HSB
Hochschule Bremen
City University of Applied Sciences

Bachelorthesis

Dynamisches Tiling auf schwachen FPGAs zur Objekterkennung mithilfe kleiner neuronaler Netze

(Dynamic Tiling on Weak FPGAs for Object Detection with Small Neural Networks)

Felix Müller

Matrikel Nr.: 5017732
Dualer Studiengang Informatik B.Sc.

23. Juni 2021

Zusammenfassung

Die Erkennung von Objekten in Bildern wird in vielen Anwendungsbereichen, wie dem autonomen Fahren oder automatisierten Produktionsketten in Fabriken, benötigt. In den letzten Jahren haben sich künstliche neuronale Netze gegenüber klassischer Methoden als überlegen gezeigt. Innerhalb dieses Feldes werden momentan gefaltete neuronale Netze für die Objekterkennung bevorzugt. Die Ausführung dieser Netze ist sehr rechenintensiv und benötigt üblicherweise starke Hardware, welche in restriktiven Bereichen, wie der Edge nicht verfügbar ist. In den letzten Jahren gab es daher einige Bemühungen diesen Rechenaufwand durch beispielsweise eine Hardwarebeschleunigung zu bewältigen. Eine Forschungsgruppe der Firma Xilinx hat auf einem FPGA eine sehr schnelle und effiziente Lösung entwickelt, die jedoch auf sehr geringe Auflösungen begrenzt ist und bisher hauptsächlich mit vorbereiteten Testdaten betrachtet wurde. Für die geringe Auflösung kann das Tiling Abhilfe schaffen, indem es Regionen innerhalb eines hochauflösenden Bildes extrahiert und auf die benötigte Größe skaliert. In dieser Arbeit wurde versucht ein möglichst konfigurierbares und effizientes Tiling auf einem FPGA umzusetzen und es zwischen einer externen Bildquelle und dem neuronalen Netz zu integrieren. Zu diesem Zweck wurden zuerst die Einschränkungen der Hardware beleuchtet und verschiedene Lösungsansätze auf deren Basis erörtert. Der vielversprechenste Ansatz wurde im Anschluss als Prototyp implementiert und hat in der Evaluation im Bereich der Geschwindigkeit und Effizienz sehr vielversprechende Ergebnisse im Vergleich mit einer Softwarelösung gezeigt, aber benötigt noch weitere Optimierung der verwendeten Ressourcen auf dem FPGA. Durch die Anbindung einer externen Bildquelle konnte weiterhin die Verarbeitung reeller Daten ermöglicht werden.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Grafiken, Skizzen, bildliche Darstellungen sowie auf Quellen aus dem Internet. Die Arbeit habe ich in gleicher oder ähnlicher Form auch auszugsweise noch nicht als Bestandteil einer Prüfungs- oder Studienleistung vorgelegt.
Ich versichere, dass die eingereichte elektronische Version der Arbeit vollständig mit der Druckversion übereinstimmt.

Felix Müller
Matrikel Nr.: 5017732
Schwanewede, den 23. Juni 2021

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel betrachtet das Problemfeld der Objekterkennung auf schwacher Hardware und beschreibt den Stand der Technik in diesem. Folgend wird eine Fragestellung im Problemfeld aufgestellt und in einem Lösungsansatz der Plan dieser Arbeit vorgestellt.

1.1 Problemfeld

Objekterkennung und -lokalisierung in zweidimensionalen Bildern sind häufige Anwendungsfälle der Bildverarbeitung, da eine Vielzahl von Problemfeldern diese Funktionalität benötigen. Einige prominente Beispiele wären zum Beispiel die Verkehrsschilderkennung [21], ein automatisches Abblenden des Fernlichts [48] bei Personenkraftfahrzeugen, die Beschriftung von Satellitenbildern [23] oder das automatische Aussortieren fehlerhafter Produkte in einer Fabrik [31]. In den meisten Fällen kann die Verarbeitung auf rechenstarker Hardware ausgeführt werden, jedoch findet in den letzten Jahren eine Verschiebung an die sogenannte „Edge“ statt. Die Edge ist definierbar als der Rand eines Netzes aus Netzwerk- und Rechenressourcen in dem Datenproduzenten und Datenkonsumenten existieren. Verschiebt man die Berechnung nun von den Konsumenten zu den Produzenten, verarbeitet also direkt an der Datenquelle, spricht man von „Edge Computing“. Gründe dafür könnten beispielsweise die geringere Latenz oder eine Verringerung der benötigten Netzwerkbandbreite sein [39]. Die verfügbare Rechenleistung an der Edge ist überlicherweise deutlich geringer im Vergleich zu einer zentralisierten Berechnung in einem Rechenzentrum. Dies ergibt sich dadurch, dass an der Edge besondere Anforderungen an die Geräte gestellt werden. Beispiel dafür sind minimierte Herstellungskosten, ein niedriger Stromverbrauch bei mobilen Geräten oder ausreichende Strahlentoleranz im Weltraum.

1.2 Stand der Technik

Dieser Abschnitt betrachtet den Stand der Technik bei der Objekterkennung mit Fokus auf künstlichen neuronalen Netzen (KNN). Basierend darauf werden folgend die Möglichkeiten zur Ausführung solcher KNN auf schwacher Hardware aufgezeigt. Da besonders bei schwacher Hardware die Bildauflösung begrenzt ist, werden zuletzt bereits erforschte Techniken zum Umgehen dieser Einschränkung erläutert.

1.2.1 Objekterkennung

Die Objekterkennung ist seit Jahren ein beliebtes Forschungsgebiet im Bereich der Bildverarbeitung. Bis 2014 verwendete der Großteil der Forschung traditionelle Methodiken, welche größtenteils die manuelle Erkennung besonderer Eigenschaften enthielten. Mit dieser Technik und diversen Anstrengungen zur Optimierung der Genauigkeit und der Geschwindigkeit wurden bis 2010 inkrementelle Fortschritte erreicht [52]. Nach einer Stagnation in den folgenden vier Jahren erreichten Girshick u.a. mit der Veröffentlichung von [20] einen gewaltigen Durchbruch durch eine Wiederbelebung und Erweiterung der Convolutional Neural Networks (CNN) mit ihrer Forschungsarbeit „Regions with Convolutional Neural Network Features“ (R-CNN). In der damals sehr beliebten „Visual Object Classes Challenge 2012“ (VOC2012) [18] wurde das bisher beste Ergebnis (30%) bei der mittleren durchschnittlichen Genauigkeit mit 53,3% geschlagen und damit eine neue Ära der Objekterkennung eingeläutet.
Das Potential neuronaler Netze wurde zwar bereits früh erkannt, wie etwa von Pal und Pal in ihrem Review zur Bildsegmentierung [33] im Jahre 1993, jedoch brachte erst die starke Verbreitung von rechenstarken Grafikkarten einen erneuten Aufschwung dieser Technik. In den folgenden Jahren nach der Veröffentlichung von R-CNN wurden schnell weitere Fortschritte gemacht. Darunter Fast R-CNN [19] und Faster-RCNN [37], welche die Ausführungsgeschwindigkeit verbessert haben. Analog dazu wurden auch Optimierungen der Genauigkeit erreicht, sodass heutzutage im bereits genannten Wettbewerb VOC2012 Ergebnisse von über 80% erreicht werden, wie zum Beispiel von RefineDet mit 83,8% [51].

1.2.2 Neuronale Netze auf schwacher Hardware

Falls starke Hardware im Sinne der verfügbaren Rechenleistung nicht verfügbar ist, da Anforderungen, wie etwa ein geringer Stromverbrauch, den Einsatz verhindern, muss auf schwächere Hardware ausge-

wichen werden. Dieser Umstand hat zur Folge, dass der Einsatz üblicherweise rechenintensiver KNN erschwert wird. Aus dieser Problemstellung hat sich ein neuer Forschungszweig namens „Tiny Machine Learning“ (TinyML) mit dem Fokus auf der Optimierung KNN für den Einsatz auf solch schwacher Hardware gebildet. Da das Training von KNN typischerweise aufwändiger, als die Ausführung, und meistens einmalig ist, wird dieses meist getrennt auf einem stärkeren System ausgeführt. Trotz des geringeren Rechenaufwands der Ausführung sind weitere Anpassungen notwendig, um den Limitierungen der Hardwareplatform, wie etwa geringer Hauptspeicher, gerecht zu werden.
Eine beliebte Hardwareplatform sind Mikocontroller, weil diese bereits weit verbreitet sind und Anforderungen, wie etwa geringe Kosten, Leistungsaufnahme und Platzbedarf, erfüllen. Ein Framework, welches diese Vorteile stark ausnutzt, ist TensorFlow Lite Micro [16]. Es nutzt die Vorarbeit des verbreiteten Frameworks TensorFlow, sowie dessen verschlanktem Derivats TensorFlow Lite, und abstrahiert mittels eines Interpreters von der Architektur des Mikrocontrollers. Ein großer Nachteil der Mikrocontroller ist die stark sequentielle Ausführung der zugrundeliegenden Prozessorarchitekturen, welche die hohe Parallelität von KNN nicht ausnutzen.
Eine vergleichsweise deutlich parallere Form von Hardware sind Grafikkarten, welche durch ihren Ursprung in der Bildverarbeitung besonders viele Floating-Point-Zahlen gleichzeitig verarbeiten können. Der Hersteller Nvidia bietet mit dem Jetson TX2 [2] Hardware, welche mit einem Stromverbrauch von 7,5W auch einen Einsatz an der Edge erlaubt.
Eine weitere Steigerung bei der Optimierung der Hardware ist anwendungspezifische Hardware, welche speziell für KNN entwickelt wird. Dies kann in der Form eines Hardwarebeschleunigers, wie etwa der „Snapdragon Neural Processing Engine“ des Herstellers Qualcomm [5], für verschiedene oder optimiert für genau einen Anwendungsfall geschehen.
Zur Forschung oder Erstellung von Protoypen sind besonders Field Programmable Gate Arrays (FPGAs) interessant, da diese die Entwicklung von Hardwareschaltungen mittels einer Hardwarebeschreibungssprache erlauben, welche nach einer Übersetzung durch entsprechende Software geladen und auf reeller Hardware ausgeführt werden kann. Dies erspart den aufwendigen Entwicklungsprozess mit diskreter Hardware oder die kosten- und zeitintensive Herstellung eigenen Silikons.
Der Nutzen von FPGAs zur Erforschung KNN hat sich zum Beispiel in einem Wettbewerb zur Entwicklung stromsparender Objekerkennungsalgorithmen für unbemannte Luftfahrzeuge [50] gezeigt. Die Teilnehmer hatten die Wahl zwischen zwei Hardwareplatformen: einem „Nvidia Jetson TX2“ [2] (auf Basis einer GPU) oder einem „Digilent PYNQ-Z1“ [34] (auf Basis eines FPGAs). Bei der Auswertung der Einreichungen kam man zu der Erkenntnis, dass die Lösungen auf dem FPGA zwar einen geringeren Durchsatz an Bildern pro Sekunde schafften, dafür aber die gleiche Genauigkeit bei nur $\frac{1}{3}$ bis $\frac{1}{2}$ des Stromverbrauchs erreicht haben.
Neben der Hardwarebeschleunigung wird auch versucht die Architektur von KNN zu optimieren. Eine solche Optimierung sind beispielsweise binäre neuronale Netze (BNN). Bei einem BNN wird anstatt von Floating-Point-Zahlen für Aktivierungen und Gewichte von klassischen Implementierungen lediglich ein Bit verwendet, welches einen Wert von +1 oder -1 darstellt. Dieser Ansatz sorgt für eine erhebliche Ersparnis an benötigtem Rechenspeicher und erlaubt besondere Optimierungen auf speziell angepasster Hardware [30].
Eine Implementierung eines BNN ist das BNN-PYNQ auf Basis des FINN-Frameworks [41]. Dabei handelt es sich um eine reine Hardwareimplementierung von verschiedenen KNN auf Mittelklasse-FPGAs der Marke Xilinx. Messungen auf einem Zynq Z7045 System-on-a-Chip (SoC) ergaben für das Gesamtsystem einen Stromverbrauch von unter 25 W und einen Durchsatz von über 12.000 Bildern/Sekunde bei der Verwendung eines CNN, jedoch bei einer Auflösung von ledigleich 32x32 Pixeln. Der Flaschenhals für diese Rahmenbedingung ist der verfügbare Block RAM auf dem FPGA, weshalb eine Erhöhung der Eingangsgröße neue Hardware oder die Verwendung von langsameren externen Speicher benötigten würde.
Eine Neuimplementation des BNN-PYNQ ist FINN [13], welches dem gleichen Namen, wie das ursprüngliche theoretische Paper trägt. Während das BNN-PYNQ hauptsächlich zur Demonstration des Potentials BNN auf FPGAs diente, versucht FINN ein universeller anwendbares Framework zu werden. Der Fokus auf stark quantisierte Netze bleibt erhalten, jedoch ist nun eine höhere Konfigurierbarkeit gegeben. Die Generierung der Schaltung für den FPGA erfolgt, wie beim BNN-PYNQ, über die Synthetisierung der Hochsprache C++. Der Unterschied zum Vorgänger ist, dass nur noch fertige Blöcke,

wie etwa Matrizenmultiplikation, als Methoden von Hand geschrieben werden. Die Aufrufe der Methoden werden automatisch durch die Transformation der Knoten des Graphen einer Netztopologie in die vorgefertigten Blöcken erzeugt.

1.2.3 Inferenz von hochauflösenden Bildern

Aktuelle Bildsensoren in Kameras erreichen problemlos sehr hohe Auflösungen, sodass selbst eine Mittelklasse Kamera, wie die Canon M50 [1], Auflösungen ferner der 20 Megapixel erreicht. Ohne Kompression entsteht pro Bild und einer Farbtiefe von 24 Bit ein Speicherbedarf von ca. 60 MB, was für eine Inferenz riesige KNN mit gewaltigem Anforderungen an Rechenkraft und Speicher erfordern würde. Moderne Netze, wie YOLOv4 [14], erreichen jedoch selbst bei einer vergleichsweise geringen Auflösung von 608x608 Pixeln auf einer starken Grafikkarte nur 33 Bilder/Sekunde beim MS COCO Datensatz [27]. Bei einer Verringerung der Auflösung auf 416x416 Pixel erhöht sich die Verarbeitungsgeschwindigkeit auf 54 Bilder/Sekunde. Eine Erhöhung der Auflösung, welche im Vergleich zu den Auflösungen heutiger Bildsensoren, sehr gering ist hat also einen hohen negativen Einfluss auf die Verarbeitungsgeschwindigkeit.
Ein weiteres Problem von Netzen mit größerer Auflösung ist der gesteigerte Bedarf an flüchtigem Hauptspeicher, da mit einer Steigerung der Datenmenge am Eingang auch die Größe einzelner Schichten im Netz steigt, weswegen mehr Daten zwischengespeichert werden müssen. Eine Ablage und Abruf der Daten aus einem nicht-flüchtigem Speicher ist dabei aufgrund der hohen Latenz nicht praktikabel. Weiterhin mangelt es besonders an der Edge an beiden Speicherarten, sodass hier Techniken zur Speicherreduktion angewendet werden müssen.
Eine Möglichkeit ist der Einsatz von Tiling, also einer Unterteilung des Bildes in Regionen geringerer Auflösung, die einzeln vom KNN ausgewertet werden und anschließend zu einem Gesamtergebnis kombiniert werden. In einer systematischen Evaluation der negativen Auswirkungen von Tiling [36] ergab sich, dass die Ergebnisse im Vergleich zu einem Netz für die volle Bildgröße schlechter ausfallen, jedoch wurde auch erkannt, dass es als Mittel zum Zweck zumindest so lange dienen muss, bis es möglich ist ausreichend große Netze für hohe Auflösungen umzusetzen. Weiterhin kamen die Autoren zu dem Schluss, dass die Qualität der Ergebnisse erheblich von gewählten Parametern, wie Tilegröße, Überlappung der Tiles oder der sogenannten „translational variance“ abhängt. Letzterer Begriff beschreibt das Phänomen, dass die Position des Objekts innerhalb eines Tiles eine Auswirkung auf das Ergebnis hat. Jenes Phänomen wurde zum Beispiel in [23] im Rahmen der semantischen Segmentation bei der Satelitenfotografie genauer untersucht. Besonders erwähnenswert ist auch die Arbeit von Ünel, Özkalaycı und Çığla, welche die Stärken von neuronalen Netzen in Kombination mit Tiling zur Erkennung kleiner Objekte in Bildern evaluiert haben [32]. Die Rechenzeit des Tilings, der in der Arbeit verwendeten Implementierung, steigt linear mit der Auflösung der Bilder, wodurch entweder bei der Bildwiederholrate oder bei der Auflösung Abstriche gemacht werden müssten. Krishna und Jawahar zeigen in ihrem Paper [25], wie mithilfe eines weiteren KNN zur Suche interessanter Bildbereiche in Kombination mit einer Hochskalierung und Entrauschen sehr gute Ergebnisse bei der Erkennung kleiner Objekte ($<$1% des Bildbereichs) erreicht werden können. Eine Erkennung von Objekten war mit diesem Ansatz teilweise bei einem Bildbereich von lediglich 16x16 Pixeln möglich.
Einige der genannten Publikation wurden erst die letzten Jahre veröffentlicht, wodurch festzuhalten ist, dass die Forschung zu Tiling für Bildverarbeitung mit KNN noch nicht abgeschlossen ist und weitere Arbeit benötigt.

1.3 Fragestellung

Die Ausführung KNN auf FPGAs ist ein relativ unerforschtes Gebiet, welches jedoch ein hohes Potential im Bereich hoher Geschwindigkeit und Effizienz bei geringem Stromverbrauch zeigt. Ein besonders interessantes Einsatzgebiet ist die Objekterkennung mithilfe einer Kamera auf der Edge mit inhärenten strengen Anforderungen an den Stromverbrauch.
Die meisten der existierenden KNN zur Objekterkennung auf kleinen FPGAs, wie zum Beispiel der Artix-7 Serie von Xilinx, sind limitiert auf geringe Auflösungen und wurden teilweise nur unter Laborbedingungen mit vorbereiteten Bildern getestet. Hervorzuheben ist das BNN-PYNQ [41], welches durch hardwareoptimierte Lösungen mittels eines BNN besonders schnell und dabei gleichzeitig stromsparend ist. In der Standartimplementierung werden Bilder mit einer Auflösung von 32x32 Pixeln und drei Farbkanälen rot-grün-blau (RGB) in einem DDR3-Speicher abgelegt und von dort aus vom KNN auf dem FPGA abgerufen und ausgewertet.
Eine direkte Übertragung eines Eingangsbildsignals ist nicht möglich, da dessen Auflösung meist um ein Vielfaches höher, als die Eingangsauflösung des KNN auf dem FPGA, ist. Eine Herunterskalierung des gesamten Bildes ist nicht zielführend, da ein Großteil der Informationen verloren geht. Besser ist die selektive Auswahl interessanter Bereiche, die mit einem deutlich geringerem Faktor skaliert werden können, falls nötig. Die Anzahl, Position und Größe der Bildbereiche variiert abhängig von der Methode zur Generierung der Regionen und muss daher in möglichst vielen Aspekten konfigurierbar sein.
Eine Softwarelösung auf einer CPU ist mittels verbreiteter Programmbibliotheken in einer einfachen Form trivial umsetzbar und lässt sich einfach anpassen.

These: Die Bildvorverarbeitung des Tilings auf der CPU ist ineffizient und kann auf einem FPGA deutlich stromsparender und gleichzeitig leistungsstärker umgesetzt werden.

Aus der These ergeben sich folgende Fragestellungen:

- Wie einfach lässt sich das Tiling auf einem FPGA umsetzen?
- Wie konfigurierbar kann das Tiling auf einem FPGA umgesetzt werden?
- Wie schnell ist das Tiling auf dem FPGA im Vergleich zur CPU?
- Wie effizient ist das Tiling auf dem FPGA im Vergleich zur CPU?
- Sofern kein zweiter FPGA verwendet wird, belegt das KNN bereits einen Teil der Ressourcen. Kann das Tiling mit den übrigen Ressourcen umgesetzt werden?
- Wie einfach kann eine externe Bildquelle an das Tiling auf dem FPGA angebunden werden?

1.4 Lösungsansatz

Zum Beweis der These werden die gegebenen Einschränkungen betrachtet und auf deren Basis verschiedene Lösungsansätze entwickelt. Die Vor- und Nachteile der Ansätze werden zueinander abgewägt und die vielversprechenste Lösung als Prototyp umgesetzt.
Zu Evaluierungszwecken wird eine optimierte Softwarelösung herangezogen. Für eine möglichst hohe Vergleichbarkeit wird die gleiche Hardware, das Digilent PYNQ-Z1, und die gleiche Implementierung für das KNN, das BNN-PYNQ, genutzt. Da das BNN-PYNQ nicht mehr weiter entwickelt wird und bereits das FINN als Nachfolger hat, soll die neue Lösung auch kompatibel mit diesem sein.
In der Evaluierung wird zuerst der Ressourcenverbrauch und das Timing betrachtet, um sicherzustellen, dass die neue Lösung tatsächlich auf reeller Hardware einsetzbar ist. Als nächstes wird die Geschwindigkeit gemessen und gegen die Softwarelösung verglichen. Folgend wird die Konfigurierbarkeit und Komplexität der Implementierung behandelt.
Die Erkenntnisse der Evaluation werden zuletzt genutzt, um die anfängliche Fragestellung zu beantworten und einen Ausblick für weitere nötige Arbeiten zu geben.

1.5 Aufbau der Arbeit

Zuerst werden in Kapitel 2 die nötigen Grundlagen für die Umsetzung des Lösungsansatzes erklärt. In Kapitel 3 werden verwandte Arbeiten vorgestellt. Kapitel 4 enhält die Konzeption und den Vergleich verschiedener Ansätze für die Umsetzung des Tilings. In Kapitel 5 wird die Implementierung eines Prototypen beschrieben und in Kapitel 6 dessen Funktionalität sichergestellt. In Kapitel 7 wird der entwickelte Prototyp unter möglichst realen Bedingungen evaluiert wird. Zuletzt werden in Kapitel 8 die Ergebnisse der Evaluation zur Beantwortung der anfänglichen Fragestellung genutzt und ein Ausblick für weitere nötige Arbeit gegeben.

2 Grundlagen

Dieses Kapitel erklärt die notwendigen Grundlagen, um den Lösungsansatz aus Kap. 1.4 umzusetzen. Als Erstes wird in Kap. 2.1 beschrieben, wie mithilfe von Tiling hochauflösende Bilder mit einem kleineren KNN verarbeitet werden können. Folgend beschreibt Kap. 2.2 die digitale Übertragung von Bilddaten. In Kap. 2.3 wird beschrieben, wie die Skalierung von Bildern funktioniert und wie diese auf einem FPGA umgesetzt werden kann. Kap. 2.4 enhält die Architekturbeschreibung der verwendeten Hardware, welche in Kap. 2.5 durch die Einführung des internen Busses erweitert wird. In Kap. 2.6 wird beschrieben, wie KNN zur Bildklassifizierung mit den existierenden Implementierungen BNN-PYNQ und FINN angebunden werden können. Zuletzt werden in Kap. 2.7 die Blockdiagramme vorgestellt, welche ein leistungsfähiges Werkzeug in der Entwicklungsumgebung Vivado zum Aufbau von Bussystemen sind.

2.1 Inferenz hochauflösender Bilder mithilfe von Tiling

Die Problematik bei der Inferenz hochauflösender Bilder mittels KNN ergibt sich durch deren begrenzte Größe, welche wiederum unter anderem durch verfügbare Hardwareressourcen begrenzt ist. Der einfachste Lösungsansatz ist eine Skalierung des gesamten Bildes auf die Eingangsgröße des KNN. Nachteil dieses Ansatzes ist der hohe Informationsverlust, welcher besonders groß ist, falls das zu erkennende Objekt nur einen kleinen Bereich des Bildes einnimmt. Wird beispielhaft die Skalierung um einen Faktor von ungefähr vier eines Bildes in Abb. 1 betrachtet, dann ist es problemlos möglich das Objekt weiterhin zu erkennen.

(a) 135x101 Pixel (Original)

(b) 34x25 Pixel

(c) 34x25 Pixel

Abbildung 1: Skalierung mit prominentem Objekt

Im Vergleich dazu ist die Erkennung der drei weniger prominenten Objekte bei identischer Zielauflösung in Abb. 2 selbst mit dem menschlichem Auge nahezu unmöglich. Durch das unterschiedliche Seitenverhältnis von Start- und Zielauflösung entsteht weiterhin eine Verzerrung der Objekte, wodurch die Erkennung zusätzlich erschwert wird.

(a) 858x311 Pixel (Original)

(b) 34x25 Pixel

(c) 34x25 Pixel

Abbildung 2: Skalierung mit kleinen Objekten

Unter der Annahme, dass nur bestimmte Bereiche des Bildes von Interesse sind, wie es beispielsweise bei der Objektlokalisierung der Fall ist, kann die Prominenz im resultierenden Bild erhöht werden, indem vor der Skalierung ein kleinerer Bildbereich ausgeschnitten wird. Dieses Vorgehen für die Extraktion eines bestimmten Bildbereiches mit anschließender Skalierung auf eine kleinere Größe wird folgend als „Tiling“ (zu deutsch „Kacheln“) bezeichnet und ein Resultat einer solchen Verarbeitung als „Tile“ (zu deutsch „Kachel“).
Folgend müssen Anzahl, Größe und skalierte Größe der Tiles bestimmt werden, damit das gesuchte Objekt einen möglichst großen Bereich des Bildes füllt. Die skalierte Größe ergibt sich durch die

Eingangsgröße des KNN. Für die Bestimmung der weiteren Parameter sind verschiedene Ansätze mit verschiedenen Vor- und Nachteilen möglich.
Das sogenannte „Sliding Window“ verwendet eine Schablone fester Größe, die mit einem konstanten Versatz über das Bild geschoben wird. Ist der Versatz identisch zur Größe, entsteht das Problem, dass ein Objekt sich zwischen zwei Tiles befinden kann, wodurch es möglicherweise nicht mehr erkannt wird. Um dem entgegenzuwirken kann der Versatz verringert werden, sodass sich die einzelnen Tiles überlappen. Im Anschluss ist es nun nötig die Einzelergebnisse wieder zusammenzuführen. Auch hier sind verschiedene Ansätze möglich, wie etwa die Wahl des Tiles mit dem höchsten Konfidenz nach der Inferenz oder das Zusammenführen naheliegender Tiles mit ausreichend hoher Konfidenz. Vorteil des Sliding Windows ist die einfache Implementierung. Nachteil ist der hohe Rechenaufwand durch viele verarbeitete Tiles, die außerhalb des Objektbereichs liegen. Des weiteren können durch die feste Größe und den festen Versatz der Schablone nur Objekte erkannt werden, die einen Bildbereich ähnlicher Größe füllen und ungefähr auf dem durch den Versatz festgelegten Raster liegen. Beides kann ausgeglichen werden durch Schablonen verschiedener Größe oder kleineren Versatz, was jedoch die Anzahl der Tiles und damit wiederum den Rechenaufwand erhöht.
Mithilfe sogenannter „Region Proposals“ kann die Anzahl und Trefferquote der ausgewählten Bildbereiche erhöht werden. Für die Generierung der Regionsvorschläge existieren erneut mehrere Möglichkeiten mit unterschiedlicher Qualität und unterschiedlichem Rechenaufwand. Ein Ansatz ist die Verwendung klassischer Bildsegmentierung. Soll beispielsweise das rechte Auto im obigen Beispielbild erkannt werden, kann dessen markante rote Farbe genutzt werden, um eine grobe Bestimmung des Bereichs zu ermöglichen. Das Resultat einer solch simplen Segmentierung kann in Abb. 3 betrachtet werden. Diese konkrete Beispiel unterliegt natürlich vielen Einschränkungen, da beispielsweise ein grüne Segmentierung durch den fehlenden Kontrast zum Hintergrund weniger zielführend wäre. Komplexere Methoden (z.B. Kantenerkennung oder der Einsatz eines weiteren KNN) können womöglich bessere Ergebnisse erzielen, allerdings muss besonders in restriktiven Bereichen, wie etwa auf der Edge, stets Aufwand und Nutzen abgewägt werden.

Abbildung 3: Regionsvorschlag mittels Bildsegmentierung anhand eines Farbkanals

Die schlußendlich verwendete Methodik muss abhängig vom Anwendungsfall und dessen Anforderungen gewählt werden. Beispielsweise kann Rechenaufwand und damit der Stromverbrauch drastisch reduziert werden, falls eine geringere Erkennungsrate oder schlechtere Positionsresultate akzeptabel sind. Zuletzt müssen die Einzelergebnisse der Tiles wieder zu einem Gesamtergebnis kombiniert werden. Auch hier gibt es wieder mehrere Möglichkeiten, wobei die wohl bekannteste die „Non-Maximum Suppression“ ist [38], welche aus einer Liste von Bereichen und derer Klassifizierungskonfidenz einen oder mehrere resultierende Bereiche errechnet.

2.2 Digitale Bildübertragung

Für einen maximalen Datendurchsatz wären im Idealfall rein parallele Schnittstellen zu bevorzugen, welche die gesamten Daten eines Bildes innerhalb eines Taktzyklus übertragen. Soll eine solche Schnittstelle nun auf echter Hardware verwendet werden, fällt auf, dass bereits für ein Bild mit geringer Auflösung von 320x240 Pixeln und einer Farbtiefe von 24 Bit ein Bedarf an 1.843.200 Flip-Flops entsteht, sobald kein reines Schaltnetz verwendet wird. Dieser Bedarf übersteigt die verfügbaren Ressourcen auf kleiner Hardware, wie dem in dieser Arbeit verwendeten Xilinx Zynq-7020, um einen Faktor von 17x [54]. In der Praxis wird daher meist ein sequentieller Datenstrom oder ein Zwischenpuffer verwendet.
Durch den Ursprung in der Darstellung auf Kathodenstrahlröhrenbildschirmen hat sich besonders die pixel- und zeilenweise Übertragung etabliert. Ein konkreter Standard ist der AXI4-Video Stream, welcher vom Hersteller des verwendeten SoC offiziell unterstützt wird. Dank paralleler Übertragung und freier Konfigurierbarkeit auf FPGAs ist es möglich bis zu einem Pixel pro Taktzyklus bei beliebiger Farbtiefe und Auflösung zu übertragen, sofern genug Hardwareressourcen zur Verfügung stehen und beide Kommunikationspartner entsprechend schnell Daten produzieren oder verarbeiten können, denn beide Seiten sind jederzeit in der Lage die Übertragung durch Verweigerung des Handschlags zu blockieren.
Der AXI4-Video Stream Standard definiert keine Maximaldauer für Übertragungsblockaden, daher muss anhand der Bildquelle/Bildsenke entschieden werden, ob Blockaden akzeptabel sind oder eine Kompensation nötig ist. Soll beispielsweise ein externes Videosignal verarbeitet werden, welches zu jeder steigenden Flanke eines Pixeltakts Daten liefert, dann müssen diese zu jeder dieser steigenden Flanken abgegriffen werden, da sie sonst verloren gehen.
Eine Möglichkeit kleine bis mittlere Verzögerungen auszugleichen ist die Verwendung einer First-In-First-Out (FIFO) Warteschlange. Solange die Datensenke die gepufferten Daten schnell genug verarbeitet und die Warteschlange so nicht volläuft, wird die Übertragung der Datenquelle nicht blockiert. Analog dazu wird die Datensenke nie blockiert, solange die Warteschlange stets schnell genug von der Datenquelle gefüllt wird.
Alternativ kann ein zufällig adressierbarer Speicher als Puffer verwendet werden, was zusätzlich den Vorteil bietet, dass die Pixel nicht zwingend sequentiell, sondern in beliebiger Reihenfolge im Puffer verarbeitet werden können. Die Größe des Puffers hängt vom Anwendungszweck und den verfügbaren Ressourcen ab. Klassische Beispiele sind die Speicherung einzelner Zeilen (Zeilenpuffer) oder die Speicherung ganzer Bilder (Bildpuffer). Zur Vermeidung von Zugriffskonflikten durch Datenquelle/-senke existiert der Puffer meist in zumindest zweifacher Form und wird ausgetauscht, sobald ein anderer Puffer komplett gefüllt wurde.

