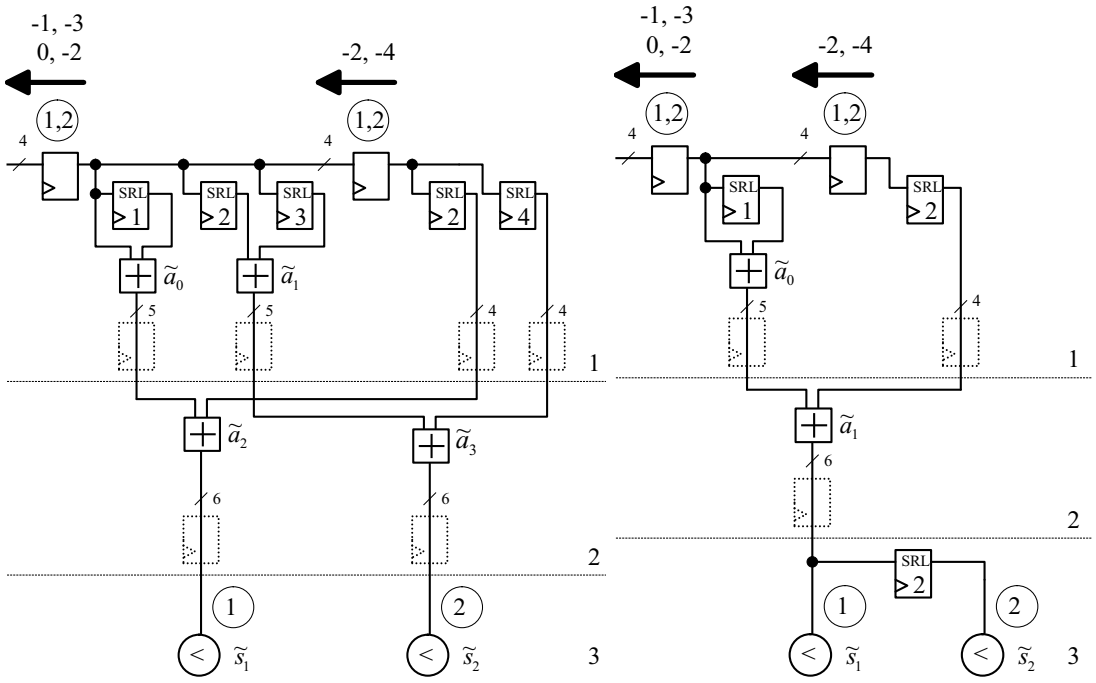
**Abbildung 5.11:** Addierergraphen mit Verschieben einzelner Templateelemente eines Templates auf einen Registerknoten.

Die Verbindungen zwischen den sieben Registern des SRAs und den Addiererbäumen sind in Abb. 5.11(a) angegeben. In Abb. 5.11(b) ist dargestellt, dass alle Templateelemente nach links auf das Register $\tilde{r}_0$ verschoben und zur Überdeckung gebracht wurden. Die Verschiebungen in negativer Richtung werden beim Aufbau des Addierergraphen zunächst durch Verzögerungsglieder mit den entsprechenden Längen ausgeglichen und dann wie bisher mit Addiererknoten zusammengefasst.

**Ressourcenbedarf** Für die erste Anordnung aus Abb. 5.11(a) werden sieben Register für die SRAs (28 FFs) und vier Addiererknoten mit $2 * 4 + 2 * 5 = 18$ LUTs für den Addiererbaum benötigt. Werden die Templateelemente gemäß Abb. 5.11(b) verschoben, so werden für das SRA nur noch ein Register (4 FFs), zusätzlich vier SRLs (16 LUTs) und wiederum vier Addiererknoten mit $2 * 4 + 2 * 5 = 18$ LUTs benötigt. Insgesamt werden vier SRLs (16 LUTs) mehr benötigt und dafür sechs Register (24 FFs) eingespart. Der Ressourcengewinn ergibt sich aus den nicht von den Addiererbäumen benötigten Registern.

**Beispiel für Gewinn-Analyse mit *virtuellen* Verzögerungsgliedern**

In Abb. 5.12 sind ein SRA bestehend aus zwei Registern und die Verbindungen der einzeln verschobenen Templateelemente von zwei Templates mit den Addiererbäumen dargestellt.




**(a)** Addiererbäume mit der Basis-Methode.

**(b)** Addiererbäume mit der Methode mit *virtuellen* Verzögerungsgliedern.

**Abbildung 5.12:** Methoden für Addiererbäume mit Verschieben einzelner Templateelemente eines Templates auf ein Register.

Wird der Addiererbaum nach der Basis-Methode aufgebaut, siehe Abb. 5.12(a), so werden zunächst die einzelnen Verschiebungen der Templateelemente durch Verzögerungsglieder ausgeglichen. Anschließend kann der weitere Aufbau des Addierergraphen nach dem im vorigen Abschnitt beschriebenen Algorithmus erfolgen.

Weniger Ressourcen werden benötigt, falls beim Aufbau des Addierergraphen zunächst nur die unbedingt notwendigen Verzögerungsglieder eingefügt werden, siehe Abb. 5.12(b). Hier wird eine Gewinn-Analyse mittels *Partialsummen*-Matrix und der Methode mit *virtuellen* Verzögerungsknoten durchgeführt, so dass zumindest lokal gesehen derjenige Knoten eingefügt wird, der zu einem minimalen Ressourcenverbrauch führt (Abschn. 5.5.2).

**Ressourcenbedarf** Für die nach der Basis-Methode aufgebauten Addiererbäume, die in Abb. 5.12(a) dargestellt sind, werden fünf SRLs (20LUTs) und vier Addierknoten mit einem Ressourcenbedarf von $2*4+2*5 = 18$ LUTs benötigt. Der in Abb. 5.12(b) dargestellte Addiererbaum, der nach der Methode mit *virtuellen* Verzögerungsgliedern aufgebaut wurde, benötigt hingegen ausschließlich drei Verzögerungsknoten mit $2*4+1*6 = 14$

LUTs und zwei Addiererknoten mit $1*4+1*5 = 9$ LUTs. Insgesamt führt die Methode mit *virtuellen* Verzögerungsknoten zu einer deutlichen FPGA-Ressourcenersparnis gegenüber der Basis-Methode.

### Algorithmus

Der im vorigen Abschnitt vorgestellte Algorithmus muss erweitert werden, weil in den Datenstrukturen, der *Translation*-Matrix $D$ nicht vorgesehen ist, dass ein Registerknoten $\tilde{r}_l$ mehrmals mit einem Addiererbaum $A_j$ zu verbinden ist.

### Initialisierung

***Connection*-Matrix $C$ und *Translation*-Matrix $D$** Die Initialisierung der *Connection*-Matrix $C$ und der *Translation*-Matrix $D$ erfolgt prinzipiell wie im vorigen Abschn. 5.5.2 beschrieben. Das Element $c_{l,j}$ der *Connection*-Matrix $C$ wird inkrementiert falls eine Verbindung zwischen Registerknoten $\tilde{r}_l$ und einem Eingang $a_{i,j}$ von Addiererbaum $A_j$ besteht. Das Element $c_{l,j}$ enthält nun die Anzahl dieser Verbindungen und ist beschränkt durch die Anzahl der Templateelemente von Template $T_j$, d.h. es gilt $0 \leq c_{l,j} \leq |T_j|$. Bisher war in den Elementen $c_{l,j}$ die Information enthalten, ob eine Verbindung zwischen Registerknoten $\tilde{r}_l$ und Addiererbaum $A_j$ besteht oder nicht, d.h. es galt $0 \leq c_{l,j} \leq 1$.

Gleichzeitig werden für die Verbindungen die Verschiebungen der einzelnen Templateelemente in die erweiterte *Translation*-Matrix $D$ eingetragen. Existiert eine Verbindung zwischen Addierereingang $a_{i,j}$ und Registerknoten $\tilde{r}_l$, so wird $c_{l,j}$ inkrementiert und $d_{l,j,e} = t_{i,j,v}$ gesetzt mit $e = c_{l,j}-1$. Der weitere Index $e$ von $d_{l,j,e}$ läuft von $e = 0, \ldots, c_{l,j}-1$. Der max. Index wird mit $e_{max} = c_{l,j} - 1$ bezeichnet. Alle Einträge $d_{l,j,0}, \ldots, d_{l,j,e_{max}}$ der *Translation*-Matrix $D$ haben unterschiedliche Werte, weil alle Templateelemente $t_{i,j}$ eines Templates $T_j$ vor dem Verschieben unterschiedlich waren.

***Partialsummen*-Matrix $P$** Die Initialisierung der *Partialsummen*-Matrix erfolgt, wie in den vorigen Abschnitten beschrieben, durch Vergleich der Einträge von jeder Zeile der *Connection*-Matrix $C$ bzw. *Translation*-Matrix $D$ mit jeder anderen Zeile von $C$ bzw. $D$. Zunächst werden alle Elemente $p_{l,m}$ der *Partialsummen*-Matix $P$ mit Null initialisiert. Hierbei ist zu beachten, dass jedes Element $d_{l,j}$ mehrere Einträge $d_{l,j,0}, \ldots, d_{l,j,e_{max}}$ besitzen kann und daher die Bedingung für die Berechnung der gemeinsamen Partialsummen geändert werden muss. Das Element $p_{l,m}$ wird inkrementiert, falls $(c_{l,j} \geq 1) \wedge (c_{m,j} \geq 1) \wedge (d_{l,j,e} = d_{m,j,e'})$ für $j = 0, \ldots, N-1$ und alle Kombinationen von $e$ und $e'$. Die Anzahl der möglichen Kombinationen für die beiden Zeilen $l$ und $m$ ist gegeben durch $\sum_{j=0}^{N-1} c_{l,j} * c_{m,j}$. Man mache sich dies anhand den entsprechenen Komponenten der beiden Zeilen $l$ und $m$ klar

$$\mathbf{c_l} = \left[ \ldots \mid c_{l,j} = 4 \mid \ldots \right], \qquad \mathbf{d_l} = \left[ \ldots \mid d_{l,j,0} \quad d_{l,j,1} \quad d_{l,j,2} \quad d_{l,j,3} \mid \ldots \right]$$
$$\mathbf{c_m} = \left[ \ldots \mid c_{m,j} = 3 \mid \ldots \right], \qquad \mathbf{d_m} = \left[ \ldots \mid d_{m,j,0} \quad d_{m,j,1} \quad d_{m,j,2} \mid \ldots \right].$$

Aufgrund der Tatsache, dass alle Einträge $d_{l,j,0}, \ldots, d_{l,j,e_{max}}$ und $d_{m,j,0}, \ldots, d_{m,j,e_{max}}$ einen unterschiedlichen Wert besitzen, kann das Element $p_{l,m}$ höchstens $\min(c_{l,j}, c_{m,j})$-mal in-

krementiert werden. Daher sind die Einträge der *Partialsummen*-Matrix durch $p_{l,m} < \sum_{j=0}^{N-1} \min(c_{l,j}, c_{m,j})$ beschränkt und nicht wie bisher durch $p_{l,m} < N$.

**Normalformvektor** $n$ Die Komponente $n_l$ des Normalformvektors $n$ ist die Summe der Elemente der Zeile $l$ der *Connection*-Matix $C$ mit $c_{l,j} - 1 > 1$, d.h. $n_l = \sum_{j=0}^{N-1}(c_{l,j} - 1)$ vorausgesetzt $c_{l,j} > 0$. Für Zeilen mit Normalform gilt $n_l = 0$. Sind alle Kompontenten von $n$ gleich Null, so besitzt die *Translation*-Matrix $D$ Normalform und für alle Einträge der *Connection*-Matrix gilt $0 \leq c_{l,j} \leq 1$.

**Weitere Datenstrukturen** Für die weiteren Datenstrukturen wie Indexliste $I$, Iterationszähler $i$ bzw. $\tilde{L}$, Zähler $L$ und $F$ für die Addierer- und Verzögerungsknoten, Tiefenvektor $t$, Summenvektor $s$ und Verzögerungsvektor $h$ erfolgt die Initialisierung wie im vorigen Abschn. 5.5.2 beschrieben.

### Einfaches Zurückführen von $C$ und $D$ auf Normalform (Basis-Methode)

Zunächst wird eine iterative Vorgehensweise vorgestellt, mit dem die erweiterte *Translation*-Matrix $D$ auf Normalform zurückgeführt wird, ohne dabei eine Gewinn-Analyse mittels *Partialsummen*-Matrix oder *virtuellen* Verzögerungsgliedern durchzuführen. Die erweiterte *Translation*-Matrix $D$ wird auf Standard-Matrizenform zurückgeführt, indem für Elemente von $c_{l,j} > 1$ und $d_{l,j,e} \neq 0$ Verzögerungsglieder mit der entsprechenden Länge $-d_{l,j,e}$ eingefügt werden.

Die folgenden Schritte werden für alle Zeilen $l$ mit $l = 0, \ldots, \tilde{R}-1$ der *Connection*-Matrix $C^{(0)}$ und *Translation*-Matrix $D^{(0)}$ nach deren Initialisierung durchgeführt.

- Zunächst wird nach einem Eintrag $c_{l,j} > 1$ der $l$-ten Zeile gesucht und der erste Eintrag mit $d_{l,j,e} \neq 0$ bestimmt mit $e = 0, \ldots, e_{max}$. In den anderen Spalten der $l$-ten Zeile wird nach gemeinsamen Einträgen $d_{l,j',e'}$ mit $d_{l,j',e'} = d_{l,j,e}$ gesucht für $e' = 0, \ldots, e'_{max}$. Für die gemeinsamen Einträge aus den unterschiedlichen Spalten $j$ wird ein Verzögerungsglied mit einem festen Wert $-d_{l,j,e}$ eingefügt.
- Die gemeinsamen Einträge mit dem Wert $d_{l,j,e}$ werden aus der alten Zeile $l$ ausgetragen und in die neue Zeile $\tilde{L} = \tilde{R} + F$ übertragen. Der Eintrag $c_{l,j}$ der alten Zeile $l$ von $C$ wird dekrementiert $c_{l,j}^{(i+1)} = c_{l,j}^{(i)} - 1$ und das Element $d_{l,j,e}$ wird aus der Liste $d_{l,j,0}, \ldots, d_{l,j,e_{max}}$ gestrichen. Außerdem werden die Einträge der neuen Zeile $c_{\tilde{L},j}^{(i+1)} = 1$ und $d_{\tilde{L},j}^{(i+1)} = 0$ gesetzt.
- Die Iterationszähler $i$ und $\tilde{L}$ sowie der Zähler für die Verzögerungsknoten $F$ wird nach dem Einfügen eines neuen Verzögerungsknoten inkrementiert. Gilt für alle Einträge $c_{l,j} \leq 1$ der Zeile $l$, so ist der Eintrag $n_l$ des Normalformvektors $n$ gleich Null und die Zeile $l$ der *Connection*-Matrix ist auf Normalform zurückgeführt.

Ist die Anzahl der neu eingefügten Verzögerungsknoten gleich $F$, so sind sowohl die *Connection*-Matrix $C$ als auch die *Translation*-Matrix $D$ von der Größe $(\tilde{R} - 1 + F) \times N$

und die *Partialsummen*-Matrix $P$ von der Größe $(\tilde{R}-1+F)\times(\tilde{R}-1+F)$. Der weitere Aufbau der Addiererbäume erfolgt nun mit dem im vorigen Abschn. 5.5.2 angegebenen Algorithmus.

### Gewinn-Analyse

Bei der Gewinn-Analyse für Addiererknoten wird, wie in den vorigen Abschnitten beschrieben, ein Addiererknoten für den max. Koeffizienten $p_{\bar{l},\bar{m}}$ der *Partialsummen*-Matrix $P$ eingefügt. Zunächst ist allerdings zu klären, ob ein Addiererknoten oder ein Verzögerungsglied eingefügt wird.

**Methode mit *virtuellen* Verzögerungsgliedern** Die Berechnung des virtuellen Gewinns erfolgt wiederum, wie im vorigen Absch. 5.5.2 beschrieben, für jede mögliche Kombination der beiden Zeilen $l$ und $m$. Bei den erweiterten Datenstrukturen wird der virtuelle Gewinn zusätzlich für $l=m$ bestimmt.

Im Folgenden wird das virtuelle Verschieben der Einträge der *Translation*-Matrix $D$ um $1\leq vd<d_{max}$ anhand den Zeilen $l$ und $m$ beschrieben. Dabei sind die Einträge der Zeile $l$ fest und die der Zeile $m$ werden virtuell um $vd$ verschoben. Bei Gleichheit der Einträge von $D$, d.h. falls $d_{l,j,e}=d_{m,j,e'}+vd$ und zusätzlich gilt $d_{m,j,e'}+vd\leq 0$, können die beiden Einträge zusammengefasst werden und der virtuelle Gewinn wird inkrementiert. Die *virtuellen* Verschiebungen und anschließende Vergleiche sind für alle Kombinationen von $e$ und $e'$ durchzuführen. Eine mehrfache Inkrementierung des virtuellen Gewinns für eine Spalte $j$ ist nicht möglich, weil die Elemente $d_{l,j,0},\ldots,d_{l,j,e_{max}}$ der erweiterten *Translation*-Matrix $D$ unterschiedlich sind.

Außerdem ist die im vorigen Abschn. 5.5.2 beschriebene, erweiterte *Tiefenminimierungs*-Regel für Verzögerungsknoten zu benutzen.

### Iterativer Aufbau der Addiererbäume

- **Änderungen der *Connection*-Matrix $C^{(i+1)}$ und der *Translation*-Matrix $D^{(i+1)}$:**

  Ist ein Addiererknoten einzufügen, so werden für die Elemente $(c^{(i)}_{\bar{l},j}\geq 1)\wedge(c^{(i)}_{\bar{m},j}\geq 1)$ und die übereinstimmenden Elemente $(d_{\bar{l},j,e}=d_{\bar{m},j,e'})$ aus den Zeilen $\bar{l}$ und $\bar{m}$ der erweiterten *Translation*-Matrix $D^{(i)}$ entfernt, d.h. $d^{(i+1)}_{\bar{l},j,e}=0$, $d^{(i+1)}_{\bar{m},j,e'}=0$ gesetzt und aus der Liste gestrichen und die Einträge $c^{(i+1)}_{\bar{l},j}$ und $c^{(i+1)}_{\bar{m},j}$ der *Connection*-Matrix $C^{(i)}$ dekrementiert. Diese werden in die neue Zeile $\tilde{L}$ der Matrizen $C^{(i+1)}$ und $D^{(i+1)}$ übertragen, d.h. $c^{(i+1)}_{\tilde{L},j}$ wird inkrementiert und $d^{(i+1)}_{\tilde{L},j}=d^{(i)}_{\bar{l},j,e}=d^{(i)}_{\bar{m},j,e'}$ gesetzt.

  Ist ein Verzögerungsglied der Länge $vd$ nach dem Addiererknoten $a_{\bar{l}}$ einzufügen, so werden die entsprechenden Einträge der *Connection*-Matrix der neuen Zeile $c^{(i+1)}_{\tilde{L},j}$ inkrementiert, vorausgesetzt $c^{(i)}_{\bar{l},j}\geq 1$ und $d^{(i)}_{\bar{l},j,e}+vd\leq 0$. Trifft diese Bedingung zu, so gilt außerdem für die Einträge der erweiterten *Translation*-Matrix $d^{(i+1)}_{\tilde{L},j}=d^{(i)}_{\bar{l},j,e}+vd$

und $d^{(i+1)}_{\bar{l},j,e} = d^{(i)}_{\bar{l},j,e} + vd$. Ist die Komponente der alten Zeile $d^{(i+1)}_{\bar{l},j,e} = 0$, so wird diese gestrichen und $c^{(i+1)}_{\bar{l},j}$ dekrementiert.

- **Änderungen der *Partialsummen*-Matrix $P^{(i+1)}$:** Die Einträge der *Partialsummen*-Matrix $P^{(i+1)}$ sind wie im vorigen Abschn. 5.5.2 beschrieben in jedem Iterationsschritt für die alten Zeilen und Spalten $\bar{l}$ und $\bar{m}$ sowie für die neue Zeile $\tilde{L}$ zu berechnen. Das Element $p_{l,m}$ wird genau dann inkrementiert, falls für die beiden Einträge der erweiterten Matrix $C$ *und* von $D$ gilt $(c_{\bar{l},j} \geq 1) \wedge (c_{\bar{m},j} \geq 1) \wedge (d_{\bar{l},j,e} = d_{\bar{m},j,e'})$ für alle möglichen Kombinationen von $e$ und $e'$. Die genauen Details wurden in diesem Abschnitt bei der Initialisierung der *Partialsummen*-Matrix $P$ beschrieben.