2.3 Bildskalierung (auf FPGAs)

Bildskalierung bezeichnet die Änderung der Größe eines Bildes. Die beliebte Bildverarbeitungsbibliothek OpenCV fasst diesen Prozess allgemein in folgender Abbildung zusammen [3]:

$$dst(x,y) = src(f_x(x,y),f_y(x,y)) \tag{1}$$

Die Funktionen *dst* und *src* sind eine Abbildung von einem Koordinatenpaar (x,y) zu dem Wert des entsprechenden Pixels, während *dst* das Bild neuer Größe und *src* die interpolierten Werte aus dem Quellbild darstellt. Die Funktionen f_x und f_y bilden ein Koordinatenpaar auf eine neue Koordinate ab. Da das Ergebnis nicht zwingend ganzzahlig ist, ist durch *src* eine Interpolation mit den diskreten Koordinaten (x,y) im Quellbild notwendig. Der einfachste Ansatz ist das Runden der Koordinaten und wird „Nächster-Nachbar-Interpolation“ (engl. nearest-neighbour) genannt. Nachteilig ist der relativ hohe Informationsverlst, weshalb komplexere Interpolationen für die Mittelung der einzelnen Pixelwerte verwendet werden können [3]. Die bilineare Interpolation ist eine lineare Interpolation, welche nacheinander in Richtung beider Achsen zwischen vier Punkten interpoliert. Interpolation höherer Ordnung, wie etwa die bikubische Interpolation, erreichen ein glatteres Ergebnis, aber benötigen mehr Rechenleistung [49]. Die genannten Interpolation werden am häufigsten verwendet, da sie universell einsetzbar und relativ einfach zu berechnen sind. Spezielle Skalierungen mit besonderen Transformation abhängig vom Bildinhalt existieren, aber sind stark anwendungsfallabhängig.
Die Wahl des verwendeten Skalierungsalgorithmus ist eine Abwägung zwischen Rechenaufwand und Qualität der Ergebnisse, wobei die Metrik der Qualität anhand der Anforderungen definiert werden

muss. Beispielsweise ist ein geglättetes Bild für einen Menschen potentiell anschaulicher, während der Fokus bei der Erkennung von Objekten eher auf der Vermeidung von Artefakten liegt.
Im Vergleich zu einer Softwareimplementation kann durch eine Hardwarebeschleunigte Lösung eine deutlich höhere Effizienz erreicht werden. In der Arbeit von Nuño-Maganda et al. wurde gezeigt, wie die bikubische Interpolation auf einem FPGA im Vergleich zu einer Softwarelösung auf einer gebräuchlichen CPU bei 24x geringerer Frequenz eine 10x schnellere Ausführungszeit erreicht [29].
Heutzutage ist die Anwendung von Bildskalierung auf FPGAs durch fertige Komponenten oder mittels einer hochsprachen Hardwaresynthese sehr einfach möglich, sodass beispielsweise C++ Quellcode mit der OpenCV Bibliothek zur Erzeugung einer Hardwareschaltung genutzt werden kann [45].

2.4 Architektur der verwendeten Hardwareplattform

Das PYNQ-Z1 ist eine Entwicklungsplatine des Herstellers Digilent (Abb. 4) auf Basis eines Xilinx ZYNQ XC7Z020-1CLG400C System-on-Chip (SoC) [34]. Dieser SoC ist das größte kostenoptimierte Modell der Zynq-Familie [54] und durch vergleichbare Hardwareressourcen zum strahlungsharten Virtex-5QV [35] ideal für Konzeptstudien deren Ergebnisse somit theoretisch auch im Weltraum oder in ähnlich eingeschränkten Bereichen eingesetzt werden können.
Der SoC kann grob in drei Bereiche eingeteilt werden: CPU, FPGA und Peripherie. Alle Komponenten innerhalb des SoC kommunizieren mittels zwei AMBA AXI3 Bussen für schnelle Kommunikation, wie Speicherzugriffe, und langsame Kommunikation, wie Konfiguration oder relativ langsame Schnittstellen (beispielsweise I2C). Die Abb. 5 zeigt die wichtigsten Komponenten und deren Zusammenschluss.

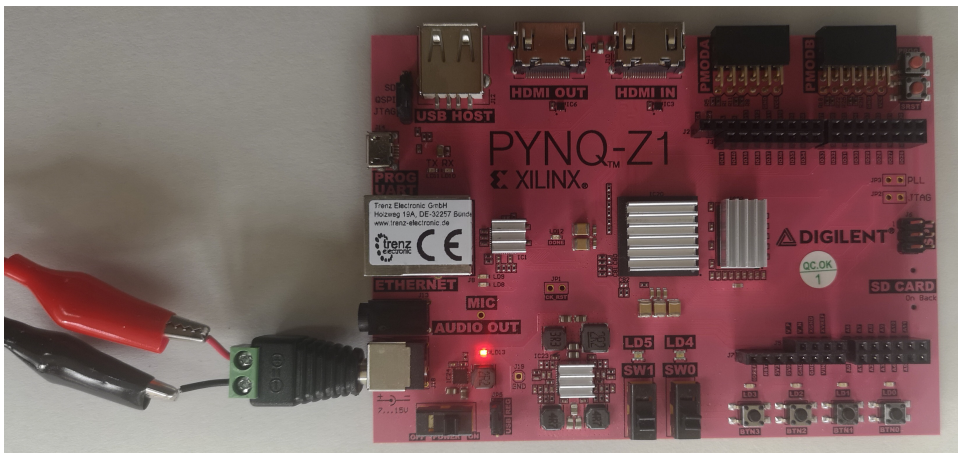


Abbildung 4: Digilent PYNQ-Z1

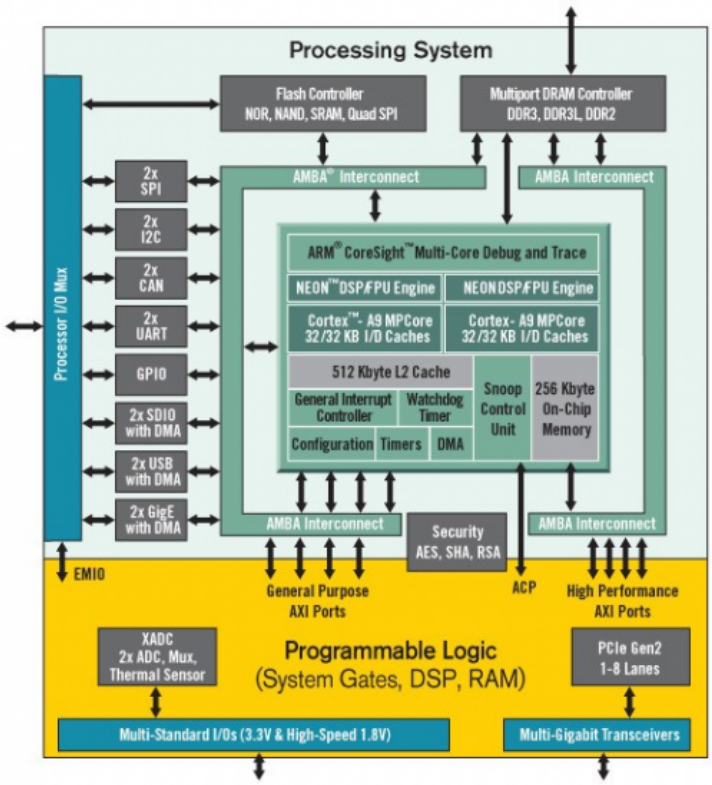


Abbildung 5: Architektur des Xilinx Zynq SoC [34]

2.4.1 CPU

Die CPU ist ein ARM Cortex-A9 Prozessor mit zwei Rechenkernen, welcher auf dem PYNQ-Z1 standartmäßig mit einer Taktfrequenz von 650 Mhz läuft. Als Cache sind jeweils 32 KB für Anweisungen/Daten auf Ebene 1 und gemeinsame 512 KB auf Ebene 2 verfügbar. Zusätzlich dazu stehen 256 KB an eigenem Arbeitsspeicher zur Verfügung, sodass kleine Programme keinen Zugriff auf externen Speicher benötigen. Das Programm kann über JTAG, einen externen Quad-SPI Flash oder von einer Micro SD Karte geladen werden. Vom Hersteller wird eine auf Ubuntu 18.04 basierende Distribution vorgesehen, die Teil der PYNQ Softwarefamilie ist. Diese Distribution erlaubt es mit dem FPGA sehr einfach über die Programmiersprache Python zu kommunizieren [34].

2.4.2 FPGA

Der FPGA ist äquivalent zu einem Xilinx Artix-7 und besteht hauptsächlich aus ca. 85.000 Logikzellen (CLB), die wiederum jeweils aus 8 Lookup Tables (LUTs) mit 6 Eingängen und 16 Flip-Flops (FFs) bestehen. Die CLBs können außerdem als RAM oder Shift Register konfiguriert werden und bieten spezielle Logik für die Implementierung von Addierern. Neben den CLBs sind 630 KB Block RAM (BRAM) verfügbar. Dieser ist in 36 Kb Blöcke augeteilt, die nochmals zweigeteilt werden können. Die Blöcke des RAMs sind in der Lage zwei völlig unabhängige Schnittstellen bereitzustellen, die beide Lesen und Schreiben können („true dual-port“). Zur Unterstützung spezieller digitaler Signalverarbeitung sind 220 sogenanne „DSP Slices“ verfügbar, die komplexe Operation wie vorzeichenbehafte Multiplikation unterstützen [53].

2.4.3 Peripherie

Der Zynq SoC enthält viele fertige Peripheriekomponenten, um CPU und FPGA bei der verwendung häufiger Schnittstellen zu untersützten. Eine der wichtigsten Periphierekomponenten ist der „Multiport

DRAM Controller“, welcher den Komponenten innerhalb des SoC mittels Direct Memory Access (DMA) über den AXI-Bus direkten Zugriff auf einen externen Hauptspeicher erlaubt. Im Falle des PYNQ-Z1 sind 512 MB DDR3-Speicher mit einem Datenbus von 16 Bit und einem maximalen Durchsatz von 1050 Mbps auf der Platine verbaut und angebunden.
Zur persistenten Speicherung von Programmen ist sowohl ein Controller für SD-Karten, als auch für Flash Speicher vorhanden. Auf dem PYNQ-Z1 sind 16 MB Flashspeicher mit einer Quad-SPI Schnittstelle verbaut.
Weiterhin existiert für fast jede gängige Schnittstelle ein entsprechender Controller, wie etwa Gigabit Ethernet, I2C, SPI oder UART [34].

2.4.4 Externe Schnittstellen

Der Zynq SoC besitzt eine Vielzahl an Ein- und Ausgängen, die auch größtenteils auf dem PYNQ-Z1 nach außen geführt ist.
Die wohl wichtigste Schnittstelle auf der Platine, sofern nicht von einer SD-Karte oder Flashspeicher gestartet wird, ist der Micro USB Anschluss, der es mittels FTDI Chip erlaubt den SoC und Flashspeicher über JTAG zu programmieren. Gleichzeitig wird die Platine mit Strom versorgt und Zugriff auf eine der UART Schnittstellen gegeben, sodass Debuggingausgaben leicht möglich sind. Die Stromversorgung über USB ist begrenzt auf 0,5A und kann nach Bedarf über einen Klinkenanschluss abgelöst werden.
Besonders interessant für die Videoverarbeitung sind zwei HDMI Anschlüsse für die Ein- und Ausgabe eines DVI Signals. Die Verwendung eines echten HDMI-Signals ist aus Lizenzgründen nicht trivial möglich.
Für die Anbindung von USB Geräten ist ein USB Typ-A Anschluss verbaut, der allerdings fest mit der CPU verbunden ist. Neben den genannten Anschlüssen existieren unter anderem verschiedene Taster und Schalter, LEDs, Gigabit Ethernet, ein Audioausgang und ein Mikrofon [34].

2.5 Advanced eXtensible Interface (AXI)

Das Advanced eXtensible Interface (AXI) ist ein synchrones und paralleles Kommunikationsprotokoll, welches vom Unternehmen ARM entwickelt wurde. Die neueste Version (AXI4) wurde 2010 als Teil der AMBA4 Spezifikation eingeführt [7] und wird offiziell vom Hersteller Xilinx für die eigenen FPGA und SoC Produktreihen unterstützt [47]. AXI4 implementiert einen logischen Bus mit Unterstützung für mehrere Master und Slaves, welcher Rückwärtskompatibel zur Vorgängerversion AXI3 ist, der auf dem verwendeten Zynq für die Kommunikation ausserhalb des FPGAs verwendet wird. Jegliche Kommunikation erfolgt von Punkt zu Punkt, sodass für den Zusammenschluss von mehr als zwei Komponenten sogenannte „Interconnects“ benötigt werden. Zwischen zwei Punkten kann die Datenbreite beliebig gewählt werden.
Innerhalb der AXI4 Spezifikation sind drei Schnittstellen für unterschiedliche Anforderungen definiert, die sich innerhalb ihrer Signale und Kommunikationsart unterscheiden.

2.5.1 AXI4

Die AXI4 Schittstelle, welche auch „AXI4 Memory-Mapped“ oder „AXI4-Full“ genannt wird, erlaubt speicheradressierte Transaktionen ohne Annahmen über die Art des Speichers zu treffen, der durch einen Slave implementiert wird. Diese universelle Unterstützung wird ermöglicht durch die Verwendung von fünf Kommunikationskanälen:

- Lese- und Schreibadresse (AR und AW)
- Lese- und Schreibdaten (R und W)
- Status einer vorhergegangen Schreiboperation (B) - Der Status einer Leseoperation ist in den Lesedaten enthalten

Die offizielle Kurzbezeichnung jedes Kanals wird in Klammern angegeben, welche folgend für T in den Signalnamen eingesetzt werden können. Nicht jeder Kanal besitzt dabei alle Signale. Die genaue Zuordnung kann der offiziellen Spezifikation [7] entnommen werden.
Eine Datenübertragung innerhalb der Kanäle findet statt, wenn beide Seiten zu einer steigenden Taktflanke bereit sind. Dies geschieht durch einen zweiseitigen Handschlag mittels zweier Signale: „TREADY“ und „TVALID“. TREADY signalisiert dabei die Bereitschaft Daten empfangen zu können und TVALID die Validität der aktuell angelegten Daten.
Lese- und Schreibtransaktionen geschehen immer als sogenannter „Burst“, was eine sequentielle Folge mehrerer Datentransfers „Beats“ ab einer spezifizierten Startadresse bedeutet. Ein Burst wird auf dem Adresskanal initialisiert, sowie parametresiert, und auf dem Datenkanal ausgeführt. Dieser gesamte Vorgang wird „Transaktion“ genannt. Als Art des Bursts kann zwischen drei Optionen gewählt werden:

- FIXED: Jeder Beat schreibt an die spezifizierte Adresse
- INCR: Nach jedem Beat wird die Adresse automatisch um einen spezifizierten Wert erhöht
- WRAP: Identisch zu INCR, außer dass nach einer spezifizierten Anzahl Transfers auf die Startadresse zurückgesetzt wird

Die Anzahl der Beats innerhalb eines Burst wird vor Beginn festgelegt und kann nach dem Start nicht mehr geändert werden. Alle Beats in einer Transaktionen haben die gleiche Wortlänge. Der letzte Transfer innerhalb eines Bursts wird durch den Generator der Daten mit dem Signal „TLAST“ signalisiert.
Ein Burst darf keine Adressgrenze von 4096 Bytes überschreiten, was eine Sicherstellung ist, dass kein Burst zwei verschiedene Slaves anspricht, da diese ebenfalls auf diese Grenzen ausgerichtet werden müssen. Ein Slave kann beispielsweise dem Adressbereich 0 bis 2048 zugeordnet sein, aber trotzdem kann der nächste Adressbereich eines Slaves erst bei 4096 beginnen. Ein Master wiederum kann mit einem Burst auf den Adressbereich 4079 bis 4095 oder 4096 bis 4128 zugreifen, aber nicht auf 4092 bis 4100.
Durch „TSTRB“ ist es weiterhin möglich unausgerichtet auf den Speicher, also nicht in Abständen von Worten, zuzugreifen. Dieses Signal enthält für jedes Byte der Wortlänge von Beats ein Bit, welches die Aktivierung dieses Bytes anzeigt. Falls nur ausgerichte Transaktionen erlaubt wären, könnte bei einer Wortlänge von vier Bytes nur wortweise auf die Adressen $4n$ (0, 4, 8, etc.) zugegriffen werden. Dies wäre unvorteilhaft, falls beispielsweise vier Bytes ab der Adresse 1 geschrieben werden sollen, da im ersten Wort ein und im zweiten Wort drei Bytes überschrieben werden, die nicht Teil der neuen Daten sind. Das TSTRB Signal schafft Abhilfe, indem es als eine Art Maske gesetzt werden kann. Beim genannten Beispiel wäre diese Maske (niederwertigstes Byte rechts) für Wort 1: „1110“ und für Wort 2: „0001“, wodurch nur die gewünschten Bytes modifiziert werden.

2.5.2 AXI4-Lite

Die AXI4-Lite Schnittstelle ist eine vereinfachte Form der kompletten AXI4 Schnittstelle und wird meistens für langsame Operationen, wie etwa das Lesen/Schreiben eines Registers, genutzt. Die größten Unterschiede zur vollständigen AXI4 Schnittstelle sind der Entfall der Bursts und die Beschränkung auf Datenbreiten mit einer Größe von 32 oder 64 Bit.

2.5.3 AXI4-Stream

Die AXI4-Schnittstelle erlaubt die sequentielle Folge mehrerer Transfers als Datenstrom zwischen einem Master und einem Slave. Ein Transfer findet äquivalent zu den Kanälen von AXI4(-Lite) immer bei steigender Taktflanke und „TREADY“/„TVALID“-Handschlag statt. Mehrere Transfers können durch das „TLAST“-Signal in Pakete gruppiert werden. Für die Daten innerhalb eines Transfers kann für jedes Byte durch „TKEEP“ die Aktivierung festgelegt werden. Zusätzlich kann durch „TSTRB“ definiert werden, ob sich um ein Positions- oder Datenbyte handelt. Das Signal „TUSER“ kann beliebig verwendet und abhängig vom Anwendungsfall definiert werden.
Eine konkrete Nutzung dieses Signals implementiert der AXI4-Video Stream für die zeilenweise Übertragung von Bilddaten, welcher „TUSER“ für die Signalisierung des ersten Pixels innerhalb eines Bilder und „TLAST“ für das letzte Pixel einer Zeile verwendet.

2.6 Anbindung von KNN zur Bildklassifizierung auf FPGAs

Zur Anbindung eines KNN auf einem FPGA ist es notwendig dessen Schnittstellen zu kennen, welche abhängig von den im Netz verarbeiteten Daten stark variieren können. Mit Bezug auf den Rest der Arbeit wird dieser Abschnitt auf KNN zur Klassifizierung von Bilddaten beschränkt. Als konkrete Implementierung wird nun als Erstes das BNN-PYNQ betrachtet, welches eine Netztopologie mit drei vollständig verbundenen Schichten (Fully Connected Network) namens „SFC“ und „LFC“, sowie eine Topologie mit gefalteten Schichten (Convolutional Network) namens „CNV“ [41] unterstützt, wobei nur letztere zum Zwecke der Objekterkennung sinnvoll anwendbar ist.
Da die Schnittstellen von den originalen Entwicklern nur auf sehr abstrakter Ebene dokumentiert wurden, basiert die folgende Beschreibung auf der Vorarbeit eines studentischen Projektes [15], welches die Schnittstellen auf niederster Ebene anhand des Quellcodes und weiteren Methoden, wie die Analyse der Daten im Speicher, nachvollzogen und dokumentiert hat.
Grundsätzlich besitzt das BNN-PYNQ zwei Schnittstellen: AXI4-Lite Slave zur Konfiguration, Steuerung und Statusabfrage und AXI4 Master für den Zugriff auf den DDR3-Hauptspeicher zum Abruf der Quellbilddaten und Ablage der Ergebnisse.
Zur Initialisierung müssen als erstes über die AXI4-Lite Schnittstelle die Parameter für das Netz (Gewichte und Schwellwerte) geladen werden. Mit diesen Parametern können für ein fixes Netz in Hardware verschiedene Trainingsdaten verwendet werden, sodass mit der gleichen Hardwareschaltung CNV verschiedene Datensätze, wie CIFAR10 [26] oder German Road Signs [22], benutzt werden können. Nach erfolgreicher Initialisierung müssen nun die zu verarbeitenden Bilddaten im Speicher abgelegt werden. Mehrere Bilder können aneinandergereiht gespeichert werden. Jedes einzelne Bild wird pixel- und zeilenweise im Speicher abgelegt mit drei Byte pro Pixel. Jedes Byte beschreibt einen von drei Farbkanälen: Rot-Grün-Blau im Bereich 0 bis 255. Bei der Ablage im Speicher muss beachtet werden, dass sich die Daten in einem kontinuierlichen Bereich des physischen Adressbereichs befindet. Die Startadresse dieses Speicherbereichs, sowie die Startadresse des Bereichs für die Ergebnisse, wird nun über AXI4-Lite zusammen mit der Anzahl Bilder konfiguriert. Nun kann die Verarbeitung über AXI4-Lite gestartet werden. Über ein Statusregister kann nachvollzogen werden, wann die Verarbeitung beendet ist. Sobald dies der Fall ist, können die Ergebnisse ab der konfigurierten Adresse im Speicher abgerufen werden. Die CNV Topologie erzeugt immer Ergebnisse für 64 Klassen selbst wenn der trainierte Datensatz, wie bei CIFAR10, weniger besitzt. Für jede dieser Klassen wird das Konfidenzlevel als 16 Bit Integer im Wertebereich 0 bis 511 gespeichert. Das Konfidenzlevel für die Klasse k vom Bild b kann also mit $64 * b + 2 * k$ Byte Versatz von der Startadresse der Ergebnisse ausgelesen werden.
Der Nachfolger FINN [13] des BNN-PYNQ besitzt erwartungsgemäß sehr ähnliche Schnittstellen, aber bietet deutlich mehr Flexibilität. Anstatt die Gewichte und Schwellwerte über AXI4-Lite zu laden, können diese direkt in die Schaltung integriert werden, sodass diese anfängliche Konfiguration nicht mehr nötig ist, was jedoch eine komplett neue Übersetzung bei der Änderung der Trainingsdaten erfordert. Die Eingabe von Bilddaten, die Inferenz und die Ausgabe der Resultate wurde in drei Komponenten unterteilt. Zwingend notwendig ist nur die Komponente für die Inferenz, welche für die Bilddaten und Resultate jeweils eine AXI4-Stream Schnittstelle besitzt. Für die Bilddaten hat diese Schnittstelle beim offiziellen Beispiel für ein CNN eine Datenbreite von einem Byte, sodass die Farbkanäle eines Pixels in der Reihenfolge Rot-Grün-Blau und mehrere Pixel zeilenweise übertragen werden. Der Ausgabestream kann bei der Übersetzung des neuronalen Netzes konfiguriert werden. Soll bei einer Klassifizierung nur der Index der Klasse mit der höchsten Konfidenze ausgegeben werden, dann ist die Datenbreite ein Byte. Jeder Transfer im Stream ist genau ein Resultat für ein komplettes Bild. Für die Konfidenz jeder Klasse ist die Datenbreite für n Klassen genau n Bytes. Jede Konfidenz ist ein vorzeichenloser 16 Bit Integer, weswegen zwei Transfers im Stream für die Resultate eines Bildes benötigt werden.
Falls die Daten statt aus einem AXI4-Stream aus einem Speicher gelesen werden sollen, kommen die zwei anderen Kompontenten zum Einsatz, welche einen „Direct Memory Access“ (DMA) implementieren und zwischen einem Datenstrom und einer Speicherdarstellung konvertieren können. Die erste Komponente liest Bilddaten sequentiell aus dem Speicher aus und gibt diese als AXI4-Stream weiter. Die zweite Komponente ist die Inverse der Ersten und schreibt einen AXI4-Stream mit Konfidenzen oder erkannter Klassen sequentiell in einen Speicher. Das Format der Daten im Speicher ist identisch zum BNN-PYNQ. Beide Komponenten werden über AXI4-Lite konfiguriert und besitzen dafür ein

Kontroll-/Statusregister, ein Register für die Startadresse im Speicher und ein Register für die Anzahl Bilder/Resultate. Die Ausführung wird über das Kontroll-/Statusregister durch Schreiben eines Bits gestartet. Ob die Ausführung beendet wurde, kann über ein weiteres Bit im gleichen Register abgerufen werden.

2.7 Blockdiagramme

Sogenannte „Block Designs“ (zu deutsch „Blockentwürfe“ oder „Blockdiagramme“) sind ein starkes Werkzeug, welches in der Vivado Entwicklungsumgebung von Xilinx verfügbar ist [46].
Blockdiagramme erlauben es mit einer grafischen Oberfläche mehrere Komponenten zu verbinden. Diese Komponenten können beispielsweise die Form von sogenannten „Intellectual-Property-Cores“ (IP-Cores) oder einfachen Quellcodedateien in Hardwarebschreibungssprachen, wie VHDL oder Verilog, annehmen. Der Begriff IP-Core steht wortwörtlich übersetzt für „Geistiges Eigentum“-Core, aber kann einfacherweise, als eine verpackte Schaltung mit definierten Schnittstellen betrachtet werden. Komponenten werden in der Oberfläche als einfache Blöcke dargestellt, wie beispielsweise in Abb. 6 für einen einfachen Addierer. Die Konfiguration der Blöcke erfolgt ebenfalls über eine grafische Oberfläche. Für die Programmiersprache VHDL sind dort beispielsweise die Generics der Entity einstellbar. Anspruchsvolle IP-Cores können hier umfängliche Bedienelemente, wie Dropdowns oder Grafiken, darstellen.

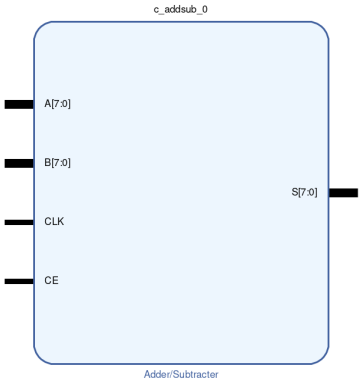


Abbildung 6: Blockdiagramm - Block eines Addierers/Subtrahierers

Mehrere Blöcke können über das Ziehen von Leitungen verbunden werden, was teilweise auch automatisch durchgeführt werden kann, falls Quelle und Ziel einer Verbindung eindeutig sind. Schnittstellen mit mehreren Leitung werden automatisch zu einem Leitungsbündel zusammengefasst, wie beim Addierer für die Bitvektoren zu sehen. Für komplexere Schnittstellen, wie AXI4, AXI4-Lite oder AXI4-Stream, sind zusätzlich noch weitere Automatisierungen möglich. Beispielsweise können automatisch die nötigen Interconnects (ebenfalls als Blöcke) in der Schaltung platziert werden und zugehörige Takt- und Resetsignale erkannt werden.
Mehrere Blöcke in einem Blockdiagramm können zur logischen Trennung in einer Hierarchie zusammengefasst werden oder mehrere Blockdiagramme verschachtelt werden. Der gesamte Funktionsumfang der Blockdiagramme übersteigt bei weitem diesem Abschnitt und kann unter anderem in [46] nachgelesen werden.
Falls sich AXI4 oder AXI4-Lite Peripherie im Blockdiagramm befinden, kann ein grafischer Adresseditor genutzt werden, um den Adressraum zu konfigurieren. Basierend auf dieser Konfiguration werden die Interconnects automatisch parametrisiert. Dieser Adresseditor ist in Abb. 7 zu sehen.

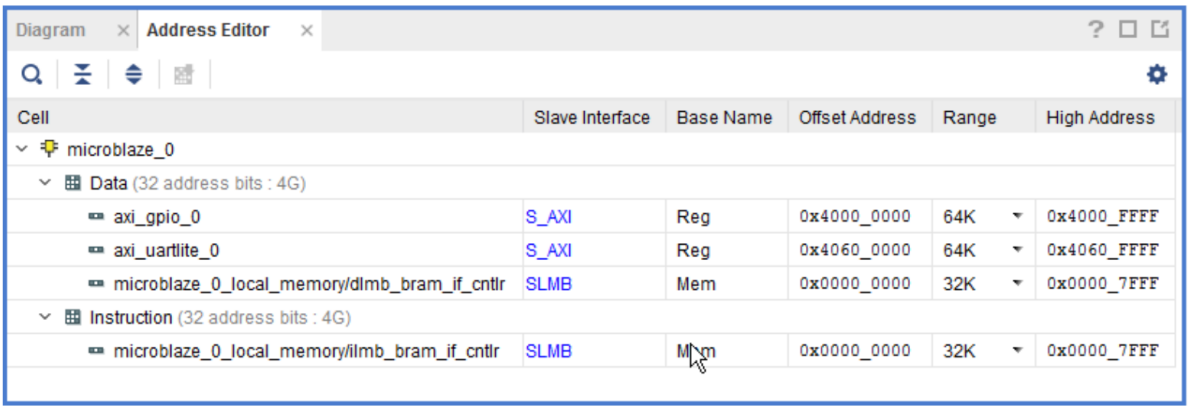


Abbildung 7: Blockdiagramm - Adresseditor [46]

Für Blockdiagramme kann eine XML-Datei mit der platzierten Peripherie und derer Adressen exportiert werden. Diese Datei kann beispielsweise für die CPU des Zynq-SoC genutzt werden, um automatisch entsprechende Header-Dateien für C/C++ zu generieren oder automatisch entsprechende Treiber für Abstraktionen in Python zu laden, damit die Peripherie auf dem FPGA adressiert werden kann.

3 Verwandte Arbeiten

Im Rahmen einer Projektarbeit unter der Betreuung von Prof. Dr. Jan Bredereke wurde im Wintersemester 2020/21 an der Hochschule Bremen eine Hardware- und Softwareplattform zur Erforschung von KNN auf FPGAs erdacht und realisiert [15], woran auch der Autor dieser Arbeit beteiligt war. Diese Plattform basiert auf einem Modellfahrzeug der Firma Elegoo. Der vom Hersteller vorgesehen Mikrocontroller wurde duch eine PYNQ-Z1 Entwicklungsplatine der Firma Digilent ersetzt. Zusätzlich zum Austausch des Mikrocontrollers wurde eine einfache USB-Webcam am Fahrzeug montiert und an die USB-Schnittstelle der CPU angebunden.
Aufbauend auf der realisierten Hardwareplattform wurde auf dem FPGA des SoCs ein in Hardware implementiertes KNN, das BNN-PYNQ, geladen. Konkret wurde dabei ein mitgeliefertes vortrainiertes CNN zur Erkennung deutscher Verkehrsschilder verwendet. Ziel war es dann ein Stoppschild in einem Bild zu lokalisieren und das Fahrzeug in Richtung dieses Schildes zu fahren. Die Lokalisierung wurde durch die Unterteilung des Bildes in Tiles umgesetzt, welche dazu auf eine Auflösung von 32x32 Pixeln skaliert, vom BNN-PYNQ klassifiziert und zuletzt anhand der jeweiligen Konfidenz der Klassifizierung ausgewertet wurden. Die Position des Stoppschildes im Bild ergibt sich durch die Position der Tiles mit der höchsten Konfidenz. Das Tiling wurde in Software auf dem Prozessor des SoCs umgesetzt. Im Vergleich zu einer mitgelieferten Beispielimplementierung der BNN-PYNQ Forschungsgruppe für Tiling mithilfe von Python, ist die im Projekt erarbeitete Version durch eine hardwarenahe Implementierung um ein Vielfaches schneller und schafft eine Verarbeitung mehrerer Bilder pro Sekunde, jedoch reicht selbst das nur ansatzweise aus, um die Fahrzeuglenkung dynamisch zu regeln. Ein Grund für den begrenzten Durchsatz ist der relativ schwache Prozessor, welcher für das Tiling nicht geeignet ist, da er keine Hardwarebeschleunigung für Bildverarbeitung besitzt und somit für die Aufteilung des Bildes, sowie die anschließende Skalierung der Tiles, die CPU bereits vollends auslastet. Ein zweiter Grund ist die Anbindung des KNN auf dem FPGA. Hierfür werden die generierten Tiles an einer speziellen physikalischen Adresse im DDR3-Speicher abgelegt, woraufhin das KNN über einen im Arbeitsspeicher abgebildeten Register instruiert wird mit der Klassifizierung zu beginnen und dafür die Bilder (Tiles) wieder aus dem Arbeitsspeicher ausliest und die Ergebnisse der Klassifizierung wieder in diesem ablegt, was durch die langsame Speicherschnittstelle womöglich die Ausführungsgeschwindigkeit einschränkt.
Eine weitere Implementation mit reellen Bilddaten ist BinaryEye [24], welche die Echtzeitverarbeitung von einem 32x32 Pixel großen Bereichs eines proprietären Bildsensors erlaubt. Die Bilddaten werden über eine parallele Schnittstelle vom FPGA aufgenommen und in einem Datenstrom an eine angepasste Version des BNN-PYNQ [41] zur Schriftzeichenerkennung basierend auf dem MNIST Datensatz übertragen. In dieser Konfiguration kann auf die Verwendung von externen dynamischen RAM verzichtet werden und so eine geringe Leistungsaufnahme von 13,78 W und ein Datendurchsatz von ca. 20.000 Bildern/Sekunde für das Gesamtsystem erreicht werden. Dabei ist zu beachten, dass die LFC-Topologie von BNN-PYNQ verwendet wurde, welche im Vergleich zu der CNV-Topologie eine deutlich geringer Ausführungszeit besitzt, da anstatt eines CNN ein simpleres Netz verwendet wird. Weiterhin kann nur genau ein Bildbereich des Sensors ohne Skalierung verwendet werden, sodass die Kamera extern auf die Schriftzeichen ausgerichtet werden müsste.
Für wenige Tiles existiert der „Multi Scaler“ als fertiger IP-Core von Xilinx [28]. Dieser IP-Core kann für bis zu acht Skalierungen konfiguriert werden, deren Eingangsbilder aus einem Speicher ausgelesen werden. Die Reduktion auf einen Teilbereich des Bildes kann durch die Konfiguration der Schrittweite und Anzahl Zeilen beim Auslesen konfiguriert werden. Ein großer Nachteil ist jedoch der sehr hohe Ressourcenverbrauch. Eine Messung des Herstellers [4] hat gezeigt, dass selbst bei sparsamster Konfiguration über 70 Block RAM Blöcke á 36 Kbit benötigt werden. Auf schwacher Hardware, wie dem Zynq-7020, bedeutet dies eine Belegung von über 50%, was keinen Platz mehr für ein KNN lässt.

4 Konzeption einer Lösung

In diesem Kapitel wird eine theoretische Umsetzung des Tilings auf einem FPGA konzeptioniert. Dazu werden zuerst in Kap. 4.1 die ausschlaggebenden Parameter für ein möglichst flexibles Tiling erörtert. Vor der eigentlichen Konzeption werden in Kap. 4.2 die Limitierungen der verwendeten Hardware betrachtet, um sicherzustellen, dass die theoretische Lösung auch praktisch umgesetzt werden könnte. Auf Basis dieser Informationen werden in Kap. 4.3 zwei Systemarchitekturen mit unterschiedlicher Anbindung der Bilddaten betrachtet und der vielversprechendere Ansatz weiter ausgebaut. Nachdem die Systemarchitektur beschrieben wurde, folgt zuletzt in Kap. 4.4 und 4.5 die Beschreibung der Komponenten in der Systemarchitektur.

4.1 Tilingparameter

Tiling beschreibt lediglich, dass ein Bildbereich ausgeschnitten und eventuell skaliert wird. Welche Bildbereiche und welche Skalierung benötigt wird, hängt davon ab, wie die Objekterkennung mit dem Tiling umgesetzt wird. Dies könnte beispielsweise als Sliding Window oder mit Region Proposals geschehen. Eventuell soll auch einfach nur genau ein Bereich in der Mitte des Bildes betrachtet werden. Damit möglichst viele dieser Ansätze mit der neuen Lösung auf dem FPGA umsetzbar sind, muss diese möglichst dynamisch konfigurierbar sein. Die Tabelle 1 enthält daher alle Parameter, die nötig sind um beliebige Bildbereiche auszuschneiden und zu skalieren.

Nr.	Parameter	Beschreibung
P1	Auflösung des Gesamtbildes	Abhängig von der Bildquelle. Typische Werte: Breite x Höhe, 320x240, 640x480, 1920x1080
P2	Farbtiefe	Anzahl und Auflösung der Farbkanäle einzelner Pixel. Typische Werte: 3 Kanäle (Rot-Grün-Blau) mit 8 Bit/Kanal 1 Kanal (Graustufen) mit 8 Bit
P3	Anzahl Tiles	Abhängig von der verwendeten Technik zur Bestimmung der interessanten Regionen.
P4	Positionen der Bildregionen	Koordinaten der Bildregionen der einzelnen Tiles im Gesamtbild mit Koordinatenursprung oben-links.
P5	Größe der Bildregionen	Höhe und Breite der Bildregionen in Pixel mit Richtung unten-rechts relativ zum Koordinatenursprung.
P6	Skalierte Auflösung	Zielauflösung für das verwendete KNN. Falls das KNN verschiedene Eingangsauflösungen unterstützt, kann dieser Schritt potentiell entfallen.
P7	Skalierungsalgorithmus	Die Qualität und der Rechenaufwand der Skalierung hängt vom verwendeten Algorithmus ab. Typische Werte: Bilinear oder Bikubisch

Tabelle 1: Tilingparameter

4.2 Limitierungen der verwendeten Hardware

Diese Arbeit verwendet bewusst schwache Hardware, um einen Einsatz in restriktiven Bereichen, wie auf der Edge zu erlauben. Dementsprechend fiel die Wahl der Hardware auf ein Modell der unteren Mittelklasse des Herstellers Xilinx, welches sowohl stromsparend als auch relativ kostengünstig ist, aber auch im Vergleich zu stärkerer Hardware weniger und schlechtere Ressourcen bietet.
Die **CPU** des SoC ist ein schwacher zweikern Arm-v9 Prozessor und wird in dieser Arbeit zur Konfiguration verwendet, da der Fokus auf der Verwendung des FPGAs liegt, wodurch die Einschränkungen der CPU vernachlässigbar sind.
Der verfügbare **Block RAM** auf dem FPGA ist mit 630 KB relativ klein, aber dafür sehr schnell bei geringer Latenz und einer flexiblen Zuordnung durch die Aufteilbarkeit in viele Blöcke. Das zum

Vergleich genutzte BNN-PYNQ belegt bei der Implementierung eines größeren Netzes, einem CNN, ungefähr 80% des BRAMs für die Speicherung von Gewichten, Schwellwerten und Zwischenergebnissen. Eine Verringerung des Speicherverbrauchs wäre möglich, aber würde potentiell die Qualität der Klassifizierungsergebnisse verschlechtern, weswegen die Implementierung des Tilings auf die 20% verbleibenden BRAM (ca. 126 KB) beschränkt wird.
Der **DDR3-Hauptspeicher** fällt mit 512 MB relativ großzügig aus und ist über einen Speichercontroller mittels AXI4 an die CPU, den FPGA und diverse weitere Peripherie verbunden. Alle angeschlossenen Komponenten teilen sich die Schnittstellen des Speichers, wodurch Zugriffe einer Komponente die Zugriffe einer anderen potentiell verzögern könnnen. Zusätzlich zu einer Verzögerung durch andere Komponenten sind Speicherzugriffe durch die Verwendung von dynamischem Speicher außerdem deutlich langsamer bei höherem Stromverbrauch im Vergleich zum Block RAM.
Der Verbrauch von Logikressourcen auf dem FPGA durch das BNN-PYNQ, wie **LUTs** oder **Flip Flops**, fällt deutlich geringer aus und lässt mehr Spielraum für das neue Tiling. Zur Verminderung der Stromverbrauchs wird allerdings auch hier versucht die verwendeten Ressourcen zu minimieren.
Die **Taktfrequenz** der Schaltung auf dem FPGA ist begrenzt durch den Aufbau seiner internen Komponenten und den Fertigungsprozess. Xilinx produziert die Chips in verschiedenen Geschwindigkeitsklassen. Die verwendete Platine besitzt einen Chip der untersten Geschwindigkeitsklasse, weshalb hohe Taktfrequenzen nur mit sehr kurzen Pfadlängen funktionieren. Die Standartfrequenz sind recht konservative 100 Mhz. Für das Tiling wird versucht gute Ergebnisse mit dieser Frequenz zu erreichen.

4.3 Systemarchitektur

Die Abb. 8 ordnet das Tiling als neue Komponente in ein Gesamtsystem aus Komponenten (Blöcke) und Schnittstellen (Linien) ein. Die *Datenquelle* ist das zu verarbeitende Videosignal, welches folgend von *Tiling* verarbeitet wird und die entstandenen Einzelbilder an ein KNN als *Datensenke* weitergegeben werden.

Abbildung 8: Architektur - Einordnung des Tilings

Der Datenfluss zur Datensenke wird durch das verwendete BNN-PYNQ vorgegeben, welches die Daten in einem über AXI4 adressierbaren Speicher erwartet. Der Nachfolger FINN kann zusätzlich zum Speicherabruf die Bilder auch direkt über einen Datenstrom annehmen (siehe Kap. 2.6). Die Anbindungsmöglichkeit über den Datenstrom wäre vorteilhaft, da die Speicherzugriffe entfallen, allerdings nicht zwingend notwendig.
Für den Datenfluss der Bilddaten zum Tiling wurden zwei Optionen betrachtet: Verwendung eines Datenstroms oder einen adressierbaren Speichers in dem die Daten zwischengespeichert werden.

4.3.1 Datenstrombasierte Lösung

In einem ersten Anlauf wurde versucht die Bilddaten aus einem sequentiellen Datenstrom heraus zu verarbeiten. Hauptargument war eine höhere mögliche Bandbreite bei geringerer Latenz und weniger Stromverbrauch durch die Vermeidung von Zugriffen auf einen vergleichsweise langsamen (externen) Speicher. Dieser Ansatz lässt sich allerdings nur unter bestimmten Umständen verwenden, da er folgende Einschränkungen mit sich bringt:

Daten müssen sequentiell verarbeitet werden:
Unter der Annahme, dass der Datenstrom keine gesamten Bilder, sondern nur Bildfragmente in einem Takt überträgt (siehe Kap. 2.2), können diese nur sequentiell verarbeitet werden. Anders ausgedrückt steht zu jedem Zeitpunkt nur ein Fragment zur Verfügung. Zur Veranschaulichung soll ein Bild in vier Teilbilder aufgeteilt werden, die in einem Datenstrom am Ende sequentiell ausgegeben werden sollen. Die Bilddaten werden mit einzelnen Pixeln als Fragmente zeilenweise übertragen. In Abb. 9 ist diese Transformation beispielhaft dargestellt. Das Originalbild hat eine Auflösung von 4x4 Pixeln, die als

Eingangsbildsignal Zeile für Zeile übertragen werden. Als Ausgangssignal werden die 2x2 Pixel großen Viertel sequentiell erwartet. Zur Umwandlung vom Eingangssignal zum Ausgangsbildsignal müssen die Daten neu geordnet werden. In der Zwischenrepräsentation wird daher der zeilenweise Datenstrom in vier Datenströme für die Bildsegmente aufgeteilt. Hier wird deutlich, dass sich die Segmente innerhalb einer Zeile überlappen. Da für das Ausgangssignal beispielsweise das gelbe Segment nach dem Grünen erwartet wird, müssen dessen Daten solange gepuffert werden, bis das vorherige Segment komplett übertragen wurde. Für das Beispiel ist die Größe und Anzahl der Puffer überschaubar, jedoch ändert sich dies schnell, sobald sich Auflösung oder die Anzahl der Segmente erhöhen.

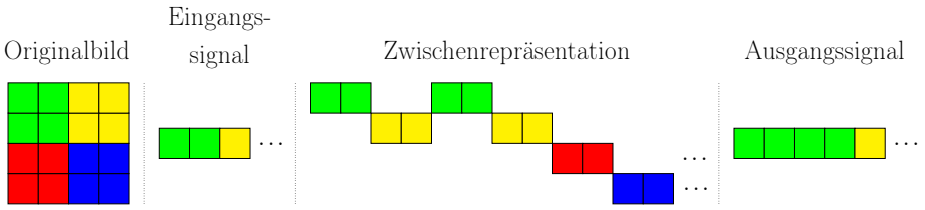


Abbildung 9: Bildaufteilung mit einem Datenstrom

Keine starken Verzögerungen möglich:
Schnittstellen für Videosignale, wie HDMI oder VGA, liefern die Bilddaten meist in festen Abständen ohne die Möglichkeit die Übertragung zu verzögern. Die Daten müssen dementsprechend in diesen Intervallen angenommen werden, da sie ansonsten verloren gehen. Zweitweise Verzögerungen können durch einen Puffer ausgeglichen werden, dessen Größe durch die Länge der maximalen Verzögerung gewählt werden muss. Können Verzögerungen zwischenzeitlich nicht wieder ausgeglichen werden, dann erreicht die Größe des Puffers schnell das gesamte Bild und die Sinnhaftigkeit des Datenstroms ist zu hinterfragen.

Schlechte Ausnutzung der verfügbaren Bandbreite:
In Videosignalen werden Daten meist nicht kontinuierlich, sondern nur zeitweise übertragen, was sich aus der Generierung und Darstellung ergibt. Beispielsweise soll ein Signal betrachtet werden, welches Bilder mit einer Auflösung von 640x480 Pixeln und einer Bildwiederholrate von 30 Bildern/Sekunde überträgt. Innerhalb der Übertragung eines Bildes gibt es zur Vereinfachung keine Verzögerungen. Die Schnittstelle für den Datenstrom kann bei einer Taktfrequenz von 100 Mhz ein Pixel pro steigender Taktflanke übertragen. Die Übertragung benötigt also $640x480 = 307.200$ Taktperioden für ein Bild und $30 * 307.200 = 9.216.000$ Taktperioden für alle Bilder innerhalb einer Sekunde. Innerhalb der restlichen 90.784.000 Taktperioden werden keine Daten übertragen, wodurch sich eine Auslastung von lediglich 9,21% ergibt. Weil durch die oben genannten Gründe die Verarbeitung synchron zum Videosignal erfolgen muss, muss die weitere Verarbeitung entweder innerhalb der aktiven Zeitintervalle erfolgen oder die Daten gepuffert werden und zwischen den Zeitintervallen fortgesetzt werden.

Schlechte Konfigurierbarkeit:
Aus den oberen Punkten ergibt sich, dass die Verarbeitung einer harten Echtzeitbedingung unterliegt. Diese Bedingung fordert, dass alle Bilddaten während ihrer Übertragung verarbeitet werden. Die Echtzeitbedingung ist hart, da bei nichteinhaltung Daten verloren gehen und Teile der Ergebnisse ungültig werden. Wie fatal dies für die weitere Verarbeitung ist, wird hier nicht weiter betrachtet.
Die Einhaltung dieser Echtzeitbedinung wird dann schwierig, wenn sich auszuschneidene Bildbereiche überschneiden. Zur Veranschaulichung soll eine Lokalisierung von Objekten mittels eines Sliding Windows (siehe 2.1) betrachtet werden, welches für die Bildregionen eine Art Schablone fester Größe über das Bild schiebt. Ist der Versatz der Verschiebung kleiner als die Breite/Höhe der Schablone, dann überschneiden sich die Bildbereiche, was häufig gewünscht ist, um auch Objekte zu erkennen, die sonst genau zwischen zwei Bereichen liegen.
Die einzelnen Bildbereiche werden zur Verarbeitung durch das KNN sequentiell im Speicher erwartet. Überschneiden sich nun zwei Regionen, dann müssen die sich überschneidenen Pixel an mehreren Adressen im Speicher abgelegt werden. Da die Anzahl Schnittstellen des Speichers stark begrenzt wird,

erfordert dies mehrere Schreibvorgänge, welche alle abgeschlossen sein müssen, bevor das nächste Pixel eintrifft. Weil dieser Zeitraum begrenzt ist, ist nur eine gewisse Anzahl Überschneidungen möglich, weshalb die Konfigurierbarkeit stark leidet. Ein weiteres Problem ist der Abruf dieser Konfiguration, da zu jedem Pixel bekannt sein muss, zu welchen Bildbereichen er gehört und basierend darauf an welcher Adresse er abgelegt werden soll. Sollen allen Parameter parallel abrufbar sein, erfordert dies viele Ressourcen auf dem FPGA. Wird wiederum ein Speicher verwendet, müssen die Parameter sequentiell ausgelesen werden, wodurch weitere kostbare Zeit verloren geht.

4.3.2 Speicherbasierte Lösung

Im Vergleich zum datenstrombasierten Ansatz kann bei einer zufällig adressierbaren Lösung mittels eines Speichers in beliebiger Reihenfolge auf die Daten zugegriffen werden. Voraussetzung ist ein Speicher mit ausreichender Größe für Bilddaten, welcher glücklicherweise durch 512 MB DDR3-Speicher auf der verwendeten Platine zur Verfügung steht. Dieser Speicher ist über einen Speichercontroller an einen AXI4-Bus angebunden und kann daher von mehreren Komponenten verwendet werden. Entsprechend kann die in Abb. 8 eingeführte Architektur, wie in Abb. 10 zu sehen, mittels eines Speichers zur Kommunikation umgesetzt werden. Anstelle der Punkt-zu-Punkt Verbindungen des logischen AXI4-Busses wurde der Bus zur Vereinfachung als Leitungsbündel dargestellt. Die Daten eines externen Videosignals werden von der Videosignalanbindung in den Speicher geschrieben und vom Tiling abgerufen. Die erzeugten Tiles werden im Speicher abgelegt und dort vom KNN ausgelesen (Variante A) oder alternativ innerhalb des FPGAs direkt per Datenstrom weitergegeben (Variante B). Unabhängig von der Übertragung der Tiles werden diese vom KNN klassifiziert und die Inferenzergebnisse im Speicher abgelegt. Diese Ergebnisse könnten nun beispielsweise von der CPU ausgewertet werden, was allerdings nicht mehr Bestandteil dieser Architektur ist.

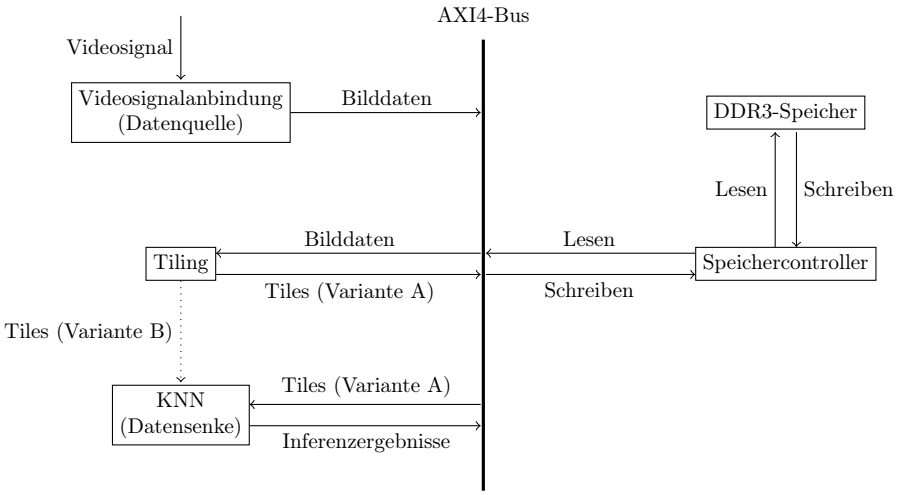


Abbildung 10: Architektur - Einordnung des Tilings in eine speicherbasierte Lösung

Durch die Verwendung des Speichers als Puffer sind die einzelnen Komponenten im Gesamtsystem voneinander entkoppelt und können unbeinträchtigt von Verzögerungen einer anderen Komponte ihre Verarbeitung vollziehen. Lediglich die maximale Gesamtdauer für die Verarbeitung eines Bildes ist weiterhin durch die Bildwiederholrate des Videosignals vorgegeben. Im Vergleich zur datenstrombasierten Lösung ist diese Echtzeitbedingung allerdings weicher, da die Verarbeitung eines Bildes beliebig lange dauern kann, wodurch die Ergebnisse für dieses stets korrekt sind. Falls das nächste Bild vor dem Ende der Verarbeitung eintrifft, kann dieses verworfen werden, wodurch lediglich die Bildwiederholrate verringert wird, anstatt dass ein Bild nur partiell verarbeitet wird.

4.3.3 Konfiguration mittels AXI4-Lite

Die Konfiguration des Tilings erfolgt analog zum verwendeten KNN über AXI4-Lite, was es erlaubt die Konfiguration sowohl von innerhalb, als auch außerhalb des FPGAs durch beispielsweise die CPU auf dem SoC durchzuführen. Die Konfiguration über die CPU hat den Vorteil, dass sich Änderungen durch kürzere Kompilierungszeiten, als bei den FPGA Werkzeugen, schneller umsetzen lassen. Die Wahl für AXI4-Lite im Vergleich zu AXI4-Stream ist damit begründet, dass die Konfiguration einfacherweise über Kontroll-/Statusregister umgesetzt werden soll, was durch eine zufällige Adressierbarkeit deutlich einfacher ausfällt. Im Vergleich zur schnelleren AXI4-Schnittstelle wurde AXI4-Lite gewählt, da es durch eine verminderte Menge an Signalen weniger Hardwareressourcen verbraucht und die höhere Geschwindigkeit für die Konfiguration nicht als nötig angesehen wird. Weiterhin hat AXI4-Lite den Vorteil, dass innerhalb des verwendeten SoC die Busse für Konfiguration und schnelle Datentransfers getrennt sind und so die Konfiguration die Speicherzugriffe nicht beeinträchtigt.
In Abb. 11 wurden die AXI4-Lite Schnittstellen in die Gesamtarchitektur eingezeichnet. Der Bus wurde zur einfacheren Darstellung erneut als Leitungsbündel statt den tatsächlichen Punkt-zu-Punkt Verbindungen dargestellt. Die Kommunikation über AXI4-Lite ist in beide Richtungen möglich, allerdings wird die AXI4-Lite Schnittstelle hier im Slave Modus verwendet, sodass nur durch einen externen Master Transaktionen initialisiert werden, womit jegliche Datenübertragung von Slave zu Master Statusinformationen enthält und von Master zu Slave konfiguriert wird. Die Übertragung findet immer in Form von Registern statt, wobei ein Register sowohl Konfigurations-, als auch Statusinformationen, enthalten kann. Beispielsweise wäre dies für das Setzen eines Startbit durch den Master der Fall, welches nach Beendigung der Verarbeitung vom Slave zurückgesetzt wird.

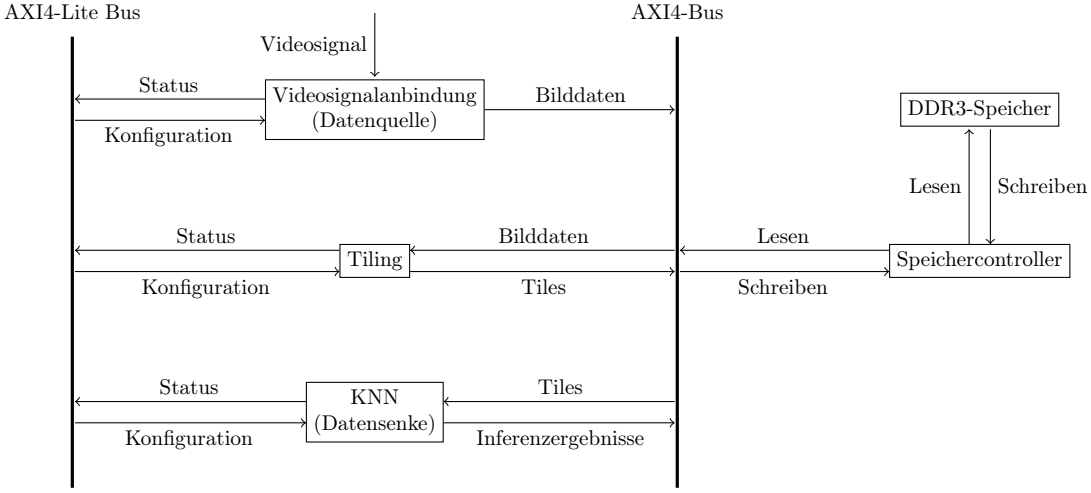


Abbildung 11: Architektur - Konfiguration

Die Architekturbeschreibung der Videosignalanbindung und des Tilings folgt in den nächsten Kapiteln. Die Schnittstellen des KNN können in Kap. 2.6 nachgelesen werden. Der Speichercontroller ist fest im SoC verbaut und wird nicht weiter betrachtet.

4.4 Architektur der Videosignalanbindung

In diesem Abschnitt wird die Architektur zur Anbindung eines externen Videosignals an das Tiling beschrieben. Durch die Verwendung eines FPGAs kann nahezu jede Bildquelle über diverse Schnittstellen, wie HDMI, VGA oder direkt am Bildsensor, angebunden werden. Da diese Schnittstellen unterschiedliche Signale und Protokolle verwenden, würde die Auslegung des Tilings für genau eine Schnittstellen andere ausschließen. Aus diesem Grund werden mehrere Umwandlungsschritte vollzogen, um das Videosignal in ein einheitliches Format zu bringen. Durch die mehrstufige Umwandlung muss beim Austausch der externen Videoschnittstelle nicht die gesamte Anbindung, sondern nur die ersten Stufen,

ausgetauscht werden. Diese Verarbeitungspipeline ist in Abb. 12 dargestellt.

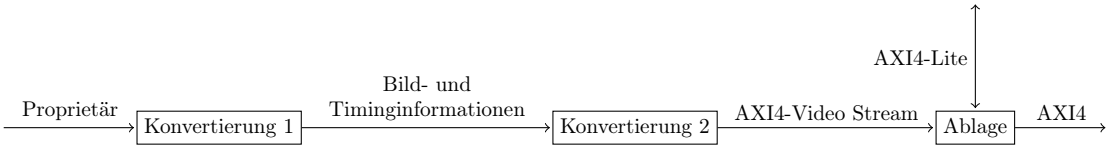


Abbildung 12: Architektur - Videosignalanbindung

Im ersten Schritt wird das proprietäre externe Signal in Bild- und Timinginformationen aufgeteilt. Die ursprüngliche Übertragungsart, wie etwa ein differentielles Paar bei HDMI oder Synchronisationssignale mit einem implizitem Takt bei VGA, wird dabei abstrahiert. Bild- und die proprietären Timinginformation werden folgend in einen AXI4-Video Stream zusammengefasst, der nur noch Bilddaten und Synchronisationssignale für Bildanfang und Zeilenende besitzt. Wird der datenstrombasierte Ansatz der Systemarchitektur verwendet, dann ist keine weitere Umwandlung nötig. Falls der speicherbasierte Ansatz verwendet wird, muss der AXI4-Video Stream zuletzt im Speicher abgelegt werden, was über die AXI4 Schnittstelle erfolgt. Die Konfiguration dieser Ablage erfolgt über AXI4-Lite, damit die Adresse der Bilddaten im Speicher eingestellt werden kann und Statusinformationen abgerufen werden können. Für die Konfiguration der ersten Konvertierungsstufen wurden keine Annahmen getroffen, da diese stark von der verwendeten Schnittstelle abhängt. Womöglich existiert für die Umwandlung für einige Schnittstellen auch eine direkte Umwandlung in einen AXI4-Video Stream oder AXI4, sodass mehrere Stufen zusammengefasst werden können.

4.5 Architektur des Tilings

In Abb. 13 ist die Teilarchitektur des Tilings dargestellt. Eingang des Tilings sind die zu verarbeitenden Bilddaten aus der Videosignalanbindung. Der *Befehlsgenerator* teilt dem *Extraktor* mit, welche Bildbereiche ausgeschnitten werden sollen. Diese Bildbereiche werden vom *Skalierer* in ihrer Größe angepasst und zuletzt von *Ablage* im Speicher abgelegt, damit die generierten Bilder/Tiles vom KNN verarbeitet werden können. Falls das neue FINN verwendet wird, kann die Ablage im Speicher entfallen und der Skalierer direkt an dieses angebunden werden. Die Abarbeitung der zu extrahierenden Tiles erfolgt sequentiell. Für eine parallele Berechnung können mehrere Instanzen des Tilings auf dem FPGA platziert werden. Der Befehlsgenerator ist dann dafür zuständig die Arbeit auf die einzelnen Instanzen zu verteilen oder es werden mehrere Befehlsgeneratoren platziert.

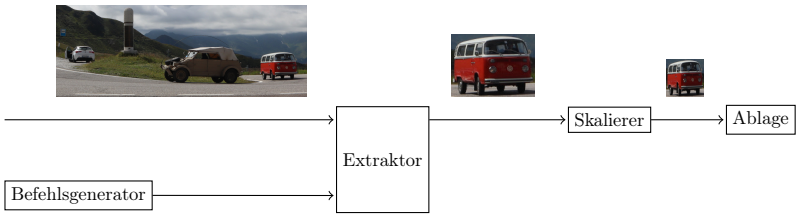


Abbildung 13: Grobarchitektur des Tilings

In Abb. 14 werden die Komponenten nun um konkrete Schnittstellen ergänzt, die in den folgenden Abschnitten für die einzelnen Komponenten genauer erläutert werden. Die Verbindung zwischen Schnittstellen sind durch Pfeile dargestellt, welche mit den enthaltenen Daten und der Art der Schnitstelle in Klammern beschriftet sind. Verbindungen, die in keiner anderen Komponente enden, sind externe Schnittstellen der Tilingkomponente und dienen zur Anbindung in ein Gesamtsystem.

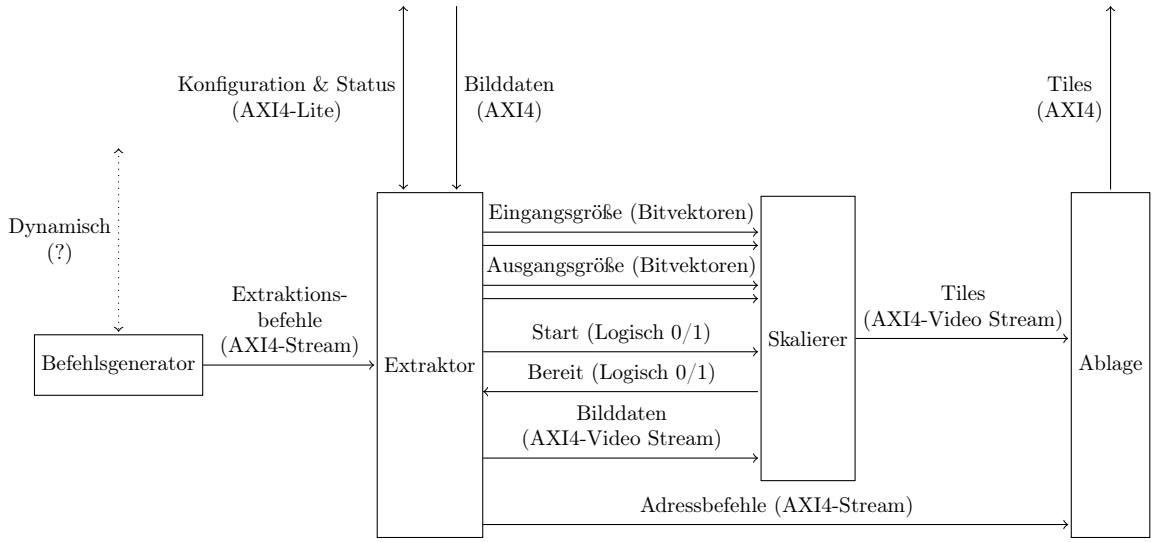


Abbildung 14: Schnittstellen der Komponenten des Tilings

Zur Kommunikation werden alle drei verfügbaren Schnittstellen der AXI4-Spezifikation genutzt. Die vollwertige AXI4-Schnittstelle wird für Zugriffe auf den DDR3-Speicher genutzt und AXI4-Lite für die Konfiguration und Statusabruf der Komponenten über Register. Zur Kommunikation zwischen Komponenten wird hauptsächlich der AXI4-Stream genutzt, welcher die Daten synchron zum Takt weitergibt. Die Datenströme können beidseitig blockiert werden, sodass die Komponenten beliebige Verzögerungen enhalten können. Dabei ist zu beachten, dass die Verzögerung einer Komponente die gesamte Komponente des Tilings verzögert.

4.5.1 Extraktor

Die Aufgabe des Extraktors ist das Auslesen von Bereichen eines Bildes aus dem Speicher, wofür eine AXI4-Schnittstelle genutzt wird. Über Konfigurationsregister mittels AXI4-Lite wird dem Extraktor die Startadresse der Bilddaten im Speicher, sowie die Höhe/Breite des Gesamtbildes mitgeteilt. Über eine AXI4-Stream Schnittstelle wird dem Extraktor folgend mitgeteilt, welcher Bereich zu extrahieren und über eine AXI4-Video Stream Schnittstelle auszugeben sind. Die Befehle haben die Form eines Bitvektors mit einer Länge von 64 (siehe Abb. 15). Alle Felder sind vorzeichenlose 16 Bit Integerwerte.

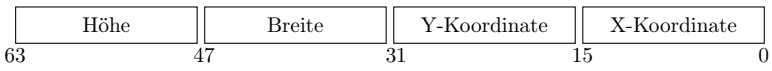


Abbildung 15: Format der Extraktionsbefehle

Gleichzeitig zur Ausgabe der Bilddaten werden über einen weiteren AXI4-Stream sogenannte „Adressbefehle“ an die Ablage weitergegeben, welche die Länge des ausgegebenen Bildes in Pixeln und die Zieladresse im Speicher enthält. Das genaue Format ist in Abb. 16 dargestellt. Beide Felder sind vorzeichenlose 32 Bit Integerwerte. Die Zieladresse wird in Bytes angegeben. Die Befehle werden direkt vom Extraktor an die Ablage, statt über den Skalierer, weitergegeben, damit keine Anpassung des Skalierers notwendig ist. Die Bildgröße könnte theoretisch von der Ablage anhand der Synchronisationssignale im Videostream erkannt werden, allerdings fehlt diesem ein Signal für das Ende eines Bildes, sodass erst zum Anfang des nächsten Bildes feststehen würden, wann das vorherige endet.

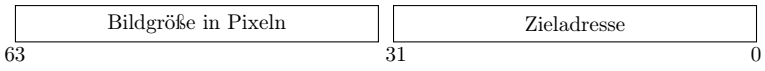


Abbildung 16: Format der Adressbefehle

4.5.2 Befehlsgenerator

Die Aufgabe des Befehlsgenerators ist die Erzeugung der Koordinaten und Größen von zu extrahierenden Bildbereichen, welche an den Extraktor über einen AXI4-Stream weitergegeben werden. Die Art der Generierung kann anwendungsfallabhängig in der Implementation gewählt werden, weshalb die weiteren Schnittstellen frei wählbar sind. Folgend werden beispielhaft drei mögliche Umsetzungen vorgestellt.
Eine arithemtische Umsetzung könnte einen internen Zähler für die Anzahl der übermittelten Befehle führen. Anhand dieses Zählers i kann der Bildbereich mittels vier mathematischen Abbildungen für die Koordinaten $x(i)$ und $y(i)$, sowie die Größe mit Breite $w(i)$ und Höhe $h(i)$, erfolgen. Bei einem maximalen Zählerstand wird der Zähler zurückgesetzt und die Generierung wiederholt sich.
Eine Umsetzung mit AXI4-Lite kann über diese Schnittstelle konfiguriert werden und speichert die Befehle in einem internen Puffer. Mit einem Zähler der übermittelten Befehle werden diese sequentiell und wiederholt ausgelesen und übertragen.
Eine Umsetzung mit einem externen Speicher liest die Befehle sequentiell und wiederholt aus und überträgt sie als Datenstrom. Die Startadresse und Anzahl der Befehle im Speicher wird über eine weitere AXI4-Lite Schnittstelle konfiguriert.
Alle Lösungen bieten verschiedene Vor- und Nachteile, die abhängig vom Anwendungsfall abgewägt werden müssen. Die arithemtische Lösung verzichtet vollständig auf einen Puffer, aber besitzt gleichzeitig die geringste Flexibilität. Die Lösungen mit internem und externem Puffer sind beide deutlich flexibler, aber benötigen einen Speicher. Der externe Speicher hat den Vorteil, dass er meist einfacher allokierbar ist, wie es beispielweise bei dem geteilten DDR3-Speicher auf dem Zynq SoC der Fall ist. Nachteil ist die höhere Latenz im Vergleich zur Lösung mit internem Puffer. Diese geringe Latenz wird allerdings nur erreicht, da zum Zeitpunkt der Hardwaresynthese exklusiver Speicher bereitgestellt wird, was bedeutet, dass dessen Größe und damit die maximale Anzahl der Befehle im Vorhinein bekannt sein muss. Zusammenfassend kann festgehalten werden, dass keine der Lösungen optimal ist.