- **Weitere Datenstrukturen:** Nach jedem Iterationsschritt werden wie bisher der Iterationszähler $i$, der Zähler aller Knoten $\tilde{L}$ und einer der beiden Zähler $L$ für die Addiererknoten und $F$ für die Verzögerungsknoten inkrementiert, je nachdem, ob ein Addierer- oder Verzögerungsknoten eingefügt wird. Die Anpassungen der weiteren Datenstrukturen wie Indexliste $I$, Verzögerungsvektor $h$, Summenvektor $s$ und Tiefenvektor $t$ erfolgt wie in Abschn. 5.5.2 beschrieben.

Der iterative Teil des Algorithmus wird beendet, falls alle Elemente der *Translation*-Matrix $D$ gleich Null sind und die im vorigen Abschn. 5.5.2 angegebene Abbruchbedingung des Algorithmus erfüllt ist. Anschließend werden den Addiererbäumen deren Wurzel- bzw. Schwellwertknoten zugewiesen.

### Bemerkungen

Die Pfade von den Blättern, den Registerknoten, zu der Wurzel des Teilgraphen $A_j$ des Addierergraphen $A$ sind nicht mehr eindeutig. Es existieren $c_{l,j}$ Pfade ausgehend von Registerknoten $\tilde{r}_l$ für den Teilgraphen $A_j$ zu seinem Wurzelknoten. Die gesamte Verzögerung der jeweiligen Pfade für die Verzögerungsglieder ist gleich den jeweiligen Einträgen $d_{l,j,e},\ e = 0, \ldots, e_{max}$ der erweiterten *Translation*-Matrix $D$ nach deren Initialisierung. Die Teilgraphen $A_j$ sind nach den in Abschn. 5.2.1 gegebenen Definitionen keine Bäume.

Außerdem ist zu beachten, dass die Anzahl der Addiererknoten für jeden Teilgraphen $A_j$ kleiner sein kann als die Anzahl der Templateelemente von Template $T_j$.

### Beispiel für einfaches Zurückführen auf Normalform (Basis-Methode)

Im Folgenden wird beschrieben, wie die Einträge der *Connection*-Matrix $C$ und der erweiterten *Translation*-Matrix $D$ für das in in Abb. 5.11 dargestellte Beispiel mit der Basis-Methode auf Normalform zurückgeführt werden. Die Indexliste $I$, Summenvektor $s$, Tiefenvektor $t$, Verzögerungsvektor $h$ und Normalformvektor $n$ sind ebenfalls angegeben.

Die initialisierten Datenstrukturen sind gegeben durch

$$\mathbf{C^{(0)}} = \left[\begin{array}{c|c} 3 & 4 \end{array}\right], \mathbf{D^{(0)}} = \left[\begin{array}{cc|ccc|cc} 0 & -2 & -3 & -2 & -3 & -5 & -6 \end{array}\right], \tag{5.34}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}, \mathbf{n}]^{(\mathbf{0})} = \left[\begin{array}{c|c|c|c|c} 0 & 7 & 0 & 0 & 5 \end{array}\right],\ \tilde{v}_0 = (0). \tag{5.35}$$

Der erste Eintrag $d_{0,j,e} \neq 0$ in der ersten Zeile ist $d_{0,1,1} = -2$. Ein weiterer Eintrag mit dem Wert $-2$ ist das Element $d_{0,2,0} = -2$. Für beide wird das gemeinsame Verzögerungsglied $\tilde{v}_0$ der Länge zwei eingefügt und mit dem Registerknoten $\tilde{r}_0$ verbunden. Die Elemente der Matrix $C$ aus der ersten Zeile werden in die zweite Zeile übertragen, die von $D$ auf Null gesetzt und aus der Liste gestrichen. Für die nachfolgenden drei Iterationsschritte sind keine Besonderheiten festzustellen.

$$\mathbf{C}^{(1)} = \left[\begin{array}{c|c} 2 & 3 \\ \hline 1 & 1 \end{array}\right], \mathbf{D}^{(1)} = \left[\begin{array}{cc|ccc} 0 & -3 & -3 & -5 & -6 \\ \hline \multicolumn{2}{c|}{0} & \multicolumn{3}{c}{0} \end{array}\right], \tag{5.36}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}, \mathbf{n}]^{(1)} = \left[\begin{array}{c|c|c|c|c} 0 & 5 & 0 & - & 3 \\ \hline 1 & 2 & 0 & 1 & 0 \end{array}\right], \tilde{v}_1 = (0). \tag{5.37}$$

$$\mathbf{C}^{(2)} = \left[\begin{array}{c|c} 1 & 2 \\ \hline 1 & 1 \\ 1 & 1 \end{array}\right], \mathbf{D}^{(2)} = \left[\begin{array}{c|cc} 0 & -5 & -6 \\ \hline 0 & \multicolumn{2}{c}{0} \\ 0 & \multicolumn{2}{c}{0} \end{array}\right], \tag{5.38}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}, \mathbf{n}]^{(2)} = \left[\begin{array}{c|c|c|c|c} 0 & 3 & 0 & - & 1 \\ \hline 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 0 & 1 & 0 \end{array}\right], \tilde{v}_2 = (0). \tag{5.39}$$

$$\mathbf{C}^{(3)} = \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{array}\right], \mathbf{D}^{(3)} = \left[\begin{array}{c|c} 0 & -6 \\ \hline 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{array}\right], \tag{5.40}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}, \mathbf{n}]^{(3)} = \left[\begin{array}{c|c|c|c|c} 0 & 2 & 0 & - & 0 \\ \hline 1 & 2 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array}\right], \tilde{v}_3 = (0). \tag{5.41}$$

In der *Connection*-Matrix $C$ sind alle Einträge $c_{l,j} \leq 1$ und daher alle Einträge des Normalformvektors $n$ gleich 0. Daher besitzt die *Translation*-Matrix $D$ Normalform. Der weitere Aufbau des Addierergraphen erfolgt nach dem im vorigen Abschn. 5.5.2 beschriebenen Algorithmus.

### Beispiel für Gewinn-Analyse mit der Methode mit *virtuellen* Verzögerungsgliedern

Die Initialisierung der Datenstrukturen für das in Abb. 5.10 dargestellte Beispiel ist gegeben durch

$$\mathbf{C}^{(0)} = \left[\begin{array}{c|c} 2 & 2 \\ 1 & 1 \end{array}\right], \mathbf{D}^{(0)} = \left[\begin{array}{cc|cc} 0 & -1 & -2 & -3 \\ \multicolumn{2}{c|}{-2} & \multicolumn{2}{c}{-4} \end{array}\right], \mathbf{P}^{(0)} = \left[\begin{array}{cc} - & - \\ 0 & - \end{array}\right], \tag{5.42}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}, \mathbf{n}]^{(0)} = \left[\begin{array}{c|c|c|c|c} 0 & 4 & 0 & 0 & 2 \\ \hline 1 & 2 & 0 & 0 & 0 \end{array}\right], \tilde{v}_0 = (0). \tag{5.43}$$

Bei der Gewinn-Analyse für die Addiererknoten wird festgestellt, dass alle Einträge $p_{l,m}$ der *Partialsummen*-Matrix gleich Null sind. Die Gewinn-Anaylse mit der Methode mit *virtuellen* Verzögerungsknoten ergibt für die erste Zeile, die mit sich selber verglichen wird, einen virtuellen Gewinn von zwei, falls ein Verzögerungsglied der Länge eins eingefügt wird. Der Vergleich einer Zeile mit sich selbst macht nur bei der erweiterten *Connection*-Matrix $D$ Sinn. Der virtuelle Gewinn ist größer als der max. Eintrag der *Partialsummen*-Matrix. Daher wird ein Verzögerungsknoten der Länge eins eingefügt und mit dem Registerknoten $\tilde{r}_0$ verbunden.

Für die zweite Zeile könnte ebenfalls ein virtueller Gewinn von zwei durch Einfügen eines *virtuellen* Verzögerungsgliedes der Länge zwei und der Länge eins erzielt werden. Von diesen beiden wäre der Verzögerungsknoten mit der Länge zwei eingefügt worden, weil bei Gleichheit derjenige mit der größten Länge eingefügt wird. Das Einfügen eines Verzögerungsknotens mit Länge eins würde zu einem Addierergraphen mit einem zusätzlichen Verzögerungsknoten und somit einem leicht erhöhten Ressourcenbedarf führen.

$$\mathbf{C}^{(1)} = \left[\begin{array}{c|c} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{array}\right], \mathbf{D}^{(1)} = \left[\begin{array}{c|c} 0 & -2 \\ -2 & -4 \\ 0 & -2 \end{array}\right], \mathbf{P}^{(1)} = \begin{bmatrix} - & - & - \\ 0 & - & - \\ \bar{2} & 0 & \end{bmatrix} \tag{5.44}$$

$$[\mathbf{I}, \mathbf{s}, \mathbf{t}, \mathbf{h}, \mathbf{n}]^{(1)} = \left[\begin{array}{c|c|c|c|c} 0 & 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 2 & 2 & 0 & 1 & 0 \end{array}\right]. \tag{5.45}$$

Nach dem ersten Iterationsschritt sind alle Einträge des Normalform-Vektors $n$ gleich Null. Die *Translation*-Matrix $D$ ist auf Normalform zurückgeführt. Beim weiteren Aufbau des Addierergraphen, der mit dem im vorigen Abschn. 5.5.2 beschriebenen Algorithmus durchgeführt wird, treten keine Besonderheiten auf.

## 5.6 Reduzierung der Templateelemente

In diesem Abschnitt wird eine Strategie vorgestellt, die für die in Abschn. 2.2.1 angegebenen Templates zu einer deutlichen Reduzierung der FPGA-Ressourcen führt, indem die Anzahl $|T_j|$ der Templateelemente $t_{i,j}$ des Templates $T_j$ reduziert wird. Auf dem FPGA wird dann nicht mehr das ursprüngliche Template-Matching gelöst. In einer weiteren Stufe kann dann z.B. auf dem Host-PC das ursprüngliche Matching für Templates mit allen Templateelementen berechnet werden, jedoch nur für die Ergebnispunkte des Matchings auf dem FPGA. Die Reduzierung der Templateelemente der Templates ist gerechtfertigt, weil das Template-Matching auf DT-Bildern und nicht etwa auf binären Kantenbildern durchgeführt wird, siehe Kapitel 2. Außerdem werden drei Regeln beschrieben, wie die Anzahl der Templateelemente bei allgemeinen Templates reduziert werden kann. Die Templateelemente sind in den folgenden Abbildungen mit unterschiedlichen Grauwerten dargestellt, je nach Zugehörigkeit zum Richtungsbereich $k$.

### Reduktionsfaktor $C_R$

Zunächst wird geklärt, um welchen Faktor $C_R$ die Anzahl der Templateelemente $|T_j|$ reduziert werden kann. Die reduzierten Templates werden im Folgenden mit $\tilde{T}_j$ bezeichnet. Dabei sind die folgenden zwei Punkte zu berücksichtigen. Zum einen darf die Zahl der Ergebnisse beim Template-Matching nicht zu groß werden und zum anderen sollen *alle* Muster im Bild gefunden werden, die auch mit den ursprünglichen Templates $T_j$ gefunden worden wären.

Eine einfache Methode zur Reduzierung der Anzahl der Templateelemente um einen Faktor $C_R$ besteht darin, jedes $C_R$-te Templateelement in der Reihenfolge, in der sie gegeben sind, zu behalten, und alle anderen zu streichen. Anhand dieser Reduzierungsstrategie wurde an realen Bildern im Labor[4] untersucht, um welchen Faktor $C_R$ die Templateelemente $t_{i,j}$ des Templates $T_j$ reduziert werden können. Das Ergebnis dieser empirischen Messungen ist, dass sich bei Reduzierungen der Anzahl der Templateelemente um einen Faktor drei bis vier die Anzahl der Ergebnisse des Matchings nur wenig erhöht und die im Bild vorhandenen Templates weiterhin gefunden werden. Bei einer größeren Reduzierung der Templateelemente steigt die Anzahl der Ergebnisse des Matchings stark an. Genauere Untersuchungen sind hierfür bisher nicht erfolgt, jedoch wünschenswert.

### Neuer Schwellwert $\Theta_j$

Für die reduzierten Templates ist der Schwellwert $\Theta_j$ für das Ähnlichkeitsmaß, siehe Gl. 2.13, anzupassen. Hierfür wird der bisherige Schwellwert $\Theta_j$ durch den Reduktionsfaktor $C_R$ geteilt. Dies ist jedoch nicht ausreichend, da gewährleistet werden soll, dass *alle* Muster, die auch mit den ursprünglichen Templates gefunden worden wären, mit den reduzierten Templates zu finden sind. Daher wird ein zusätzlicher Ausgleichswert $C_A$ eingeführt, der von der verwendeten Metrik der Distanztransformation und der Art der Reduzierung der Templates $T_j$ abhängt und für den gilt $C_A > 1$. Der neue Schwellwert, der mit $\tilde{\Theta}_i$ bezeichnet wird, ist dann gegeben durch

$$\tilde{\Theta}_i = \frac{C_A}{C_R}\,\Theta_i, \quad C_A > 1. \tag{5.46}$$

Im Rahmen dieser Arbeit wurde ein empirischer Ausgleichswert $C_A = 1.1$ bis $1.2$ gewählt. Genauere Untersuchungen sind hierbei ebenfalls durchzuführen.

### Kreisförmige Templates

Bei der Reduzierung der Templateelemente $t_{i,j}$ der zwölf kreisförmigen Templates $T_j, j = 0, \ldots, 11$ wird folgende Vorgehensweise vorgeschlagen, dessen Ergebnis in Abb. 5.13 für die unterschiedlichen kreisförmigen Templates zu sehen ist. Zunächst werden die Templatelemente $t_{i,j}$ jeder zweiten Zeile gelöscht. In den verbleibenden Zeilen wird etwas weniger als jedes zweite Templateelement gestrichen, was für alle Templates $T_j$ nach einem möglichst ähnlichen Muster geschieht.

[4] Sowohl Innen- und Außenansichten im Labor.

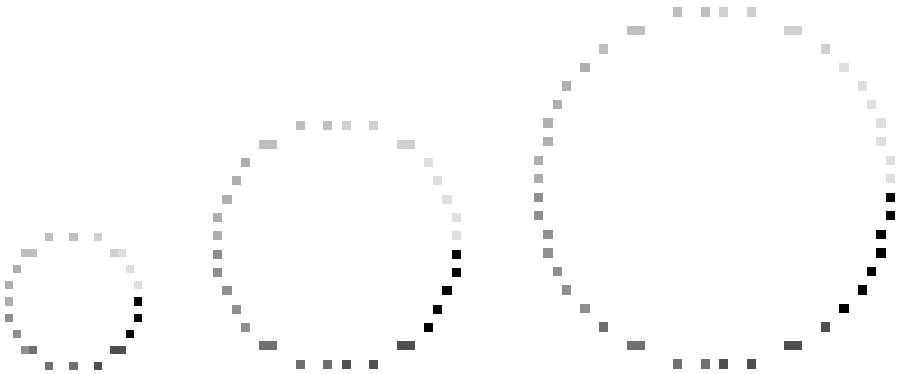


**Abbildung 5.13:** Reduzierung der Anzahl der Templateelemente bei den kreisförmigen Templates aus Abb. 2.8.

Dies führt zu einer Reduktion bei dem kleinsten kreisförmigen Template um einen Faktor $C_R = 2.7$, bei dem größten Kreistemplate um einen Faktor $3.2$, insgesamt um einen mittleren Faktor von $3.0$, siehe auch Tab. 5.1. Dieser Reduktionsfaktor stellt nach den empirischen Betrachtungen des vorigen Abschnitts einen sinnvollen Wert dar.

| $T_j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | $\sum$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lvert T_j \rvert$ | 64 | 72 | 80 | 86 | 96 | 104 | 112 | 120 | 128 | 132 | 144 | 152 | 1290 |
| $\lvert \tilde{T}_j \rvert$ | 24 | 26 | 28 | 30 | 32 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 434 |
| $C_R$ | 2.7 | 2.8 | 2.9 | 2.9 | 3.0 | 2.9 | 2.9 | 3.0 | 3.0 | 3.0 | 3.1 | 3.2 | 3.0 |

**Tabelle 5.1:** Anzahl der Templateelemente der ursprünglichen und reduzierten kreisförmigen Templates.

### Dreieckige Templates

Bei den dreieckigen Templates $T_j, j = 12, \ldots, 23$ wird in den diagonalen Seiten jede zweite Zeile, die jeweils zwei Templateelemente besitzt, gelöscht, siehe Abb. 5.14 und 5.15, was eine Reduktion um einen Faktor $C_R = 3$ zur Folge hat. In der horizontalen Seite wurde jedes zweite Templateelement eliminiert was zu einer Reduktion von $C_R = 2$, insgesamt zu einer Reduktion von $C_R = 2.6$ führt, siehe Tab. 5.2.

| $T_j$ | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | $\sum$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lvert T_j \rvert$ | 63 | 71 | 79 | 87 | 95 | 103 | 111 | 119 | 127 | 135 | 143 | 151 | 1284 |
| $\lvert \tilde{T}_j \rvert$ | 24 | 27 | 30 | 33 | 36 | 39 | 42 | 45 | 48 | 51 | 54 | 57 | 486 |
| $C_R$ | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 |

**Tabelle 5.2:** Anzahl der Templateelemente der ursprünglichen und reduzierten dreieckigen Templates.

Für die kreisförmigen und dreieckigen Templates ergibt sich insgesamt eine Reduzierung der Templateelemente um einen Faktor $C_R = 2.7$.

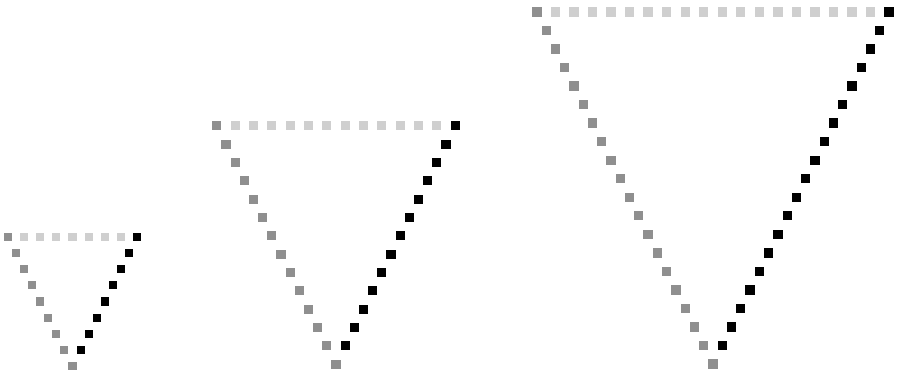


**Abbildung 5.14:** Reduzierung der Anzahl der Templateelemente bei den dreieckigen Templates mit Spitze nach unten aus Abb. 2.10.

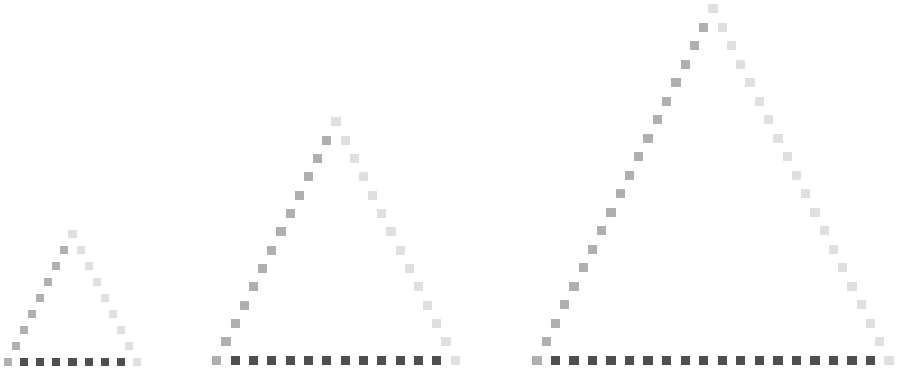


**Abbildung 5.15:** Reduzierung der Anzahl der Templateelemente bei den dreieckigen Templates mit Spitze nach oben aus Abb. 2.9.

### Allgemeine Kriterien

Im Folgenden werden drei Regeln angegeben, nach denen die Anzahl der Templateelemente eines Templates reduziert werden kann.

- Die Abstände zwischen den verbleibenden Templateelementen eines Templates sollten ungefähr gleich groß sein und ein max. Abstand nicht überschritten werden.
- Das Herausnehmen jeder zweiten Zeile der Templates kann bewirken, dass die Templates besser zur Überlappung gebracht werden können. Diese künstliche Verdichtung der Templateelemente führt letztendlich dazu, dass auf einem um einen Faktor zwei gröberen Gitter gearbeitet wird.
- Außerdem ist darauf zu achten, dass die Reduzierung der Templateelemente $t_{i,j}^k$ in jedem der $M$ Richtungsbereiche $k$ ungefähr gleich groß ist.

### Berechnung des Template-Matchings

Nach der Berechnung der Distanzmaße mit den reduzierten Templates $\tilde{T}_j$ auf dem FPGA werden die positiven Matching-Ergebnisse nochmals mit den ursprünglichen Templates

$T_j$ z.B. auf dem Host-PC berechnet. Der Nachteil besteht jedoch darin, dass die DT-Bilder vom FPGA-Koprozessor zum PC zu übertragen sind. Die verbleibenden Ergebnisse des Matchings sind anschließend für den RBF-Klassifikator relevant.

Eine andere Möglichkeit für die Berechnung des Matchings mit den ursprünglichen Templates besteht darin, das Matching mit der in Abschn. 4.2.1 beschriebenen, elementaren Implementierungsstrategie auf dem FPGA durchzuführen. Der Transfer der DT-Bilder vom FPGA-Koprozessor zum PC entfällt und der FPGA-Ressourcenbedarf für die elementare Implementierungsstrategie ist gering.

## 5.7 Verschieben der Templates

In diesem Abschnitt wird untersucht, wie durch das Verschieben der in Abschn. 2.2.1 dargetellten Templates $T_j$ und den in den Abb. 5.13 bis 5.15 dargestellten reduzierten Templates $\tilde{T}_j$ möglichst viele Überdeckungen einzelner Templatepunkte zu erzeugen sind. Die Verschiebungen haben zum einen Auswirkungen auf den FPGA-Ressourcenverbrauch der SRAs bzw. oSRAs aus Abschn. 5.1 und zum anderen auf den der Addiererbäume, die mit den in den Abschn. 5.3 bis 5.5 beschriebenen Algorithmen aufgebaut werden.

Für die Verschiebungen der Templates wurde kein systematischer Ansatz verwendet. Diese wurden mit einfachen Strategien realisiert, mit dem Ziel, möglichst viele Überdeckungen zwischen den Templateelementen zu erzielen. Die Verschiebungen für die Templates als Ganzes sind in Abschn. 5.7.1, und die für die einzelnen Templateelemente in Abschn. 5.7.2 angegeben. Der FPGA-Ressourcenbedarf für die SRAs und die optimierten Addiererbäume für die ursprünglichen und die verschobenen Templates ist in Abschn. 5.8 angegeben.

### 5.7.1 Verschieben der Templates als Ganzes

Im Folgenden werden die Verschiebungen der Templates für die jeweiligen Templateklassen angegeben und graphisch illustriert.

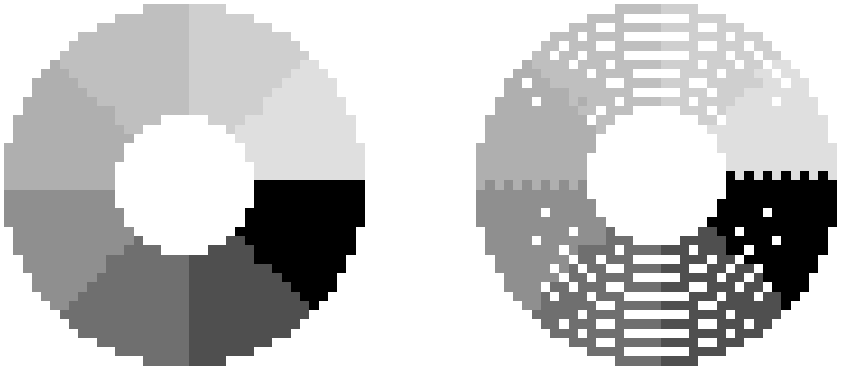
#### Kreisförmige Templates

Die unterschiedlich skalierten kreisförmigen Templates $T_j, j = 0, \ldots, 11$, die in Abb. 5.16(a) dargestellt sind, können nicht gut zur Überdeckung gebracht werden, weil sich die kreisförmigen Templates in ihren lokalen Krümmungen stark unterscheiden.

Wird jedes zweite kreisförmige Template $T_j$, $j = 0, \ldots, 11$ um ein Pixel in negativer y-Richtung verschoben

$$v_{j,x} = (0,0,0,0,0,0,0,0,0,0,0,0) \tag{5.47}$$
$$v_{j,y} = (-1,0,-1,0,-1,0,-1,0,-1,0,-1,0), \tag{5.48}$$

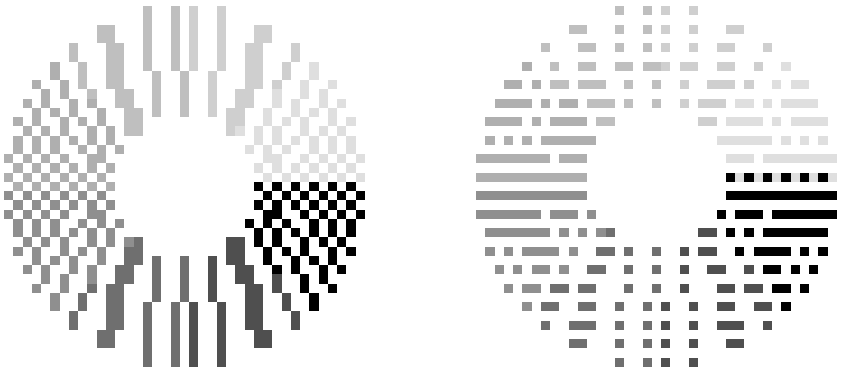
so lassen sich einige Templateelemente zur Überdeckung bringen, siehe Abb. 5.16(b). Bei den reduzierten kreisförmigen Templates, siehe Abb. 5.17(a), bei denen jede zweite Zeile



(a) (b)

**Abbildung 5.16:** (a) Unverschoben. (b) Verschieben jedes zweiten kreisförmigen Templates als Ganzes in vertikaler Richtung.

der Templateelemente gelöscht wurde, kann bei der obigen Strategie zur Verschiebung der Templates ebenfalls eine höhere Überdeckung erzielt werden, siehe Abb. 5.17(b), weil dieses Vorgehen zu einer künstlichen Verdichtung der Templateelemente führt.



(a) (b)

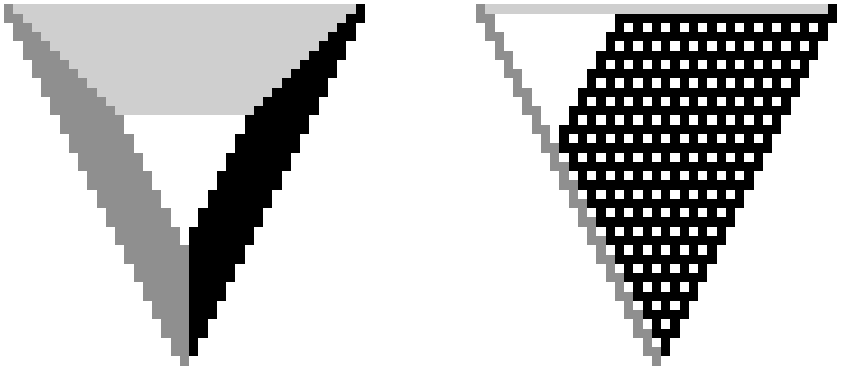
**Abbildung 5.17:** (a) Unverschoben. (b) Verschieben jedes zweiten reduzierten kreisförmigen Templates als Ganzes in vertikaler Richtung.

### Dreieckige Templates

Bei den dreieckigen Templates $T_j$, $j = 12, \ldots, 35$ kann eine hohe Anzahl an Überdeckungen erzielt werden, indem diese in eine der drei Ecken übereinander geschoben werden, siehe Abb. 5.18(b) oder Abb. 5.20(b).

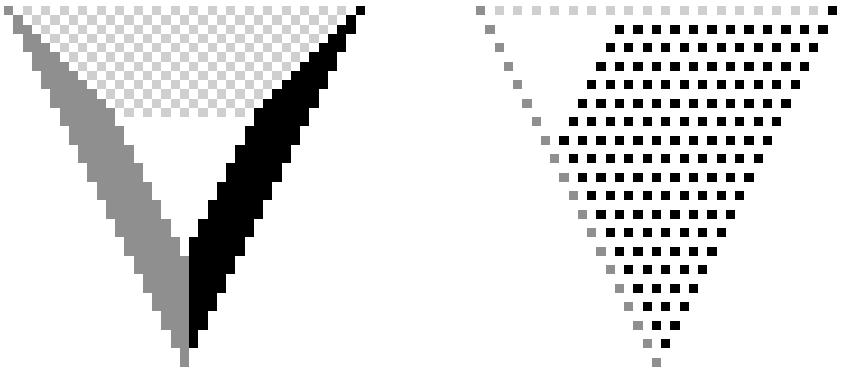
### Dreieckige Templates mit Spitze nach unten

Die dreieckigen Templates mit Spitze nach unten $T_j$, $j = 12, \ldots, 23$ werden zunächst in die linke obere Ecke geschoben, siehe Abb. 5.18(b). Die Verschiebungen der reduzierten dreieckigen Templates mit Spitze nach unten sind in Abb. 5.19(b) dargestellt.



(a) (b)

**Abbildung 5.18:** (a) Unverschoben. (b) Verschieben der dreieckigen Templates mit Spitze nach unten als Ganzes in die linke obere Ecke.



(a) (b)

**Abbildung 5.19:** (a) Unverschoben. (b) Verschieben der reduzierten dreieckigen Templates mit Spitze nach unten als Ganzes in die linke obere Ecke.

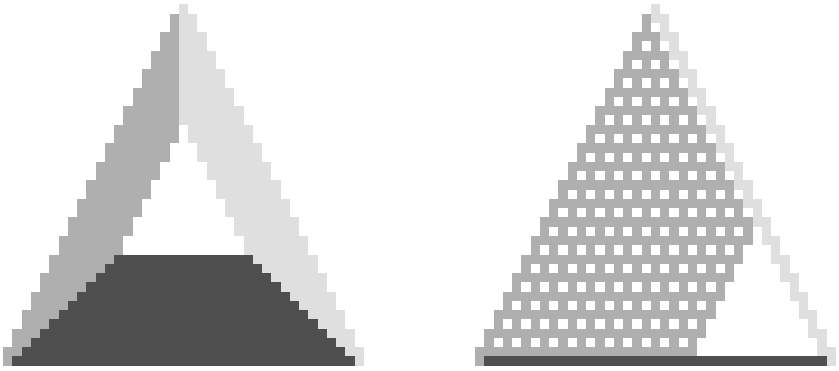
Der Verschiebungsvektor für das Verschieben der Templates in die linke obere Ecke ist $(v_{j,x}, v_{j,y}) = (j - 11, j - 11)$. Zusätzlich erfolgt eine Verschiebung in positiver x- und y-Richtung für alle Templates von $(v_{j,x}, v_{j,y}) = (6, 6)$, so dass sich die Templateelemente mit denen der kreisförmigen Templates besser überlappen. Dies wird aus den Abb. 5.22 und 5.23 ersichtlich, in denen eine Gesamtübersicht über alle verschobenen kreisförmigen und dreieckigen Templates gegeben ist. Der Verschiebungsvektor der beiden Einzelverschiebungen ist gegeben durch

$$v_{j,x} = (-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6) \quad (5.49)$$

$$v_{j,y} = (-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6). \quad (5.50)$$

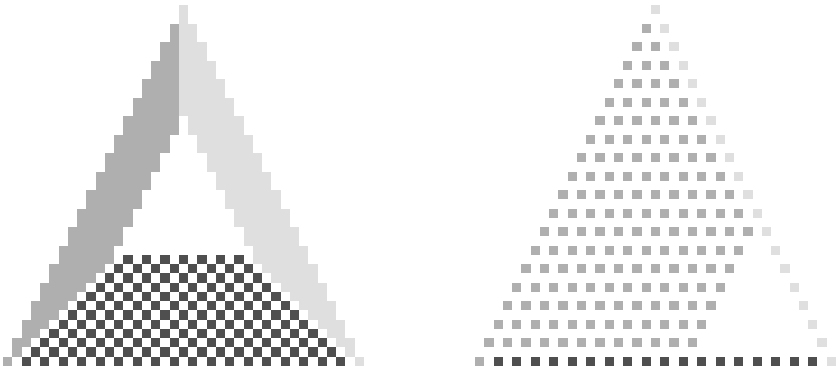
### Dreieckige Templates mit Spitze nach oben

Die verschobenen dreieckigen Templates mit Spitze nach oben sind in Abb. 5.20(b) dargestellt und die der reduzierten dreieckigen Templates mit Spitze nach oben in Abb. 5.21(b).



(a) (b)

**Abbildung 5.20:** (a) Unverschoben. (b) Verschieben der dreieckigen Templates mit Spitze nach oben als Ganzes in die rechte untere Ecke.



(a) (b)

**Abbildung 5.21:** (a) Unverschoben. (b) Verschieben der reduzierten dreieckigen Templates mit Spitze nach oben als Ganzes in die rechte untere Ecke.

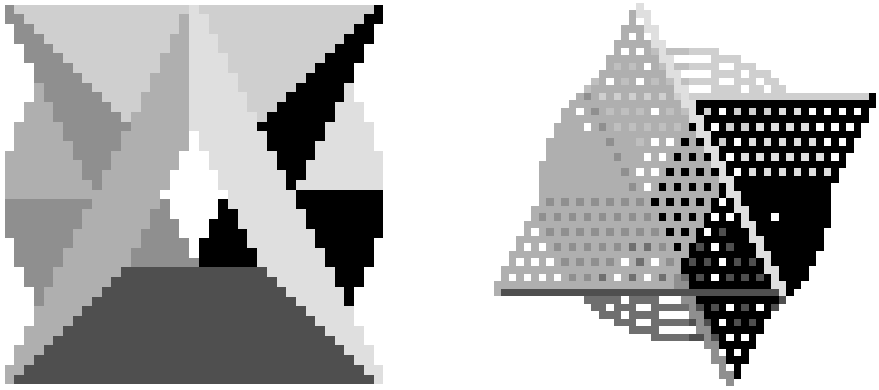
Die Verschiebungen der dreieckigen Templates mit Spitze nach oben $T_j$, $j = 24, \ldots, 35$ lassen sich wiederum in zwei Verschiebungen aufteilen. Als Erstes werden die Templates in die rechte untere Ecke geschoben. Der Verschiebungsvektor ist gegeben durch $(v_{j,x}, v_{j,y}) = (11 - j, 11 - j)$. Zweitens erfolgt eine Verschiebung in negativer x- und y-Richtung für alle Templates mit $(v_{j,x}, v_{j,y}) = (-6, -6)$, womit höhere Überlappungen mit den kreisförmigen Templates erzielt werden. Dies wird aus Abb. 5.22 und 5.23 ersichtlich, in der alle Templates der unterschiedlichen Templateklassen dargestellt sind. Insgesamt ist der Verschiebungsvektor gegeben durch

$$v_{j,x} = (5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6) \quad (5.51)$$

$$v_{j,y} = (5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6). \quad (5.52)$$

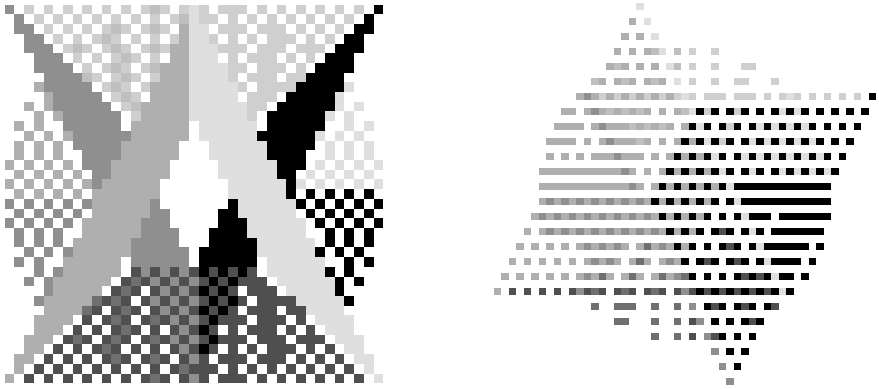
### Kreise und Dreiecke

Einen Überblick über alle verschobenen kreisförmigen und dreieckigen Templates ist in Abb. 5.22 angegeben. Die Verschiebungen für die reduzierten Templates in Abb. 5.23(b) sind so erfolgt, dass nur in jeder zweiten Zeile Templateelemente vorhanden sind.



(a) (b)

**Abbildung 5.22:** Gesamtübersicht über die kreisförmigen und dreieckigen Templates. (a) Unverschoben. (b) Verschieben der Templates als Ganzes.



(a) (b)

**Abbildung 5.23:** Gesamtübersicht über die reduzierten kreisförmigen und dreieckigen Templates. (a) Unverschoben. (b) Verschieben der reduzierten Templates als Ganzes.

### 5.7.2 Verschieben von einzelnen Templateelementen

Für die kreisförmigen und dreieckigen Templates wird im Folgenden angegeben, wie deren einzelne Templateelemente verschoben werden. Verschiebungen sind nur in negativer x-Richtung erlaubt, weil diese durch Verzögerungsglieder, deren Verzögerungen positiv sind, ausgeglichen werden, siehe Abschn. 5.5.

#### Kreisförmige Templates

Bei den kreisförmigen Templates ist es aufgrund der unterschiedlichen lokalen Krümmungen schwierig eine Strategie zur Verschiebung der einzelnen Templateelemente zu finden. Die Templateelemente einer Zeile des Richtungsbereichs $k$ werden in negativer x-Richtung auf dasjenige Templateelement der Zeile geschoben, welches sich am weitesten links befindet. Dies wird für die Templateelemente für jeden der $M$ Richtungbereiche getrennt durchgeführt. Hiermit lassen sich viele Überdeckungen zwischen den einzelnen

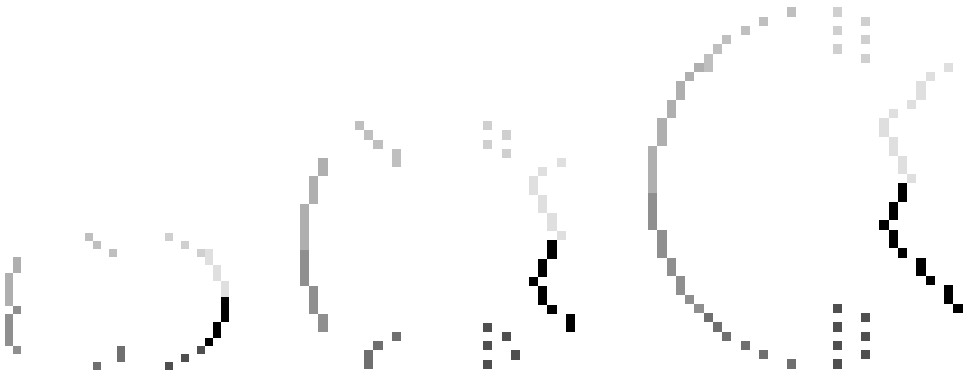


**Abbildung 5.24:** Kreisförmige Templates mit Verschieben einzelner Templateelemente.

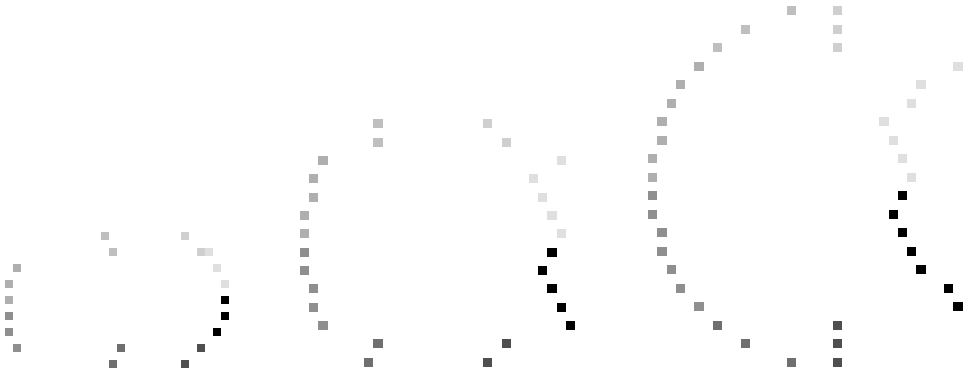
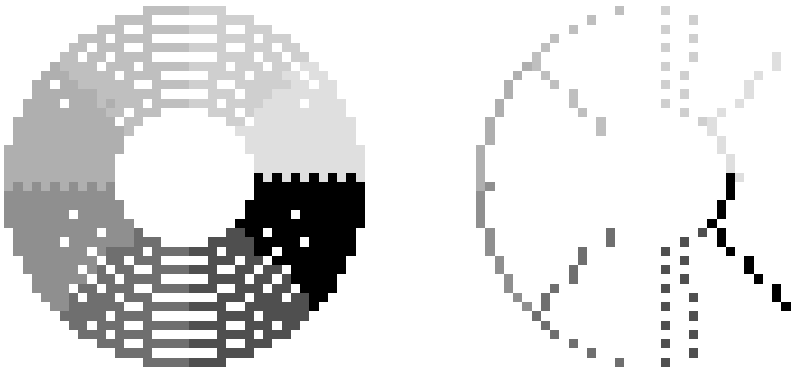


**Abbildung 5.25:** Reduzierte kreisförmige Templates mit Verschieben einzelner Templateelemente.

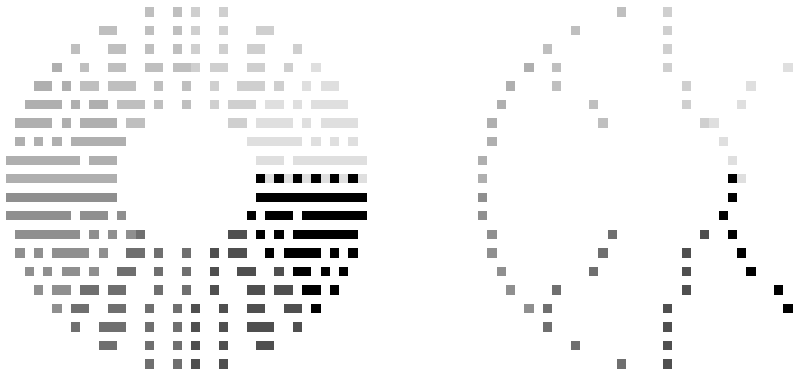
Templateelementen erzeugen. Die kreisförmigen Templates mit einzeln verschobenen Templateelementen sind in Abb. 5.24 dargestellt und die der reduzierten Templates in Abb. 5.25.