4.5.3 Skalierer

Die Aufgabe des Skalierers ist eine Veränderung der Größe eines Eingangsbildsignals durch die Anwendung von Interpolation zur Bildskalierung. Der Skalierer besitzt sowohl für das Eingangs- und Ausgangssignal eine AXI4-Video Stream Schnittstelle. Das Eingangssignal enthält das Bild in originaler Größe, welches in skalierter Größe als Ausgangssignal ausgegeben wird. Die Eingabe- und Ausgabegröße wird über jeweils zwei Bitvektoren als vorzeichenloser Integerwert konfiguriert. Die Eingabegröße kann theoretisch über die Synchronisationssignale des AXI4-Video Streams erkannt werden, jedoch wäre die Auflösung und damit der Skalierungsfaktor erst nach einer Bildübertragung bekannt ist, weshalb die gesamten Bilddaten gepuffert werden müssten. Wird die Auflösung vor der Übertragung konfiguriert, dann kann die Skalierung bereits während der Übertragung erfolgen, wodurch Puffergröße und Latenz deutlich geringer ausfallen. Die Aktivierung der Konfiguration und der Beginn der Skalierung erfolgt über ein Startsignal. Sobald der Skalierer das gesamte Bild verarbeitet hat, wird ein Bereitschaftssignal aktiviert, wodurch der Extraktor weiß, wann die nächste Konfiguration anzulegen ist.

4.5.4 Ablage

Die Aufgabe der Ablage ist einen Bilddatenstrom in einen Speicher zu schreiben. Die Bilddaten werden über AXI4-Video Stream empfangen und über AXI4 im Speicher abgelegt. Über einen weiteren AXI4-Stream werden Adressbefehle erthalten, welche angeben an welcher Adresse im Speicher das Bild abgelegt werden soll. Ein zweiter Wert im Befehl enthält die Anzahl abzulegender Pixel, wodurch bekannt ist, wann der nächste Befehl zu starten ist. Die Trennung von Video- und Befehlsdaten wurden gewählt, um für Erstere einen unveränderten AXI4-Video Stream und dadurch eine vorhandene Komponente für den Skalierer nutzen zu können. Die Anzahl der Pixel im Adressbefehl ist notwendig, um das Ende eines Bildes zu erkennen, da der AXI4-Video Stream nur ein Startsignal für ein Bild besitzt, wodurch das Ende des vorherigen erst mit dem Anfang des nächsten Bildes bekannt wäre.

5 Realisierung eines Prototypen

Mittels eines Prototypen soll eine konkrete Implementierung der erdachten Lösungsansätze aus der Konzeption umgesetzt werden. Mit dieser Implementierung soll später evaluiert werden, ob sich die anfängliche These auf echter Hardware bewahrheitet. Als Erstes werden funktionale und nicht-funktionale Anforderung an den Prototypen aufgestellt. Folgend wird die verwendete Software und und verwendete Programmbibliotheken beschrieben. Zuletzt werden ausgewählte Aspekte der Implementierung erläutert.

5.1 Anforderungsanalyse

In diesem Abschnitt werden aus der konzeptionierten Lösung in Kap. 4 Anforderungen an einen Prototypen abgeleitet. Die Anforderungen wurden dabei so gewählt, dass sie im begrenzten Zeitrahmen umsetzbar, aber trotzdem ausreichend für eine ausführliche Evaluierung, sind.
Die ausgewählten Tilingparameter aus Kap. 4.1 werden in Klammern referenziert, um sicherzustellen, dass alle Parameter P1-P7 beachtet wurden.

5.1.1 Funktionale Anforderungen

F1 Der speicherbasierte Ansatz wird umgesetzt

F2 Es wird ein externes Bildsignal angebunden

F3 Die erzeugten Tiles werden im DDR3-Speicher passend für das BNN-PYNQ/FINN abgelegt

F4 Folgende Tilingparameter sind zur Laufzeit konfigurierbar:

- (a) Größe des Eingangsbildes (P1)
- (b) Anzahl Tiles (P3)
- (c) Positionen der Bildregionen im Eingangsbild (P4)
- (d) Größen der Bildregionen im Eingangsbild (P5)
- (e) Skalierte Größe der Tiles (P6)

F5 Folgende Tilingparameter sind statisch:

- (a) Farbtiefe (P2) ist 24 Bit RGB passend für das BNN-PYNQ/FINN
- (b) Skalierungsalgorithmus (P7)

F6 Es wird eine Gesamtschaltung aus externer Bildquelle, Tiling und KNN aufgebaut

5.1.2 Nicht-Funktionale Anforderungen

N1 Die verwendete Hardwareplatform ist das Digilent PYNQ-Z1

N2 Die Implementierung des Tilings erfolgt auf dem FPGA der Zielplattform

N3 Die Gesamtschaltung ist synthetisierbar

5.2 Verwendete Software und Programmbibliotheken

Es wurde hauptsächlich mit den offiziellen Werkzeugen des SoC Herstellers Xilinx gearbeitet. Vitis 2020.2 wurde zur Programmierung des Prozessors des SoC mittels der Programmiersprache C++ genutzt, da es eine sehr gute Integration mit den FPGA Werkzeugen besitzt. Für die Entwicklung der Schaltung auf dem FPGA wurde Vivado 2020.2 verwendet. Die neu implementierten Komponenten wurden hauptsächlich in der Hardwarebeschreibungssprache VHDL geschrieben und zusammen mit anderen fertigen Komponenten in sogenannte „Block Designs“/Blockdiagramme zusammengeführt. Die Wahl der Hardwarebeschreibungssprache VHDL statt einer synthetisierbaren Hochsprache, wie C/C++, war die

höhere Kontrollmöglichkeit, welche unter den vielen Abstraktionen verloren geht oder nur schwer durch spezielle Compilerdirektiven wieder zu erreichen ist.
Neben der mit Vivado und Vitis mitgelieferten Bibliotheken für Treiber und IP-Cores wurde die Vivado Library von Digilent verwendet [6]. Diese enthält verschiedene IP-Cores zur Unterstützung der verbauten Peripherie der eigen hergestellten Entwicklungsplatinen, wie zum Beispiel dem verwendeten Digilent PYNQ-Z1.

5.3 Ausgewählte Aspekte

Dieser Abschnitt beschreibt ausgewählte Aspekte der Implementierung des Prototypen. Zuerst wird das konkret verwendete Datenformat der Bilddaten im Speicher in Kap. 5.3.1 beschrieben, folgend die Anbindung eines externen Bildsignals in Kap. 5.3.2, die Komponenten des Tilings in Kap. 5.3.3, die Methoden zur Konfiguration mittels CPU in Kap. 5.4 und zuletzt der Aufbau einer Gesamtschaltung aller Komponenten in Kap. 5.5.. Jeglicher Quellcode und Kommentare in diesem wurden auf Englisch verfasst, da die Schnittstellen, wie etwa AXI4, für ihre Signale ebenfalls englische Namen verwenden.

5.3.1 Datenformat der Bilddaten

Das verwendete KNN erwartet pro Pixel drei Bytes, die aufeinanderfolgend im Speicher liegen (siehe 2.6), weshalb sinnvollerweise auch alle anderen Komponenten, welche über AXI4 angebunden sind, dieses Format für ihre Puffer im DDR3-Speicher nutzen. Bursts für Lese- und Schreibtransaktionen werden auf dem Adresskanal initialisiert und auf dem Datenkanal durchgeführt (für eine genauere Beschreibung siehe 2.5). Sowohl Adressen, als auch Daten, werden in Worten angegeben, welche in ihrer Länge in Bytes als Zweierpotenzen konfiguriert werden können. Dementsprechend sind Wortlängen von beispielsweise 1, 2, 4 oder 8 möglich, aber nicht die 3 Byte eines Pixels. Die einfachste Lösung wäre eine Wortlänge von einem Byte, allerdings benötigt dies drei Taktzyklen zur Übertragung eines Pixels, was die Bandbreite der AXI4-Schnittstelle bei weitem nicht ausnutzt. Auch die Schnittstelle des angebundenen DDR3-Speichers mit 16 Bit Datenbreite und Zugriffen bei steigender und fallender Taktflanke durch die doppelte Datenrate hat deutlich mehr Potential, weshalb die verwendete Wortlänge möglichst nah an diesem Wert liegen sollte. AXI4 erlaubt über das *WSTRB*-Signal die Übertragung von weniger Bytes, als die Wortlänge, weshalb eine Wortlänge von vier mit einer Maske für drei Bytes verwendet werden könnte, jedoch wird die Bandbreite der Schnittstelle dementsprechend nur zu 75% ausgenutzt. Da ein Speicher adressiert wird, welcher pro Adresse nur einmal beschrieben wird, kommt nur der *INC* Bursttyp in Frage, welcher nach jedem Beat automatisch die Adresse um eine gesetzte Anzahl Bytes erhöht, jedoch erneut nur in einem Abstand von Zweierpotenzen. Der genannte Ansatz für eine Wortlänge von vier Bytes mit drei belegten Bytes funktioniert daher nur für eine Burstlänge von eins, da keine auf drei Bytes ausgerichtete Inkrementierung möglich ist.
Zur schlußendlichen Lösung wurde ein Packen der Pixel in Worte genutzt, anstatt Pixel einzeln zu lesen oder zu schreiben. Das erste gemeinsame Vielfache der Wortlänge von vier Bytes und der Pixellänge von drei Bytes sind zwölf Bytes oder drei Worte. Daraus ergeben sich drei Formate für Worte, welche sich stetig wiederholen. Zur Illustration wurde dies in Abb. 17 festgehalten, welche einen wortadressierbaren Speicher mit einer Wortlänge von vier Byte darstellt. Die Buchstaben dienen zur Unterscheidung der einzelnen Pixel, wobei jeder Pixel drei Bytes belegt.

Daten (Worte):	···	JIII	HHHG	GGFF	FEEE	DDDC	CCBB	BAAA
Adresse (Bytes):	28	24	20	16	12	8	4	0

Abbildung 17: Packen der Pixeldaten in Worte

Die drei Worttypen können zusammengefasst werden in BAAA, BBAA und BBBA mit dem niederwertigsten Byte rechts. Die Buchstaben dienen zur Unterscheidung zweier Pixel. Im ersten Typ ist Pixel A komplett enthalten, aber nur ein Byte von Pixel B, welcher im zweiten Typ als Pixel A fortgesetzt wird. Gleichzeitig beginnt ein neuer Pixel B, der im letzten Typ als Pixel A beendet wird und zusätzlich

ein neuer kompletter Pixel B enthalten ist. Der Worttyp w_{typ} ergibt sich abhängig von der Adresse a in Bytes folgendermaßen:

$$w_{typ} \in \begin{cases} \text{BAAA falls } a\%4 \leq 1 \\ \text{BBAA falls } a\%4 = 2 \\ \text{BBBA falls } a\%4 = 3 \end{cases} \tag{2}$$

Die Burstlänge l_{worte} in Worten steht zu der Burstlänge in Pixeln l_{pixel} in folgendem Zusammenhang:

$$l_{worte}(l_{pixel}) = \left\lceil \frac{3 * l_{pixel}}{4} \right\rceil \tag{3}$$

Abhängig von der gewählten Länge kann es vorkommen, dass das erste und/oder letzte Wort in einem Burst Daten enthält, die nicht Teil des gewünschten Pixel sind. Bei einem Lesevorgang können diese Daten ignoriert werden, jedoch darf ein Schreibvorgang diese nicht überschreiben. Anstatt die alten Daten abzurufen und erneut mitzuschreiben, kann vom *WSTRB*-Signal Gebrauch gemacht werden und nur die Bytes maskiert werden, die tatsächlich Teil des Bursts sind. Ein beispielhafter Burst für fünf Pixel ist in Abb. 18 zusammen mit dem Wert des *WSTRB*-Signals bei welchem jedes Bit die Aktivierung eines Bytes anzeigt dargestellt. Die unterschiedlichen Pixel sind erneut mit Buchstaben dargestellt, während ein ? Daten anzeigt, die nicht Teil des Burst sind und deren Werte zum Schreibzeitpunkt nicht bekannt sein müssen.

WSTRB:	0001	1111	1111	1111	1100
Daten (Worte):	???E	EEDD	DCCC	BBBA	AA??
Adresse (Bytes):	20	16	12	8	4

Abbildung 18: Beispiel für einen gepackten Burst mit 5 Pixeln ab Adresse 6

5.3.2 Anbindung des Eingangsbildsignals

Die Anbindung eines externen Bildsignals erfordert die Konvertierung dieses Signals in ein Format, welches vom Tiling verarbeitet werden kann. Das Tiling erwartet die Bilddaten analog zum BNN-PYNQ pixel- und zeilenweise im DDR3-Speicher mit einer Farbtiefe von 24 Bit. Beispielhaft wurde ein DVI-Signal über einen HDMI Anschluss verwendet, welches als proprietärer Datenstrom eintrifft. Dieser Datenstrom musste nun in das erwartete Format umgewandelt werden. Damit auch andere Schnittstellen verwendet werden könnten, wurde vorerst eine Umwandlung in ein universelles Zwischenformat vorgenommen.
Dafür war der AXI4-Video Stream geradezu prädestiniert, da dieser ausschließlich Daten und zwei Synchronisationssignale enthält: Anfang des Bildes und Ende einer Zeile. Dieser Minimalismus erlaubt es nahezu jedes Format in einen solchen Stream umzuwandeln, wozu bereits viele IP-Cores existieren.
Für die auf der verwendeten Hardware vorhandenen Anschlüsse stellt der Hersteller Digilent einen entsprechenden IP-Core namens „DVI to RGB Video Decoder“ [17], welcher mithilfe eines Referenztaktes ein differentielles DVI-Signal in Timing- und Bilddaten umwandelt. Der IP-Core „Video In to AXI4-Stream“ von Xilinx [42] wandelt diese Daten folgend in einen AXI4-Video Stream um. Dieser Zusammenschluss von Komponenten wurde zur logischen Trennung in einem Blockdiagramm festgehalten, sodass die gesamte Umwandlung mit einem Block (siehe Abb. 19) erfolgen kann. Der interne Aufbau kann in Abb. 41 im Anhang betrachtet werden.

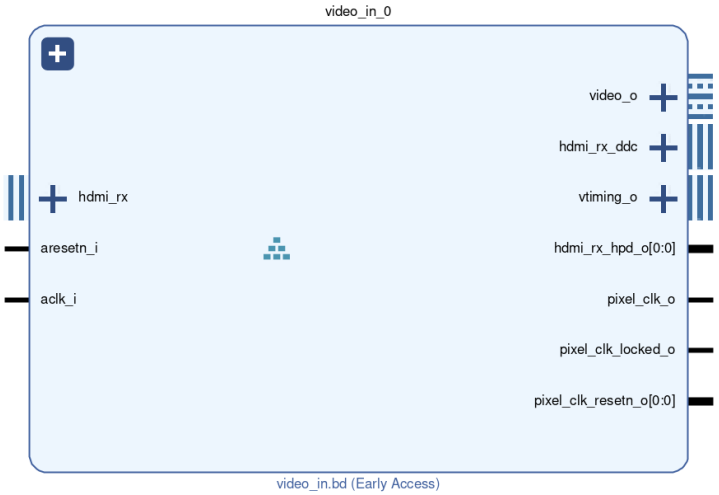


Abbildung 19: Komponente - Umwandlung von DVI zu AXI4-Video Stream

Als letzter Schritt galt es nun diesen Datenstrom im Speicher abzulegen, wofür der „AXI Video DMA“ von Xilinx [9] verwendet wurde, welcher den Datenstrom im erwarteten Format in den Hauptspeicher schreibt. Die Konfiguration erfolgte über AXI4-Lite, die hauptsächlich aus folgenden Parametern bestand: Höhe & Breite des Bildes, Zieladresse im Speicher und die Aktivierung der automatischen Wiederholung. Letztere schreibt wiederholt nach einmaligem Start die eintreffenden Bilder, sodass die Ausführung komplett eigenständig erfolgt. In einer erweiterten Konfiguration könnten mehrere Zieladressen für zyklisch verwendete Puffer definiert werden. Dies ist vorteilhaft, da die Daten eines Puffers verarbeitet werden können, ohne dass diese sich dabei verändern. Aus Zeitgründen wurde in dieser Arbeit auf die Anwendung verzichtet. Die Anbindung des Video DMA an den AXI4-Video Stream kann im Anhang in Abb. 43 betrachtet werden.

5.3.3 Umsetzung der Tilingkomponente

Dieser Abschnitt beschreibt die Umsetzung der Komponenten des Tilings, was den Extraktor, den Skalierer, die Ablage und einen beispielhaften Befehlsgenerator enthält.

Skalierer
Der Skalierer nimmt eine Bildsignal mit einer definierten Auflösung und reduziert oder erhöht dessen Auflösung abhängig der Konfiguration durch Interpolation und gibt dieses Signal neuer Größe aus. Da es bereits einige fertige Lösungen auf FPGAs für diese Aufgabe gibt, wurde eine solche zur Zeitersparnis gesucht. Weiterhin ist eine solche Lösung wahrscheinlich leistungsstärker, effizienter und ausgereifter, als eine komplett eigene Lösung, die in der Kürze der Zeit entwickelt werden könnte. Ausschlaggebend für die Auswahl war eine freie Verfügbarkeit, AXI4-Video Stream als Videoschnittstelle, eine einfache Anbindung mittels IP-Core und eine Anpassbarkeit der Schnittstellen. Betrachtet wurden drei Alternativen:

Xilinx Video Scaler v8.1 [44]
Dieser IP-Core unterstützt die AXI4-Video Stream Schnittstelle und wird mit Vivado mitgeliefert. Leider wurde dieser IP-Core vom Xilinx LogiCORE IP Video Multi Scaler abgelöst und wurde vom Hersteller als veraltet deklariert. Da davon auszugehen ist, dass er in zukünftigen Vivado Versionen entfernt wird, wurde die Verwendung ausgeschlossen.

Xilinx LogiCORE IP Video Multi Scaler v1.2 [28]
Dieser IP-Core besitzt viele Konfigurationsmöglichkeiten, wird aktiv weiterentwickelt und mit Vivado mitgeliefert, jedoch erfolgt die Ein- und Ausgabe der Videodaten über eine AXI4 Schnittstelle. Weiterhin ist der Ressourcenverbrauch viel zu hoch und übersteigt die 20% verwendbaren Block RAM um ein Vielfaches [4].

Digilent Video Scaler 1.0 [43]
Dieser IP-Core ist enthalten in der frei verfügbaren Digilent Vivado Library, welche für die IP-Cores der HDMI-Schnittstellen des PYNQ-Z1 bereits verwendet wurde. Als Videoschnittstelle ist AXI4-Video Stream vorgesehen, während die Konfiguration standartmäßig über fünf Register mittels AXI4-Lite geschieht: Breite/Höhe des Eingangsbildes, Breite/Höhe des Ausgangsbildes und die Steuerung zum einmaligen oder wiederholten Starten. Der Verbrauch an Block RAM ist mit 12 Blöcken mäßig. Der Quellcode ist offen, verwendet C++ mit der verbreiteten OpenCV Bibliothek und besteht fast ausschließlich aus 18 Quellcodezeilen. Standartmäßig ist die Farbtiefe auf 24 Bit RGB und der Skalierungsalgorithmus auf bilinear eingestellt, was aber leicht geändert werden könnte.

Auswahl
Schlußendlich wurde sich für den Digilent Video Scaler entschieden, da dieser die im Vorraus gewählten Kriterien erfüllt und leicht angepasst werden kann.
Nach der Auswahl wurde vom offenen Quellcode gebrauch gemacht und einige Anpassungen vorgenommen. Die maximale Auflösung wurde von 4096x4096 auf 1024x1024 verringert, was den Block RAM Verbrauch geviertelt hat, jedoch damit der limitierende Faktor für die maximale Größe der ausgeschnittenen Bildbereiche ist. Diese Einschränkung wurde jedoch akzeptiert, da das verwendete KNN standartmäßig nur Auflösungen von 32x32 unterstützt, wovon 1024x1024 bereits ein 32-facher Skalierungsfaktor ist. Die AXI4-Lite Konfigurationsschnittstelle wurde durch eine einfachere Schnittstelle mit Start-, Leerlauf-, Fertigstellungs- und Bereitschaftssignal ersetzt. Die Eingabe der Eingangs- und Ausgangsgröße erfolgt zusammen mit dem Startsignal über vier Bitvektoren als vorzeichenloser Integerwert mit einer Länge von 32 Bit. Der erzeugte IP-Core kann in Abb. 20 betrachtet werden. Da der Quellcode noch nicht auf Vitis HLS umgestellt wurde, wurde er mit dem Vorgänger Vivado HLS in der Version 2019.1 übersetzt.

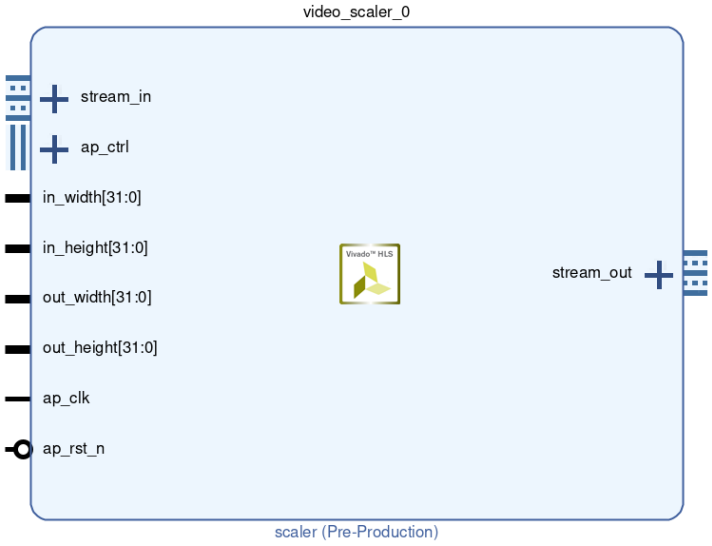


Abbildung 20: Komponente - Skalierer

Die Schnittstellen „stream_in“ und „stream_out“ sind die AXI4-Video Streams für das originale und skalierte Bildsignal. Das Leitungsbündel „ap_ctrl“ enthält das Signal zum Starten des Skalierers und die verschiedenen Statussignale. Die Eingangssignale „in_width“, „in_height“, „out_width“ und „out_height“ erhalten die Eingangs- und Ausgangsauflösung der Bilddaten, die zu Beginn zusammen mit dem Startsignal zwischengespeichert werden. Die gesamte Schaltung läuft synchron zum Taktsignal „ap_clk“ und wird bei niedrigem Pegel synchron zum Takt über „ap_rst_n“ zurückgesetzt.

Befehlsgenerator
Der Befehlsgenerator erzeugt einen AXI4-Stream mit Befehlen für den Extraktor, welche die Koordinaten und die Größe eines Bildbereichs enthalten. Für die Erzeugung dieser Befehle standen mehrere Möglichkeiten zur Auswahl, die in Kap. 4.5.2 aufgeführt wurden. Zum Zwecke des Prototypen wurde sich für das Auslesen der Befehle aus einem externen Speicher entschieden, da dieser Ansatz eine maximale Anpassbarkeit der Befehle bietet, deren Format in Abb. 15 beschrieben ist. Die Anzahl der Befehle ist ausschließlich durch die Größe des Speichers begrenzt, was bei dem verwendeten 512 MB DDR3-Speicher mehrere millionen Befehle bedeutet, die individuell konfiguriert werden können. Die tatsächliche Generierung der Befehle und Ablage im Speicher kann durch die CPU auf dem SoC geschehen. Dies hat den Vorteil, dass Anpassungen durch deutlich kürzere Kompilierungszeiten sehr schnell möglich sind. Sofern die Generierung der Befehle einmalig ist, können potentielle Geschwindigkeits- und Effizienzvorteile einer Generierung auf dem FPGA vernachlässigt werden.
Für die Implementierung wurde die Hardwarebeschreibungssprache VHDL gewählt. Die Ausgabe der Befehle erfolgt über eine AXI4-Stream Schnittstelle mit dem Namen „M_AXIS_COMMANDS“. Zur Konfiguration wurde eine AXI4-Lite Schnittstelle namens „S_AXI_CTRL“ verwendet. Der Abruf der Daten aus dem Hauptspeicher erfolgt über eine AXI4-Lite Schnittstelle namens „M_AXI_SRC“. Ein weiterer Ausgang „idle_o“ zeigt den Leerlauf der Komponente bei hohem Pegel an. Alle Schnittstellen sind synchron zu einem gemeinsamen Taktsignal „ACLK“ und werden bei tiefem Pegel von „ARESETN“ synchron zum Takt zurückgesetzt. Der erzeugte Block für das Blockdiagramm kann mit seinen Schnittstellen in Abb. 21 betrachtet werden.

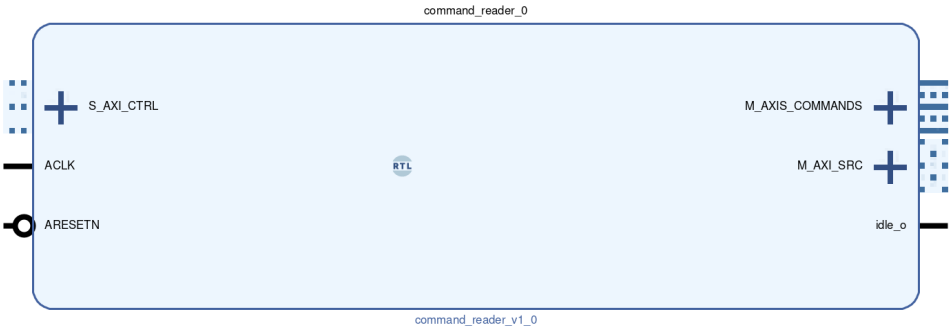


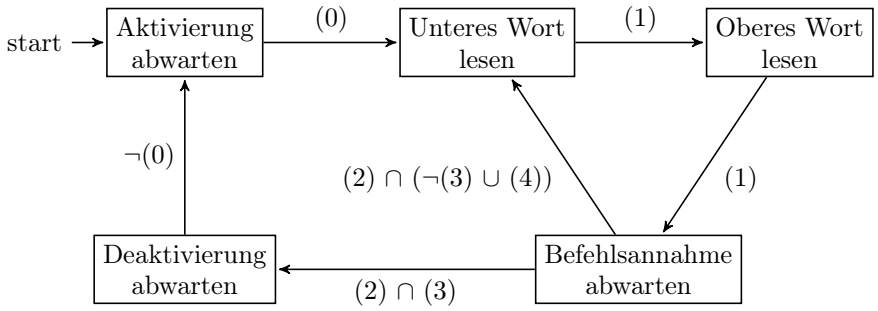
Abbildung 21: Komponente - Befehlsgenerator

Die Konfiguration erfolgt über vier Register, die über AXI4-Lite geschrieben oder gelesen werden können (siehe Tabelle 2). Als Basis für die AXI4-Lite Implementation wurde ein Quellcodebeispiel aus Vivado verwendet und entsprechend angepasst. Bevor das Auslesen gestartet wird, müssen zwei Register gesetzt werden: Die Startadresse der Befehle im Speicher und die Anzahl der Befehle, die ab dieser Adresse ausgelesen werden sollen. Beide Register dürfen nur beschrieben werden, falls die Ausführung nicht aktiv ist. Nach dem Schreiben dieser zwei Register kann die Ausführung über das Setzen des ersten Bits im Steuerungsregister gestartet werden. Nach erfolgter Ausführung muss das Bit für einen erneuten Start manuell zurückgesetzt werden. Durch ein zweites Bit kann festgelegt werden, ob das Auslesen nach Fertigstellung automatisch wiederholt wird. Das Startbit kann dann genutzt werden, um diese automatische Wiederholung zu pausieren. Erst durch die Deaktivierung des Wiederholungsbits wird die Ausführung gestoppt. Über ein Bit im Statusregister kann ausgelesen werden, ob aktuell eine Ausführung aktiv ist.

Offset	Name	Beschreibung	Aufteilung
00h	Steuerung	Breite des Bildes im Speicher in Pixeln	**Bit 1:** 1: Start, 0: Stop **Bit 2:** 1: Autom. Neustart, 0: Sonst **Bit 3-31:** Reserviert
04h	Status (Nur Lesbar)	Aktueller Zustand	**Bit 0:** 1: Leerlauf, 0: Sonst **Bit 1-31:** Reserviert
08h	Startadresse der Befehle	Startadresse der Befehle	**Bit 0-31:** Startadresse
0Ch	Befehlmenge	Anzahl der auszulesenden Befehle	**Bit 0-31:** Anzahl

Tabelle 2: Konfigurationsregister des Befehlsgenerators

Das Auslesen der Befehle aus dem Speicher über die Schnittstelle „M_AXI_SRC“ erfolgt über AXI4-Lite mit einer Datenbreite von 32 Bit. Die langsamere AXI4-Lite Schnittstelle statt der kompletten AXI4 Schnittstelle wurde gewählt, da es sich beim Lesen der Befehle um keine zeitkritische Aufgabe handelt und der Ressourcenaufwand für die Dekodierung so deutlich verringert werden kann. Das Lesen der Befehle ist nicht zeitkritisch, da die Ausführung eines Befehls deutlich längert dauert, als das Auslesen. Erst bei besonders kleinen Bildbereichen kann die AXI4-Lite Schnittstelle zum Flaschenhals werden, allerdings macht dann vermutlich ein neuer Generator mehr Sinn, da dieser mit einer Befehlslänge von acht Byte vermutlich zu ineffizient ist.
Das eigentliche Auslesen erfolgt durch einen einfachen Zustandsautomaten, der synchron zur steigenden Flanke des Takts geschaltet wird und in Abb. 22 dargestellt ist. Falls keine der Übergangsbedingungen zu einer steigenden Taktflanke wahr ist, wird implizit im Zustand verweilt. Zu Anfang wird auf ein Startsignal gewartet, wonach die niederen 32 Bit des Befehls und folgend die oberen 32 Bit aus dem Speicher gelesen werden. Nun wird solange gewartet, bis der Befehl über den Handschlag des AXI4-Streams angenommen wird. Wurde die konfigurierte Anzahl Befehle noch nicht erreicht, dann wird der nächste Befehl gelesen, ansonsten wird geprüft, ob der automatische Neustart aktiv ist. Falls ja, wird erneut der erste Befehl gelesen und der Befehlszähler zurückgesetzt. Falls nein, wird auf die Deaktivierung des Startsignals gewartet, wonach in den Startzustand gewechselt wird.



Bedingungen:
(0): Startsignal
(1): Lesevorgang abgeschlossen
(2): Befehl angenommen
(3): Anzahl Befehle erreicht
(4): Automatischer Neustart aktiv

Abbildung 22: Zustandsautomat für den Befehlsgenerator

Ablage
Die Ablage empfängt die skalierten Bildregionen vom Skalierer und einen Adressbefehl vom Extraktor, wonach die Bilddaten an der Adresse innerhalb des Befehls im Speicher abgelegt werden. Das genaue Format der Adressbefehle wurde in Kap. 4.5.1 beschrieben. Für die Implementation wurde die Hardwarebeschreibungssprache VHDL gewählt und folgende Schnittstellen vorgesehen: Ein AXI4-Video Stream „S_AXIS_VIDEO“ für das Eingangsbildsignal, ein AXI4-Stream „S_AXIS_COMMANDS“ für die Eingabe von Befehlen, eine komplette AXI4 Schnittstelle „M_AXI“ für das Schreiben der Daten in den Speicher und ein einfaches Signal „idle_o“, welches einen hohen Pegel besitzt, sobald alle Eingabedaten verarbeitet wurden. Alle Schnittstellen teilen sich ein gemeinsames Taktsignal „ACLK“ und einen gemeinsamen Reset „ARESETN“, welcher bei niedrigem Pegel und synchron zum Takt ausgeführt wird. Die Burstlänge der AXI4-Transaktionen kann über Generics als Zweierpotenz konfiguriert werden. Der generierte Block ist dargestellt in Abb. 23.

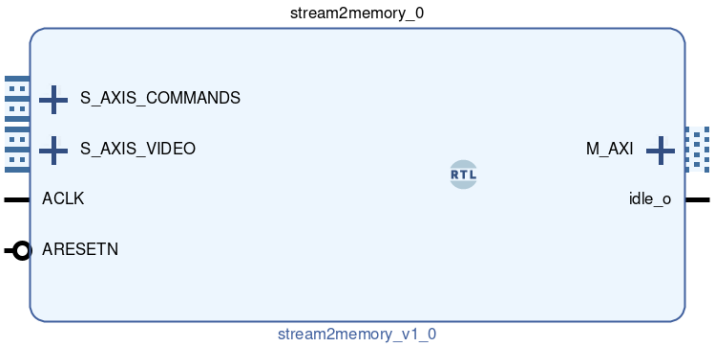
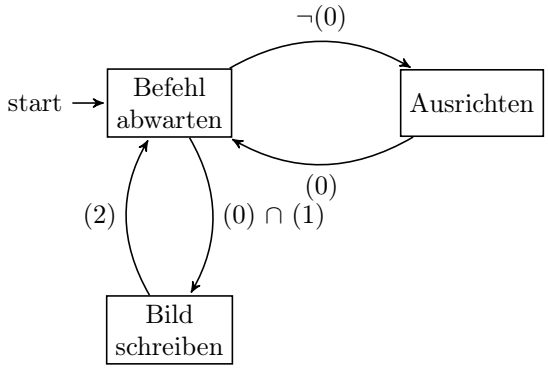


Abbildung 23: Komponente - Ablage

Die grundlegende Funktionalität kann als Zustandsautomat dargestellt werden (siehe Abb. 24), welcher synchron zur steigenden Flanke des Taktes bei der Validität einer Übergangsbedingung schaltet. Ist keine Bedingung wahr, dann wird implizit im aktuellen Zustand verweilt. Im Startzustand wird zuerst geprüft, ob das anliegende Bildsignal den Anfang eines Bildes anzeigt. Ist dies nicht der Fall, dann werden im Zustand „Ausrichten“ solange Pixel verworfen, bis dies der Fall ist. Die Notwendigkeit für diesen Vorgang kann durch verschiedene Umstände erfolgen: Der letzte Befehl enthielt die falsche Länge, wodurch Daten des nächsten Bildes als Teil des vorherigen geschrieben werden. Der Generator des Videosignals liefert falsche Bildgrößen durch beispielsweise eine falsche Konfiguration des Skalierers oder ein Puffer enthält zu Anfang invalide Daten. Wenn einer dieser Fälle eintrifft, erholt sich die Ablage durch das Ignorieren dieser Daten. Sobald das Bildsignal auf den Anfang ausgerichtet ist und ein Befehl verfügbar ist, wird dieser Befehl ausgeführt, indem dessen Daten an der Adresse im Befehl abgelegt werden. Durch die angegebene Länge im Befehl ist bekannt, wann alle Daten geschrieben wurden und wieder in den Startzustand gewechselt werden soll.



Bedingungen:
(0): Nächster Pixel ist Bildanfang
(1): Befehl verfügbar
(2): Schreibvorgang abgeschlossen

Abbildung 24: Zustandsautomat für die Ablage

Der Schreibvorgang im Zustand „Bild schreiben“ ist nicht trivial und erforderte eine verschachtelten Zustandsautomaten, weil für einen maximalen Datendurchsatz eine AXI4-Schnittstelle mit einer Burstlänge >1 verwendet wird, welche allerdings nur wortbasierte Transaktionen erlaubt (für mehr Informationen siehe Kap. 5.3.1). Dieser Zustandsautomat ist synchron zum gleichen Takt, wie der übergreifende Automat, und in Abb. 25 dargestellt.
Vor den Automaten wurde eine abgewandelte FIFO Warteschlange geschaltet mit einer Größe vom doppelten der über Generics konfigurierten Burstlänge. Diese FIFO wurde ebenfalls in VHDL geschrieben und wurde im Gegensatz zu einer herkömmlichen Implementation um weitere Signal ergänzt. Die Eingabe der Daten erfolgt auf klassische Weise durch eine Schreibaktivierung und ein Eingangssignal für die Daten. Zur Ausgabe stehen zusätzlich zu der Leseaktivierung und den aktuellen Daten die letzten und nächsten Daten zur Verfügung. Ergänzend zur Leseaktivierung kann weiterhin ein Signal zum Überspringen aktiviert werden, sodass zur nächsten Taktflanke die übernächsten Daten zur Verfügung stehen. Als Statussignal steht ein Signal für einen leeren und vollen Puffer, sowie der aktuelle Füllstand, zur Verfügung. Größe und Datenbreite der FIFO werden über Generics konfiguriert.
Nun folgt das Vorgehen des Zustandsautomaten der Schreibvorgänge, welcher starken Gebrauch von den Erweiterungen der FIFO macht. Zu Beginn wird im Startzustand darauf gewartet, dass der äußere Automat im entsprechenden Zustand ist, andernfalls bleibt dieser Zustandsautomat inaktiv. Im nächsten Zustand „Burst vorbereiten“ werden folgende Signale auf dem Adresskanal der AXI4-Schnittstelle vorbereitet: Startadresse des Bursts a_{burst} mittels Ausrichten der gewünschten Adresse $a_{original}$ auf die Wortlänge von vier Bytes. Diese Ausrichtung lässt sich durch $a_{burst} = a_{original} - (a_{orignal} \% 4)$ berechnen, was sich sehr effizient in Hardware durch das Ersetzen der unteren zwei Bits durch 0 umsetzen lässt. Als Burstlänge wird die über Generics konfigurierte Länge verwendet, solange die ausgerichtete Adresse auf diese ausgerichtet ist. Falls dies nicht der Fall ist, wird der erste Burst entsprechend verkürzt. Diese Ausrichtung ist notwendig, da laut AXI4-Spezifikation kein 4096 Byte Segment im Speicher in einem Burst überschritten werden darf (siehe Kap. 2.5). Solange ein Burst eine Zweierpotenz als Länge besitzt und die Startadresse auf derer Länge ausgerichtet ist, wird diese Einschränkung stets eingehalten. Der Bursttyp bleibt konstant auf „INC“ mit einem Inkrement von vier Bytes. Der Füllstand der FIFO wird nun genutzt, um zu Prüfen, ob ausreichend Pixel für einen Burst zur Verfügung stehen, damit innerhalb eines Bursts keine Verzögerungen entstehen und zu jeder Taktflanke ein Wort übertragen werden kann. Ist dies der Fall, dann wird in den Zustand „Burst starten“ gewechselt.
Im Zustand „Burst starten“ wird der berechnete Schreibburst nun beim AXI4 Slave beantragt. Sobald dieser die Konfiguration annimmt, wird gewechselt in den Zustand „Daten schreiben“.

In diesem werden nun die tatsächlichen Daten in Worten ab der Startadresse geschrieben. Der jeweils aktive Worttyp für die aktuelle Adresse wird anhand der Formel aus Kap. 5.3.1 berechnet. Der Worttyp gibt an, wie zwei Pixel in ein Wort gepackt werden. Durch die modifizierte FIFO können der letzte, aktuelle und nächste Pixel in der Warteschlange zu einem Wort kombiniert werden. Ist beispielsweise der aktuelle Worttyp BAAA, dann wird für Pixel A der aktuelle Pixel in der Warteschlange und für B der nächste Pixel benötigt. Der Worttyp AAAB stellt einen Sonderfall da, da hier zwei Pixel gleichzeitig fertiggestellt werden, weshalb das Signal zum Überspringen in der FIFO genutzt wird. Dadurch, dass vor Beginn sowieso ausreichend Daten zur Verfügung stehen müssen, ist sichergestellt, dass die FIFO während eines Bursts nie leer läuft. Weitere Sonderfälle sind der erste und letzte Burst eines Bildes, da das zu schreibende Wort nicht nur neue Daten enthält (siehe Beispiel in Abb. 18). Basierend auf Worttyp und Position des Wortes wird daher anhand der Maskierung mittels WSTRB festgelegt, welche Bytes Daten enthalten, sodass „fremde“ Daten im Wort unverändert bleiben. Wurde der gesamte Burst geschrieben, aber noch nicht alle Daten des Bildes, dann wird ein neuer Burst vorbereitet und ausgeführt. Ist das Bild vollständig geschrieben, dann wird folgend auf einen Zustandswechsel des äußeren Automaten gewartet, was diesem gleichzeitig die Fertigstellung signalisiert. Hat der äußere Zustandsautomat den Zustand gewechselt, dann wird in den Startzustand zurückgekehrt.

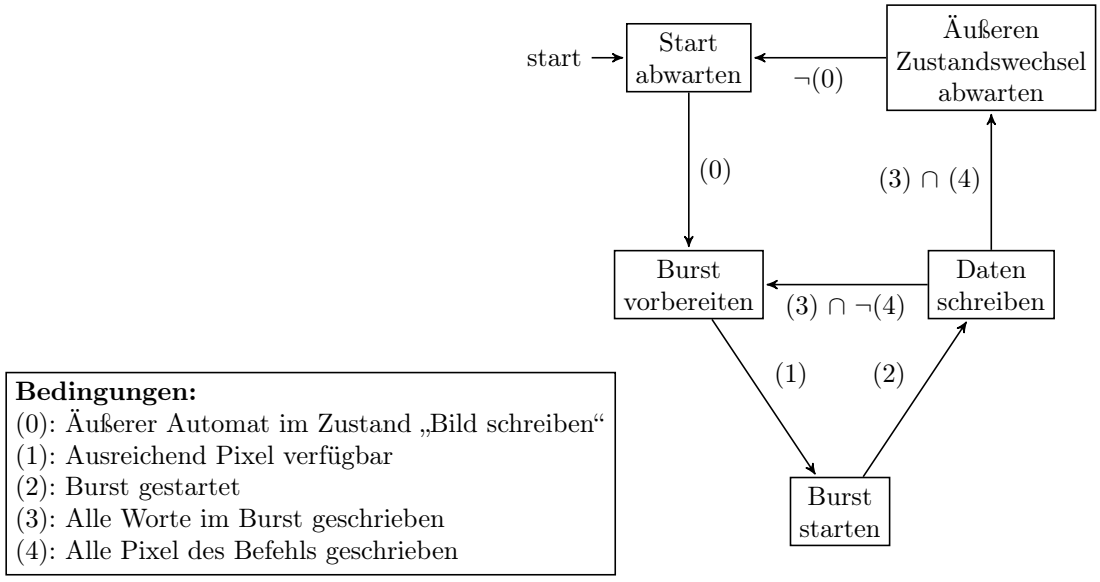


Abbildung 25: Zustandsautomat für Schreibvorgänge der Ablage

Extraktor

Der Extraktor ist der Hauptakteur im Tiling und koordiniert Skalierer und Ablage anhand der erhaltenen Extraktionsbefehle vom Befehlsgenerator und wurde in der Hardwarebeschreibungssprache VHDL umgesetzt. Die Extraktionsbefehle enthalten die Koordinaten und die Größe eines Bildbereichs (siehe Kap. 4.5.1). Die Eingangsauflösung für den Bildbereich wird dem Extraktionsbefehl entnommen, während die Zielauflösung der über AXI4-Lite für alle Tiles konfiguriert wird. Entsprechend dieser beiden Auflösungen wird der Skalierer konfiguriert. Gleichzeitig wird über einen Adressbefehldatenstrom ein Kommando an die Ablage ausgegeben, welches die Anzahl Pixel im skalierten Bildbereich und die Zieladresse im Speicher enthält. Erstere ergibt sich aus der Multiplikation von Höhe und Breite der Zielauflösung, während letztere anhand eines internen Zählers bestimmt wird. Dieser Zähler wird automatisch mit jedem Bildbereich um die Anzahl Pixel in der Zielauflösung mal drei Bytes erhöht und nach dem letzten Extraktionsbefehl, anhand des „Last“-Signals, zurückgesetzt auf die über AXI4-Lite konfigurierte Startadresse der Tiles. Das genaue Format der Adressbefehle ist in Kap. 4.5.1 beschrieben. Falls die Ablage nicht verwendet und der Adressbefehldatenstrom nicht benötigt wird, muss das „Ready“-Signal des Datenstroms konstant auf einen hohen Pegel gesetzt werden, da sonst die Verarbeitung des Extraktors blockiert wird. Der generierte Block für das Blockdiagramm ist in Abb. 26 dargestellt. Die Schnittstelle

„S_AXI_CTRL“ ist die AXI4-Lite Konfigurationsschnittelle, „S_AXIS_COMMANDS“ der Extraktionsbefehledatenstrom, „M_AXIS_COMMANDS“ der Adressbefehldatenstrom, „M_AXIS_VIDEO“ der AXI4-Video Stream für die Pixel des extrahierten Bildbereichs und zuletzt ist „M_AXI“ die AXI4-Schnittstelle zum Auslesen der Bilddaten aus dem DDR3-Speicher. Der Skalierer wird über die Signale „scaler_*_width“ und „scaler_*_height“ jeweils für Eingabe- und Ausgabeauflösung als 32 Bit vorzeichenloser Integer konfiguriert. Die Ausführung des Skalierers wird mit „scaler_start_o“ gestartet und die Fertigstellung mit „scaler_ready_i“ erkannt. Der Ausgang „idle_o“ gibt an, ob die Ausführung des Skalierers momentan aktiv ist. Alle Schnittstellen teilen sich das Taktsignal „ACLK“ und das zu diesem Takt synchrone Resetsignal „ARESETN“, welches bei niedrigem Pegel aktiv ist.

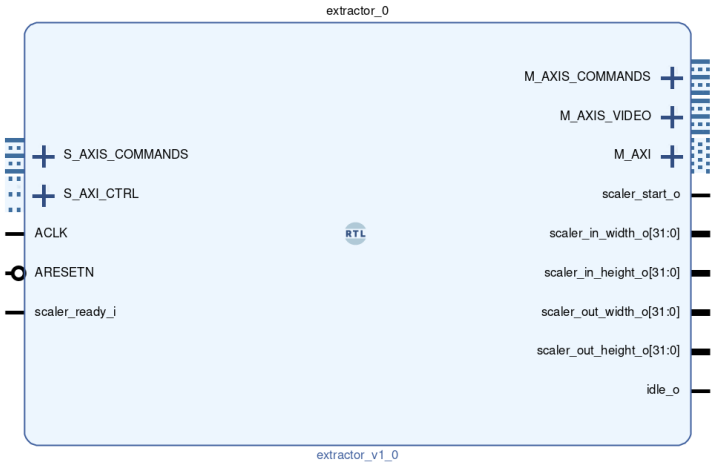
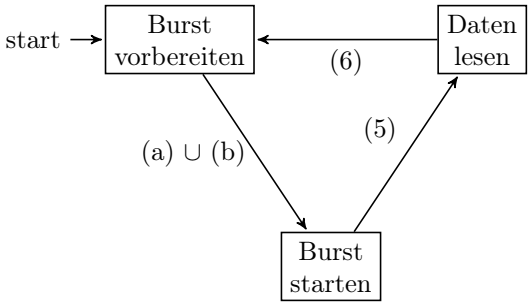


Abbildung 26: Komponente - Extraktor

Zur Entkopplung der Datenströme werden intern drei FIFO Warteschlangen verwendet für die gelesenen Daten aus dem Speicher, die Adressbefehle und die Skaliererkonfiguration.
Die tatsächliche Verarbeitung erfolgt mit einem Zustandsautomaten, der sehr ähnlich zu dem der Ablage ist. Er schaltet ebenfalls zu jeder steigenden Flanke und verbleibt implizit im aktuellen Zustand, falls keine der Übergangsbedingungen wahr ist. In Abb. 27 befindet sich eine Darstellung des Automaten zusammen mit den Übergangsbedingungen.
Die Verarbeitung startet nach einem Reset mit einem neuen Bildbereich im Zustand „Burst vorbereiten“, wozu auf einen Extraktionsbefehl in der entsprechenden Warteschlange gewartet wird. Zusätzlich muss Platz für einen Befehl in der Skaliererkonfigurationswarteschlange und in der Adressbefehlswarteschlange sein, sowie für die Daten eines Bursts in der Lesedatenwarteschlange. Ist dies der Fall, dann wird der Extraktionsbefehl angenommen und die entsprechenden Befehle für Skalierer und Ablage in den Ausgabewarteschlangen platziert. Basierend auf dem Extraktionsbefehl wird die Konfiguration des ersten Bursts gesetzt. Die Startadresse wird auf die Wortlänge von vier Bytes ausgerichtet. Falls die resultierende Adresse nicht auf die Burstlänge ausgerichtet ist, wird die Länge um die Differenz verkürzt. Die Ausrichtung auf die Burstlänge erfolgt, um sicherzustellen, dass kein Burst eine Grenze von 4096 Bytes überschreitet (siehe Kap. 2.5), was bei Ausrichtung auf Bursts mit Zweierpotenzen als Länge nicht passieren kann.
Im Zustand „Burst starten“ wird der Burst auf dem Adressekanal anhand der Konfiguration angefordert. Sobald dies erfolgt ist und der Burst gestartet wurde, wird in den Zustand „Daten lesen“ gewechselt. In diesem werden die ausgelesenen Worte des Burst in Pixel entpackt und in der Lesedatenwarteschlange abgelegt. Sobald alle Worte des Bursts empfangen wurde, wird wieder in den Zustand „Burst vorbereiten“ gewechselt.
Hier gibt es nun drei Optionen für einen Übergang, die allesamt vorraussetzen, dass genug Platz in der Lesedatenwarteschlange für die Daten eines Bursts ists. Falls das Auslesen einer Zeile noch nicht abgeschlossen wurde, dann wird die Startadresse für den Burst lediglich um die Burstlänge erhöht. Ist eine Zeile abgeschlossen, dann wird die Startadresse des Bursts auf den Beginn der nächsten Zeile im

Speicher gesetzt. Sind alle Zeilen eines Bildbereichs abgeschlossen, dann beginnt die Ausführung, wie oben beschrieben, von vorne.



Bedingungen:
(0): Neuer Bildbereich
(1): Extraktionsbefehl verfügbar
(2): Platz für einen Burst in der Lesedatenwarteschlange
(3): Skaliererkonfigurationswarteschlange voll
(4): Adressbefehlswarteschlange voll
(5): Burst gestartet
(6): Alle Worte im Burst gelesen
(a): $(0) \cap (1) \cap (2) \cap \neg(3) \cap \neg(4)$
(b): $\neg(0) \cap (2)$

Abbildung 27: Zustandsautomat des Extraktors

Die Berechnung der Adressen prüft nicht, ob sich der Bildbereich tatsächlich im Eingangsbild befindet, weshalb nur Extraktionsbefehle mit Bereichen komplett innerhalb des Bildes korrekt verarbeitet werden. Ein ungültiger Bereich erzeugt potentiell invalide Daten, aber führt zu keinem Fehlerzustand. Für Bereiche, die sich teilweise innerhalb des Bildes befinden, wurde keine spezielle Behandlung implementiert, da keine Annahme getroffen werden sollte, welche Werte die Pixel ausserhalb des Bildes haben sollen.
Für die Konfiguration des Extraktors über AXI4-Lite wurde der Beispielquellcode für eine einfache Registerkonfiguration aus Vivado angepasst und die folgenden Register in Tabelle 3 eingesetzt. Jeder Register besitzt eine Länge von vier Byte, sodass die Adressen genau diesen Versatz haben. Die Konfiguration ist nur erlaubt, wenn sich der Extraktor im Leerlauf befindet. Falls innerhalb der Verarbeitung die Konfiguration verändert wird, ist das Verhalten undefiniert.

Offset	Name	Beschreibung	Aufteilung
04h	Status (Nur Lesbar)	Zustand: Leerlauf ist aktiv, wenn keine weiteren Befehle zu verarbeiten sind. Fertig ist zusätzlich aktiv, falls die Ausgabewarteschlangen leer sind	**Bit 0:** 1: Leerlauf, 0: Sonst **Bit 1:** 1: Fertig, 0: Sonst **Bit 2-31:** Reserviert
08h	Startadresse der Bilddaten	Startadresse der Bilddaten	**Bit 0-31:** Startadresse
0Ch	Startadresse der Tiles	Startadresse ab der die erzeugten Tiles abgelegt werden	**Bit 0-31:** Startadresse
24h	Eingangsbreite	Breite des Bildes im Speicher in Pixeln	**Bit 0-15:** Breite **Bit 16-31:** Reserviert
28h	Eingangshöhe	Höhe des Bildes im Speicher in Pixeln	**Bit 0-15:** Höhe **Bit 16-31:** Reserviert
2Ch	Ausgangsbreite	Breite der skalierten Bildbereiche in Pixeln	**Bit 0-15:** Breite **Bit 16-31:** Reserviert
30h	Ausgangshöhe	Höhe der skalierten Bildbereiche in Pixeln	**Bit 0-15:** Höhe **Bit 16-31:** Reserviert

Tabelle 3: Konfigurationsregister des Extraktors

5.4 Konfiguration durch die CPU

Viele der Komponenten auf dem FPGA sind zur Laufzeit über AXI4-Lite konfigurierbar. Diese Konfiguration könnte innerhalb des FPGAs erfolgen, allerdings müsste bei neuen Parametern die Schaltung neu erstellt werden oder weitere Schnittstellen über beispielsweise physische Schalter geschaffen werden. An dieser Stelle zeigt sich der Vorteil des verwendeten SoCs, welcher neben dem FPGA unter anderem eine CPU enthält. Die Software der CPU kann sehr schnell und unabhängig der Hardwareschaltung angepasst werden. Da die Konfiguration keine besonders rechenintensive Aufgabe darstellt, können eventuelle Vorteile einer Lösung auf dem FPGA vernachlässigt werden.
Die Entwicklung der Software erfolgte mit Xilinx Vitis 2020.2, welches explizit für die Entwicklung von Software für eingebettete Systeme entwickelt wurde. Als Programmiersprache wurde C++ gewählt. Auf ein Betriebssystem wurde verzichtet, sodass ein direkter Zugriff auf den physischen Speicher möglich war.
Dieser Zugriff auf den physischen Speicher macht die Konfiguration über AXI4-Lite sehr einfach, da der AXI4-Bus über den Speichercontroller im Adresseraum verfügbar ist. Im unteren Adressraum liegt der DDR3-Speicher von 0 bis 512 MB. Die Adressen der restlichen Peripherie können in Vivado über den Adresseditor eines Blockdiagramms konfiguriert werdenm (siehe Kap. 2.7), wodurch die Interconnects als eine Art Router konfiguriert werden. Als Beispiel soll eine Schaltung betrachtet werden, deren AXI4-Lite Schnittstelle an Adresse 2^{29} (Bytes) im Speicher startet. Die Adressbreite der Schnittstelle ist 16 Bit, sodass der Speicherbereich bei $2^{29} + 65535$ endet. Die Daten werden als 32 Bit Worte betrachtet, die jeweils ein Register darstellen. Ein Speicherzugriff der CPU an Adresse 2^{29} greift somit auf das erste Register zu, $2^{29} + 4$ auf das Zweite und $2^{29} + 4n$ auf das nte Register.
Zur Abstraktion wurden für solche Registerzugriffe der Komponenten Methoden geschrieben. Eine solche Methode ist in Listing 1 dargestellt und dient zur Konfiguration der Startadresse der Eingangsbilddaten im Extraktor. Dass mit dieser Methode ein Hardwarebeschleuniger und keine Software konfiguriert wird, bleibt den Anwendern verborgen. Eine Ausnahme ist die Übergabe der Startadresse zur Konfiguration der jeweiligen Komponente im Speicher. Diese Adressen könnten theoretisch durch die erzeugten Headerdateien von Vivado/Vitis (siehe Kap. 2.7) auch automatisch erkannt werden und wurden nur zu Testzwecken des Prototypen manuell konfigurierbar gemacht.

```
void extractorSetInputBaseAddress(u32 baseAddress, u32 address) {
    /**
    * @brief    Sets the base address of the input image buffer
    *
    * @param    baseAddress: Base address of the extractor
    * @param    address: Base address of the input image buffer
    */
    Xil_Out32(baseAddress + EXTRACTOR_REG_INPUT_BASE_ADDRESS, address);
}
```

Listing 1: Methode zur Adressenkonfiguration der Eingangsbilddaten des Extraktors

Mehrere solcher Methoden wurden weiterhin zusammengefasst, um die gesamte Konfiguration durch einen Methodenaufruf möglich zu machen. Die Initialisierungsmethode für den Extraktor ist in Listing 2 dargestellt.

```
void extractorInit(u32 baseAddress, u32 iBaseAddress, u32 iWidth, u32 iHeight,
                   u32 oBaseAddress, u32 oWidth, u32 oHeight) {
    /**
    * @brief   Initializes all important parameters of an extractor at once
    *
    * @param   iBaseAddress: Base address of the extractor
    * @param   iBaseAddress: Base address of the input image buffer
    * @param   iWidth: Width of the input image in pixels
    * @param   iHeight: Height of the input image in pixels
    * @param   oBaseAddress: Base address of the output tile buffer
    * @param   oWidth: Width of the scaled tiles in pixels
    * @param   oHeight: Height of the scaled tiles in pixels
    */
    extractorSetInputBaseAddress(baseAddress, iBaseAddress);
    extractorSetOutputBaseAddress(baseAddress, oBaseAddress);
    extractorSetInputWidth(baseAddress, iWidth);
    extractorSetInputHeight(baseAddress, iHeight);
    extractorSetOutputWidth(baseAddress, oWidth);
    extractorSetOutputHeight(baseAddress, oHeight);
}
```

Listing 2: Methode zur Initialisierung des Extraktors

Für die Konfiguration des Befehlsgenerators wurde zusätzlich ein Datentyp erstellt (siehe Listing 3), der die Definition der Bildbereiche sehr simpel macht. Die Definition mehrere Bildbereiche erfolgt durch einen Array dieses Datentypen. Die Werte „xOffset“/„yOffset“ sind die Koordinaten an der oberen linken Ecke und „width“/„height“ die Breite/Höhe des Bildbereichs.

```
typedef struct {
        u16 xOffset; u16 yOffset;
        u16 width; u16 height;
} command_t;
```

Listing 3: Datentyp für die Konfiguration von Bildbereichen für den Befehlsgenerator

5.5 Aufbau einer Gesamtschaltung

Nach der Umsetzung der einzelnen Komponenten galt es diese in einer Gesamtschaltung bestehend aus externem Bildsignal, Tiling und KNN zu integrieren. Für die Anbindung des externen Bildsignals und das Tiling wurden die neu implementierten Komponenten verwendet. Als KNN wurde FINN eingesetzt, da dieses im Vergleich zum Vorgänger, dem BNN-PYNQ, deutlich einfacher zu konfigurieren ist. Da es sich bei FINN selbst nur um ein Framework handelt, musste noch ein mit diesem erstelltes KNN gefunden werden. Hierfür wurde das offizielle CNV Beispiel der Entwickler für ein CNN verwendet, welches mit dem CIFAR10 Datensatz [26] trainiert wurde. Die einzelnen Bilder werden in einer Auflösung von 32x32 Pixeln und einer Farbtiefe von 24 Bit RGB erwartet. Das Ergebnis nach der Inferenz eines Bildes ist der Index der Klasse mit der höchsten Konfidenz. Bei CIFAR10 wird zwischen 10 Klassen unterschieden, sodass der Index einen Wert von 0 bis 9 einnimmt. Grundsätzlich erlaubt es FINN einen Datenstrom oder einen Speicher für die Ein-/Ausgabe von Daten zu verwenden (siehe Kap. 2.6). Für die Ausgabe wurde der von FINN mitgelieferte Block zur Ablage im Speicher verwendet, damit die Ergebnisse durch einfachen Speicherzugriff von der CPU abgerufen und ausgegeben werden können. Für die Eingabe der Daten wurden zwei verschiedene Schaltungen aufgebaut.
In Abb. 28 ist eine Schaltung dargestellt, welche die Tiles für das KNN aus dem Speicher ausliest. Verantwortlich dafür ist der „input_dma“, welcher die Bilddaten aus dem Speicher als Datenstrom an das eigentliche KNN „finn“ weitergibt. Die Ergebnisse der Inferenz werden als Datenstrom von „output_dma“ entgegen genommen und im Speicher abgelegt. Alle Blöcke wurden vom FINN-Framework automatisch generiert. In diesem Zusammenschluss ist die Schnittstelle für die Eingabe der Tiles äquivalent zum BNN-PYNQ.

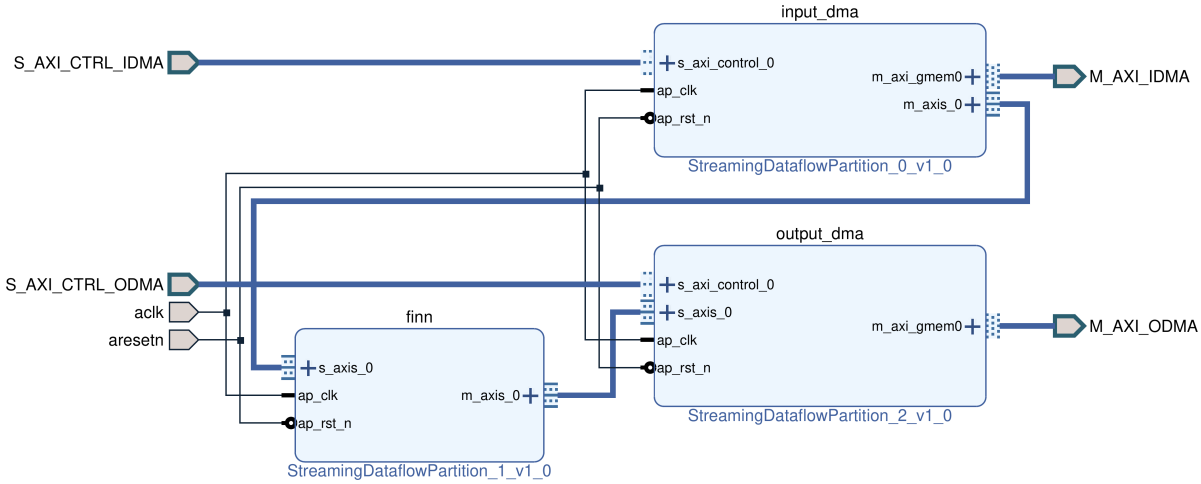


Abbildung 28: Blockdiagramm - FINN Anbindung (Tiles im Speicher)

Der Aufbau des Tilings als Gegenstück dieser Schaltung befindet sich in Abb. 29 auf der folgenden Seite. Der Block „command_reader“ ist der Befehlsgenerator, „extractor“ der Extraktor, „scaler“ der Skalierer und „stream2memory“ die Ablage. Die Schnittstellen „S_AXI_CTRL_*“ sind die AXI4-Lite Konfigurationsschnittstellen von Befehlsgenerator und Extraktor. Die Schnittstellen „M_AXI_*“ sind die AXI4-Schnittstellen zum Speicherzugriff durch Befehlsgenerator, Extraktor und Ablage.

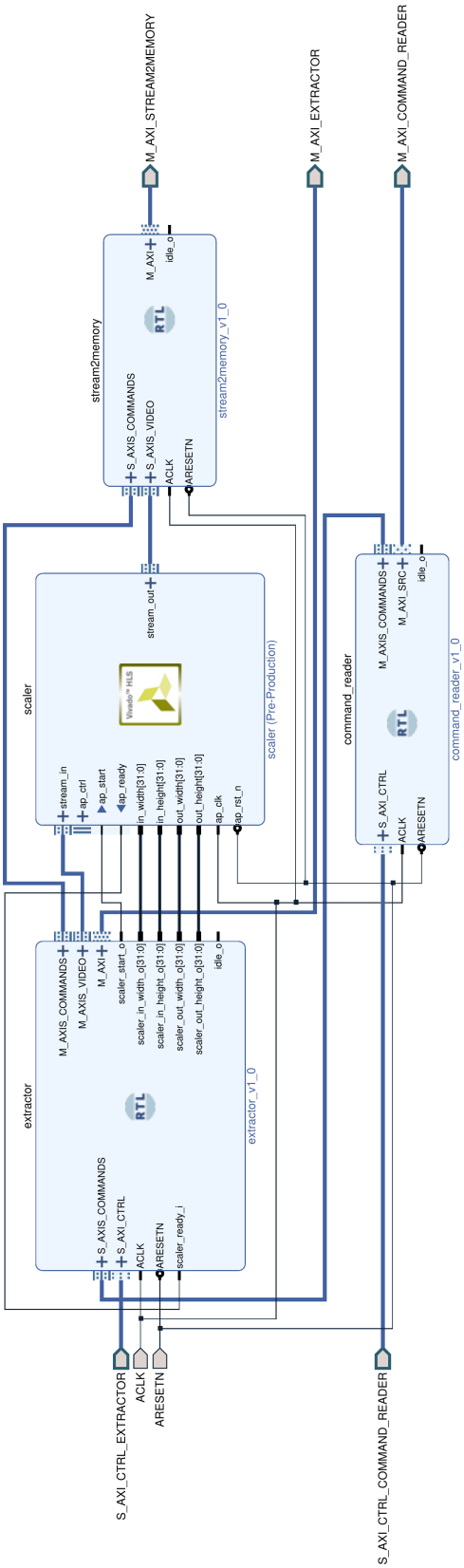


Abbildung 29: Blockdiagramm - Tiling der Gesamtschaltung (Tiles im Speicher)

Im BNN-PYNQ waren alle drei Blöcke fest in einem kombiniert, weshalb ausschließlich der Abruf und die Ablage der Daten aus einem Speicher möglich war. Da die fertig verarbeiteten Tiles den Skalierer als Datenstrom verlassen, kann zur Anbindung auf zwei Komponenten verzichtet werden: Die Ablage beim Tiling und der „input_dma“ beim KNN, sodass Tiling und KNN direkt auf dem FPGA verbunden sind und nicht mehr den Umweg über den Speicher nehmen müssen. Der reduzierte Aufbau für das KNN ist in Abb. 30 dargestellt. Weil das KNN einen Datenstrom mit einer Breite von einem Byte erwartet, während das Tiling alle drei Farbkanäle durch eine Datenbreite von drei Byte auf einmal weitergibt, war der Block „width_converter“ nötig, der entsprechend konvertiert und standartmäßig in Vivado verfügbar ist [11].