Die Verschiebungen der einzelnen Templateelemente aller kreisförmigen Templates sind in Abb. 5.26(b) angegeben und für die reduzierten Templates in Abb. 5.27(b).



(a) (b)

**Abbildung 5.26:** Verschieben der kreisförmigen Templates. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.



(a) (b)

**Abbildung 5.27:** Verschieben der reduzierten kreisförmigen Templates. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.

### Dreieckige Templates mit Spitze nach unten

Bei den dreieckigen Templates mit Spitze nach unten werden die Templateelemente der rechten diagonalen Seite nach links auf die Templateelemente des sich am weitesten links befindlichen Templates verschoben, siehe Abb. 5.28. Für die reduzierten dreieckigen Templates mit Spitze nach unten sind die Verschiebungen der einzelnen Tempalteelemente in Abb. 5.29 dargestellt.

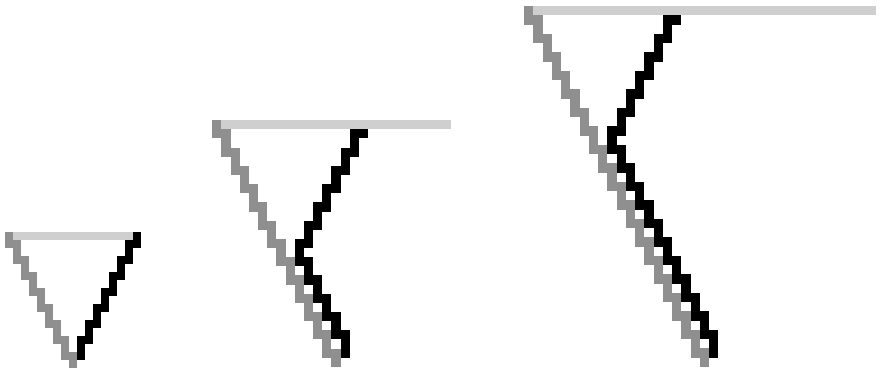


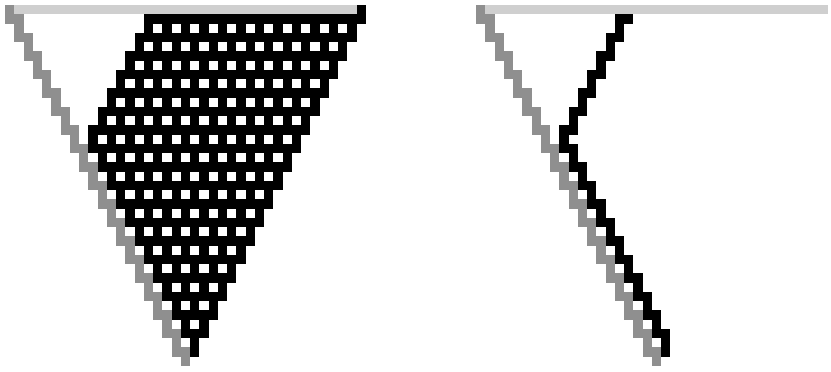
**Abbildung 5.28:** Dreieckige Templates mit Spitze nach unten mit Verschieben einzelner Templateelemente.

Die Verschiebungen der einzelnen Templateelemente aller dreieckigen Templates mit Spitze nach unten sind in Abb. 5.30(b) angegeben und die aller reduzierten Templates in Abb. 5.31(b).

### Dreieckige Templates mit Spitze nach oben

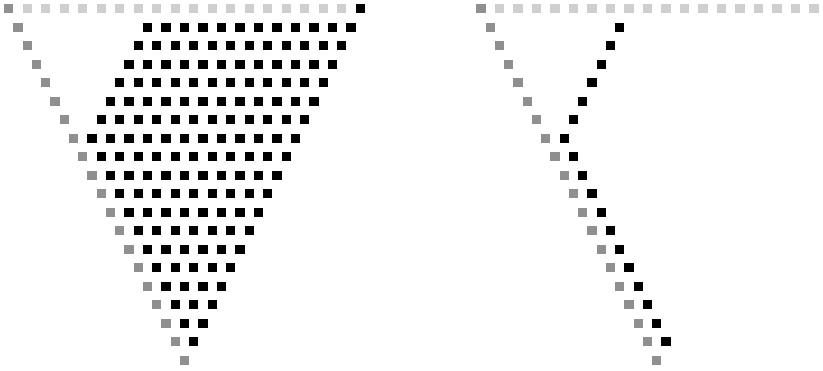
Bei den dreieckigen Templates mit Spitze nach oben werden die Templateelemente der linken diagonalen Seite nach links auf die Templateelemente des größten dreieckigen Templates verschoben, siehe Abb. 5.32. Für die reduzierten dreieckigen Templates mit

**Abbildung 5.29:** Reduzierte dreieckige Templates mit Spitze nach unten mit Verschieben einzelner Templateelemente.



(a) (b)

**Abbildung 5.30:** Verschieben der dreieckigen Templates mit Spitze nach unten. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.



(a) (b)

**Abbildung 5.31:** Verschieben der reduzierten dreieckigen Templates mit Spite nach unten. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.

Spitze nach oben sind die Verschiebungen der einzelnen Tempalteelemente in Abb. 5.33 dargestellt.

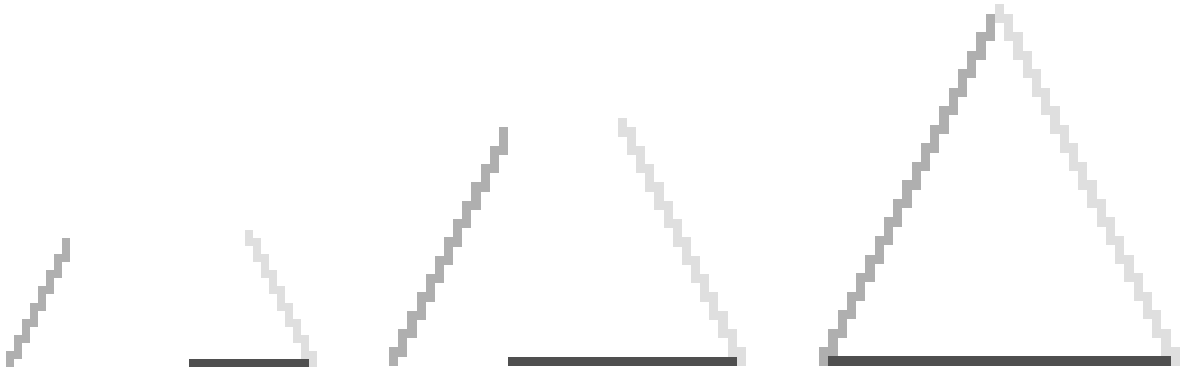
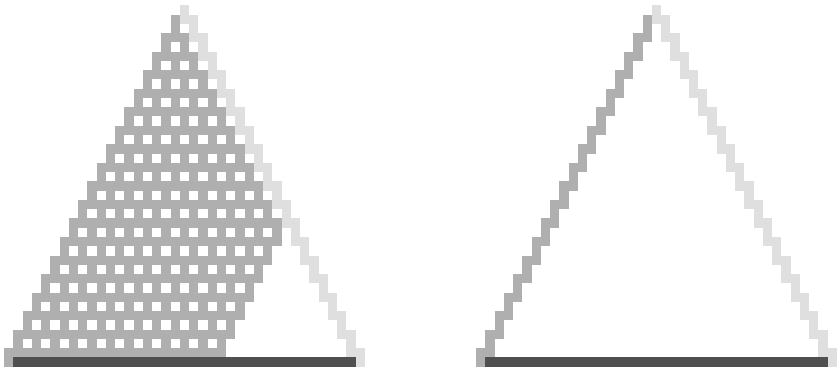


**Abbildung 5.32:** Dreiecke Templates mit Spitze nach oben mit Verschieben einzelner Templateelemente.

**Abbildung 5.33:** Reduzierte dreieckige Templates mit Spitze nach oben mit Verschieben einzelner Templateelemente.

Die Verschiebungen der einzelnen Templateelemente der dreieckigen Templates mit Spitze nach oben sind in Abb. 5.34(b) angegeben und für die reduzierten Templates in Abb. 5.35(b).



(a) (b)

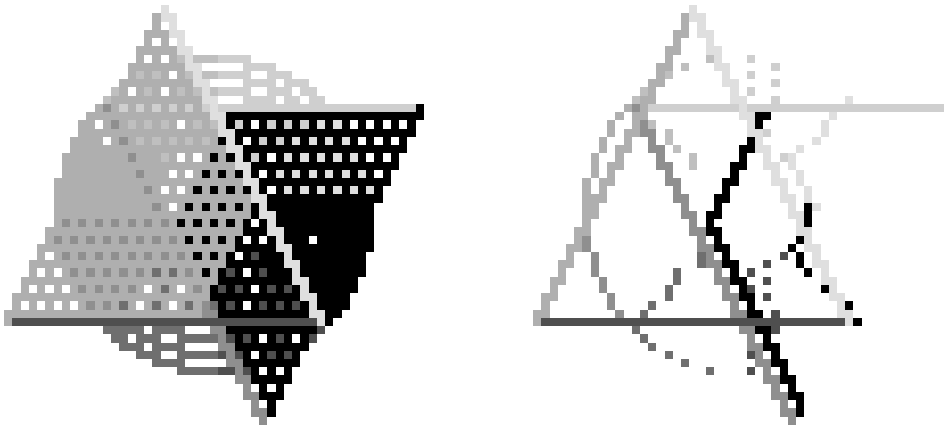
**Abbildung 5.34:** Verschieben der dreieckigen Templates mit Spitze nach oben. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.

(a) (b)

**Abbildung 5.35:** Verschieben der reduzierten dreieckigen Templates mit Spite nach oben. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.

### Kreise und Dreiecke

Die kreisförmigen und dreieckigen Templates $T_j$, bei denen die einzelnen Templateelemente verschoben wurden, sind in Abb. 5.36(b) dargestellt, die der reduzierten Templates $\tilde{T}_j$ in Abb. 5.37(b). Es ist ersichtlich, dass für die SRAs nur noch sehr wenige Register benötigt werden.



(a) (b)

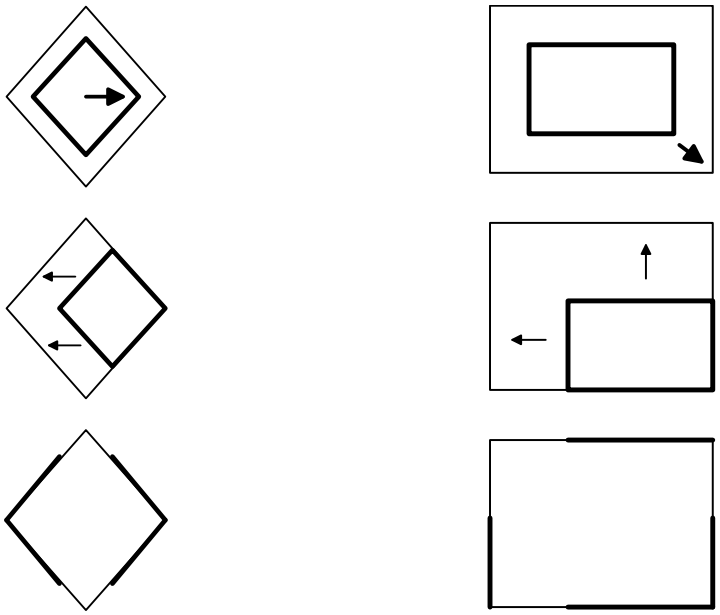
**Abbildung 5.36:** Verschieben der kreisförmigen und dreieckigen Templates. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.

### Ausblick auf Rechtecke und Rauten

Des Weiteren wird angegeben, wie bei rautenförmigen und rechteckigen Templates die einzelnen Templatepunkte verschoben werden können. Bei rautenförmigen Templates lassen sich wie in Abb. 5.38(a) dargestellt alle vier Seiten zur Überdeckung bringen. Die Verschiebungen würden in den Addiererbäumen wiederum durch Verzögerungsglieder (SRLTs) ausgeglichen werden.

(a) (b)

**Abbildung 5.37:** Verschieben der reduzierten kreisförmigen und dreieckigen Templates. (a) Verschieben als Ganzes. (b) Verschieben einzelner Templateelemente nach links in horizontaler Richtung.



(a) (b)

**Abbildung 5.38:** Verschieben einzelner Templateelemente bei (a) rautenförmigen und (b) rechteckigen Templates.

Bei den Rechtecken können ebenfalls alle vier Seiten des Rechtecks zur Überdeckung gebracht werden. Allerdings müssen die Zwischenergebnisse in den Addiererbäumen für die obere Seite des Rechtecks über mehrere Bildzeilen verzögert werden. Diese Verzögerungen würden dann mit dem internen BlockRAM, und nicht wie bisher, mit den SRLTs des Virtex-II FPGAs realisiert werden.

**Ausblick**

Ein systematischer Ansatz für das optimale Verschieben der in dieser Arbeit eingesetzten Templates ist die Formulierung und Lösung eines ganzzahligen Optimierungsproblems. Hierzu ist ein Modell des verwendeten FPGAs einzusetzen, welches dessen Ressourcen in Form von FFs und LUTs und das Verbindungsnetzwerk enthält. Eine diskrete Zielfunktion soll den Ressorcenverbrauch für die SRAs und den Addierergraphen abhängig von den Verschiebungen der Templates beschreiben. Das ganzzahlige Optimierungsproblem mit den Verschiebungen als Parameter besteht darin, diese Zielfunktion zu minimieren. Des Weiteren können Nebenbedingungen für das diskrete Optimierungsproblem eingefügt werden, welche die verfügbaren Ressourcen des FPGAs beschränken. Die Anzahl der Parameter ergibt sich aus den Verschiebungen der Templates als Ganzes und den Verschiebungen einzelner Templateelemente zu insgesamt $2N + \sum_{j}^{N-1} |T_j|$. Für die im Rahmen dieser Arbeit untersuchten Templates ist die Anzahl der Freiheitsgrade $2 * 36 + 3858 = 3930$.

## 5.8 Zusammenfassung und Ergebnisse

In diesem Abschnitt wurden vier Methoden zur Reduzierung des FPGA-Ressourcenbedarfs vorgestellt für die in Kapitel 4 vorgeschlagene Implementierungsstrategie für das parallele Template-Matching.

Dies sind im einzelnen:

- Generierung von optimierten SRAs (oSRAs) in Abschn. 5.1.
- Aufbau von optimierten Addiererbäumen ohne und mit Verschiebungen der Templateelemente der Templates in den Abschn. 5.3 bis 5.5.
- Reduzierung der Anzahl der Templateelemente der Templates in Abschn. 5.6.
- Verschiebungen der in dieser Arbeit benutzten Templates als Ganzes oder von einzelnen Templateelementen in Abschn. 5.7.

Die Auswirkungen der Verschiebungen der Templateelemente wurden anhand von Beispielen auf den FPGA-Ressourcenbedarf von SRAs bzw. optimierten SRAs und von binären Addiererbäumen bzw. optimierten Addierergraphen untersucht. Anhand den Beispielen wurde gezeigt, dass Verschiebungen, die zu vielen Überlappungen zwischen den Templateelementen der jeweiligen Templates führen, günstige Auswirkungen auf den FPGA-Ressourcenverbrauch haben. Ebenfalls wurde klar, dass es lohnend ist, einzelne Templateelemente zu verschieben, auch wenn diese durch Verzögerungsglieder auszugleichen sind. Als zusätzliche Erweiterung wurde erlaubt, mehrere Templateelemente von einem Template auf eine Position zu verschieben. Für die unverschobenen Templates wurde ein neuer Algorithmus zum Aufbau optimierter Addiererbäume entwickelt, der für die verschiedenen Arten der Verschiebungen erweitert wurde. Für Templates mit einzeln verschobenen Templateelementen wurde eine Basis-Methode und die sog. Methode mit *virtuellen* Verzögerungsgliedern entwickelt.

Für die im Rahmen dieser Arbeit verwendeten kreisförmigen und dreieckigen Templates wurden in Abschn. 5.6 eine Reduzierung der Anzahl der Templateelemente vorgeschlagen. Für die reduzierten und ursprünglichen Templates sind in Abschn. 5.7 Verschiebungen als Ganzes angegeben. Bei den kreisförmigen Templates konnten die einzelnen Templateelemente gut zur Überdeckung gebracht werden, indem mehrere Templateelemente von jeweils einem Template übereinander geschoben wurden. Bei den dreieckigen Templates tritt dieser Sonderfall nicht auf. Für Templates mit einzeln verschobenen Templateelementen wird beim Aufbau der Addiererbäume die Methode mit *virtuellen* Verzögerungsgliedern eingesetzt.

In Zukunft ist die Integration der vier Optimierungsstrategien und den daraus reslutierenden optimierten SRAs und optimierten Addiererbäume in die FPGA-Implementierung für das parallele Template-Matching aus Kapitel 4 durchzuführen.

### 5.8.1 Ressourcenverbrauch für die SRAs

Die in Abschn. 5.1 beschriebenen Optimierungsstrategien zum Aufbau der oSRAs wurden für die im Rahmen dieser Arbeit verwendeten Templates angewandt. Dies sind die ursprünglichen Templates $T_j, j = 0, \ldots, 35$ aus Abschn. 2.2.1, die aus jeweils zwölf Templates der Templateklassen Kreise, Dreiecke mit Spitze nach oben und Dreiecke mit Spitze nach unten bestehen und die hiervon reduzierten Templates $T_j$ aus Abschn. 5.6. Auf diese wurden außerdem die unterschiedlichen Strategien zum Verschieben der Templates als Ganzes, siehe Abschn. 5.7.1, und dem Verschieben einzelner Templateelemente, siehe Abschn. 5.7.2, angewandt.

#### Ressourcenverbrauch der kreisförmigen Templates

Der Ressourcenverbrauch für den Aufbau der SRAs bzw. oSRAs für die in den Abb. 5.16 und 5.26 dargestellten kreisförmigen Templates $T_j, j = 0, \ldots, 11$ sind in Tab. 5.3 angegeben, der für die kreisförmigen reduzierten Templates $\tilde{T}_j, j = 0, \ldots, 11$ aus den Abb. 5.17 und 5.27 in Tab. 5.4. Für die SRAs sind dies die Anzahl an Register (Reg) und für die oSRAs die Anzahl an Register und Verzögerungsglieder (SRLTs) in Abhängigkeit der $M$ Richtungsbereiche $k$.