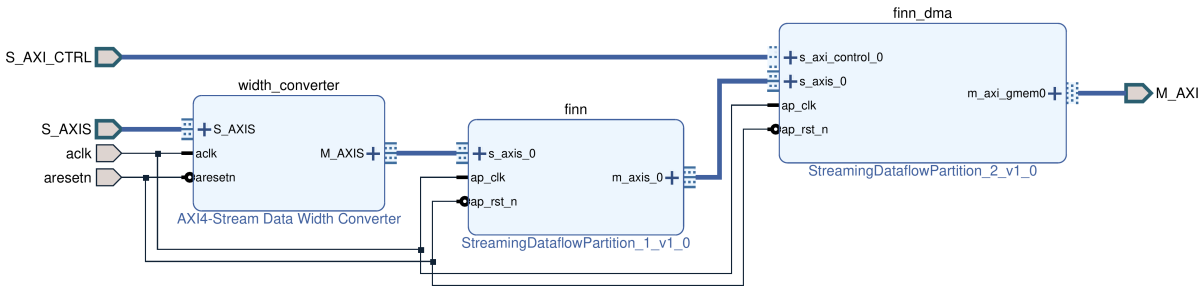


Abbildung 30: Blockdiagramm - FINN Anbindung (Tiles als Datenstrom)

Der entsprechende Aufbau des Tilings unterscheidet sich im Vergleich zu Abb. 29 hauptsächlich durch den Entfall der Ablage. Zusätzlich wurde das „Ready“-Signal des Adressbefehldatenstroms konstant auf logisch hoch gesetzt, da dieser ohne die Ablage nicht benötigt wird. Die Informationen aus den Adressbefehlen sind für das KNN nicht interessant, da es konstant von einer Auflösung von 32x32 ausgeht. Da die Unterschiede der Schaltung nur gering sind, wurde die entsprechende Abbildung 42 in den Anhang verschoben.
Auf den nächsten zwei Seiten folgen zwei Diagramme für eine Gesamtschaltung mit den beiden Varianten. Der Unterschied beider Schaltung lässt sich sehr schön an der Verbindung zwischen „tiling“ und „finn“ erkennen. Bei der Schaltung mit den Tiles im Speicher sind beide nicht direkt, sondern über „zynq“ verbunden, welcher den SoC und die Anbindung des DDR3-Speichers repräsentiert. Innerhalb dieses Blocks befinden sich ein Block für den Zynq, Interconnects und die Synchronisation des Resetsignals, welche alle von Vivado automatisch generiert und daher nicht weiter behandelt werden.
Der Block „hdmi_in_out“ enthält die Konvertierung des externen Bildsignals, die Ablage dessen im Speicher und gleichzeitig einer erneuten Ausgabe des Eingangsvideosignals über DVI, damit das Videosignal gleichzeitig zur Verarbeitung auf einem Monitor betrachtet werden kann.

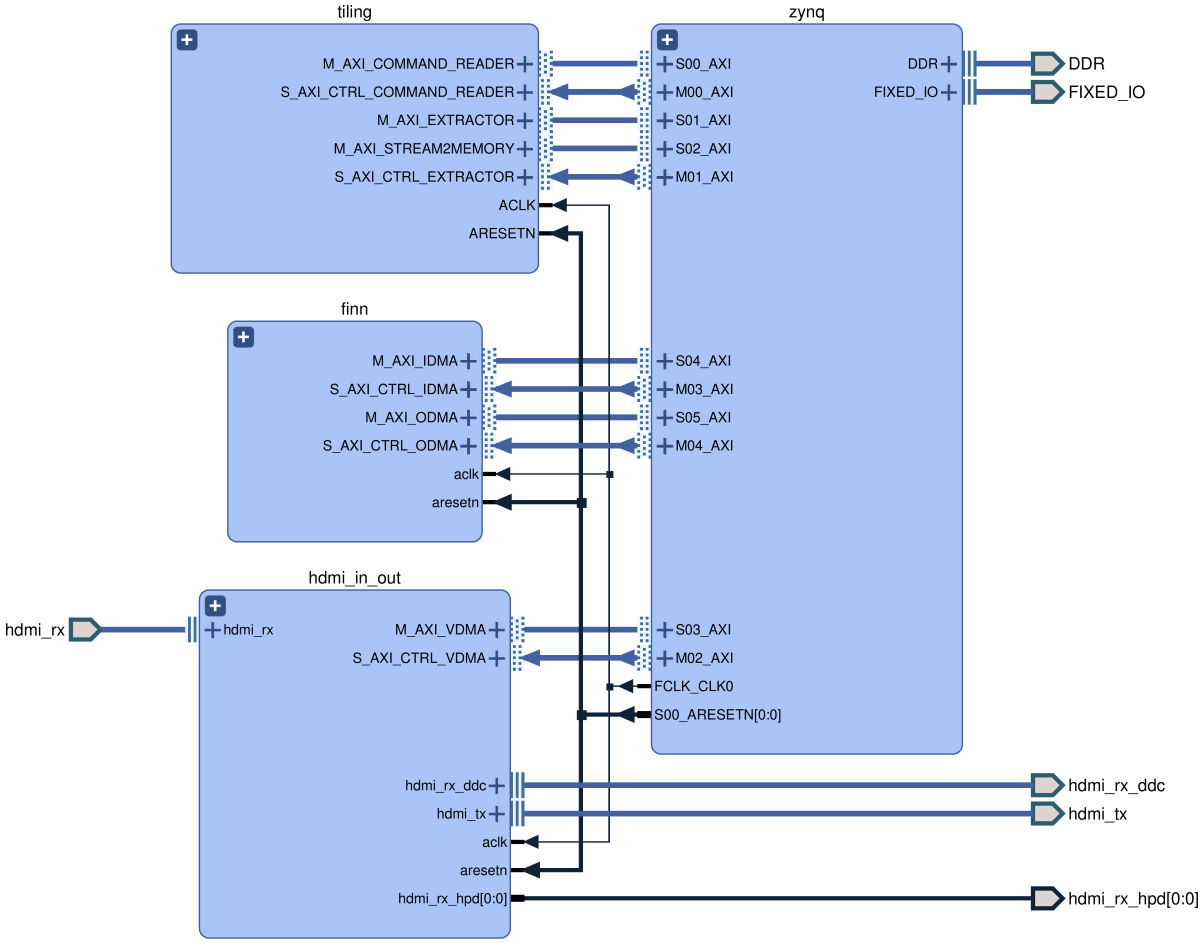


Abbildung 31: Blockdiagramm - Gesamtschaltung mit FINN (Tiles im Speicher)

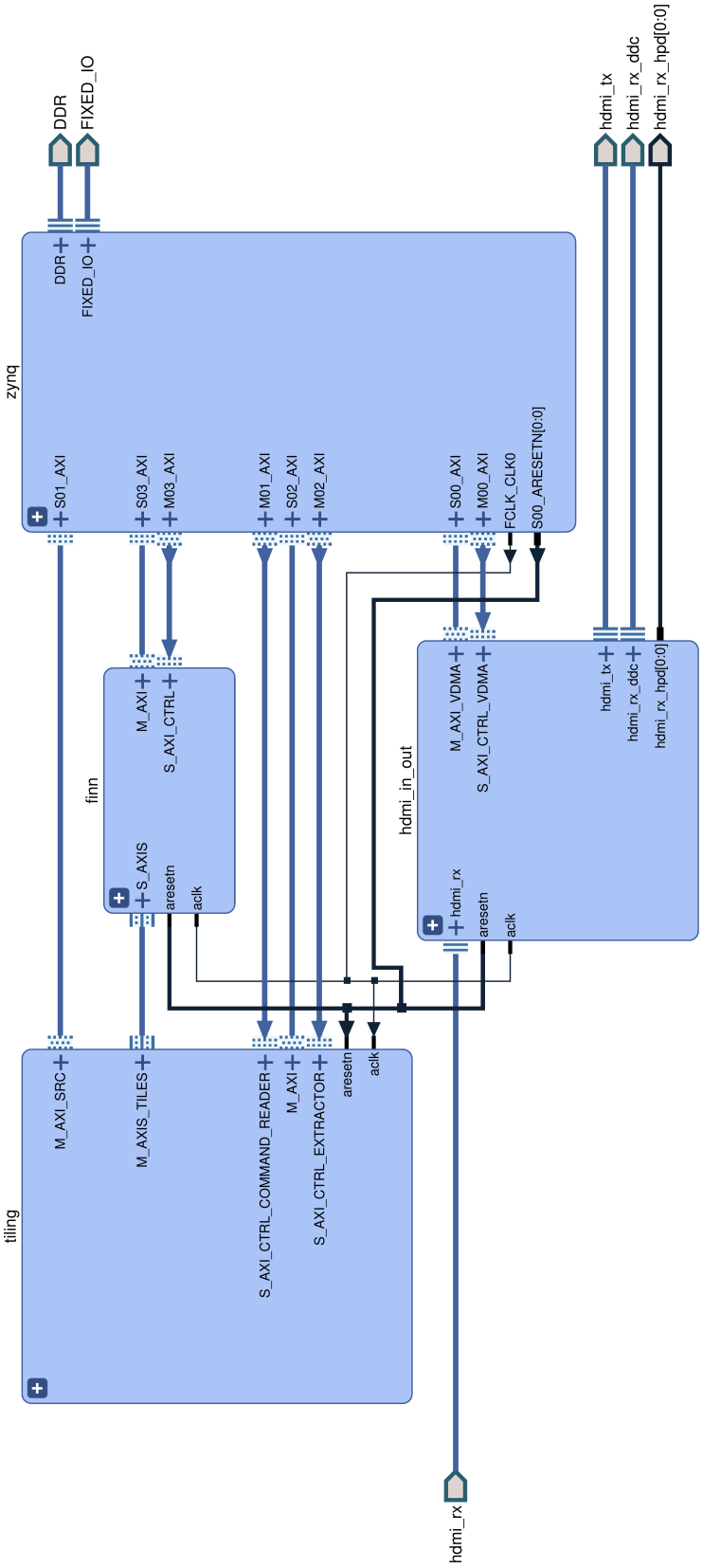


Abbildung 32: Blockdiagramm - Gesamtschaltung mit FINN (Tiles als Datenstrom)

6 Verifikation des Prototypen

Dieses Kapitel beschreibt die Verifikation der Funktionalität des erstellten Prototypen. Als Erstes werden die Tests der einzelnen Komponenten in Kap. 6.1 betrachtet. Die Beschreibung für Tests der Integration mehrerer Komponenten in Teilsysteme folgt in Kap. 6.2 und wird durch die Systemtests in Kap. 6.3 nochmals erweitert. Zuletzt folgt eine Validation in Kap. 6.4, um sicherzustellen, dass die richtigen Anforderungen an den Prototypen umgesetzt wurden.

6.1 Komponententests

In den Komponententest wurden die einzelnen Komponenten des Tilings in isolierter Form anhand ihrer Schnittstellen (als „Black Box“) verifiziert.

6.1.1 Testziele

Es sollten die erstellten Komponenten Befehlsgenerator, Extraktor, Skalierer und Ablage getestet werden. Die restlichen Komponenten werden erst in den späteren Tests beachtet, da die Funktionalität auf Komponentenebene bereits durch die Hersteller geprüft wurde. Für jede Komponente müssen als Eingabewerte sowohl eindeutig richtige, als auch Grenzwerte, getestet werden.

6.1.2 Testwerkzeuge

Alle Komponenten wurden mittels Testbenches in Vivado 2020.2 getestet, welche Blockdiagramme zur Einbindung der zu testenden Komponente und VHDL zur Stimulation der Eingabesignale und Überprüfung der Ausgabesignale nutzen. Anstatt die Testkomponente direkt in VHDL zu instanziieren, wurden die Blockdiagramme genutzt, da die erstellten Komponenten durch AXI4 teils komplexe Schnittstellen mit vielen Leitungsbündeln besitzen. Die Verifikation dieser Schnittstellen ist ein sehr komplexes Unterfangen, sodass zu diesem Zwecke ein fertiger IP-Core von Xilinx der „AXI Verification IP“ [10] verwendet wurde, welcher bei Verletzung der Spezifikation Warnungen und Fehler erzeugt. Zusammen mit diesem IP-Core wurden die Testkomponenten folgend an weitere IP-Cores angebunden, um die Architektur des Zynq-SoC zu simulieren. Besonders hervorzuheben ist dabei der „AXI BRAM Controller“ [8] zusammen mit dem „Block Memory Generator“ [12], die zusammen in der Verhaltenssimulation einen über AXI4 angebundenen Speicher emulieren können. Der Inhalt dieses Speicher kann sowohl im Simulator inspiziert, als auch über eine zweite Speicherschnittstelle zur Befüllung oder Auslesen des Speichers zum Abgleich der Daten genutzt werden.

6.1.3 Testspezifikation und -durchführung

Die Testbenches wurden als Teil der Implementation zur testgetriebenen Entwicklung umgesetzt. Zuerst wurde jeweils eine fehlschlagende Testbench entwickelt, die folgend durch eine erste Implementation in einen erfolgreichen Zustand versetzt wurde. Dieser Vorgang wurde folgend im klassischen Zyklus der testgetriebenen Entwicklung wiederholt, welcher in Abb. 33 dargestellt ist.

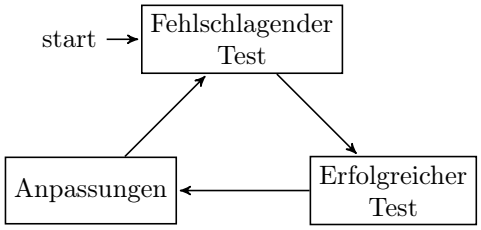


Abbildung 33: Methode zur Erstellung der Komponententests

Am Anfang des Quellcodes jeder Testbenches wurden die Testziele in jedem Zyklus ergänzt und nach der Umsetzung der Ziele eine Begründung für die Erfülltheit hinzugefügt. Zusätzlich zu den Testfolgen

im Quellcode verifiziert der „AXI Verication IP“ die AXI4-Schnittstelle, soweit vorhanden, und bricht die Simulation ab, falls die Spezifikation nicht eingehalten wurde.
Die erstellten Blockdiagramme für die einzelnen Testbenches sind im Anhang E abgebildet.

6.1.4 Testergebnisse

Der letzte Stand der Testbenches, welche alle erfolgreich durchlaufen, kann auf dem beiliegenden Datenträger im Ordner *Vivado/dynamic_tiling/vhdl/* betrachtet werden. Die Zuordnung der Dateinamen ist folgendermaßen: Befehlsgenerator → command_readerTb.vhd, Extraktor → extractorTb.vhd, Skalierer → scalerTb.vhd und Ablage → stream2memoryTb.vhd.
Die Protokolle in Form von Konsolenausgaben der letzten Durchführung der Testbenches befinden sich im Anhang F.1.

6.2 Integrationstests

Um sicherzustellen, dass die Schaltung auch auf echter Hardware funktioniert, wurden auf dieser Integrationstests der Teilsysteme ausgeführt.

6.2.1 Testziele

Die tatsächliche Hardware unterscheidet sich leicht von der Simulationsumgebung, weshalb diese Unterschiede erkannt und betrachtet werden sollen.

6.2.2 Testspezifikation- und durchführung

Der erste Test wurde zur Verfikation der Konfiguration und Anbindung der Komponenten zur Umwandlung und Ablage des externen DVI-Signals im Speicher durchgeführt. Dieser Test ist notwendig, da zwar die einzelnen Komponenten von den Herstellern getestet wurden, aber noch nicht die aufgebaute Gesamtschaltung dieser Arbeit. Weiterhin wird ein externes Signal verwendet, weshalb die externe Schnittstelle richtig verbunden und der richtige Referenztakt eingestellt sein muss, da ansonsten das Signal nicht richtig erkannt wird. Es wird nur eine Bildquelle und eine Auflösung getestet, weil eine falsche Anbindung überhaupt nicht funktionieren würde. Falls diese eine Konfiguration funktioniert, ist durch die Komponententests der Hersteller auch die Funktionalität für andere Konfiguration sichergestellt. An die Platine wurde dazu ein Raspberry Pi 4 angeschlossen, die Auflösung auf 640x480 gesetzt und eine herkömmliche Desktopoberfläche anzeigt. Der VDMA wurde folgend so konfiguriert, dass er das aktuelle Bild im DDR3-Speicher ab einer definierten Startadresse ablegt. Der Speicherbereich des Bildes wurde danach durch ein Programm auf der CPU des SoCs über die serielle Schnittstelle ausgegeben, die wiederum über USB an einen weiteren Hostrechner angebunden wurde. Auf diesem wurden empfangenen Daten mittels Python in eine PNG-Bilddatei umgewandelt. Die Bilddatei wurde geöffnet und visuell mit der Ausgabe des Raspberry Pis auf einem Monitor abgeglichen. Der Test ist erfoglreich, falls beide Darstellungen identisch sind.
Der zweite Test diente zur Verifikation der Funktionalität des Tilings ausserhalb der Simulation. Im Vergleich zu den Testbenches unterscheidet sich der verwendete Speicher, sodass hier das größte Fehlerpotential vermutet wird. Für das Tiling wird nur genau eine Konfiguration getestet. Falls die Speicheranbindung für diese funktioniert, ist durch die Komponententest dies auch für andere sichergestellt. Durch die CPU wurde im Speicher ein generiertes Testbild abgelegt, wonach das Tiling so konfiguriert wurde, dass genau ein Tile mit genau der Testbildgröße extrahiert und an einer anderen Adresse im Speicher abgelegt wurde. Der Zielspeicherbereich wurde zuvor mit Nullen gefüllt, um sicherzustellen, dass die Daten tatsächlich durch das Tiling abgelegt werden. Der Test ist erfolgreicht, falls beide Speicherbereiche die gleichen Daten enthalten. Ein Protokoll des Tests wurde über die serielle Schnittstelle der CPU ausgegeben und in einer Textdatei festgehalten.

6.2.3 Testergebnisse

Für die Anbindung der externen Bildquelle und das Tiling wurden jeweils mögliche Fehlerquellen entsprechend der Testziele betrachtet und diese durch zwei Tests abgedeckt, die beide erfolgreich durchgeführt werden konnten.
Das ausgelesene Bild aus dem ersten Test und ein Protokoll der Konsolenausgaben des zweiten Tests können im Anhang F.2 eingesehen werden.

6.3 Systemtests

Zuletzt wurde das in Kap. 5.5 entwickelte Gesamtsystem betrachtet, um die Funktionalität im Zusammenschluss von externem Videosignal, Tiling und KNN sicherzustellen.

6.3.1 Testziele

Die Integrationstests haben bereits gezeigt, dass die Teilsysteme der Bildsignalanbindung und des Tilings im Zusammenschluss ihrer Komponenten auf Hardware wahrscheinlich funktionieren. Nun soll noch einmal ein Zusammenschluss der beiden Teilsysteme und des KNN getestet werden, um die Funktionalität eines Gesamtsystems sicherzustellen. Dazu soll getestet werden, dass der Datenfluss von der externen Bildquelle bis zu einem Inferenzergebnisse durch das KNN funktioniert. Jegliche Verarbeitung der Bilddaten muss durch das Gesamtsystem auf dem FPGA geschehen. Nur die Konfiguration und Ausgabe der Ergebnisse darf durch die CPU erfolgen. Es muss sowohl die Anbindung des Tilings an das KNN über den DDR3-Speicher, als auch über den Datenstrom innerhalb des FPGAs getestet werden. Als KNN soll das FINN verwendet werden, da es beide Anbindungen unterstützt. Das BNN-PYNQ wird nicht separat getestet, da es über den Speicher eine identische Schnittstelle, wie das FINN, hat.

6.3.2 Testwerkzeuge

Für den Systemtest wurde eine externe Bildquelle benötigt, wozu ein Raspberry Pi 4 über HDMI an das PYNQ-Z1 angeschlossen wurde. Die Auflösung wurde auf 640x480 gesetzt. Auf dem Pi wurden mit Python vier Bilder in den Größen 32x32, 64x64, 128x128 und 256x256 aus dem CIFAR10 Datensatz [26] an unterschiedlichen Stellen auf dem Bildschirm angezeigt. Eine beispielhafte Darstellung ist in Abb. 34 zu sehen.

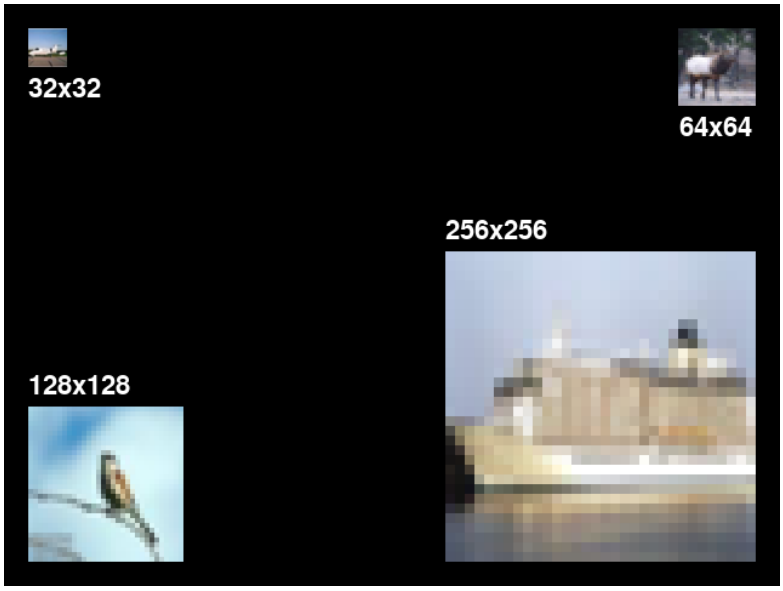


Abbildung 34: Anzeige von Testbildern für die Systemtests

Das Tiling wurde daraufhin durch die CPU mit den in Kap. 5.4 beschriebenen Methoden konfiguriert. Der Extraktor wurde so konfiguriert, dass er die abgelegten Bilddaten des externen Bildsignals liest, alle Tiles auf eine Größe von 32x32, passend für FINN, skaliert und die generierten Tiles an einer bestimmten Stelle im Speicher ablegt. Der Befehlsgenerator wurde instruiert vier Befehle zu konfigurieren, welche genau die Bildregionen treffen, wo der Pi die Bilder anzeigt. Die genauen Befehle sind in Tabelle 4 aufgeführt. Für das Auslesen der Tiles aus dem Speicher durch FINN wurde die Adresse des entsprechenden DMAs auf die im Extraktor konfigurierte Zieladresse der Tiles gesetzt und die Anzahl der Bilder entsprechend der Anzahl der Befehle auf vier gesetzt. Für die Anbindung von FINN über den Datenstrom entfällt diese Konfiguration, während die restliche Konfiguration des Tilings unverändert bleibt. Für die Ablage der Inferenzresultate musste zuletzt der zweite DMA von FINN mit der Zieladresse für die Ergebnisse und erneut die vier für die Anzahl konfiguriert werden.

Nr.	X	Y	Breite	Höhe
0	20	20	32	32
1	556	20	64	64
2	20	332	128	128
3	364	204	256	256

Tabelle 4: Extraktionsbefehle für die Systemtests

Als nächstes wurde über die serielle Schnittstelle eine Möglichkeit geschaffen die Ausführung zu starten und die Inferenzergebnisse auszugeben, indem ein ASCII „f“ an die CPU geschickt wird. Dazu wird, falls vorhanden, zuerst der DMA zum Auslesen der Tiles für FINN gestartet, danach der Befehlsgenerator und zuletzt der DMA zur Ablage der Ergebnisse von FINN. Diese Ergebnisse wurden über Speicherzugriffe ausgelesen und zur einfacheren Interpretation zusammen mit dem Namen der Klasse ausgegeben. Für die Bilder in Abb. 34 ist die Ausgabe beispielsweise folgende:

```
Results:
0 (airplane)
4 (deer)
2 (bird)
8 (ship)
```

Listing 4: Beispielausgabe des Systemtests

6.3.3 Testspezifikation und -durchführung

Jeweils für die direkte Anbindung von FINN an das Tiling und die Anbindung über den DDR3-Speicher wurden die folgenden Tests durchgeführt.
Als Erstes wurden vier Bilder unterschiedlicher Klassen angezeigt, um sicherzustellen, dass tatsächlich unterschiedliche Tiles vom KNN verarbeitet werden. Der Test ist erfolgreich, wenn vier unterschiedliche Klassifizierungen ausgegeben werden.
Als Zweites wurde an allen vier Positionen zwei mal das gleiche Bild angezeigt. Die beiden Bilder wurden so gewählt, dass sie bei korrekter Anbindung auch tatsächlich vom KNN erkannt werden würden, da dieses die Bilder aus dem Datensatz nicht mit hunderprozentiger Genauigkeit klassifiziert. Der Test ist erfolgreich, falls für beide Bilder an allen vier Positionen die richtige Klassifizierung ausgegeben wird.
Für alle Tests wurde eine Bildschirmaufnahme der Anzeige getätigt und die serielle Ausgabe dokumentiert.

6.3.4 Testergebnisse

In Tabelle 5 sind die Ergebnisse der Systemtests aufgeführt. Die Spalte Anzeige enthält eine Bildschirmaufnahme der Anzeige des Raspberry Pis. Die Bildschirmaufnahme für den ersten Test ist in Abb. 34 nochmal größer dargestellt. Die Spalten der Ausgaben enthalten die jeweilige serielle Ausgabe der Inferenzergebnisse für Speicher-/Datenstromanbindung. Die Reihenfolge der Ergebnisse ist oben-links, oben-rechts, unten-links und unten-rechts. In der letzten Spalte wurde ein Haken gesetzt, falls die oben genannte Erfolgsbedingung erfüllt wurde, was für alle Tests der Fall war.
Die Bildschirmaufnahmen und Ausgabeprotokolle befinden zusätzlich zur Tabelle auch in hoher Auflösung auf dem beigelegten Datenträger unter *Protokolle/Tests/Systemtests*.

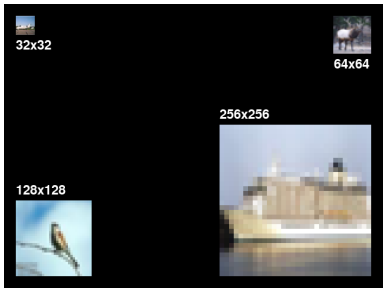
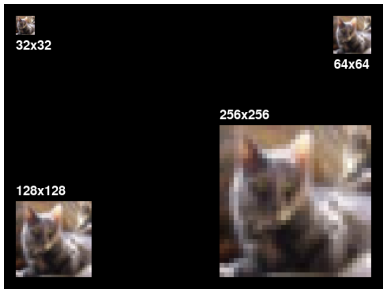
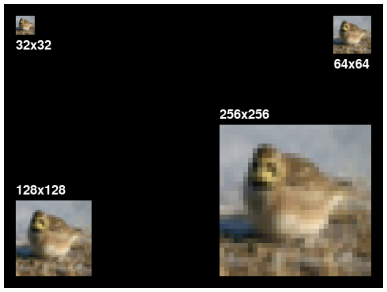
Nr.	**Anzeige**	**Ausgabe (Speicher)**	**Ausgabe (Datenstrom)**	**OK?**
1		Results: 0 (airplane) 4 (deer) 2 (bird) 8 (ship)	Results: 0 (airplane) 4 (deer) 2 (bird) 8 (ship)	✓
2a		Results: 3 (cat) 3 (cat) 3 (cat) 3 (cat)	Results: 3 (cat) 3 (cat) 3 (cat) 3 (cat)	✓
2b		Results: 2 (bird) 2 (bird) 2 (bird) 2 (bird)	Results: 2 (bird) 2 (bird) 2 (bird) 2 (bird)	✓

Tabelle 5: Extraktionsbefehle für die Systemtests

Die gesamte Verarbeitung der Bilddaten ist für diesen Test durch die Gesamtschaltung auf dem FPGA mit Anbindung von FINN über Speicher/Datenstrom erfolgt, wodurch alle Testziele erfüllt wurden.

6.4 Validation

Zur Sicherstellung, dass die Anforderungen nicht nur richtig, sondern auch die richtigen Anforderungen umgesetzt wurden, wird folgend die Implementation mit den Anforderungen aus Kap. 5.1 anhand ihrer Identifikationsnummer verglichen.
Die Anforderungen **F1** und **F2** wurden umgesetzt, da die Bilder eines externen DVI-Signals im Speicher abgelegt und von dort aus weiterverarbeitet werden. Das Ablageformat **F3** wurde entsprechend umgesetzt und durch den Systemtest nachgewiesen, dass die erzeugten Tiles tatsächlich vom KNN verarbeitet werden können.
Die Auflösung des Gesamtbildes **F4a** kann über AXI4-Lite im Extraktor konfiguriert werden. Die Farbtiefe **F5a** wurde nicht konfigurierbar gemacht, da es starke Änderungen an der Architektur der Komponenten erfordert, die nicht trivial zur Laufzeit anpassbar sind. Aus Zeitgründen wurde sich daher auf die geforderte Farbtiefe von 24 Bit RGB beschränkt. Die Anzahl Tiles **F4b**, die Positionen der Bildregionen **F4c** und die Größe der Bildregionen **F4d** wird über den Befehlsdatenstrom des Extraktors bestimmt. Die Generierung der Befehle erfolgt durch den Befehlsgenerator, welcher eine konfigurierbare Anzahl Befehle mit Position und Größe der Bildregionen aus einem Speicher liest. Die skalierte Auflösung **F4e** kann über AXI4-Lite im Extraktor konfiguriert werden, welcher wiederum den Skalierer konfiguriert. Der Skalierungsalgorithmus **F5b** kann im Prototypen nicht konfiguriert werden und ist daher festgesetzt auf bilineare Skalierung. Um den Skalierer konfigurierbar zu machen, müsste dessen Quellcode um eine entsprechende Auswahlschnittstelle angepasst werden, welche vom Extraktor konfiguriert werden könnte, der wiederum die Konfiguration über die bereits vorhandene AXI4-Lite Schnittstelle erhält. Die Anforderung **F6** wurde durch den Aufbau einer solchen Schaltung für den Systemtest erfüllt.
Die Anforderung **N1** ist erfüllt, da alle erstellten Komponenten tatsächlich auf dem PYNQ-Z1 lauffähig sind, was durch Integrations- und Systemtests gezeigt wurde. Das Tiling wurde entsprechend **N2** auf dem FPGA umgesetzt und wird lediglich von der CPU konfiguriert. Wie durch **N3** gefordert, ist die Schaltung synthetisierbar, was analog zu **N1** durch den Systemtest gezeigt wurde.

7 Evaluation

In diesem Kapitel werden verschiedene Aspekte der implementierten Lösung einzeln evaluiert, um im folgenden Kapitel mit den gewonnenen Erkenntnissen die Fragestellungen aus Kap. 1.3 zu beantworten. Als Erstes wird in Kap. 7.1 der Verbrauch der Ressourcen auf dem FPGA durch die verschiedenen Komponenten betrachtet. Als Nächstes folgt in Kap. 7.2 eine Betrachtung der Pfadlängen innerhalb der Schaltung und ob diese bei der gewählten Frequenz ein Problem darstellen. In Kap. 7.3 wird die Verarbeitungsgeschwindigkeit der neuen Lösung gemessen und gegen eine Softwarelösung verglichen. Zuletzt folgt eine Beschreibung der Problematiken bei der Umsetzung auf einem FPGA.

7.1 Ressourcenverbrauch

Dieser Abschnitt misst und analysiert den Ressourcenverbrauch der Komponenten auf dem FPGA. Diese Betrachtung ist notwendig, da alle Komponenten des Gesamtsystems auf dem FPGA Platz finden müssen, ansonsten kann die Schaltung nicht für die Zieldhardware synthetisiert werden.

7.1.1 Durchführung der Messung

Zur Messung des Ressourcenverbrauchs wurden die integrierten Analysewerkzeuge von Vivado genutzt. Als Schaltung wurde ein Blockdiagramm mit einer Instanz des Tilings, der Anbindung eines externen Bildsignals und dem BNN-PYNQ erstellt, welches in Abb. 43 im Anhang betrachtet werden kann. Auf den Ressourcenverbrauch auf dem FPGA haben die zur Laufzeit konfigurierbaren Parameter keinen Einfluss, weshalb folgend nur die Parameter zum Zeitpunkt der Hardwaresynthese betrachtet werden. Sowohl für den Extraktor, als auch die Ablage, kann die Burstlänge über Generics konfiguriert werden. Anhand des gewählten Werten wird intern die Größe einer Warteschlange gewählt, weshalb mit höherer Länge der Ressourcenverbrauch steigt. Zum Vergleich wurden Extraktor und Ablage jeweils mit den Burstlängen 1, 8, 16 und 256 gemessen. Die verwendete AXI4-Schnittstelle unterstützt Längen von 1 bis 256, weshalb diese als äußere Grenzen gewählt wurden. Der adressierte DDR3-Speicher ist jedoch über den AXI3-Bus angebunden, welcher nur Längen von 1 bis 16 unterscheidet, weshalb die Grenze und eine Mittlere zusätzlich betrachtet wurden. Der Skalierer und Befehlsgenerator sind nicht zur Hardwaresynthese konfigurierbar. Für die Parameter zur Anbindung eines externen Bildsignals wurde nur die Konfiguration mit Standartparametern betrachtet. Das BNN-PYNQ wurde in der Konfiguration „cnvW1A1“ verwendet.
Nach der Synthese und Implementation in Vivado 2020.2 wurde die Ansicht „Implemented Design“ geöffnet und in der Liste „Netlist“ über die Hierarchie des Blockdiagramms die jeweils zu betrachtende Komponente ausgewählt. Im Bereich „Cell Properties“ wurde der Reiter „Statistics“ ausgewählt und die aufgeführten Werte notiert. Zusätzlich wurden Bildschirmaufnahmen getätigt, die auf dem beigelegten Datenträger im Ordner *Protokolle/Evaluation/Ressourcenverbrauch* zu finden sind. Dieser Vorgang wurde für die restlichen Komponenten wiederholt

7.1.2 Messergebnisse

Die Tabelle 6 enthält die benötigten Logikressourcen für die einzelnen Komponenten. Die Bezeichner haben folgende Bedeutung: FLOP_LATCH → Flip Flops und Latches, LUT → Lookup-/Verweistabellen, CARRY → Übertragslogik, MUXFX → Multiplexer und MULT → Multiplizierer.