Die Anzahl der benötigten Register und SRLTs für die kreisförmigen Templates ist sowohl für die SRAs als auch die oSRAs mehr oder weniger gleichverteilt über alle Richtungsbereiche $k$. Für die als Ganzes verschobenen Templates ist zu beachten, dass sich die Anzahl der benötigten Register für die SRAs leicht erhöht. Dies liegt an den vertikalen Verschiebungen von jedem zweiten kreisförmigen Template. Vergleicht man die Anzahl der Register des SRAs der unverschobenen Templates mit der Anzahl der Register und den SRLTs des oSRA der Templates mit einzeln verschobenen Templateelementen, so verringert sich diese um einen Faktor $6.9$ für die originalen und um einen Faktor $11.8$ für die reduzierten Templates.

| | Kreisförmige Templates $T_j$, $j = 0, \ldots, 11$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 0 | 196 | 131 | 12 | 210 | 130 | 23 | 135 | 15 | 14 |
| 1 | 182 | 126 | 11 | 195 | 99 | 28 | 75 | 15 | 14 |
| 2 | 196 | 132 | 12 | 210 | 100 | 28 | 150 | 15 | 14 |
| 3 | 182 | 126 | 12 | 196 | 124 | 24 | 84 | 14 | 13 |
| 4 | 196 | 131 | 12 | 196 | 127 | 20 | 84 | 14 | 13 |
| 5 | 196 | 138 | 11 | 196 | 96 | 25 | 140 | 14 | 13 |
| 6 | 182 | 120 | 12 | 169 | 83 | 24 | 65 | 13 | 12 |
| 7 | 182 | 126 | 12 | 196 | 118 | 23 | 112 | 14 | 13 |
| $\sum$ | 1512 | 1030 | 94 | 1568 | 877 | 195 | 845 | 114 | 106 |
| % | 100 | 68.1 | 6.2 | 100 | 56.0 | 12.4 | 100 | 13.5 | 12.5 |

**Tabelle 5.3:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der kreisförmigen Templates ohne und mit Verschiebungen.

| | Reduzierte kreisförmige Templates $\tilde{T}_j$, $j = 0, \ldots, 11$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRL | Reg | Reg | SRL | Reg | Reg | SRL |
| 0 | 182 | 60 | 57 | 195 | 54 | 26 | 120 | 8 | 8 |
| 1 | 168 | 49 | 34 | 180 | 32 | 23 | 75 | 8 | 7 |
| 2 | 182 | 52 | 40 | 195 | 33 | 27 | 135 | 8 | 10 |
| 3 | 182 | 56 | 54 | 182 | 49 | 21 | 78 | 7 | 6 |
| 4 | 182 | 60 | 57 | 169 | 52 | 17 | 78 | 7 | 6 |
| 5 | 182 | 54 | 41 | 169 | 32 | 23 | 117 | 7 | 10 |
| 6 | 168 | 47 | 33 | 156 | 28 | 20 | 65 | 7 | 6 |
| 7 | 182 | 56 | 54 | 182 | 48 | 26 | 117 | 7 | 9 |
| $\sum$ | 1428 | 434 | 370 | 1428 | 328 | 183 | 785 | 59 | 62 |
| % | 100 | 30.4 | 25.9 | 100 | 23.0 | 12.8 | 100 | 7.5 | 7.9 |

**Tabelle 5.4:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der reduzierten kreisförmigen Templates ohne und mit Verschiebungen.

### Ressourcenverbrauch der dreieckigen Templates mit Spitze nach unten

Der Ressourcenverbrauch für den Aufbau der SRAs bzw. oSRAs für die in den Abb. 5.18 und 5.30 dargestellten dreieckigen Templates mit Spitze nach unten $T_j, j = 12, \ldots, 23$ sind in Tab. 5.5 angegeben, der für die reduzierten Templates $\tilde{T}_j, j = 12, \ldots, 23$ aus den Abb. 5.19 und 5.31 in Tab. 5.6.

Die Anzahl der benötigten Register sind für die unterschiedlichen Richtungsbereiche ungleichmäßig verteilt. Für die unverschobenen dreieckigen Templates mit Spitze nach un-

| | Dreieckige Templates mit Spitze nach unten $T_j$, $j = 12, \ldots, 23$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 0 | 722 | 188 | 41 | 1140 | 474 | 193 | 456 | 56 | 37 |
| 3 | 780 | 201 | 49 | 780 | 58 | 76 | 780 | 58 | 76 |
| 6 | 444 | 312 | 14 | 37 | 37 | 0 | 37 | 37 | 0 |
| $\sum$ | 1946 | 701 | 104 | 1957 | 569 | 269 | 1273 | 151 | 113 |
| % | 100 | 36.0 | 5.3 | 100 | 29.1 | 13.7 | 100 | 11.9 | 8.9 |

**Tabelle 5.5:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der dreieckigen Templates mit Spitze nach unten ohne und mit Verschiebungen.

| | Reduzierte dreieckige Templates mit Spitze nach unten $\tilde{T}_j$, $j = 12, \ldots, 23$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 0 | 703 | 162 | 41 | 1110 | 162 | 202 | 444 | 19 | 36 |
| 3 | 780 | 174 | 51 | 780 | 19 | 57 | 780 | 20 | 57 |
| 6 | 420 | 150 | 152 | 35 | 18 | 17 | 35 | 18 | 17 |
| $\sum$ | 1903 | 486 | 244 | 1925 | 200 | 276 | 1259 | 57 | 110 |
| % | 100 | 25.5 | 12.8 | 100 | 10.4 | 14.3 | 100 | 4.5 | 8.7 |

**Tabelle 5.6:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der reduzierten dreieckigen Templates mit Spitze nach unten ohne und mit Verschiebungen.

ten fällt auf, dass die diagonalen Seiten der Dreiecke zu einem ungefähr doppelt so hohen Ressourcenbedarf führen als deren horizontalen Seiten. Für die als Ganzes verschobenen Templates wird ersichtlich, dass sich die Anzahl der Register im SRA 0 deutlich erhöht und die im Richtungsbereich sechs sehr stark verringert. Dies liegt daran, weil sich die horizontalen Seiten der Dreiecke im Richtungsbereich sechs überlappen, während die diagonalen Seiten von Richtungsbereich 0 voneinander weg geschoben wurden. Insgesamt kann für die ursprünglichen Templates der FPGA-Ressourcenbedarf an Registern bzw. Verzögerungsgliedern um einen Faktor $7.4$ verringert werden und für die reduzierten Templates um einen Faktor $11.4$.

### Ressourcenverbrauch der dreieckigen Templates mit Spitze nach oben

Der Ressourcenverbrauch für den Aufbau der SRAs bzw. oSRAs für die in den Abb. 5.20 und 5.34 dargestellten dreieckigen Templates mit Spitze nach oben $T_j, j = 24, \ldots, 35$ sind in Tab. 5.7 angegeben, der für die reduzierten Templates $\tilde{T}_j, j = 24, \ldots, 35$ aus den Abb. 5.21 und 5.35 in Tab. 5.8.

Bei den als Ganzes verschobenen dreieckigen Templates mit Spitze nach oben erhöht sich die Anzahl der benötigten Register für Richtungsbereich vier und wird minimal für Rich-

| | Dreieckige Templates mit Spitze nach oben $T_j$, $j = 24, \ldots, 35$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 1 | 444 | 312 | 14 | 37 | 37 | 0 | 37 | 37 | 0 |
| 4 | 722 | 188 | 41 | 1140 | 474 | 193 | 722 | 56 | 74 |
| 7 | 780 | 201 | 49 | 780 | 180 | 76 | 780 | 58 | 76 |
| $\sum$ | 1946 | 701 | 104 | 1957 | 569 | 218 | 1539 | 151 | 150 |
| % | 100 | 36.0 | 5.3 | 100 | 29.1 | 11.1 | 100 | 9.8 | 9.8 |

**Tabelle 5.7:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der reduzierten dreieckigen Templates mit Spitze nach oben ohne und mit Verschiebungen.

| | Reduzierte dreieckige Templates mit Spitze nach oben $\tilde{T}_j$, $j = 24, \ldots, 35$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 1 | 420 | 150 | 152 | 35 | 18 | 17 | 35 | 18 | 17 |
| 4 | 703 | 162 | 41 | 1110 | 162 | 202 | 703 | 19 | 54 |
| 7 | 780 | 174 | 51 | 780 | 20 | 57 | 780 | 20 | 57 |
| $\sum$ | 1903 | 486 | 244 | 1925 | 200 | 276 | 1518 | 57 | 128 |
| % | 100 | 25.5 | 12.8 | 100 | 10.4 | 14.3 | 100 | 3.8 | 8.4 |

**Tabelle 5.8:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der reduzierten dreieckigen Templates mit Spitze nach oben ohne und mit Verschiebungen.

tungsbereich eins. Dies liegt daran, dass die horizontalen Seiten der Dreiecke aus Richtungsbereich eins übereinandergeschoben und die aus Richtungsbereich vier auseinander geschoben wurden. Insgesamt konnte der Ressourcenbedarf an Registern bzw. SRLTs um einen Faktor $6.5$ für die ursprünglichen Templates und um einen Faktor $10.3$ für die reduzierten Templates verringert werden.

### Ressourcenverbrauch der kreisförmigen und dreieckigen Templates

Es folgt nun eine Zusammenfassung des Ressourcenbedarfs für alle Templates. Der Ressourcenverbrauch für den Aufbau der SRAs bzw. oSRAs für die in den Abb. 5.22 und 5.36 dargestellten kreisförmigen und dreieckigen Templates $T_j, j = 0, \ldots, 35$ sind in Tab. 5.9 angegeben, der für die reduzierten Templates $\tilde{T}_j, j = 0, \ldots, 35$ aus den Abb. 5.23 und 5.37 in Tab. 5.10.

Der Ressourcenbedarf für alle kreisförmigen und dreieckigen Templates konnte insgesamt um einen Faktor $5.5$ bzw. $8.3$ für die ursprünglichen bzw. reduzierten Templates verringert werden. In dieser Arbeit wurden DT-Bilder mit vier Bit Pixeln verwendet. Dies führt für die SRAs der unverschobenen Templates zu einem Ressourcenbedarf von $17728$ Flip-Flops (FFs) und für die Templates mit einzeln verschobenen Templateelementen zu

| | Kreisförmige und dreieckige Templates $T_j$, $j = 0, \ldots, 35$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 0 | 722 | 305 | 47 | 1140 | 523 | 169 | 722 | 71 | 74 |
| 1 | 518 | 325 | 18 | 570 | 126 | 43 | 555 | 51 | 39 |
| 2 | 196 | 132 | 12 | 210 | 100 | 28 | 150 | 15 | 14 |
| 3 | 780 | 315 | 55 | 1014 | 182 | 84 | 1014 | 72 | 76 |
| 4 | 722 | 305 | 47 | 1140 | 522 | 170 | 722 | 69 | 72 |
| 5 | 196 | 138 | 11 | 196 | 96 | 25 | 140 | 14 | 13 |
| 6 | 518 | 319 | 18 | 481 | 111 | 35 | 481 | 49 | 33 |
| 7 | 780 | 315 | 55 | 1014 | 176 | 86 | 780 | 72 | 79 |
| $\sum$ | 4432 | 2154 | 263 | 5765 | 1836 | 640 | 4564 | 413 | 400 |
| % | 100 | 48.6 | 5.9 | 100 | 31.8 | 11.1 | 100 | 9.0 | 8.8 |

**Tabelle 5.9:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der kreisförmigen und dreieckigen Templates ohne und mit Verschiebungen.

| | Reduzierte kreisförmige und dreieckige Templates $\tilde{T}_j$, $j = 0, \ldots, 35$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Ohne Verschieben | | | Verschieben als Ganzes | | | Verschieben einzeln | | |
| k | SRA | oSRA | | SRA | oSRA | | SRA | oSRA | |
| | Reg | Reg | SRLT | Reg | Reg | SRLT | Reg | Reg | SRLT |
| 0 | 703 | 218 | 93 | 1110 | 193 | 186 | 703 | 27 | 54 |
| 1 | 490 | 178 | 136 | 540 | 47 | 55 | 525 | 25 | 49 |
| 2 | 182 | 52 | 40 | 195 | 33 | 27 | 135 | 8 | 10 |
| 3 | 780 | 226 | 100 | 1014 | 69 | 88 | 1014 | 27 | 76 |
| 4 | 703 | 218 | 93 | 1110 | 191 | 188 | 703 | 26 | 60 |
| 5 | 182 | 54 | 41 | 169 | 32 | 23 | 117 | 7 | 10 |
| 6 | 490 | 176 | 136 | 455 | 42 | 49 | 455 | 24 | 45 |
| 7 | 780 | 226 | 100 | 1014 | 68 | 92 | 819 | 27 | 63 |
| $\sum$ | 4310 | 1348 | 739 | 5607 | 675 | 708 | 4471 | 171 | 367 |
| % | 100 | 31.2 | 17.1 | 100 | 12.0 | 12.6 | 100 | 3.8 | 8.2 |

**Tabelle 5.10:** Anzahl der Register bzw. SRLTs für die $k$ SRAs bzw. oSRAs der reduzierten kreisförmigen und dreieckigen Templates ohne und mit Verschiebungen.

$1652$ FFs und $1600$ LUTs. Dies entspricht einer Auslastung des Virtex-II XC2V3000-FPGAs von $62$ % an FFs für die unverschobenen Templates und $5.8$ % an FFs bzw. $5.6$% an LUTs für die Templates mit einzeln verschobenen Templateelementen. Für die reduzierten Templates mit einzeln verschobenen Templateelementen ist eine Auslastung an FFs und LUTs des Virtex-II XC2V3000 FPGAs von $3.1$ % zu veranschlagen. Der Ressourcenbedarf für die SRAs bzw. oSRAs für alle Templates ist zusammenfassend in Diagramm 5.39 illustriert.

Der Ressourcenbedarf an Block-RAM, der abhängig ist von der Ausdehnung $D_y^k$ der SRAs

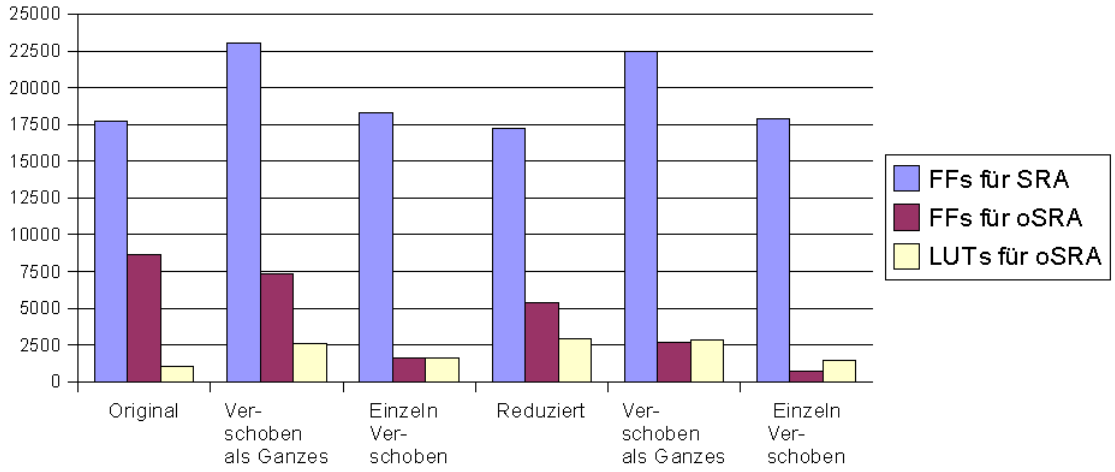



**Abbildung 5.39:** FPGA-Ressourcenbedarf für die SRAs bzw. oSRAs für alle originalen (links) und reduzierten (rechts) kreisförmigen und dreieckigen Templates für vier Bit DT-Pixel.

bzw. oSRAs in y-Richtung der jeweiligen Richtungsbereiche $k$, siehe Abschn. 4.3.1, ist für alle in dieser Arbeit beschriebenen Templatekonfigurationen identisch. Es werden jeweils 28 BlockRAMs der Virtex-II Familie benötigt.

### 5.8.2 Ressourcenverbrauch für die Addiererbäume

In den Abschn. 5.3 bis 5.5 sind die Algorithmen zum Aufbau von optimierten Addierergraphen angegeben. Diese unterscheiden sich darin, ob die Templates unverschoben, als Ganzes verschoben (Abschn. 5.7.1) oder einzelne Templateelemente verschoben (Abschn. 5.7.2) sind. Die Algorithmen wurden für die im Rahmen dieser Arbeit verwendeten ursprünglichen Templates $T_j$ aus Abschn. 2.2.1 und die hiervon reduzierten Templates $\tilde{T}_j$ aus Abschn. 5.6 angewandt.

Für den FPGA-Ressourcenbedarf ist nicht nur die Anzahl der Addierer sondern auch deren Bitbreite von Relevanz. Der Ressourcenbedarf an LUTs auf dem FPGA berechnet sich aus der Anzahl der Addierer multipliziert mit deren max. Bitbreite der beiden Eingangssignale.

#### Ressourcenverbrauch der kreisförmigen Templates

Der Ressourcenverbrauch für den Aufbau der binären Addiererbäumen und optimierten Addierergraphen für die in den Abb. 5.22 und 5.36 dargestellten kreisförmigen Templates $T_j, j = 0, \ldots, 11$ sind in Tab. 5.11 angegeben, der für die reduzierten Templates $\tilde{T}_j, j = 0, \ldots, 11$ aus den Abb. 5.23 und 5.37 in Tab. 5.12. Für Templates mit einzeln verschobenen Templateelementen wurde der optimierte Addierergraph mit der Gewinn-Analyse mit der Methode mit *virtuellen* Verzögerungsgliedern aufgebaut. Dies führt zu einem um ca. 15% geringeren Ressourcenbedarf als das einfache Zurückführen auf Normalform (Basis-Methode).

| | | Kreisförmige Templates $T_j$, $j = 0, \ldots, 11$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Binäre- | Optimierte Addiererbäume | | | | | | |
| | | original | original | | Verschieben | | Verschieben einzeln | | |
| t | b | t = b | t | b | t | b | t | b | |
| 1 | 4 | 645 | 515 | 515 | 432 | 432 | 84 | 87 | |
| 2 | 5 | 322 | 255 | 255 | 218 | 218 | 74 | 92 | |
| 3 | 6 | 161 | 126 | 126 | 106 | 106 | 70 | 90 | |
| 4 | 7 | 78 | 64 | 64 | 55 | 55 | 63 | 84 | |
| 5 | 8 | 39 | 32 | 32 | 26 | 26 | 49 | 57 | |
| 6 | 9 | 19 | 21 | 22 | 19 | 22 | 33 | 26 | |
| 7 | 10 | 11 | 12 | 12 | 12 | 14 | 27 | 20 | |
| 8 | 11 | 3 | 4 | 3 | 8 | 3 | 23 | 3 | |
| 9 | 12 | 0 | 0 | 0 | 0 | 0 | 18 | 0 | |
| 10 | 13 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | |
| 11 | 14 | 0 | 0 | 0 | 0 | 0 | 6 | 0 | |
| 12 | 15 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | |
| $\sum$ Add (LUTs) | | 1278 | 1029 | | 877 | | 459 (2859) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 189 (1135) | | |
| $\sum$ LUTs | | 6328 | 5146 | | 4418 | | 3994 | | |
| Einfügen von Registerstufen nach jeder dritten Stufe | | | | | | | | | |
| $\sum$ Reg (FFs) | | 190 (1433) | 161 (1194) | | 146 (1076) | | 179 (1308) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 3 (18) | | |

**Tabelle 5.11:** Anzahl der Addiererknoten, Registerknoten und Verzögerungsglieder der kreisförmigen Templates ohne und mit Verschiebungen. Die Addiererknoten sind aufgelistet nach deren Stufentiefe $t$ und Bitbreite $b$.

Die Analyse des Ressourcenbedarfs erfolgt gemeinsam für die ursprünglichen und reduzierten Templates. Für die unverschobenen Templates ergibt sich für den Addierergraphen eine leichte Ressourcenersparnis, weil eine geringe Anzahl von bereits vorhandenen Überdeckungen zwischen den Templates durch den Algorithmus zum Aufbau des optimierten Addierergraphen ausgenutzt wird. Der Ressourcenbedarf für die als Ganzes verschobenen Templates verringert sich weiterhin, weil die durch das Verschieben der Templates zusätzliche Überdeckungen zwischen den Templates erzeugt werden konnten. Für die Templates mit einzeln verschobenen Templateelementen, werden wesentlich weniger Ressourcen für die Addierer benötigt. Allerdings wird dieser Vorteil durch die zusätzlichen Verzögerungsglieder, mit denen die einzelnen Verschiebungen auszugleichen sind, fast wieder kompensiert. Hierbei ist außerdem zu beachten, dass sich die Tiefe des Addiererbaums von acht auf zwölf erhöht.

Der Ressourcenbedarf für die Register, die in jeder dritten Stufe in den Addierergraphen eingefügt werden, ist für die Templates mit einzeln verschobenen Templateelementen am Größten. Dies liegt daran, weil der Addierergraph für diese am Tiefsten ist.

Insgesamt lässt sich der Ressourcenbedarf für die LUTs um einen Faktor $1.6$ für die ursprünglichen kreisförmigen Templates und um einen Faktor $1.2$ für die reduzierten kreisförmigen Templates verringern. Die Ressourcenersparnis fällt bei den kreisförmigen

| | | Reduzierte kreisförmige Templates $\tilde{T}_j,\ j = 0, \ldots, 11$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Binäre- | Optimierte Addiererbäume | | | | | | |
| | | original | original | | Verschieben | | Verschieben einzeln | | |
| t | b | t= b | t | b | t | b | t | b | |
| 1 | 4 | 217 | 217 | 217 | 160 | 160 | 46 | 46 | |
| 2 | 5 | 106 | 106 | 106 | 78 | 78 | 41 | 45 | |
| 3 | 6 | 53 | 53 | 53 | 39 | 39 | 38 | 47 | |
| 4 | 7 | 27 | 27 | 27 | 24 | 25 | 28 | 35 | |
| 5 | 8 | 12 | 12 | 12 | 15 | 18 | 18 | 21 | |
| 6 | 9 | 7 | 7 | 7 | 11 | 7 | 17 | 10 | |
| 7 | 10 | 0 | 0 | 0 | 0 | 0 | 11 | 0 | |
| 8 | 11 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | |
| 9 | 12 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | |
| $\sum$ Add (LUTs) | | 422 | 422 | | 327 | | 204 (1194) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 96 (574) | | |
| $\sum$ LUTs | | 2064 | 2064 | | 1646 | | 1768 | | |
| Einfügen von Registerstufen nach jeder dritten Stufe | | | | | | | | | |
| $\sum$ Reg (FFs) | | 65 (471) | 65 (471) | | 63 (457) | | 67 (491) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 0 | | |

**Tabelle 5.12:** Anzahl der Addiererknoten, Registerknoten und Verzögerungsglieder der reduzierten kreisförmigen Templates ohne und mit Verschiebungen. Die Addiererknoten sind aufgelistet nach deren Stufentiefe $t$ und Bitbreite $b$.

Templates nicht so hoch aus, weil diese nur schlecht zur Überdeckung gebracht werden konnten. Allerdings ist zu beachten, dass durch die Templates mit einzeln verschobenen Templateelementen sehr viele Ressourcen beim Aufbau der SRAs bzw. oSRAs eingespart werden konnten.

### Ressourcenverbrauch der dreieckigen Templates

Der Ressourcenverbrauch für den Aufbau der binären Addiererbäumen und optimierten Addierergraphen für die in den Abb. 5.22 und 5.36 dargestellten dreieckigen Templates mit Spitze nach unten $T_j, j = 12, \ldots, 23$ sind in Tab. 5.13 angegeben, der für die reduzierten Templates $\tilde{T}_j, j = 12, \ldots, 23$ aus den Abb. 5.23 und 5.37 in Tab. 5.14.

Für die dreieckigen Templates mit Spitze nach oben $T_j, j = 24, \ldots, 35$ und für die reduzierten Templates $\tilde{T}_j, j = 24, \ldots, 35$ ändert sich die Anzahl der benötigten Addierer und Verzögerungsglieder nur minimal, so dass derer Ressourcenbedarf nicht gesondert angegeben wird.

Es erfolgt wiederum eine gemeinsame Analyse des Ressourcenbedarfs für die ursprünglichen und die reduzierten dreieckigen Templates. Für die unverschobenen dreieckigen Templates ergibt sich immerhin eine beachtliche Ressourcenersparnis von einem Faktor $1.7$. Diese kann für die als Ganzes verschobenen Templates nochmals auf insgesamt einen Faktor $2.2$ verbessert werden. Für die beiden Seiten des Dreiecks, die nach dem Verschie-

| | | Dreieckige Templates mit Spitze nach unten $T_j$, $j = 12, \ldots, 23$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Binäre- | Optimierte Addiererbäume | | | | | | |
| | | original | original | | Verschieben | | Verschieben einzeln | | |
| t | b | t = b | t | b | t | b | t | b | |
| 1 | 4 | 636 | 340 | 340 | 276 | 276 | 65 | 66 | |
| 2 | 5 | 324 | 174 | 174 | 138 | 138 | 37 | 37 | |
| 3 | 6 | 162 | 83 | 83 | 75 | 75 | 19 | 19 | |
| 4 | 7 | 78 | 41 | 41 | 32 | 32 | 3 | 3 | |
| 5 | 8 | 39 | 25 | 31 | 16 | 16 | 2 | 9 | |
| 6 | 9 | 19 | 23 | 25 | 9 | 14 | 3 | 20 | |
| 7 | 10 | 11 | 14 | 13 | 2 | 14 | 4 | 16 | |
| 8 | 11 | 3 | 10 | 3 | 2 | 3 | 4 | 3 | |
| 9 | 12 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 10 | 13 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 11 | 14 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 12 | 15 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 13 | 16 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 14 | 17 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 15 | 18 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | |
| 16 | 19 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 17 | 20 | 0 | 0 | 0 | 2 | 0 | 2 | 0 | |
| 18 | 21 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| 19 | 22 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| $\sum$ Add (LUTs) | | 1272 | 710 | | 569 | | 172 (1025) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 22 (161) | | |
| $\sum$ LUT | | 6308 | 3651 | | 2895 | | 1186 | | |
| Einfügen von Registerstufen nach jeder dritten Stufe | | | | | | | | | |
| $\sum$ Reg (FFs) | | 191 (1433) | 130 (958) | | 99 (746) | | 56 (400) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 19 (159) | | 30 (171) | | |

**Tabelle 5.13:** Anzahl der Addiererknoten, Registerknoten und Verzögerungsglieder der dreieckigen Templates mit Spitze nach unten ohne und mit Verschiebungen. Die Addiererknoten sind aufgelistet nach deren Stufentiefe $t$ und Bitbreite $b$.

ben als Ganzes übereinanderliegen, werden nur wenige Addiererressourcen benötigt, die allerdings zu einem sehr tiefen Addierergraphen führen. Die dritten Seiten der Dreiecke, überlappen sich nach dem Verschieben schlechter als zuvor, weil sie zusätzlich auseinandergezogen wurden und führen zu einem entsprechend höheren Ressourcenbedarf. Bei den dreieckigen Templates mit einzeln verschobenen Templateelementen konnten alle Seiten optimal zur Überdeckung gebracht werden. Diese können sehr günstig mit gemeinsamen Addierknoten zusammengefasst werden, was zu einer Ressourcenersparnis an LUTs gegenüber den unverschobenen Templates um einen Faktor $5.3$ und bei den reduzierten Templates um einen Faktor $4.6$ führt. Vergleicht man die Anzahl der benötigten Verzögerungsglieder für die dreieckigen Templates mit derjenigen für die kreisförmigen Templates, so ist festzustellen, dass die Verschiebungen der einzelnen Templateelemente

| | | Reduzierte dreieckige Templates mit Spitze nach unten $\tilde{T}_j,\ j = 12, \ldots, 23$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Binäre- | Optimierte Addiererbäume | | | | | | |
| | | original | original | | Verschieben | | Verschieben einzeln | | |
| t | b | t = b | t | b | t | b | t | b | |
| 1 | 4 | 240 | 240 | 240 | 97 | 97 | 23 | 23 | |
| 2 | 5 | 120 | 120 | 120 | 43 | 43 | 7 | 7 | |
| 3 | 6 | 60 | 60 | 60 | 22 | 22 | 3 | 3 | |
| 4 | 7 | 30 | 30 | 30 | 12 | 12 | 2 | 9 | |
| 5 | 8 | 15 | 15 | 15 | 5 | 14 | 3 | 13 | |
| 6 | 9 | 9 | 9 | 9 | 2 | 11 | 3 | 12 | |
| 7 | 10 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 8 | 11 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 9 | 12 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 10 | 13 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 11 | 14 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 12 | 15 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 13 | 16 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | |
| 14 | 17 | 0 | 0 | 0 | 2 | 0 | 2 | 0 | |
| 15 | 18 | 0 | 0 | 0 | 2 | 0 | 2 | 0 | |
| 16 | 19 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| $\sum$ Add (LUTs) | | 474 | 474 | | 199 | | 67 (420) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 11 (89) | | |
| $\sum$ LUT | | 2331 | 2331 | | 1030 | | 509 | | |
| Einfügen von Registerstufen nach jeder dritten Stufe | | | | | | | | | |
| $\sum$ Reg (FFs) | | 75 (546) | 75 (546) | | 56 (399) | | 38 (242) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 15 (99) | | 27 (163) | | |

**Tabelle 5.14:** Anzahl der Addiererknoten, Registerknoten und Verzögerungsglieder der reduzierten dreieckigen Templates mit Spitze nach unten ohne und mit Verschiebungen. Die Addiererknoten sind aufgelistet nach deren Stufentiefe $t$ und Bitbreite $b$.

bei den dreieckigen Templates mit wesentlich weniger Verzögerungsgliedern ausgeglichen werden können.

Durch das Einfügen von Registern nach jeder dritten Stufe, ergibt sich für die Templates mit einzeln verschobenen Templateelementen ein wesentlich geringerer Ressourcenbedarf als für die unverschobenen. Dies liegt daran, weil der für die Templates mit einzeln verschobenen Templateelementen generierte Addierergraph sehr schlank ist.

### Ressourcenverbrauch der kreisförmigen und dreieckigen Templates

Der Ressourcenverbrauch für den Aufbau der binären Addiererbäumen und optimierten Addierergraphen für die in den Abb. 5.22 und 5.36 dargestellten kreisförmigen und dreieckigen Templates $T_j, j = 0, \ldots, 35$ sind in Tab. 5.15 angegeben, der für die reduzierten

Templates $\tilde{T}_j, j = 0, \ldots, 35$ aus den Abb. 5.23 und 5.37 in Tab. 5.16.

| | | Kreisförmige und dreieckige Templates $T_j$, $j = 0, \ldots, 35$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Binäre- | Optimierte Addiererbäume | | | | | | |
| | | original | original | | Verschieben | | Verschieben einzeln | | |
| t | b | t = b | t | b | t | b | t | b | |
| 1 | 4 | 1917 | 1088 | 1088 | 894 | 894 | 214 | 217 | |
| 2 | 5 | 970 | 580 | 580 | 454 | 455 | 149 | 167 | |
| 3 | 6 | 485 | 297 | 297 | 246 | 249 | 108 | 128 | |
| 4 | 7 | 234 | 153 | 153 | 126 | 125 | 69 | 90 | |
| 5 | 8 | 117 | 91 | 105 | 68 | 67 | 53 | 71 | |
| 6 | 9 | 57 | 66 | 73 | 40 | 53 | 38 | 60 | |
| 7 | 10 | 33 | 40 | 39 | 19 | 42 | 34 | 50 | |
| 8 | 11 | 9 | 25 | 9 | 11 | 9 | 30 | 9 | |
| 9 | 12 | 0 | 4 | 0 | 4 | 0 | 25 | 0 | |
| 10 | 13 | 0 | 0 | 0 | 4 | 0 | 16 | 0 | |
| 11 | 14 | 0 | 0 | 0 | 4 | 0 | 13 | 0 | |
| 12 | 15 | 0 | 0 | 0 | 4 | 0 | 10 | 0 | |
| 13 | 16 | 0 | 0 | 0 | 4 | 0 | 7 | 0 | |
| 14 | 17 | 0 | 0 | 0 | 4 | 0 | 7 | 0 | |
| 15 | 18 | 0 | 0 | 0 | 4 | 0 | 7 | 0 | |
| 16 | 19 | 0 | 0 | 0 | 4 | 0 | 6 | 0 | |
| 17 | 20 | 0 | 0 | 0 | 4 | 0 | 4 | 0 | |
| 18 | 21 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| 19 | 22 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| $\sum$ Add | | 3822 | 2344 | | 1894 | | 792 (4808) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 226 (1438) | | |
| $\sum$ LUTs | | 18944 | 12091 | | 9752 | | 6246 | | |
| Einfügen von Registerstufen nach jeder dritten Stufe | | | | | | | | | |
| $\sum$ Reg (FFs) | | 573 (4299) | 440 (3253) | | 349 (2605) | | 274 (2011) | | |
| $\sum$ SRLT (LUTs) | | 0 | 2 (12) | | 37 (309) | | 53 (320) | | |

**Tabelle 5.15:** Anzahl der Addiererknoten, Registerknoten und Verzögerungsglieder der kreisförmigen und dreieckigen Templates ohne und mit Verschiebungen. Die Addiererknoten sind aufgelistet nach deren Stufentiefe $t$ und Bitbreite $b$.

Insgesamt ergibt sich für den Addierergraphen der kreisförmigen und dreieckigen Templates mit einzeln verschobenen Templateelementen eine Ressourcenersparnis um einen Faktor $3.0$ gegenüber den binären Addiererbäumen und für die reduzierten Templates um einen Faktor $2.4$. Der Ressourcenbedarf, der durch das Einfügen von Registern in jeder dritten Registerstufe entsteht, ist für die Templates mit einzeln verschobenen Templateelementen immerhin um einen Faktor $1.8$ geringer. Der FPGA-Ressourcenbedarf für die Addiererbäume der unterschiedlichen Template-Konfigurationen ist zusammenfassend in Diagramm 5.40 für die ursprünglichen und reduzierten Templates illustriert.

| | | Reduzierte kreisförmige und dreieckige Templates $\tilde{T}_j$, $j = 0, \ldots, 35$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Binäre- | Optimierte Addiererbäume | | | | | | |
| | | original | original | | Verschieben | | Verschieben einzeln | | |
| t | b | t = b | t | b | t | b | t | b | |
| 1 | 4 | 697 | 688 | 688 | 333 | 333 | 92 | 92 | |
| 2 | 5 | 346 | 343 | 343 | 160 | 161 | 55 | 59 | |
| 3 | 6 | 173 | 172 | 172 | 85 | 85 | 44 | 53 | |
| 4 | 7 | 87 | 90 | 90 | 48 | 49 | 32 | 53 | |
| 5 | 8 | 42 | 42 | 42 | 27 | 47 | 24 | 48 | |
| 6 | 9 | 25 | 25 | 25 | 15 | 29 | 23 | 33 | |
| 7 | 10 | 0 | 0 | 0 | 4 | 0 | 17 | 0 | |
| 8 | 11 | 0 | 0 | 0 | 4 | 0 | 9 | 0 | |
| 9 | 12 | 0 | 0 | 0 | 4 | 0 | 8 | 0 | |
| 10 | 13 | 0 | 0 | 0 | 4 | 0 | 6 | 0 | |
| 11 | 14 | 0 | 0 | 0 | 4 | 0 | 6 | 0 | |
| 12 | 15 | 0 | 0 | 0 | 4 | 0 | 6 | 0 | |
| 13 | 16 | 0 | 0 | 0 | 4 | 0 | 6 | 0 | |
| 14 | 17 | 0 | 0 | 0 | 4 | 0 | 5 | 0 | |
| 15 | 18 | 0 | 0 | 0 | 4 | 0 | 4 | 0 | |
| 16 | 19 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| $\sum$ Add (LUTs) | | 1370 | 1360 | | 704 | | 338 (2033) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 0 | | 122 (783) | | |
| $\sum$ LUTs | | 6726 | 6690 | | 3627 | | 2816 | | |
| Einfügen von Registerstufen nach jeder dritten Stufe | | | | | | | | | |
| $\sum$ Reg (FFs) | | 226 (1690) | 216 (1568) | | 176 (1262) | | 140 (951) | | |
| $\sum$ SRLT (LUTs) | | 0 | 0 | | 30 (198) | | 32 (148) | | |

**Tabelle 5.16:** Anzahl der Addiererknoten, Registerknoten und Verzögerungsglieder der reduzierten kreisförmigen und dreieckigen Templates ohne und mit Verschiebungen. Die Addiererknoten sind aufgelistet nach deren Stufentiefe $t$ und Bitbreite $b$.

### 5.8.3 Diskussion der Ergebnisse und Ausblick

Der FPGA-Ressourcenbedarf für die SRAs und die binären Addiererbäume bzw. für die oSRAs und optimierten Addierergraphen ist zusammenfassend für die unterschiedlichen Template-Konfigurationen in Tab. 5.17 angegeben und in Diagramm 5.41 illustriert.

Durch das Anwenden der Optimierungsstrategien für den Aufbau der SRAs, siehe Abschn. 5.1, und den Aufbau der optimierten Addiererbäumen, siehe Abschn. 5.3 bis 5.5, konnte der gesamte FPGA-Ressourcenbedarf an FFs und LUTs für die im Rahmen dieser Arbeit verwendeten originalen Templates um einen Faktor $3.6$ gegenüber den nichtoptimierten SRAs bzw. binären Addiererbäumen reduziert werden. Die Auslastung des Virtex-II XC2V3000 FPGAs, der $28.672$ LUTs und $28.672$ FFs besitzt, sinkt für die ursprünglichen kreisförmigen und dreieckigen Templates für die LUTs von $66.1\%$ auf $28.5\%$ und für die FFs von $77\%$ auf $12.8\%$. Die Ressourcen der jeweiligen Virtex-II FPGAs sind in Tab. 1.1 angegeben. Der Ressourcenbedarf an internem BlockRAM ist für FPGAs der Virtex-II Fa-

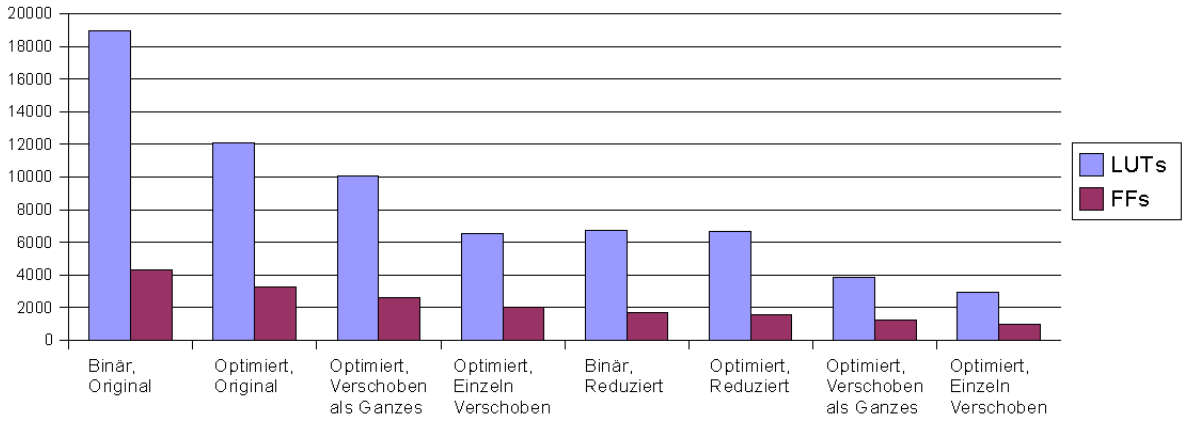



**Abbildung 5.40:** FPGA-Ressourcenbedarf für die binären Addiererbäume bzw. optimierten Addierergraphen für alle originalen (links) und reduzierten (rechts) kreisförmigen und dreieckigen Templates für vier Bit DT-Pixel.

| | Kreisförmige und dreieckige Templates $T_j,\ j = 0, \ldots, 35$ | | | |
|---|---|---|---|---|
| | Binäre- | Optimierte Addiererbäume | | |
| | original | original | Verschieben | Verschieben einzeln |
| $\sum$ LUTs | 18944 | 13155 | 12621 | 8166 |
| $\sum$ FFs | 22027 | 11869 | 9949 | 3667 |
| | Reduzierte kreisförmige und dreieckige Templates $\tilde{T}_j,\ j = 0, \ldots, 35$ | | | |
| $\sum$ LUTs | 6726 | 9646 | 6657 | 4432 |
| $\sum$ FFs | 18930 | 6960 | 3962 | 1653 |

**Tabelle 5.17:** FPGA-Ressourcenbedarf für die SRAs und binären Addiererbäume bzw. oSRAs und optimierten Addierergraphen für alle originalen (oben) und reduzierten (unten) kreisförmigen und dreieckigen Templates für vier Bit DT-Pixel. In jeder dritten Stufe des Addierergraphen wurden Register eingefügt.

milie ab dem Virtex-II XC2V500 aufwärts ausreichend. Für die SRAs und oSRAs werden sowohl für die ursprünglichen als auch reduzierten kreisförmigen und dreieckigen Templates jeweils 28 interne RAM-Blöcke eines Virtex-II FPGAs benötigt. Der Virtex-II 500 FPGA besitzt 32 von diesen.