Komponente	FLOP_LATCH	LUT	CARRY	MUXFX	MULT
BNN-PYNQ	41.417	30.744	3.130	2.993	24
Anbindung vom externen Videosignal	804	627	28	24	0
Externes Bildsignal im Speicher ablegen	2.279	1.636	56	0	0
Kommandogenerator	276	192	16	0	0
Extraktor					
- Burstlänge: 1	1.724	943	80	130	3
- Burstlänge: 8	2.101	1.074	81	210	3
- Burstlänge: 16	2.523	1.236	81	234	3
- Burstlänge: 256	15.300	5.409	82	2.730	3
Skalierer	6.041	5.630	745	0	40
Ablage					
- Burstlänge: 1	765	470	55	64	0
- Burstlänge: 8	1.130	724	55	173	0
- Burstlänge: 16	1.538	971	55	298	0
- Burstlänge: 256	13.631	8.223	55	5.082	0

Tabelle 6: Messergebnisse - Ressourcenverbrauch (Logik)

Die Tabelle 7 enthält den Verbrauch sonstiger Ressourcen. Die Bedeutung der Bezeichner ist folgendermaßen: BMEM → Block RAM, DMEM → Verteilter RAM aus Schieberegistern/Logik, CLK → Erzeugung oder Anpassung von Taktsignalen und IO → Ressourcen zur Anbindung an physische Anschlüsse des FPGAs.

Komponente	BMEM	DMEM	CLK	IO
BNN-PYNQ	124	3.154	0	0
Anbindung vom externen Videosignal	1	49	7	14
Externes Bildsignal im Speicher ablegen	2,5	188	0	0
Kommandogenerator	0	0	0	0
Extraktor				
- Burstlänge: 1	0	0	0	0
- Burstlänge: 8	0	0	0	0
- Burstlänge: 16	0	0	0	0
- Burstlänge: 256	0	0	0	0
Skalierer	3	95	0	0
Ablage				
- Burstlänge: 1	0	0	0	0
- Burstlänge: 8	0	0	0	0
- Burstlänge: 16	0	0	0	0
- Burstlänge: 256	0	0	0	0

Tabelle 7: Messergebnisse - Ressourcenverbrauch (Speicher/Taktgeneratoren/IO)

7.1.3 Analyse der Messergebnisse

Der Ressourcenverbrauch des Tilings und der Anbindung einer externen Bildquelle ist ausreichend gering, um eine Schaltung inklusive BNN-PYNQ für die Zielhardware zu übersetzen.
Nun soll einmal der RAM Verbrauch genauer betrachtet werden. Die verwendete Hardware besitzt 140 Block RAM Blöcke á 36 Kbit von denen 124 durch das BNN-PYNQ belegt werden. Von den restlichen

16 Blöcken werden durch die Anbindung des externen Signals 3,5 und durch das Tiling 3 Blöcke benötigt, was sehr sparsam ist. Gleichermaßen fällt auch der Verbrauch von generiertem RAM sehr gering aus. Extraktor und Skalierer verbrauchen keine speziellen Ressourcen für RAM. Durch die Messungen mit verschiedenen Burstlängen ist erkennbar, dass die Puffer der FIFO Warteschlangen aus einer Mischung verschiedener Logikressourcen umgesetzt wurden. Dies ist wahrscheinlich durch die relativ spezielle Implementierung der Warteschlange der Fall, da diese Zugriff auf mehrere Werte gleichzeitige erlaubt, wodurch die dedizierten RAM Implementierungen nicht anwendbar sind. Die Implementierung von Puffern aus Logikressourcen ist vergleichsweise aufwendiger und unter normalen Umständen nicht wünschenswert, jedoch unter den Umständen dieses Anwendungsfalls vorteilhaft, da mehr RAM Ressourcen dem BNN-PYNQ zur Verfügung stehen.
Der Verbrauch an restlichen Ressourcen fällt im Vergleich zum BNN-PYNQ vernachlässigbar gering aus.

7.2 Timinganalyse

Da alle entwickelten Komponenten zur steigendenen Flanke eines Systemtakts geschaltet werden, werden Signale zu diesem Zeitpunkt in Flip Flops zwischengespeichert. Das Umschalten dieser FlipFlops unterliegt verschiedenen physikalischen Einschränkungen, wodurch dies nicht mit beliebiger Geschwindigkeit und zu beliebigen Zeitpunkten erfolgen kann. Damit ein Flip Flop zuverlässig schaltet, müssen durch den Hersteller spezifizierte Grenzwerte bezüglich Vorbereitungs- und Haltezeit eingehalten werden. Falls Grenzwerte nicht eingehalten werden, kann es schnell zu pseudozufälligen Fehlern kommen, die nur schwer nachvollziehbar sind, weshalb das Risiko eines solchen Fehlerzustands durch eine vorherige Messung und Analyse in diesem Abschnitt verringert werden soll.

7.2.1 Durchführung der Messung

Zur Messung des Timings wurde Vivado 2020.2 genutzt, welches die resultierenden Zeiten auf den verschiedenen generierten Pfaden der Schaltung automatisch berechnet. Der wichtigste Wert ist der „Worst Negative Slack“ (WNS), welcher die restliche Zeit auf dem längsten Pfad beschreibt. Abgezogen von der Länge der Taktperiode ergibt dieser Wert die Dauer des Pfades. Sofern kein einziger Pfad die Länge der Taktperiode der eigenen Taktdomäne überschreitet, wird das Timing eingehalten. Für die Messung wurde ein Blockdiagramm einer Minimalschaltung mit den Komponenten des Tilings, sowie einer Minimalanzahl Komponenten zur Anbindung an den AXI-Bus des SoC, erstellt (siehe Abb. 39 im Anhang). Alle Komponenten wurden an ein Taktsignal mit einer Frequenz von 100 Mhz (Taktperiode: 10ns) angebunden. Folgend wurde die Synthese und Implementation in Vivado ausgeführt und der „Timing Report“ als Textdatei gespeichert. Aus diesem wurden folgend der WNS für Vorbereitung (Setup) und Haltezeit (Hold) und die verantwortliche Komponente für den längsten Pfad entnommen.
Als nächstes wurde das Blockdiagramm um das BNN-PYNQ, sowie die Einbindung des externen Bildsignals, ergänzt (siehe Abb. 43 im Anhang) und erneut analog zur Minimalschaltung der Timingbericht erzeugt und die Ergebnisse notiert. Diese weitere Messung wurde vorgenommen, da bei steigender Ausnutzung der Ressourcen des FPGAs die Platzierung der einzelnen Schaltnetze auf der Chipfläche für die Werkzeuge schwieriger wird und eine potentiell weniger optimale Verteilung erzeugt wird. Verschiedene Ressourcen sind nur an bestimmten Positionen innerhalb des FPGAs vorzufinden, weshalb verbundene Schaltnetze weiter voneinander entfernt sein können oder mit schlechteren Mitteln umgesetzt werden und sich somit die Pfadzeiten ändern.

7.2.2 Messergebnisse

Die Tabelle 8 enthält den WNS und die maximale Pfaddauer für die Implementation der zwei Blockdiagramme, sowie die verantwortliche Komponente für den längsten Pfad.

	Minimalschaltung	Gesamtschaltung
Taktfrequenz (Mhz)	100	100
Taktperiode (ns)	10	10
WNS Setup (ns)	0,409	0,013
WNS Hold (ns)	0,013	0,003
Verantwortliche Komponente	Skalierer	Skalierer

Tabelle 8: Messergebnisse - Timing

7.2.3 Analyse der Messergebnisse

Alle Pfadlängen befinden sich innerhalb der gewählten Taktperiode, weshalb die zeitlichen Beschränkungen grundsätzlich eingehalten werden. Die maximalen Pfadzeiten befinden sich jedoch sehr nah am Maximum und lassen bei der gewählten Taktfrequenz einen Spielraum von deutlich weniger als einer Nanosekunde. Theoretisch könnte die Taktfrequenz verringert werden, was jedoch die Leistungsfähigkeit der Schaltung senkt. Verursacher für die maximalen Pfade ist bei beiden Schaltungen die Skaliererkomponente des Tilings, welche vollkommen identisch in beiden Schaltungen ist. Trotz äquivalenter Skalierer hat sich die maximale Pfadzeit erwartungsgemäß von der Minimalschaltung zur Gesamtschaltung verschlechtert. Dies lässt sich dadurch erklären, dass durch die zusätzlichen Komponenten die einzelnen Logikressourcen des Skalierers weiter voneinander entfernt platziert wurden. Für eine zuverlässige und einfache Verwendbarkeit müssen zwingend die Ursachen für die langen Pfade des Skalierers unabhängig einer Gesamtschaltung untersucht werden. In der Bearbeitungszeit dieser Arbeit konnte die Ursache nicht gefunden werden, da der Skalierer mit einer Hochsprache umgesetzt wurde und durch die hohe Abstraktion nur wenig Einblicke in die interne Funktionsweise möglich waren. Der aus der Hochsprache generierte Quellcode in einer Hardwarebeschreibungssprache brachte auch keine weiteren Erkenntnisse.

7.3 Verarbeitungsgeschwindigkeit

In diesem Abschnitt wird die Verarbeitungsgeschwindigkeit des Tilings auf dem FPGA gemessen, ausgewertet und gegen eine Softwarelösung verglichen.

7.3.1 Durchführung der Messung

Für die Messung wurde ein Blockdiagramm ausschließlich mit dem Tiling und den minimal notwendigen Komponenten zur Anbindung an den AXI4-Bus des SoCs erstellt, welches im Anhang in Abb. 44 betrachtet werden kann. Vom Tiling wurden vier parallele Instanzen platziert, die separat an die vier nativen „High Performance Ports“ zum SoC angebunden sind. Die Geschwindigkeit wurde anhand der Rechenzeit von 10.000 Tiles/Instanz für eine Zielgröße von 32x32 anhand sechs verschiedener Auflösungen gemessen: 64x64, 96x96, 100x100, 128x128, 150x150 und 160x160. Diese Auflösungen wurden dem Messaufbau einer Softwarelösung entnommen [15], die später zum Vergleich genutzt wird. Durch die Liste der Auflösungen werden sowohl ganzzahlige, als auch bruchteilige, Skalierungsfaktoren untersucht. Betrachtet werden ausschließlich quadratische Bildbereiche, da auch das erwartete Bild für das KNN mit quadratisch ist. Durch den internen Aufbau sollte die Rechenzeit mit den Anzahl Zeilen linear skalieren, allerdings sollte dies noch einmal genauer betrachtet werden, falls zukünftig rechteckige Bildbereiche benötigt werden. Die Messung der Zeit erfolgte durch die CPU des SoCs anhand eines Timers, der mit dem Takt der CPU von 650 Mhz zählt. Zuerst wurden die Befehle für jede Instanz des Tilings und eine Auflösung im Speicher vorbereitet und Befehlsgenerator/Extraktor konfiguriert. Danach wurde der aktuelle Stand des Zählers zwischengespeichert und die gewünschte Anzahl Instanzen gestartet. Durch Polling der Statusregister des Tilings wurde in einer Schleife auf die Fertigstellung gewartet und danach erneut der Zählerstand zwischengespeichert. Die Differenz der beiden Zählerstände ergaben die benötigte Rechenzeit in Takten, welche folgend anhand der Taktfrequenz in Mikrosekunden umgerechnet und durch die Anzahl Tiles geteilt wurden. Dieser Vorgang wurde in einer Schleife für alle Auflösungen wiederholt, über eine serielle Schnittstelle die Zeiten ausgegeben und diese in einer Textdatei zum Protokoll festgehalten. Diese Messung wurde zuerst für die Burstlängen 8, 16, 32, 64, 128 und 256 mit einer

Instanz getestet, da mit längerer Burstlänge theoretisch weniger Verzögerungen entstehen und die Geschwindigkeit steigen sollte. Die Längen wurden gewählt, da das Tiling nur Zweierpotenzen unterstützt und AXI4 auf eine Burstlänge von 256 begrenzt ist. Als Nächstes wurde die Messung für eine unterschiedliche Anzahl paralleler Instanzen bei einer festen Burstlänge von 32 wiederholt. Maximal wurden vier parallele Instanzen getestet, da sich die Schaltung mit weiteren nicht mehr synthesieren lies. Die Ressourcen auf dem FPGA wurden zwar nicht vollständig ausgenutzt, allerdings konnte der Platzierer und Router die Funktionsblöcke auf dem FPGA nicht dynamisch genug zuteilen. Die genaue Ursache dafür wurde nicht weiter untersucht. Zuletzt wurden die Messergebnisse einer Softwarelösung aus dem Projektbericht der Ersteller [15] für einen Vergleich übertragen, welche zur Messung den gleichen SoC und die gleiche Methode zur Zeitmessung genutzt haben.

7.3.2 Messergebnisse

Tabelle 9 enthält die durchschnittliche Rechenzeit pro Tile abhängig der Burstlänge in 4 Byte Worten. Es ist zu beachten, dass ein Wort nicht einem Pixel entspricht (siehe Kap. 5.3.1).

Burstlänge	**64x64**	**96x96**	**100x100**	**128x128**	**150x150**	**160x160**
8	170	366	387	633	845	974
16	125	254	266	437	574	661
32	105	203	214	360	444	507
64	105	194	202	333	404	464
128	126	220	228	361	432	486
256	178	295	308	458	542	601

Tabelle 9: Messergebnisse - Rechenzeit (μs/Tile) abhängig der Burstlänge (4 Byte Worte)

Mit den Werten aus Tabelle 9 ist in Abb. 35 die Rechenzeit abhängig der Burstlänge für die Auflösung 64x64, 128x128 und 160x160 grafisch dargestellt.

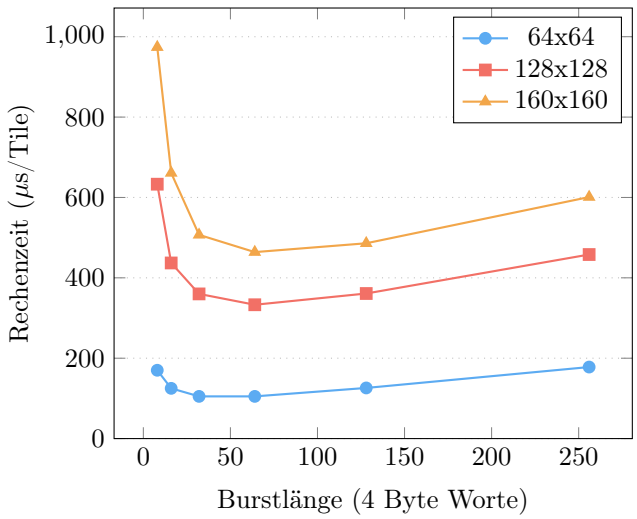


Abbildung 35: Messergebnisse - Diagramm der Rechenzeit abhängig der Burstlänge (4 Byte Worte)

Ebenfalls mit den Werte aus Tabelle 9 ist in Abb. 36 die Rechenzeit abhängig der Anzahl Pixel der gewählten Auflösung (Höhe $*$ Breite) dargestellt. Für die Rechenzeit wurde der jeweils schnellste Wert der verschiedenen Burstlängen gewählt. Die Punkte auf der Linie liegen an den Stellen der gemessenen Auflösungen.

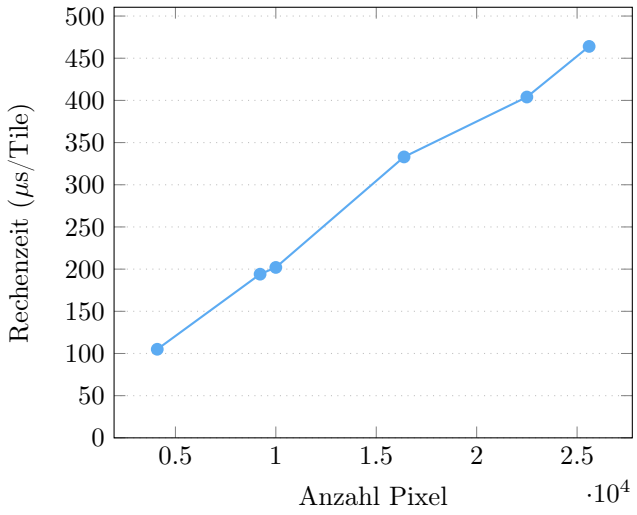


Abbildung 36: Messergebnisse - Diagramm der Rechenzeit abhängig der Auflösung (Anzahl Pixel)

Tabelle 10 enthält die Messergebnisse der Rechenzeit abhängig der Anzahl paralleler Instanzen des Tilings bei einer Burstlänge von 32.

Parallelität	**64x64**	**96x96**	**100x100**	**128x128**	**150x150**	**160x160**
1	105	203	214	360	444	507
2	59	114	120	199	251	285
3	40	77	81	133	168	192
4	30	58	61	101	127	145

Tabelle 10: Messergebnisse - Rechenzeit (μs/Tile) abhängig der Parallelität

Mit den Werten aus Tabelle 10 stellt Abb. 37 den Zusammenhang von Anzahl Instanzen des Tilings und Rechenzeit für die Auflösungen 64x64, 128x128 und 160x160 dar.

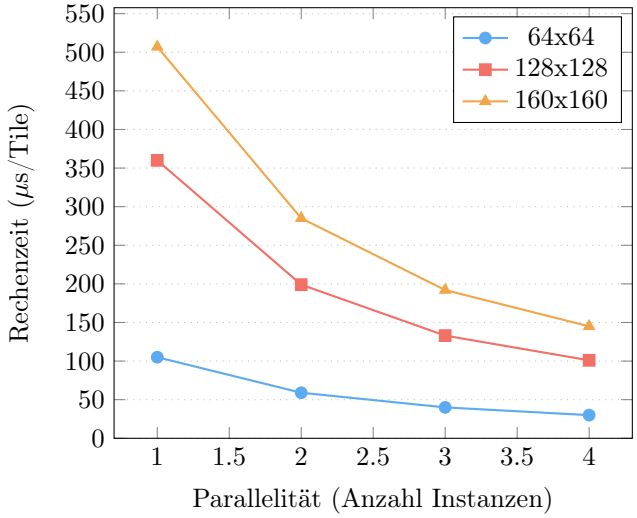


Abbildung 37: Messergebnisse - Diagramm der Rechenzeit abhängig der Parallelität

Tabelle 11 enthält die Messwerte einer Softwarelösung mit bilinearer Skalierung zum Vergleich mit der neuen Lösung. Die Werte wurden unverändert dem Projektbericht der Autoren entnommen [15].

64x64	**96x96**	**100x100**	**128x128**	**150x150**	**160x160**
60	243	247	270	312	316

Tabelle 11: Messergebnisse - Softwarelösung mit bilinearer Skalierung [15]

Zum Vergleich wurden die Rechenzeiten der Softwarelösung, sowie der Hardwarelösung mit einer und vier Instanzen, für verschiedene Auflösungen in Abb. 38 dargestellt.

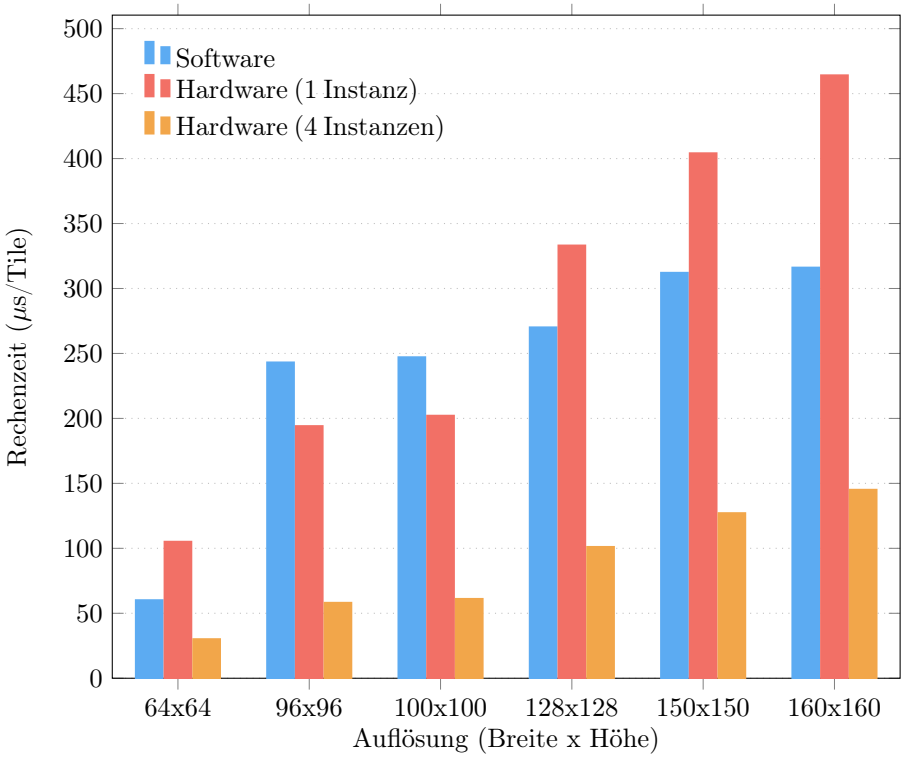


Abbildung 38: Messergebnisse - Vergleich der Rechenzeiten der Software und Hardwarelösung

7.3.3 Analyse der Messergebnisse

Die Messergebnisse für verschiedene Burstlängen zeigen, dass eine höhere Burstlänge nicht zwingend eine geringere Rechenzeit bedeutet. In Abb. 35 ist zu erkennen, dass bei einer Burstlänge von 32/64 ein Minimum existiert. Dies lässt sich dadurch erklären, dass der Extraktor den Bildbereich zeilenweise liest. Ist dieser Bildbereich schmaler als gesamte Bild, dann liegen die einzelnen Zeilen des Bildbereichs nicht sequentiell im Speicher, sodass für jede Zeile zumindest ein Burst notwendig ist. Sobald ein Burst gestartet wurde, kann er nicht abgebrochen werden. Ist die Burstlänge nun länger, als notwendig für eine Zeile des Bildes, werden Daten gelesen, die nicht benötigt werden. Durch diese nutzlosen Lesevorgänge wird die Verarbeitung verzögert und die Rechenzeit verlängert. Hier wäre eine Optimierung der Implementierung sinnvoll, welche die Burstlänge basierend auf der Bildbreite dynamisch verkürzt. Die maximale Burstlänge könnte weiterhin zur Hardwaresynthese konfiguriert werden, um den Ressourcenverbrauch zu limitieren. Trotzdessen ist es interessant, dass Längen über 16 überhaupt einen Vorteil bringen, da die Speicheranbindung durch AXI3 auf diese Länge limitiert ist. Ein Interconnect zwischen der Schaltung auf dem FPGA und dem Speichercontroller muss also irgendeine Form von Optimierung durchführen, wie beispielsweise eine Verdopplung der Datenbreite, wodurch sich auch die Datenmenge

bei gleicher Burstlänge verdoppelt. Ein Burst mit Länge von 16 und Datenbreite von 64 Bit wäre dann äquivalent zu Länge 32 und 32 Bit Datenbreite. Welche Optimierungen genau verwendet werden, konnte im Zeitrahmen nicht mehr analysiert werden. Zukünftig wäre dies sinnvoll, damit die Optimierungen direkt in die Komponenten des Tilings integriert werden können.
Trotz suboptimaler Burstlängen steigt die Rechenzeit abhängig der Anzahl Pixel relativ linear, wie in Abb. 36 zu erkennen ist. Für Extraktor und Ablage war dies vorhersehbar, da die Anzahl der Lese-/Schreibvorgänge in einem linearen Verhältnis zur Anzahl Pixel stehen. Beim Skalierer bestand die Vermutung, dass bruchteilige Skalierungsfaktoren langsamern sind, was sich aber nicht bestätigt hat.
Der Vergleich verschiedener Mengen von parallelen Instanzen des Tilings in Abb. 37 zeigt, dass eine höhere Parallelität für eine geringere Rechenzeit sorgt, jedoch mit abnehmendem Vorteil. Daraus lassen sich mehrere Schlüsse ziehen. Die Speicherbandbreite ist ausreichend für sehr hohe Verarbeitungsgeschwindigkeiten, allerdings ist zu beachten, dass bei der Anbindung des BNN-PYNQs dieses ebenso Bandbreite benötigt. Ein weiterer Verbraucher ist die CPU, welche jedoch durch die Auslagerung des Tilings auf das FPGA deutlich weniger belastet wird. Eine weitere Steigerung der Geschwindigkeit könnte potentiell durch die Optimierung der Implementierung des Tilings erreicht werden, da eine einzelne Instanz die Speicherbandbreite kaum ausnutzt. Über AXI4 und einer Wortlänge von vier Bytes bei einer Taktfrequenz von 100 Mhz wäre theoretisch eine Datendurchsatz von 400 MB/s möglich. Der verbaute DDR3-Speicher schafft einen maximalen Durchsatz von 1066 Mbps oder 133,25 MB/s. Der Vergleich ist allerdings nicht vollkommen fair, da AXI4 keine kontinuierlichen Transfers, sondern nur Bursts erlaubt, die wiederum initialisiert werden müssen, was den Durchsatz verringert. Trotzdem besteht beim Tilings noch einiges an Potential, um mit einer Instanz mehr Durchsatz bei ähnlichem Ressourcenverbrauch zu erreichen. In dieser Arbeit fehlte leider die Zeit, um die verantwortlichen Flaschenhälse bei der Verarbeitung zu identifizieren.
Im Vergleich mit der Softwarelösung kann der Prototyp auf dem FPGA deutlich höhere Geschwindigkeiten erreichen. Bei einer Auflösung von 64x64, 128x128, 150x150 und 160x160 ist die Softwarelösung schneller als die Hardwarelösung mit einer Instanz. Mit vier Instanzen liegt die neue Lösung über alle Auflösung hinweg mit deutlichem Abstand vorne mit einem maximalen Faktor von 4x bei einer Auflösung von 96x96. Nach einer Optimierung der neuen Lösung ist davon auszugehen, dass selbst eine Instanz die Softwarelösung über alle Auflösungen hinweg schlägt.
Bei der Softwarelösung fällt auf, dass die benötigte Zeit bei 64x64 deutlich geringer ist, als bei den restlichen Auflösungen. Hier scheint eine besondere Optimierung für den Skalierungsfaktor von genau zwei stattzufinden. Diese Optimierung könnte in weiterer Arbeit nocht genauer betrachtet werden, um diese eventuell auch in der neuen Lösung einzusetzen.

7.4 Komplexität der Implementation

Die Implementierung einer Softwarelösung ist durch etablierte Programmbibliotheken, wie OpenCV, relativ trivial und leicht testbar. Ein einfacher Entwicklungsprozess ist entscheident zur Minimierung der Kosten, eine simple Verifizierbarkeit und gute Wartbarkeit. Die Entwicklung für einen FPGA ist ein vergleichsweise kleineres Feld, weshalb folgend die Hürden einer Implementierung für diesen beschrieben werden.
Die höchste Komplexität hatte die Implementierung der AXI4-Schnittstellen in VHDL, welche eine sehr umfangreiche Spezifikation einhalten müssen. Wird die Spezifikation nicht genaustens beachtet, dann kommt es schnell zu irritierenden Fehlerwirkungen. Es kann beispielsweise schnell zu einem Deadlock kommen, falls die Reihenfolge der Handschläge auf den Kommunikationskanälen nicht eingehalten wird oder Schreib-/Lesevorgänge fehlschlagen, sobald eine 4096 Byte Speichergrenze in einem Burst überschritten wird. Letzteres fiel bei den Komponententests zuerst nicht auf, da diese größtenteils unter der ersten solchen Grenze getestet haben. Diese und viele weitere Details der Spezifikation haben die eigene Implementation in VHDL stark erschwert. Eine Implementation mittels Hardwaresynthese einer Hochsprache, wie Vitis HLS, könnte die Komplexität übernehmen, aber verschleiert viele Details, die für Optimierungen trotzdem bekannt sein müssen.
Die Suche von Fehlerzuständen nach der Erkennung von Fehlerwirkungen erforderte die Inspektion einer Vielzahl von Signalen. Die verwendete Enwicklungsumgebung Vivado hat hier viel Unterstützung geboten. Der Simulator hat es erlaubt den genauen internen Zustand jeder Komponente zu jedem Zeitpunkt zu inspizieren. Auf der echten Hardware war es dank der angebundenen CPU leicht möglich verschiedene Speicherbereiche mittels serieller Schnittstelle auszugeben. Weiterhin konnte mittels dem „System Logic Analyzer“ IP-Core von Vivado [40] der interne Zustand ausgewählter Signale zur Laufzeit auf der echten Hardware inspiziert werden.

8 Zusammenfassung

Dieses Kapitel enthalt eine Zusammenfassung der Ergebnisse und gibt konkrete Antworten für die anfängliche Fragestellung. Zuletzt wird ein Ausblick für weitere nötige Forschungsarbeiten gegeben.

8.1 Ergebnisse

Anhand der Ergebnisse der Evaluation werden nun die Fragen zur These aus Kap. 1.3 beantwortet.

Wie einfach lässt sich das Tiling auf einem FPGA umsetzen?
Eine optimierte Implementierung des Tilings ist auf einem FPGA deutlich schwieriger umzusetzen, als auf einer CPU, da für letztere deutlich mehr Werkzeuge und fertige Programmbibliotheken zur Verfügung stehen.

Wie konfigurierbar kann das Tiling auf einem FPGA umgesetzt werden?
Nahezu alle unter Kap. 4.1 definierten Parameter können im Prototypen konfiguriert werden. Für die fehlende Anpassbarkeit der restlichen Parameter war hauptsächlich das begrenzte Zeitbudget verantwortlich, sodass eine Weiterentwicklung dies ändern könnte. Dank der direkten Anbindung der Komponenten in den Adressbereich der CPU ist die Konfiguration sehr simpel und kann leicht modifiziert werden. Aus Sicht der Softwareentwickler handelt es sich um herkömmliche Methodenaufrufe. Die Hardwarebeschleunigung wird vollkommen abstrahiert.

Wie schnell ist das Tiling auf dem FPGA im Vergleich zur CPU?
Der Prototyp auf dem FPGA kann deutlich schneller, als die Lösung auf der CPU sein. Der große Vorteil des FPGA ist die echte Parallelität, sodass mehrere Instanzen des Tilings gleichzeitig arbeiten können, während die verwendete CPU konstant zwei Rechenkerne besitzt. Allerdings ist dabei zu beachten, dass alle Instanzen von der Speicherschnittstelle abhängen, weshalb weitere Instanzen ab einem gewissen Punkt keine höhere Geschwindigkeit bringen.

Wie effizient ist das Tiling auf dem FPGA im Vergleich zur CPU?
Die Lösung auf dem FPGA erreicht bei einer Taktfrequenz von 100 Mhz eine höhere Geschwindigkeit, als die CPU bei 650 Mhz, was den Vorteil der Hardwarebeschleunigung aufzeigt. Während die CPU ihre Instruktionen aus dem Speicher auslesen, dekodieren und ausführen muss, ist die Hardwareschaltung selber die Instruktion und kann jeden Takt für tatsächliche Arbeit verwenden.

Sofern kein zweiter FPGA verwendet wird, belegt das KNN bereits einen Teil der Ressourcen. Kann das Tiling mit den übrigen Ressourcen umgesetzt werden?
Durch die erfolgreiche Synthese und Platzierung des Prototypen auf der Zielplatform konnte gezeigt werden, dass dies möglich ist.

Wie einfach kann eine externe Bildquelle an das Tiling auf dem FPGA angebunden werden?
Durch das vorhandene Ökosystem an IP-Cores für HDMI/DVI und AXI4-Video Stream war die Anbindung relativ einfach. Lediglich die stark verteilte Dokumentation der IP-Cores und die vielfach veralteten Referenzschaltungen machten die Konfiguration etwas mühselig. Sofern in einer weiteren Arbeit ebenfalls das PYNQ-Z1 verwendet wird, kann die im Prototypen verwendete Anbindung sehr einfach in einem Blockdiagramm platziert werden, wodurch die Anbindung durch diese Arbeit nun nahezu trivial ist.