Für die folgenden Abschätzungen wird von einer realistischen Auslastung des Virtex-II FPGAs von jeweils 100% an LUTs und FFs und einger gesamten Auslastung von diesen beiden zusammen von 60% ausgegangen. Für die SRAs und die binären Addiererbäume für die ursprünglichen kreisförmigen und dreieckigen Templates ist der Virtex-II 3000 FPGA zu $71.5\%$ ausgelastet und für die reduzierten Templates zu $44.7\%$. Die Ressourcen des Virtex-II 3000 FPGAs sind nur für die reduzierten kreisförmigen und dreieckigen Templates ausreichend. Für das Template-Matching aller ursprünglichen Templates muss auf einen Virtex-II XC2V4000 FPGA zurückgegriffen werden. Werden ausschließlich die Ressourcen für die oSRAs und den optimierten Addierergraphen für die Templates mit einzeln verschobenen Templateelementen betrachtet, so sind die Ressourcen des Virtex-II XC2V1000 FPGAs mit einer gesamten Auslastung von $57.7\%$ für die ursprünglichen Tem-

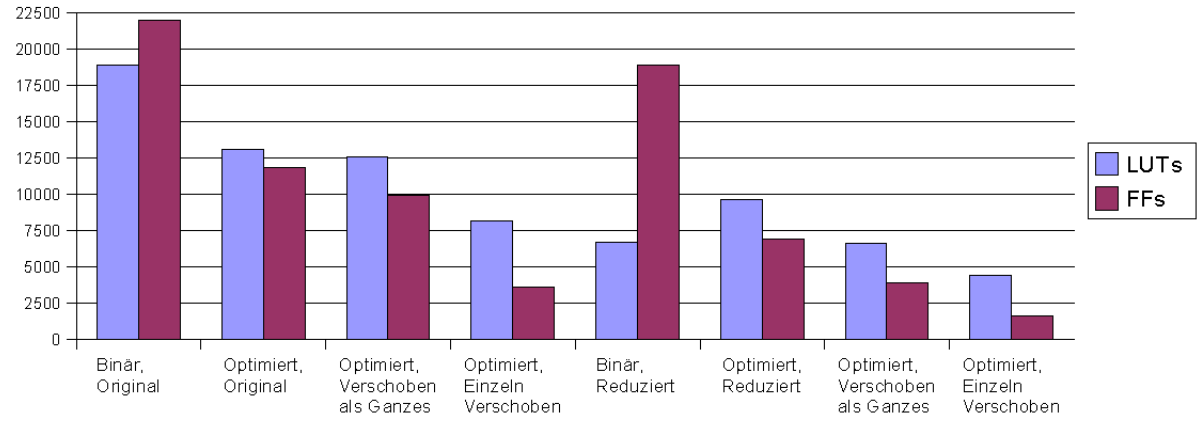



**Abbildung 5.41:** FPGA-Ressourcenbedarf für die SRAs und binären Addiererbäume bzw. oSRAs und optimierten Addierergraphen für alle originalen (links) und reduzierten (rechts) kreisförmigen und dreieckigen Templates für vier Bit DT-Pixel.

plates ausreichend und die des Vitrex-II 500 mit einer gesamten Auslastung von $42.6\%$ für die reduzierten Templates. Durch das Anwenden der Optimierungsstrategien für die SRAs und die Addiererbäume lassen sich diese für die ursprünglichen kreisförmigen und dreieckigen Templates anstelle auf einem Virtex-II 4000 FPGA auf einem Virtex-II 1000 abbilden und für die reduzierten Templates anstelle auf einem Virtex-II 3000 auf einem Virtex-II 500. Für die reduzierten Templates kommen sogar die mittleren FPGAs der preisgünstigen Spartan-3 Familie [33] in Frage.

Werden die in Abschn. 4.3 beschriebenen `resort-` und Schwellwertmodule und das Steuerungsmodul hinzugenommen, so sind die Ressourcen des Virtex-II 1500 FPGAs für die ursprünglichen Templates und die Ressourcen des Virtex-II 1000 oder eines größeren Spartan-3 FPGAs für die reduzierten Templates ausreichend. Dies trifft ebenfalls zu falls die Module zur Berechnung der DT-Bilder hinzugenommen werden.

In Zukunft ist die Integration der vier Optimierungsstrategien für die Implementierung des FPGA-Designs für das parallele Template-Matching durchzuführen. Offen bleiben die Auswirkungen auf das *Routing* auf dem FPGA. Für die oSRAs sind hierbei keine Schwierigkeiten zu erwarten, weil sich die Verbindungstopologie nicht ändert und nach wie vor jedes Register bzw. Verzögerungsglied mit genau einem nachfolgenden Register bzw. Verzögerungsglied verbunden wird und sich damit die Anzahl der Signalleitungen auf dem FPGA nicht erhöht. Man kann sogar davon ausgehen, dass sich die Anforderungen an die *Routing*-Ressourcen auf dem FPGA verringern, weil jeweils ein Register (FF) und SRAs (LUT) besser in ein Slice bzw. CLB des FPGAs gepackt werden können und diese Verbindungen intern im Slice bzw. über die CLB-*Feedback*-Pfade realisiert werden und nicht über das externe *Routing*-Netzwerk. Für die max. Frequenz, mit der das ausführbare Design getaktet werden kann, ist daher keine Verschlechterung zu erwarten. Im Gegenteil, man kann sogar von einer leicht erhöhten max. Taktfrequenz ausgehen.

Bei den binären Addierergraphen sind jedoch im Verhältnis zu der Anzahl der Addierer höhere Anforderungen an das *Routing*-Netzwerk zu erwarten. Dies liegt daran, weil ein Addierer mehrere nachfolgende Knoten besitzten kann und daher das Ausgangssignal an

mehrere Nachfolger geleitet werden muss. Bei einem *fanout* der Addierer bzw. Verzögerungsglieder im Mittel von $1.3$ und einem max. *fanout* von $8$ ist ein höherer *Routing*-Ressourcenbedarf zu erwarten. Dieser wird sich negativ auf die Auslastung der *Routing*-Ressourcen des FPGAs bzw. die max. Frequenz, mit der das Design getaktet werden kann, auswirken. Die genauen Auswirkungen auf die max. Auslastung des FPGAs und die max. Taktfrequenz, die voraussichtlich klein sein werden, sind allerdings schwer abzuschätzen. Diese sind auch davon abhängig, ob kurze Verbindungen in den Slices bzw. CLBs oder lange Verbindungen auf dem FPGA realisiert werden müssen. Um lange Signalwege auf dem FPGA zu vermeiden, kann es sinnvoll sein, Änderungen am Algorithmus zum Aufbau der optimierten Addierergraphen bei der Gewinn-Analyse durchzuführen, siehe Abschn. 5.3.2. Bei der Gewinn-Analyse könnte ein zusätzliches Kriterium eingeführt werden, welches die Nachbarschaftsbeziehungen der Addierer- bzw. Verzögerungsknoten enthält.

Für weitere Templateklassen wie Rauten und Rechtecke, die im Rahmen dieser Arbeit nicht näher untersucht wurden, kann man davon ausgehen, dass der Ressourcenbedarf für das Matching dieser Templates in derselben Größenordnung wie derjenige für die Dreiecke liegt, vorausgesetzt die Anzahl und Größe der rautenförmigen und rechteckigen Templates sind mit denjenigen der dreieckigen Templates vergleichbar. Dies liegt daran, weil die rautenförmigen und rechteckigen Templates ebenfalls sehr gute Überdeckungseigenschaften besitzen, die in Abschn. 5.7.2 beschrieben wurden.

Kapitel 6

# Ergebnisse und Ausblick

Ziel dieser Arbeit war die Entwicklung einer FPGA-Implementierung für den von D. Gavrila gegebenen Algorithmus zur Erkennung von Verkehrszeichen. Es konnte gezeigt werden, dass das Template-Matching mit distanztransformierten Bildern für 36 Templates und Bilder der Größe $512 \times 512$ in Echtzeit mit einer Bildwiederholrate von $30$ Hz auf einem FPGA-basierten hybriden Bildverarbeitungssystem durchführbar ist.

Typischerweise sind Algorithmen zur Verkehrszeichenerkennung der menschlichen Wahrnehmung ähnlich und bestehen aus den Teilschritten *Sehen*, *Erkennen* und *Entscheiden*. Der kantenbasierte Algorithmus von D. Gavrila mit einem Template-Matching auf distanztransformierten orientierten Kantenbildern zur Bestimmung eines ROIs im Bild und einer anschließenden Klassifikation mit einem RBF-Netzwerk ist hierfür sehr robust und weist hohe Erkennungsraten auf. Die RBF-Klassifikation ist für wenige Matching-Ergebnisse nicht rechenzeitkritisch und effizient auf einem Standard-Prozessor ausführbar. Es zeigte sich, dass die Erzeugung der acht DT-Bilder (einschließlich vorausgehender Kantengenerierung mittels Sobel-Filter, Eliminierung von Störpixel im Kantenbild und Zuweisung der Kantenpixel in einen der acht diskreten Richtungsbereiche) und besonders die Berechnung des Template-Matchings für die eingesetzten kreisförmigen und dreieckigen Templates sehr rechenintensiv und für CPUs ungeeignet ist.

Untersuchungen ergaben, dass FPGAs für die Berechnung der DT-Bilder und des Matchings für die verwendeten 36 Templates geeignet sind, weil die Verarbeitungseinheiten der geringen Bitbreite der Rechenoperationen optimal angepasst und die Daten diesen optimal zugeführt werden können. Als Basisplattform wurde ein hybrides Bildverarbeitungssystem bestehend aus einem Host-PC mit Standard-Prozessor und dem an der Universität Mannheim entwickelten FPGA-Koprozessor MPRACE mit einem modernen Virtex-II XC2V3000 eingesetzt, welches sich als geeignet zur Umsetzung des Algorithmus zur Erkennung von Verkehrszeichen erwies. Die FPGA-Implementierung wurde mit der an der Universität Mannheim entwickelten Hardwarebeschreibungssprache CHDL durchgeführt.

Die Umsetzung des gesamten Algorithmus (Erzeugung der acht DT-Bilder und der Berechnung des Matchings für die in dieser Arbeit verwendeten 36 Templates) einschließlich Bildaufnahme auf Hardware wurde für einen einzigen FPGA-Baustein (Virtex-II XC2V3000) entworfen. Ein vorhandenes CHDL-Modul zur Bildaufnahme (*Sehen*) wurde er-

weitert, so dass neben den Bildern in Originalgröße die um einen Faktor vier reduzierten und geglätteten Bilder in einen anderen Speicherbereich des SRAM des MPRACE-Boards geschrieben werden konnten. Hierfür wurden etwas mehr als $1\%$ der Ressourcen des verwendeten FPGAs benötigt. Auf den reduzierten Bildern wurde die Suche nach größeren Mustern (abwechselnd mit kleineren Mustern in den Originalbildern) durchgeführt.

Eine Analyse für die FPGA-Implementierung zur Erzeugung der DT-Bilder (*Erkennen*) ergab, dass alle hierfür benötigten Module parallel und nach dem Pipeline-Prinzip aufgebaut werden konnten. Das zur Kantenerzeugung eingesetzte Sobel-Filter konnte effizient implementiert werden, weil dieses auf kleinen Nachbarschaften arbeitet und dessen Festkomma-Multiplikationen durch einfache Additionen bzw. Subtraktionen realisiert werden konnten. Für den Sobel und die anschließende Binarisierung wurden etwas mehr als $1\%$ der Ressourcen des eingesetzten FPGAs benötigt. Der in Software sequentiell und rekursiv implementierte *Cleaning*-Operator, der max. drei zusammenhängende Kantenpixel (Störpixel) eliminiert, konnte so umgestaltet werden, dass er ebenfalls auf kleinen Nachbarschaften arbeitet und nach dem Pipeline-Prinzip aufgebaut und in die gesamte Pipeline eingefügt werden konnte. Hierfür war ein Ressourcenbedarf von etwas mehr als $1\%$ für den verwendeten FPGA notwendig. Als zusätzliche Merkmale werden die Orientierungen bzw. Richtungen der Kanten benutzt. Die in Software über eine große *Lookup*-Tabelle realisierte Zuweisung der Kantenpixel in die acht diskreten Richtungsbereiche konnte durch einfache Vergleichsoperationen realisiert und mit einem FPGA-Ressourcenbedarf von nahezu $1\%$ implementiert werden. Für die Berechnung der den acht diskretisierten Richtungsbereichen entsprechenden DT-Bildern wurde die *chamfer*-2-3 Metrik eingesetzt, wobei deren max. Werte auf vier Bit abgeschnitten wurden. Für die Berechnung der DT wurde die sequentielle Methode, die in Vorwärts- und Rückwärts-Richtung durchzuführen ist, ausgewählt. Eine Analyse ergab, dass für die doppelt so schnell auszuführende parallele Methode ein nahezu 16-facher FPGA-Ressourcenbedarf benötigt worden wäre. Die acht sequentiellen DT-Module, die sowohl für die Berechnung der Distanztransformation in Vorwärts- als auch Rückwärts-Richtung eingesetzt wurden, konnten bitgenau an die erforderliche Rechengenauigkeit von fünf Bit angepasst und parallelisiert werden mit einem FPGA-Ressourcenbedarf (für den XC2V3000) von insgesamt $4\%$. An dieser Stelle sind die Auswirkungen von DT-Metriken mit geringer Genauigkeit und kleinerem Wertebereich für das Template-Matching auf distanztransformierten Bildern genauer zu untersuchen. Außerdem mussten für eine schnelle Ausführung des nachfolgenden Template-Matchings jeweils acht benachbarte DT-Pixel in einem Datenwort gespeichert werden. Für die hierfür benötigten acht Module ergab sich insgesamt ein Ressourcenbedarf für den verwendeten FPGA von ca. $2\%$.

Alle Module zur Erzeugung der DT-Bilder konnten derart implementiert werden, dass ihnen pro Takt genau ein Pixel bzw. Zwischenergebniss zugeführt werden kann und pro Takt genau ein Ergebnispixel mit einer festen Latenzzeit ausgegeben wird. Die Module konnten zu zwei Pipelines, einer für die Vorwärts- und einer für die Rückwärts-Richtung zusammengefasst werden. Insgesamt wurden zwei unabhängige SRAM-Bänke des MPRACE-Boards benutzt, weil die Zwischenergebnisse nach Ausführung der ersten Pipeline in eine weitere SRAM-Bank auf dem MPRACE-Board geschrieben wurden. Für alle Module konnte eine hohe Parametrisierbarkeit bezüglich der Bitbreite und Bildgröße erreicht werden. Die datenverarbeitenden Module und das Steuerungsmodul wurden derart aufgebaut, dass während der Ausführung des Designs die Bildgröße verändert

werden konnte. Daher konnten die DT-Bilder abwechselnd für Bilder mit Originalgröße und den reduzierten Bildern berechnet werden. Für beide Pipelines wurde für Bilder der Größe $512 \times 512$ eine Rechenzeit von $8.1$ ms benötigt, mit einer Bildwiederholrate von $122$ Hz. Für die zusätzliche Berechnung der DT-Bilder für die reduzierten Bilder wurden insgesamt $10.2$ ms benötigt mit einer Bildwiederholrate von $98$ Hz. Für die Erzeugung der acht DT-Bilder sind die Ressourcen des Virtex-II XC2V3000 FPGAs zu $12\%$ ausgelastet.

Für das rechenzeitaufwändige Template-Matching (*Entscheiden*) wurden mehrere FPGA-Implementierungsstrategien untersucht. Für ein hierarchisches Template-Matching wurde eine sequentielle Implementierungsstrategie mit zeilenweiser Parallelisierung beschrieben, die jedoch nicht implementiert wurde, weil beim hierarchischen Matching die Ausführungszeiten stark vom Bildinhalt abhängig sind.

Als Hauptbestandteil dieser Arbeit wurde ein paralleler Ansatz für das FPGA-basierte Template-Matching implementiert, mit welchem das Matching in der geforderten Echtzeitbedingung ausgeführt werden konnte. In einer Matrix von Registern (SRAs), die abhängig von der Form der Templates aufgebaut wird, wurden die für die Templates relevanten Daten gespeichert. Auf die Register wurde von binären Addiererbäumen, die das Herz der Verarbeitung darstellen und die den Templateelementen entsprechenden DT-Pixel aufsummieren, zugegriffen. Für jedes Template wurde zunächst ein eigener binärer Addiererbaum eingesetzt. Alle Module für das Template-Matching konnten zu einer großen Pipeline zusammengefasst werden. Ein Steuerungsmodul konnte derart konzipiert werden, dass den acht SRAs jeweils in jedem Takt genau ein DT-Pixel zugeführt werden konnte und die Verarbeitung in einem Durchlauf nicht unterbrochen werden musste. Für die Ausführung des parallelen Template-Matchings auf dem FPGA war genau eine externe SRAM-Bank des MPRACE-Boards notwendig, aus welcher die DT-Pixel ausgelesen und den SRAs zugeführt wurden. Die Matching-Ergebnisse wurden im internen Block-RAM des FPGAs gespeichert.

Alle datenverarbeitende Module und das Steuerungsmodul für das Template-Matching konnten derart implementiert werden, dass sie bezüglich der Bitbreite der Daten und der Bildgröße parametrisierbar sind. Zusätzlich sind während der Ausführung des Designs alle Module bezüglich der Bildgröße veränderbar. Die datenverarbeitenden Module der Pipeline und die Adressgenerierung beim Steuerungsmodul konnten abhängig von der Form der Templates generiert werden. Das parallele Matching konnte für $24$ Templates mit Bilder der Größe $512 \times 512$ in $4.1$ ms auf dem FPGA-Koprozessor MPRACE ausgeführt werden. Für die Berechnung der DT-Bilder und des Matchings für $36$ Templates wurden insgesamt $12.4$ ms benötigt mit einer Bildwiederholrate von $80$ Hz. Wurden zusätzlich die reduzierten Bilder berechnet, so konnte dies in $15.6$ ms mit $64$ Hz durchgeführt werden. Der parallele Ansatz für das Template-Matching ist jedoch mit einem sehr hohen FPGA-Ressourcenbedarf verbunden. Für die acht SRAs für die in dieser Arbeit verwendeten 36 Templates werden $47\%$ und für deren Addiererbäume $74\%$ der Ressourcen des Virtex-II XC2V3000 FPGAs benötigt. Insgesamt konnten nur 24 Templates bei einer Auslastung von $100\%$ auf dem XC2V3000 des MPRACE-Boards berechnet werden.

Im letzten Teil dieser Arbeit wurden zur Reduzierung des FPGA-Ressourcenbedarfs für die parallele FPGA-Implementierung des Template-Matchings vier neue Optimierungsstrategien entwickelt. Es wurde untersucht, wie die Register in den SRAs, auf die nicht von den Addiererbäumen zugegriffen wird, durch ressourcensparende Verzögerungsglie-

der (SRLTs) zu ersetzten sind. Als weitere Strategie wurde ein iterativer Algorithmus zum Aufbau von optimierten Addiererbäumen zunächst für unverschobene Templates entwickelt, der für Templates mit verschobenen Templateelementen erweitert wurde. Grundlegend für die verschiedenen Varianten des Algorithmus war, dass Teilsummen für sich überdeckende Templateelemente verschiedener Templates gemeinsam ausgeführt wurden. Zunächst wurde für den in jedem Iterationsschritt einzufügenden Addiererknoten bestimmt, für welche beiden Registerknoten der SRAs bzw. bereits dem Addierergraphen hinzugefügte Addiererknoten die meisten gemeinsamen Partialsummen zu berechnen sind. Als weiteres Kriterium hierfür wurde die *Tiefenminimierungs*-Regel eingeführt, welche einen flachen Addierergraphen zur Folge hat. Der Algorithmus wurde für als Ganzes verschobene Templates und für Templates mit einzeln verschobenen Templateelementen erweitert. Für den letzteren Fall waren die einzelnen Verschiebungen durch Verzögerungsglieder auszugleichen. In jedem Iterationsschritt ist beim Aufbau der optimierten Addiererbäume zunächst zu entscheiden, ob ein Addierer- oder ein Verzögerungsknoten in den Addierergraphen eingefügt wird. Hierfür wurde einerseits eine Basis-Methode und andererseits die sog. Methode mit *virtuellen* Verzögerungsgliedern entwickelt. Bei letzterer wurden für alle möglichen einzufügenden Verzögerungsknoten deren Längen variiert und der sich daraus ergebende virtuelle Gewinn ermittelt. Als weitere Strategie wurde erlaubt, die Anzahl der Templateelemente der Templates zu reduzieren und mit den reduzierten Templates das Matching durchzuführen. Dies erwies sich als gerechtfertigt, weil das Matching auf DT-Bildern mit Richtungsinformation und nicht auf binären Kantenbildern durchgeführt wird. Für die in dieser Arbeit eingesetzten Templates wurde die Anzahl der Templateelemente um einen Faktor $2.6$ reduziert. Im Hinblick auf ein robustes Template-Matching sind weiterführende Untersuchungen durchzuführen, inwieweit sich die Anzahl der Templateelemente in Abhängigkeit der eingesetzten DT-Metrik reduzieren lässt. Zur weiteren Reduzierung des FPGA-Ressourcenbedarfs wurde das Verschieben von Templates als Ganzes oder deren einzelnen Templateelementen vorgeschlagen, mit dem Ziel, möglichst viele Überdeckungen zwischen den Templateelementen unterschiedlicher Templates zu erzielen. Die Templateelemente der dreieckigen Templates konnten bei Verschiebungen einzelner Templateelemente optimal zur Überdeckung gebracht werden, die der kreisförmigen Templates mit unterschiedlichen lokalen Krümmungen weniger gut. Bei letzteren wäre der Einsatz eines systematischen Optimierungsansatzes, der als Parameter die Verschiebungen der Templateelemente enthält, vorteilhaft. Die aus den Verschiebungen der ursprünglichen und reduzierten Templates resultierenden oSRAs und optimierte Addiererbäume sind in Zukunft in das Design für das parallele Template-Matching zu integrieren. Für die optimierten Addiererbäume ist zu untersuchen, ob sich negative Auswirkungen auf das *Routing* auf dem FPGA ergeben und der Algorithmus zum Aufbau der optimierten Addiererbäume ggf. anzupassen ist.

Für die ursprünglichen und reduzierten Templates wurden die Auswirkungen aller vier Optimiertungsstrategien auf den FPGA-Ressourcenverbrauch für die optimierten SRAs und die optimierten Addiererbäumen gegenüber den ohne Optimierungen erzeugten SRAs und binären Addiererbäumen untersucht. Für die oSRAs und optimierten Addiererbäume konnte der FPGA-Ressourcenbedarf für die 36 ursprünglichen (reduzierten) Templates um einen Faktor $1.6\,(1.5)$, für die als Ganzes verschobenen Templates um einen Faktor $1.8\,(2.4)$ und für die Templates mit einzeln verschobenen Templateelementen um einen Faktor $3.5\,(4.2)$ reduziert werden. Vergleicht man den FPGA-Ressourcenbedarf für

die originalen Templates (SRAs und binäre Addiererbäume) mit den reduzierten Templates (oSRAs und optimierte Addiererbäume), so verringert sich dieser insgesamt um einen Faktor $6.7$. Es ist zu erwarten, dass die Berechnungen der DT-Bilder und des Matchings für 36 Templates anstelle auf einem Virtex-II XC2V4000 FPGA auf einem Virtex-II XC2V1500 FPGA und für die reduzierten Templates auf einem Virtex-II XC2V1000 FPGA bzw. größeren FPGA der neuen Spartan-3 Familie ausgeführt werden können.

Durch die hier vorgelegte Arbeit konnte gezeigt werden, dass die Berechnung von acht DT-Bildern (mit der *chamfer*-2-3 Metrik) und des Template-Matchings mit 36 Templates für Bilder der Größe $512 \times 512$ für einen robusten Algorithmus zur Verkehrszeichenerkennung in Echtzeit auf einer FPGA-basierten Hardware durchführbar ist. Das parallele Template-Matching ist für das FPGA jedoch sehr ressourcenaufwändig. Ein FPGA, auf den sich das gesamte Design abbilden lässt, ist zurzeit sehr kostspielig und eine Integration von FPGA-basierter Hardware in das Fahrzeug zur Verkehrszeichenerkennung scheint daher in absehbarer Zeit nicht möglich. Jedoch konnte durch die Entwicklung und Implementierung der vier Optimierungsstrategien für das parallele Template-Matching der FPGA-Ressourcenbedarf wesentlich reduziert werden. Eines bleibt jedoch offen. Die aus den Optimierungen resultierenden optimierten SRAs und optimierten Addiererbäume sind in die FPGA-Implementierung für das parallele Template-Matching zu integrieren und möglicherweise negative Auswirkungen auf das *Routing* auf dem FPGA zu untersuchen. Dann wird die Ausführung des Matchings auf den preiswerten FPGAs der neuen Spartan-3 Familie, die bereits teilweise auf dem Markt erhältlich sind, möglich sein. Mit dem parallelen Ansatz für das Template-Matching einschließlich Optimierungen lassen sich aller Voraussicht nach weitere Verkehrszeichenklassen wie Rauten, Rechtecke oder Sechsecke auf einem größeren FPGA der Spartan-3 Familie abbilden. Einer erfolgreichen Integration von FPGA-basierter Hardware im Fahrzeug zur Verkehrszeichenerkennung steht dann nichts mehr im Wege.

# Literaturverzeichnis


[1] D.M. Gavrila, Multi-feature Hierarchical Template Matching Using Distance Transforms. In International Conference on Pattern Recognition, pp. 439-444, Brisbane, 1998.

[2] D.M. Gavrila, Traffic Sign Recognition Revisited, Proc. of the 21st DAGM Symposium für Mustererkennung, pp. 86-93, Springer-Verlag, Bonn, 1999.

[3] D.M. Gavrila, V. Philomin, Real-Time Object Detection for "Smart" Vehicles. In Proc. Int. Conf. on Computer Vision, pp. 87-93, 1999.

[4] U. Franke, D.M. Gavrila, S. Görzig, F. Lindner, F. Paetzhold and C. Wöhler, Autonomous Driving approaches Downtown. IEEE Intelligent Systems, vol. 13, nr. 6, pp. 40-48, 1998.

[5] D.M. Gavrila and J. Giebel, Virtual Sample Generation for Template-based Shape Matching. Proc. of IEEE Conference on Computer Vision and Pattern Recognition, vol. I, pp. 676-681, Kauai, U.S.A., 2001.

[6] B. Jähne, Digitale Bildverarbeitung. Springer-Verlag, 4. Auflage, 1997.

[7] B. Jähne, H. Hausseker, P. Geißler, Handbook on Computer Vision and Applications. Academic Press, 1999.

[8] T. Lehmann, W. Oberschelp, E. Pelikan, R. Repges, Bildverarbeitung für die Medizin. Springer-Verlag, 1997.

[9] R. Klette, P. Zamperoni, Handbook of Image Processing Operators. Wiley, 1996.

[10] P. Soille, Morphologische Bildverarbeitung. Springer-Verlag, 1998.

[11] J. Canny, A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 8, No. 6, 1986.

[12] G.K. Lang, P. Seitz, Robust classification of arbitrary objekt classes based on hierarchical spatial feature matching. Machine Vision and Applications, Vol. 10, pp. 123-135 1997.

[13] P.V.C. Hough, A Method and Means for Recognizing Complex Patterns. U.S. Patent 3,069,654, 1962.

[14] J. Schürmann, Pattern Classification. John Wiley & Sons, 1996.

[15] K. Fukunaga, Introduction to Statistical Pattern Recognition. Academic Press, New York, 2nd edition, 1990.

[16] B.D. Ripley, Pattern Recognition and Neural Networks. Cambridge University Press, 1996.

[17] C. M. Bishop, Neural Networks for Pattern Recognition. Oxford University Press, 1995.

[18] A.K. Jain and Robert P.W. Duin and J. Mao, Statistical Pattern Recognition: A Review. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 22, No. 1, pp. 4-37, 2000.

[19] F. Rosenblatt, The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain. Psych Rev 65, pp. 386-408, 1958.

[20] H. Ritter, T. Martinetz, K. Schulten, Neuronale Netze. Addison-Wesley, 2. Auflage, 1991.

[21] J. Anderson, A. Pellinoisz and E. Rosenfeld, Neurocomputing 2: Directions for Research. Cambridge Mass, MIT Press, 1990.

[22] C.G. Looney, Pattern Recognition using Neural Networks, Oxford Univerity Press, 1997.

[23] N. Cristianini and J. Shawe-Taylor, An Introduction to Support Vector Machines (and other kernel-based learning methods). Cambridge University Press, 2000.

[24] G. Piccioli, E. De Micheli and M. Campani, A robust method for road sign detection and recognition, Image and Vision Computing, 14:209-223, 1996.

[25] P. Paclik, J. Novovicova, Road Sign Classification without Color Information. Proceedings of 6th Conference of Advaced School of Imaging and Computing, ASCI, Lommel, Belgium, 2000.

[26] P. Paclik, J. Novovicova, P. Somol, P. Pudil, Road Sign Classification using Laplace Kernel Classifier. 11th Scandinavian Conference on Image Analysis, SCIA'99, Kangerlussuaq, Greenland, pp. 275-282, 1999.

[27] M. Betke and N. Makris, Fast Object Recognition in Noisy Images Using Simulated Annealing. In International Conference on Computer Vision, pp. 523-530, 1995.

[28] Y. Aoyagi and T. Asakura, A study on traffic sign recognition in scene image using genetic algorithms and neural networks. In Proc. IEEE Conf. on Industrial Electronics, Control and Instrumentation, pp. 1838-1843, Taipei, Taiwan, 1996.

[29] J. Mirua, T. Kanda, Y. Shirai, An Active System for Real-Time Traffic Recognition. Proc. 2000 IEEE Int. Conf. on Intelligent Transportation Systems, pp. 52-57, Dearborn, MI, 2000.

[30] T. Schnitger and U. Handmann, Fusion von Bildanalyseverfahren mittels einer neuronalen Kopplungsstruktur. Internal Report IRINI 98-01, Institut für Neuroinformatik, Ruhr-Universität Bochum, D-44780 Bochum, Germany, Apr. 1998.

[31] L. Priese, R. Lakmann, and V. Rehmann. Automatische Verkehrszeichenerkennung mittels Echtzeit-Farbbildanalyse. Automatisierungstechnik, 45(12), 1997.

[32] A. de la Escalera, L. Moreno, M. Salichs, and J. Armingol. Road traffic sign detection and classification. IEEE Transactions on Industrial Electronics, 44(6), 1997.

[33] http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp.

[34] XC4000E and XC4000X Series Field Programmable Gate Arrays, Product Specification. http://www.xilinx.com/partinfo/databook.htm#xc4000. May 14 (Version 1.6), 1999.

[35] Virtex-II Platform FPGA Handbook, http://www.xilinx.com. December 3 (Version 1.3), 2001.

[36] O. Brosch, P. Dillinger, K. Kornmesser, A. Kugel, R. Männer, M. Sessler, H. Simmler, H. Singpiel, S. Rühl, R. Lay and K.-H. Noffz, L. Levinson, MicroEnable - A Reconfigurable FPGA Coprocessor. 4th Worksh. on Electronics for LHC Experiments, pp. 402-406, Rome, Italy, 1998.

[37] A. Kugel, RACE-1 - A PCI-64 based High Performance FPGA Co-Processor. http://www-li5.ti.uni-mannheim.de/fpga/race/, 2003.

[38] http://www.silicon-software.de/subboards.html.

[39] TM-1040 Progressive Scan High Resoultion Shutter Camera. http://www.pulnix.com/Imaging/c-1040.html, Juni, 1999.

[40] W. Erhard. Rechnerarchitektur. Einführung in die Grundlagen. Teubner, Stuttgart, 1995.

[41] S.M. Mueller, W.J. Paul, Computer Architecture. Springer-Verlag, 2000.

[42] H. Scharr, Digitale Bildverarbeitung und Papier: Texturanalyse mittels Pyramiden und Grauwertstatistiken am Beispiel der Papierformation. Diplomarbeit, Fakultät für Physik und Astronomie, Universität Heidelberg, 1996.

[43] H. Scharr, Optimal Operators in Digital Image Processing. Dissertation, Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg, 2000.

[44] J. Weickert, B.M. ter Haar Romeny, M.A. Viergever, Efficient and reliable schemes for nonlinear diffusion filtering. IEEE Trans. Image Proc., 1998.

[45] J. Weickert, Anisotropic diffusion in image processing. ECMI Series, Teubner, Stuttgart, 1998.

[46] P.L. Rosin and G.A.W. West, Salience distance transforms. Graphical Models and Image Processing, vol. 57, 6, pp. 483-521, 1995.

[47] P. Seitz and G.K. Lang and B. Gilliard and J. C. Pandazis, The Robust Recognition of Traffic Signs from a Moving Car. Mustererkennung 1991: 13. DAGM-Symposium, B. Radig, Springer-Verlag, Berlin, Heidelberg, 287-294, 1991.

[48] G. Borgefors, An improved version of the chamfer matching algorithm. 7th International Conference on Pattern Recognition, Vol. 2, pp. 1175-1177, Montreal, Canada, 1979.

[49] G. Borgefors, Distance Transformations in Digital Images. Computer Vision, Graphics, and Image Processing 34, pp. 344-371, 1986.

[50] G. Borgefors, Hierarchical chamfer matching: A parametric edge matching algorithm. IEEE Trans. Pattern Analysis and Machine Intell., 10(6), 849-865, 1988.

[51] O. Forster, Analysis 1. Differential- und Integralrechnung einer Veränderlichen, Vieweg-Verlag, 2001.

[52] H. Barrow, J. Tenenbaum, R. Bolles and H. Wolf, Parametric correspondence and chamfer matching: Two new techniques for image matching. In International Joint Conference on Arificial Intelligence (IJCAI), pp. 659–663, 1977.

[53] M.A. Butt and P. Maragos, Optimum design of chamfer distance transforms, IEEE Transactions on Image Processing, vol. 7(10) pp. 1477-1484, 1998.

[54] J.H. Takala and J. O.Viitanen, Distance Transform Algorithm for Bit-Serial SIMD Architectures. Computer Vision and Image Understanding, Vol. 74(2), pp. 150-161 1999.

[55] D.P. Huttenlocher, G.A. Klanderman and W.J. Rucklidge, Comparing images using the haussdorff distances. IEEE trans. on Pattern Analysis and Machine Intelligence, 9(15), pp. 850-863, 1993.

[56] C. Olson, A probabilistic formulation for Hausdorff matching, In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 150-156, 1998.

[57] C.F. Olson and D.P. Huttenlocher, Automatic target recognition by matching oriented edge pixels. IEEE Transactions on Image Processing, 6(1), pp. 103-113, 1997.

[58] W.J. Rucklidge, Locating objects using the Hausdorff distance. In Proceedings of the 5th International Conference on Computer Vision, pp. 457-464, Cambridge, USA, 1995.

[59] T. Trenschel, Blob-Analyse - Bestimmung von Formparametern beliebig geformter Objekte auf FPGAs in Echtzeit. Diplomarbeit, Universität Heidelberg, 2001.

[60] P. Dillinger, Einsatz von FPGA Prozessoren als Datenreduktionshardware in den Bildverarbeitung. Dissertation, Universität Mannheim, 2003.

[61] P. Dillinger, S. Hezel, H. Lauer, FPGAs zur Echtzeit-Bildverarbeitung mit 1D/2D-FIR-Filteroperationen. Image Processing and Machine Vision, VDI Berichte 1572, Düsseldorf, pp. 213-218, 2000.

[62] S. Hezel, R. Männer, Schnelle Berechnung von 2-D FIR-Filteroperationen mittels FPGA-Koprozessor mEnable. in W. Förstner e.a. (Hrsg.), Mustererkennung 1999, 21. DAGM Symposium, Springer-Verlag, pp. 250-257, 1999.

[63] S. Hezel, A. Kugel, R. Männer, D. Gavrila, FPGA-Based Template Matching Using Distance Transforms. 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), pp. 89-97, 2002.

[64] S. Ruehl, P. Dillinger, S. Hezel, R. Männer, Generative Developement System for FPGA Processors with Active Components. 11th International Conference on Field-Programmable Logic and Applications (FPL), pp. 513-522, 2001.

[65] N. Ratha, A. Jain and D. Rover, Convolution on Splash 2. IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, pp. 204-213, Los Alamitos, CA, 1995.

[66] T.T. Do, H. Kropp, C. Reuter, P. Pirsch, A Flexible Implementation of High-Performance FIR Filters on Xilinx FPGAs. Lecture Notes in Computer Science: Field Programmable Logic and Applications (8th International Workshop FPL'98), pp. 441-445, September 1998.

[67] G.R. Goslin, A Guide to Field Programmable Gate Arrays (FPGAs) for Application-Specific Digital Signal Processing Performance. www.xilinx.com/appnotes/dspguide.pdf. 1995.

[68] T. Ikenaga, T. Ogura, Real-Time Morphology Processing Using Highly parallel 2-D Cellular Automata CAM$^2$. IEEE Transactions on Image Processing, Vol. 9, No. 12, 2000.

[69] D. Benyamin, J. Villasenor and W. Mangione-Smith, Reconfigurable Computing for Electro-Optical Sensor Processing. Final Report 1996-1997 for MICRO Project 96-186, 1997.

[70] K. Kornmesser, A. Kugel, R. Männer, The FPGA Development System CHDL. Proceedings of the 2001 Symposium on FPGAs for Custom Computing Machines, 2001.

[71] K. Kornmesser, Das FPGA-Entwicklungssystem CHDL. Eine vollständige, C++-basierte Entwicklungsumgebung für FPGA-Koprozessoren. Dissertation, Universität Mannheim, 2003.

[72] B. Stroustrup, Die C++ Programmiersprache. 4. aktualisierte Auflage, Addison-Wesley, 2000.

[73] F. Klefenz, K.-H. Noffz, R. Zoz, R. Männer, ENABLE - A Systolic 2nd Level Trigger Processor for Track Finding and e/p Discrimination for ATLAS/LHC IEEE Nucl. Sci. Symp. Beitrag in Tagungsband in Proc. IEEE Nucl. Sci. Symp. 1993.

[74] T. Kean, A. Duncan, A 800 Mpixel/sec Reconfigurable Image Correlator on XC6216. In Proceedings of FPL 97, pp. 382-391, 1997.

[75] J. Gause, P.Y.K. Cheung, and W. Luk, Reconfigurable Shape-Adaptive Template Matching Architecture. In Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2002, Napa, California, April 2002.

[76] J. Villasenor, B. Schoner, K. Chia, and C. Zapta, et.al., Configurable Computing Solutions for Automatic Target Recognition. Proceedings of the 1996 Symposium on FPGAs for Custom Computing Machines, pp. 70-79, 1996.

[77] A. Zakerolhosseini, P. Lee, and Ed Horne, An FPGA-based object recognition machine. Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL), Lecture Notes in Computer Science 1482, pp. 228-237, Springer-Verlag, 1998.

[78] D. Xu and R. Sotudeh, A Flexible VLSI Parallel Processing System for Block-Matching Motion Estimation in Low Bit-Rate Video Coding Applications. ACPC, pp. 257-264, 1999.

[79] R.D. Turney, C.H. Dick, Real Time Image Rotation and Resizing, Algorithms and Implementations. www.Xilinx.com, 1999.

[80] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs. Oxford University Press, 1999.

[81] Andarka consulting group, www.andarka.com.

[82] A. Steger, Diskrete Strukturen. Band 1, Springer-Verlag, 2001.

[83] C. Meinel, T. Theobald, Algorithmen und Datenstrukturen im VLSI-Design. Springer-Verlag, 1998.

[84] J. Clark, D.A. Holton, Graphentheorie. Spektrum Akademischer Verlag. 1994.

[85] G. De Micheli, Synthesis and Optimizaton of Digital Circuits. McGraw-Hill, 1994.

[86] T. Sasao, Logic Synthesis and Optimization. Kluwer Academic Publishers, 1998.

[87] G.L. Nemhauser and L.A. Wolsey. Integer and Combinatorial Optimization. John Wiley and Sons, New York, 1988.

# Danksagung

Ich danke Herrn Prof. R. Männer für die Möglichkeit diese Arbeit an seinem Lehrstuhl durchzuführen und für die hilfreiche und freundliche Unterstützung während dieser Arbeit.

Herrn Prof. P. Fischer danke ich für die Übernahme des Zweitgutachtens.

Für die Bereitstellung des C-Codes zur Verkehrszeichenerkennung und den fachlichen Rat nicht nur bei unserer gemeinsamen Veröffentlichung bedanke ich mich bei Dr. D. Gavrila von Daimler-Chrysler.

A. Kugel danke ich für die Betreuung und die Überlassung diverser Hardware, insbesondere dem MPRACE-Board mit Zusatzkarten.

Bei M. Müller und C. Hinkelbein bedanke ich mich besonders für die Unterstützung bei Fragen zur uelib.

Ich danke K. Kornmesser für die hilfreiche Unterstützung in CHDL.

Meinen Zimmerkollegen O. Brosch, G. Lienhart und P. Dillinger danke ich für die lockere und motivierende Arbeitsatmosphäre.

Besonders möchte ich mich auch bei G. Lienhart bedanken für die vielen fruchtbaren Diskussionen und für die Durchsicht des Manuskripts und den daraus resultierenden Anregungen.

Allen Mitgliedern und ehemaligen der FPGA-Gruppe sei für die kollegiale Zusammenarbeit und das gemeinsame Kaffeetrinken gedankt.

A. Seeger und C. Glasbrenner danke ich für die vorbildliche logistische Unterstützung.

An dieser Stelle möchte ich mich bei allen bedanken, die bisher nicht erwähnt wurden und ohne die diese Arbeit nicht zustandegekommen wäre.

Schließlich möchte ich meiner Mutter danken für Ihre fortwährende Unterstützung und allen Freunden, die für den notwendigen Ausgleich sorgten.