8.2 Ausblick

Der erstellte Prototyp ist theoretisch anwendbar und erreicht bereits gute Ergebnisse, jedoch besteht noch ein hohes Potential für Optimierungen. Eine Optimierung der Pfadlängen in der Schaltung ist zwanghaft notwendig, da die maximalen Pfadlängen sehr nah an der Grenze des Möglichen sind. Der Ressourcenverbrauch könnte verringert werden, allerdings ist dies nicht unbedingt notwendig, da die Geschwindigkeit einer Instanz des Tilings bei weitem noch nicht ausgereizt ist. Wahrscheinlich kann daher mit weniger Instanzen und entsprechend weniger Ressourcenverbrauch die gleiche Geschwindigkeit erreicht werden.
In der Evaluation wurde für die Geschwindigkeitsmessungen aus Zeitgründen lediglich die Anbindung des KNN über den Speicher betrachtet, obwohl auch die direkte Anbindung von FINN über einen Datenstrom bereits funktioniert, wie der Systemtest gezeigt hat. Nun müsste die Evaluierung für die Datenstromanbindung wiederholt werden, um festzustellen, welche Anbindung überlegen ist.
Die Effizienz im Vergleich zur Softwarelösung auf der CPU wurde nur anhand der benötigten Taktzyklen betrachtet, was jedoch nicht direkt auf den Stromverbrauch übertragen werden kann, da ein Taktzyklus auf CPU und FPGA nicht zwingend den selben Stromverbrauch haben und besonders auf dem FPGA die tatsächliche Schaltung und derer Zustandswechsel von Signalen einen Unterschied macht. Als Teil dieser Arbeit wurde versucht diesen Stromverbrauch mit einem Multimeter zu messen, was aber keine zuverlässigen Ergebnisse geliefert hat. Zwei der Hauptprobleme waren zum Einen, dass sich der Stromverbrauch der CPU unter Last kaum ändert, was sich vermutlich durch die konstante Taktfrequenz erklären lässt und das für die Messung der Lösungen auf dem FPGA die CPU nicht einfach schlafen gelegt werden konnte, da diese an weitere Funktionalität, wie die Taktgeneration für den FPGA, gekoppelt ist.
Zuletzt gilt es nun die Anbindung einer externen Bildquelle und das neue Tiling zu nutzen, um einen echten Anwendungsfall umzusetzen und diesen tatsächlich auf der Edge einzusetzen. Dazu ist es nötig einen Algorithmus für die Erzeugung der Bildregionen zu implementieren und die Inferenzergebnisse für die daraus erzeugten Tiles zu verarbeiten.

8.3 Fazit

Diese Arbeit hat ihm Rahmen der Implementierung des Tilings auf FPGAs verschiedene Einschränkungen der schwachen Hardware beleuchtet, was gezeigt hat, dass die Umsetzung schwieriger war, als anfangs gedacht. Trotzdessen konnte eine Lösung konzeptioniert und auch in einem Prototypen umgesetzt werden, welcher vielversprechende Ergebnisse in der Evaluation geliefert hat. Durch die Integration von der Anbindung eines externen Bildsignals, Tiling und KNN ist es nun weiterhin möglich einen echten Anwendungsfall mit einer echten Bildquelle umzusetzen.

A Literaturverzeichnis

[1] *(Online) Mirrorless | EOS M50 EF-M 15-45mm IS STM Kit | Canon USA*. URL: https://www.usa.canon.com/internet/portal/us/home/products/details/cameras/eos-dslr-and-mirrorless-cameras/mirrorless/eos-m50-ef-m-15-45mm-is-stm-kit/ (besucht am 27. 04. 2021).

[2] *(Online) NVIDIA Jetson TX2: Hochleistungs-KI für Edge-Anwendungen*. de-de. URL: https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-tx2/ (besucht am 27. 04. 2021).

[3] *(Online) OpenCV: Geometric Image Transformations*. URL: https://docs.opencv.org/master/da/d54/group__imgproc__transform.html (besucht am 16. 05. 2021).

[4] *(Online) Performance and Resource Utilization for Video Multi-Scaler v1.0*. URL: https://www.xilinx.com/html_docs/ip_docs/pru_files/v-multi-scaler.html (besucht am 20. 06. 2021).

[5] *(Online) Snapdragon Neural Processing Engine SDK: Features Overview*. URL: https://developer.qualcomm.com/docs/snpe/overview.html (besucht am 27. 04. 2021).

[6] *(Online) Vivado Library - Digilent Reference*. URL: https://reference.digilentinc.com/vivado:library (besucht am 11. 05. 2021).

[7] *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*. en. ARM. 2003, S. 306.

[8] *AXI Block RAM (BRAM) Controller v4.1 LogiCORE IP Product Guide (PG078)*. en. Xilinx, Inc. 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_1/pg078-axi-bram-ctrl.pdf.

[9] *AXI VDMA v6.2 LogiCore IP Product Guide (PG020)*. en. Xilinx. 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_2/pg020_axi_vdma.pdf.

[10] *AXI Verification IP v1.1 LogiCORE IP Product Guide (PG267)*. en. Xilinx, Inc. 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_vip/v1_1/pg267-axi-vip.pdf.

[11] *AXI4-Stream Infrastructure IP Suite v3.0 LogiCORE IP Product Guide*. en. Xilinx, Inc. 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf.

[12] *Block Memory Generator v8.4 LogiCORE IP Product Guide (PG058)*. en. Xilinx, Inc. 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf.

[13] Michaela Blott u. a. „FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks“. en. In: *arXiv:1809.04570 [cs]* (Sep. 2018). Comment: to be published in ACM TRETS Special Edition on Deep Learning. arXiv: 1809.04570 [cs].

[14] Alexey Bochkovskiy, Chien-Yao Wang und Hong-Yuan Mark Liao. „YOLOv4: Optimal Speed and Accuracy of Object Detection“. en. In: *arXiv:2004.10934 [cs, eess]* (Apr. 2020). arXiv: 2004.10934 [cs, eess].

[15] Prof. Dr. Jan Bredereke u. a. *Object Recognition on FPGA for Autonomous Steering of a Vehicle in Real Time*. en. 2021.

[16] Robert David u. a. „TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems“. en. In: *arXiv:2010.08678 [cs]* (März 2021). arXiv: 2010.08678 [cs].

[17] *DVI-to-RGB (Sink) 2.0 IP Core User Guide*. en. Digilent, Inc. Sep. 2019. URL: https://github.com/Digilent/vivado-library/blob/master/ip/dvi2rgb/docs/dvi2rgb.pdf.

[18] M. Everingham u. a. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. en. URL: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[19] Ross Girshick. „Fast R-CNN“. en. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. Santiago, Chile: IEEE, Dez. 2015, S. 1440–1448. ISBN: 978-1-4673-8391-2. DOI: 10.1109/ICCV.2015.169.

[20] Ross Girshick u. a. „Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation“. en. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. Juni 2014, S. 580–587. DOI: 10.1109/CVPR.2014.81.

[21] Michael Hilgers. *Elektrik und Mechatronik*. de. Wiesbaden: Springer Fachmedien Wiesbaden, 2016. ISBN: 978-3-658-12748-0 978-3-658-12749-7. DOI: 10.1007/978-3-658-12749-7.

[22] Sebastian Houben u. a. „Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark“. en. In: *International Joint Conference on Neural Networks*. 1288. 2013.

[23] Bohao Huang u. a. „Tiling and Stitching Segmentation Output for Remote Sensing: Basic Challenges and Recommendations“. en. In: abs/1805.12219 (2018), S. 9.

[24] Petar Jokic, Stephane Emery und Luca Benini. „BinaryEye: A 20 Kfps Streaming Camera System on FPGA with Real-Time On-Device Image Recognition Using Binary Neural Networks“. en. In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. Juni 2018, S. 1–7. DOI: 10.1109/SIES.2018.8442108.

[25] H. Krishna und C. V. Jawahar. „Improving Small Object Detection“. en. In: *2017 4th IAPR Asian Conference on Pattern Recognition (ACPR)*. - Proposal Network
- Supersampling / Upscaling with ANN. Nov. 2017, S. 340–345. DOI: 10.1109/ACPR.2017.149.

[26] Alex Krizhevsky, Geoffrey Hinton u. a. „Learning Multiple Layers of Features from Tiny Images“. en. In: Citeseer, 2009.

[27] Tsung-Yi Lin u. a. „Microsoft COCO: Common Objects in Context“. en. In: *arXiv:1405.0312 [cs]* (Feb. 2015). Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list. arXiv: 1405.0312 [cs].

[28] *Multi-Scaler v1.2 LogiCORE IP Product Guide (PG325)*. en. Xilinx, Inc. Apr. 2021. URL: https://www.xilinx.com/support/documentation/ip_documentation/v_multi_scaler/v1_2/pg325-v-multi-scaler.pdf.

[29] M.A. Nuno-Maganda und M.O. Arias-Estrada. „Real-Time FPGA-Based Architecture for Bicubic Interpolation: An Application for Digital Image Scaling“. en. In: *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*. Sep. 2005, 8 pp.–1. DOI: 10.1109/RECONFIG.2005.34.

[30] E. Nurvitadhi u. a. „Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC“. en. In: *2016 International Conference on Field-Programmable Technology (FPT)*. Dez. 2016, S. 77–84. DOI: 10.1109/FPT.2016.7929192.

[31] R. Ozdemir und M. Koc. „A Quality Control Application on a Smart Factory Prototype Using Deep Learning Methods“. en. In: *2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)*. Bd. 1. Sep. 2019, S. 46–49. DOI: 10.1109/STC-CSIT.2019.8929734.

[32] F Ozge Unel, Burak O Ozkalayci und Cevahir Cigla. „The Power of Tiling for Small Object Detection“. en. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2019.

[33] Nikhil R Pal und Sankar K Pal. „A review on image segmentation techniques“. In: *Pattern Recognition* 26.9 (1993), S. 1277–1294. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(93)90135-J. URL: https://www.sciencedirect.com/science/article/pii/003132039390135J.

[34] *PYNQ-Z1 Board Reference Manual*. en. Digilent, Inc. Apr. 2017. URL: https://reference.digilentinc.com/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf.

[35] *Radiation-Hardened, Space-GradeVirtex-5QV Family Data Sheet: Overview (DS192)*. en. Xilinx, Inc. Nov. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf.

[36] G. Anthony Reina u. a. „Systematic Evaluation of Image Tiling Adverse Effects on Deep Learning Semantic Segmentation“. en. In: *Frontiers in Neuroscience* 14 (Feb. 2020), S. 65. ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00065.

[37] Shaoqing Ren u. a. „Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks“. en. In: *arXiv:1506.01497 [cs]* (Jan. 2016). Comment: Extended tech report. arXiv: 1506.01497 [cs].

[38] Rowel Atienza (author). *Advanced Deep Learning With TensorFlow 2 and Keras: Apply DL Techniques, GANs, VAEs, Deep RL, SSL, Object Detection, Semantic Segmentation, and More.* en. Packt Publishing, 2020. ISBN: 978-1-83882-572-0.

[39] W. Shi u. a. „Edge Computing: Vision and Challenges“. en. In: *IEEE Internet of Things Journal* 3.5 (Okt. 2016), S. 637–646. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198.

[40] *System Integrated Logic Analyzer v1.1 LogiCORE IP Product Guide (PG261).* en. Xilinx, Inc. 2021. URL: https://www.xilinx.com/support/documentation/ip_documentation/system_ila/v1_1/pg261-system-ila.pdf.

[41] Yaman Umuroglu u. a. „FINN: A Framework for Fast, Scalable Binarized Neural Network Inference“. en. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Feb. 2017). Comment: To appear in the 25th International Symposium on Field-Programmable Gate Arrays, February 2017, S. 65–74. DOI: 10.1145/3020078.3021744. arXiv: 1612.07119.

[42] *Video In to AXI4-Stream v4.0 LogiCORE IP Product Guide (PG043).* en. Xilinx. 2021. URL: https://www.xilinx.com/support/documentation/ip_documentation/v_vid_in_axi4s/v4_0/pg043_v_vid_in_axi4s.pdf.

[43] *Video Scaler 1.0 IP Core User Guide.* en. Digilent, Inc. Jan. 2019. URL: https://github.com/Digilent/vivado-library/blob/master/ip/video_scaler/doc/video_scaler.pdf.

[44] *Video Scaler v8.1 (PG009).* en. Xilinx, Inc. Nov. 2015. URL: https://www.xilinx.com/support/documentation/ip_documentation/v_scaler/v8_1/pg009_v_scaler.pdf.

[45] *Vitis Unified Software Platform Documentation (UG1393).* en. Xilinx, Inc. Feb. 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1393-vitis-application-acceleration.pdf.

[46] *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994).* en. Xilinx, Inc. 2019, S. 263. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug994-vivado-ip-subsystems.pdf.

[47] *Vivado Design Suite: AXI Reference Guide (UG1037).* en. Xilinx, Inc. Juli 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[48] Klaus Peter Weibler. „Kameragesteuerte Optimierung Des Scheinwerferlichts“. In: *ATZelektronik* 4 (Mai 2009), S. 42–49.

[49] Joab R Winkler. „Numerical Recipes in C: The Art of Scientific Computing, Second Edition“. en. In: *Endeavour* 17.4 (Jan. 1993), S. 201. ISSN: 01609327. DOI: 10.1016/0160-9327(93)90069-F.

[50] Xiaowei Xu u. a. „DAC-SDC Low Power Object Detection Challenge for UAV Applications“. en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.2 (Feb. 2021), S. 392–403. ISSN: 0162-8828, 2160-9292, 1939-3539. DOI: 10.1109/TPAMI.2019.2932429.

[51] Shifeng Zhang u. a. „Single-Shot Refinement Neural Network for Object Detection“. en. In: *arXiv:1711.06897 [cs]* (Jan. 2018). Comment: 14 pages, 7 figures, 7 tables. arXiv: 1711.06897 [cs].

[52] Zhengxia Zou u. a. „Object Detection in 20 Years: A Survey“. en. In: *arXiv:1905.05055 [cs]* (Mai 2019). Comment: This work has been submitted to the IEEE TPAMI for possible publication. arXiv: 1905.05055 [cs].

[53] *Zynq-7000 SoC Data Sheet: Overview (DS190)*. en. Xilinx, Inc. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[54] *Zynq-7000 SoC Product Selection Guide*. en. Xilinx, Inc. 2019. URL: https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf.

B Abbildungsverzeichnis

C Tabellenverzeichnis

D Listingverzeichnis

E Blockdiagramme

Dieser Anhang enthält Grafiken der Blockdiagramme für die Implementation und Simulation.

E.1 Implementation

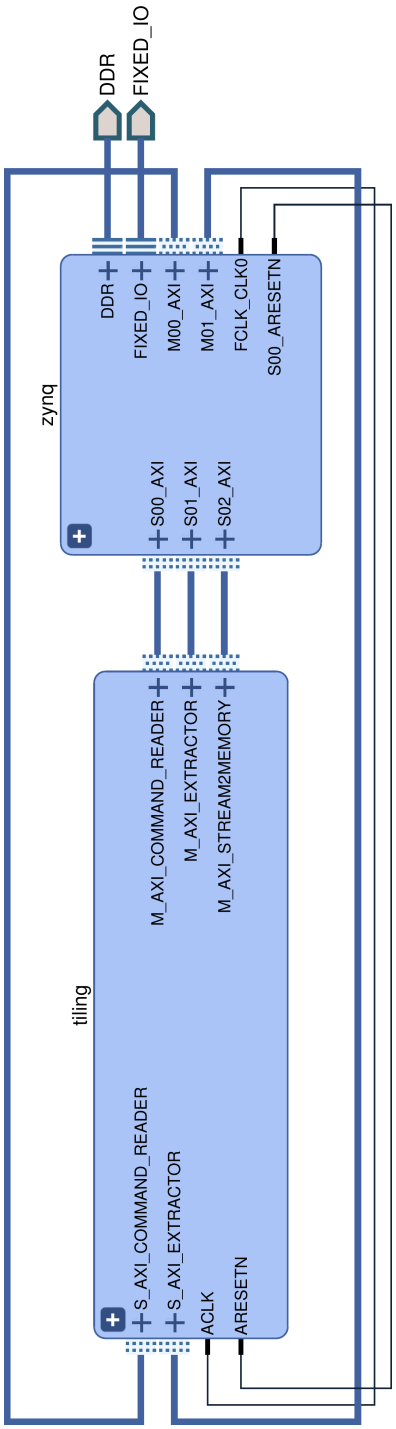


Abbildung 39: Blockdiagramm - Timingmessung in einer Minimalschaltung

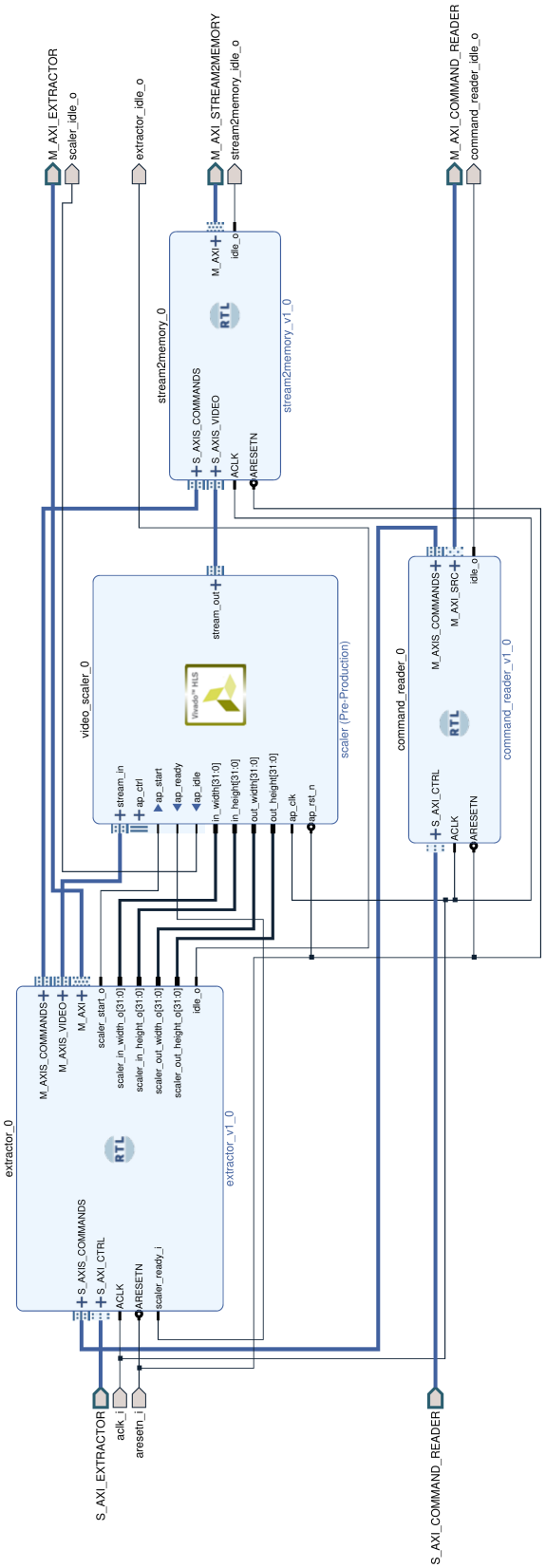


Abbildung 40: Blockdiagramm - Tiling

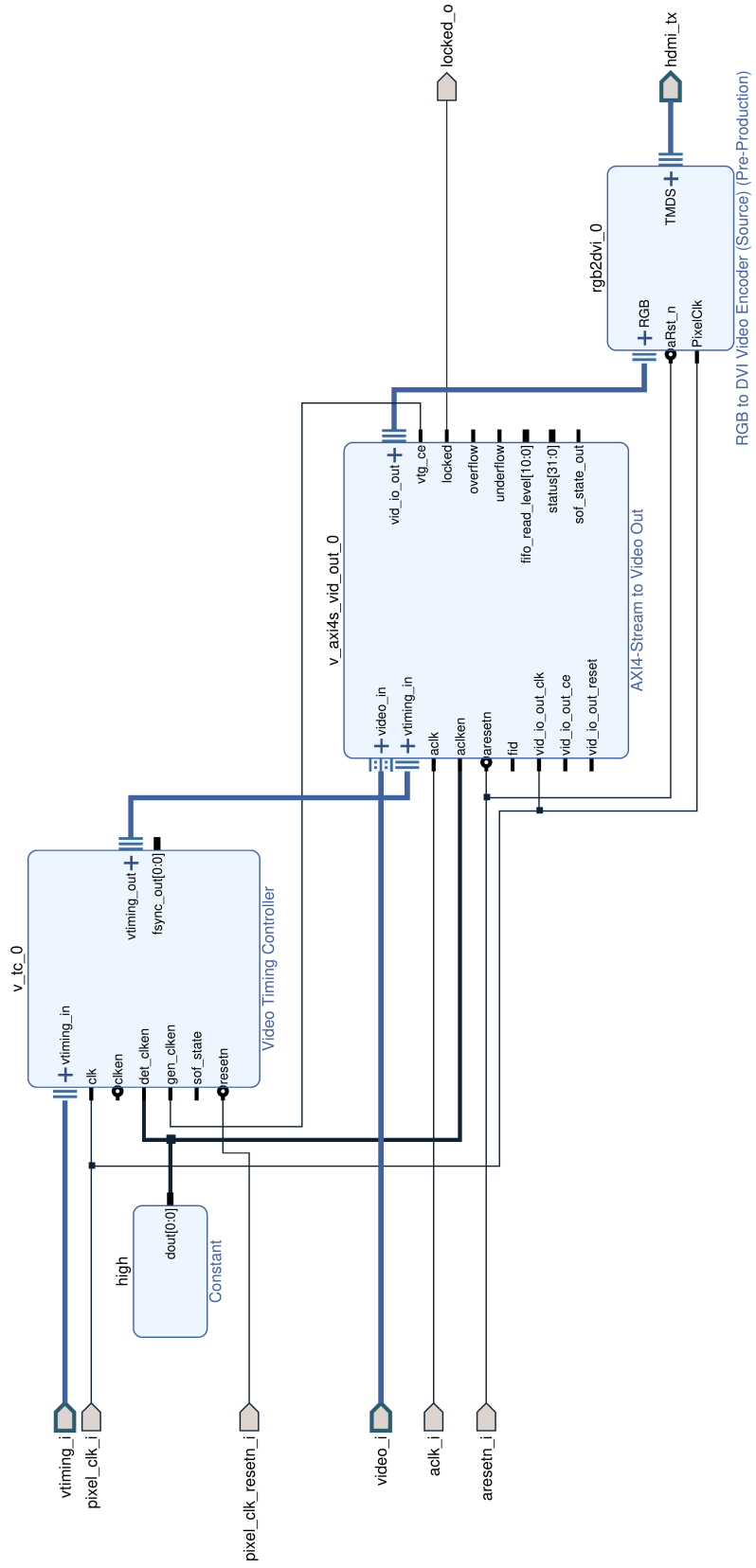


Abbildung 41: Blockdiagramm - Umwandlung von HDMI/DVI in einen AXI4-Video Stream

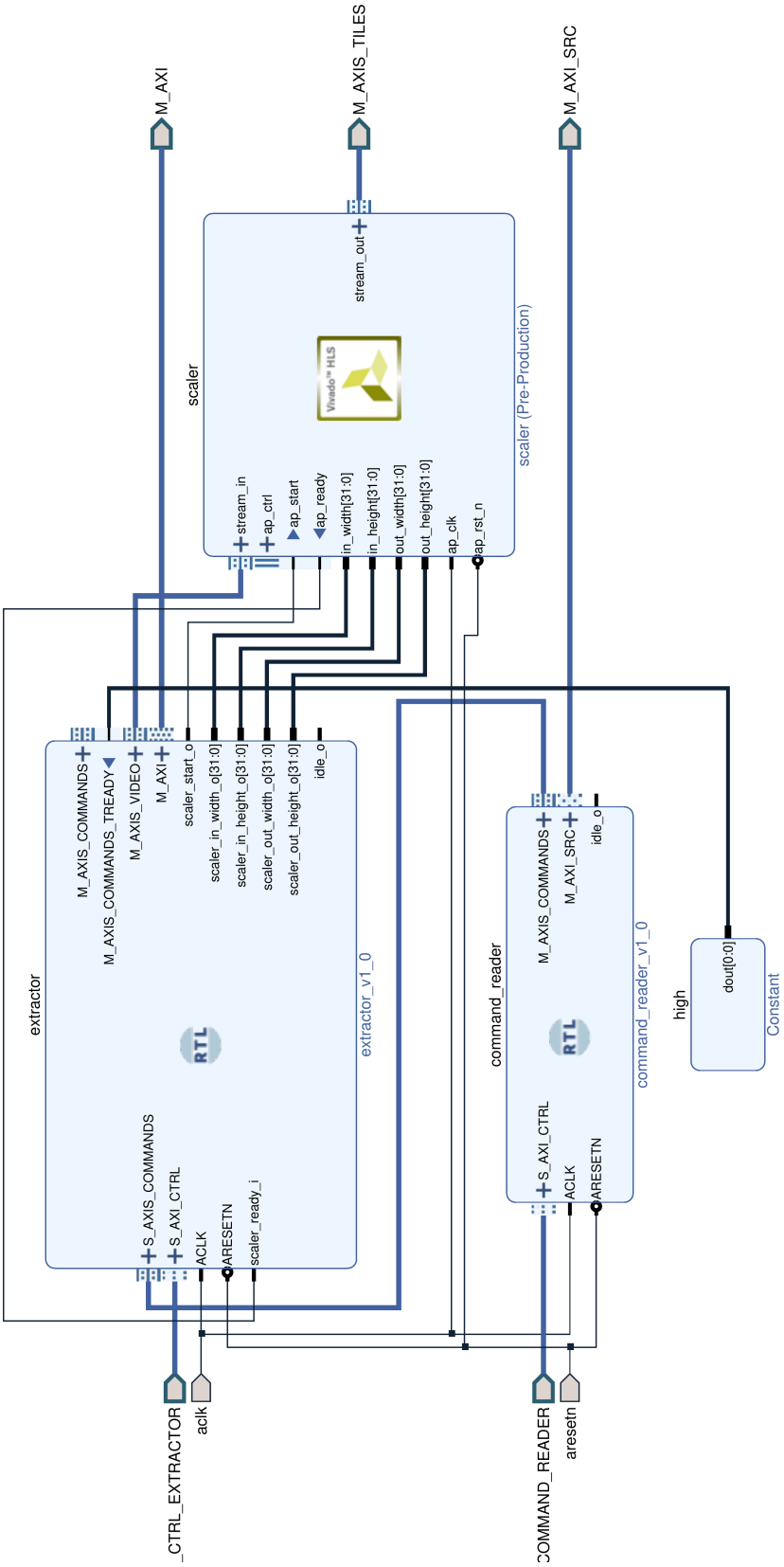


Abbildung 42: Blockdiagramm - Tiling der Gesamtschaltung (Bilder als Datenstrom)

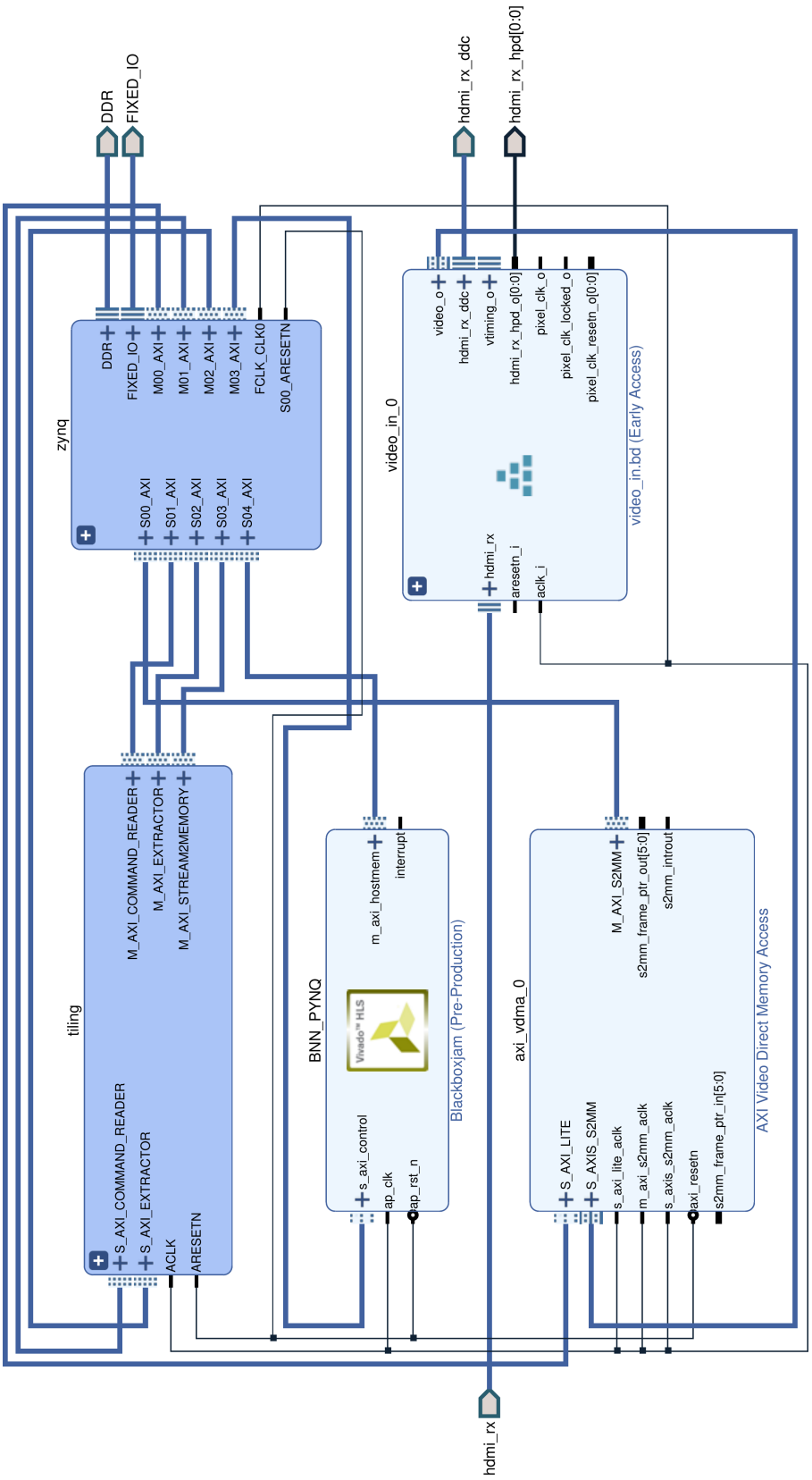


Abbildung 43: Blockdiagramm - Ressourcen- und Timingmessung

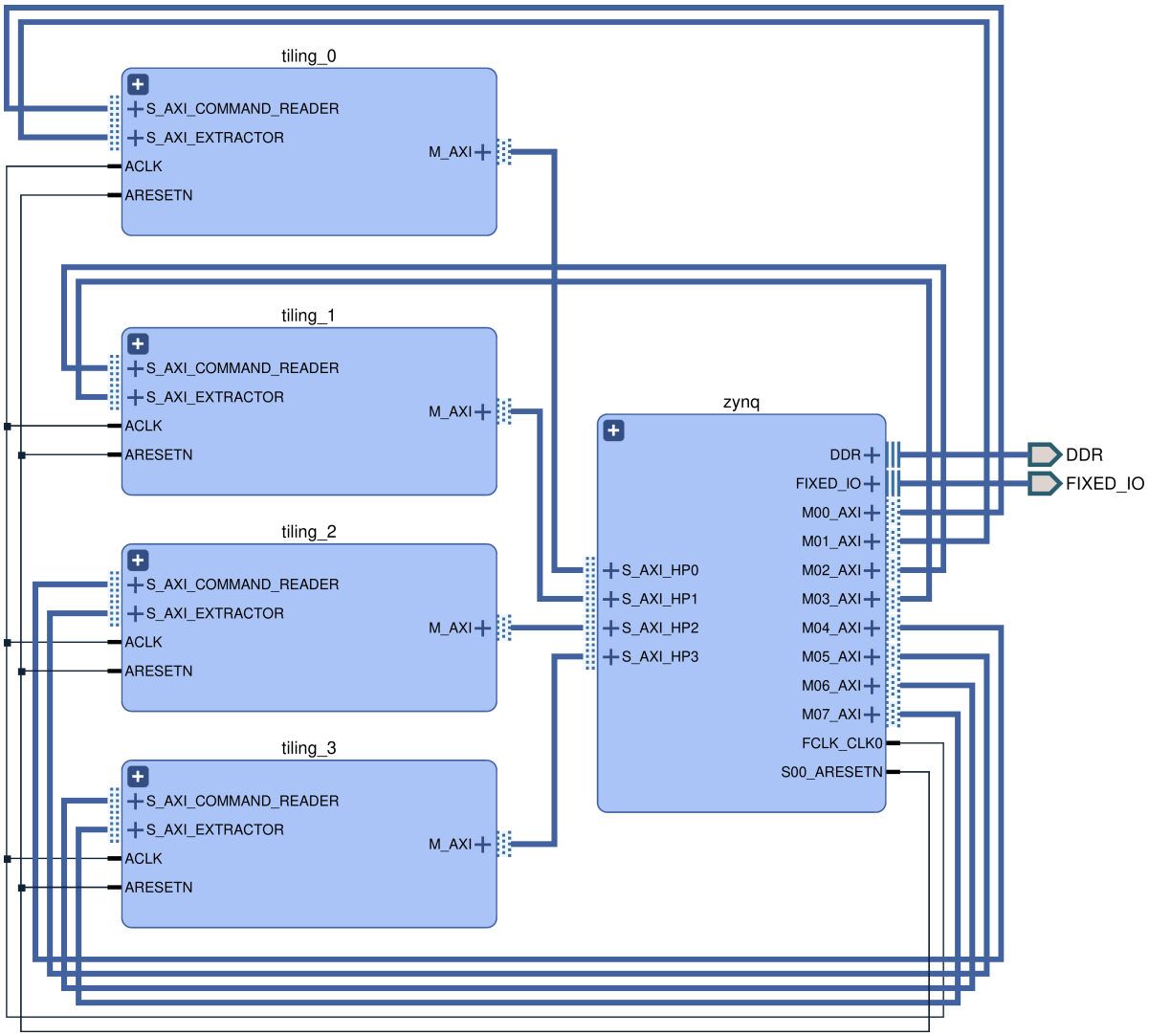


Abbildung 44: Blockdiagramm - Geschwindigkeitsmessung

E.2 Simulation

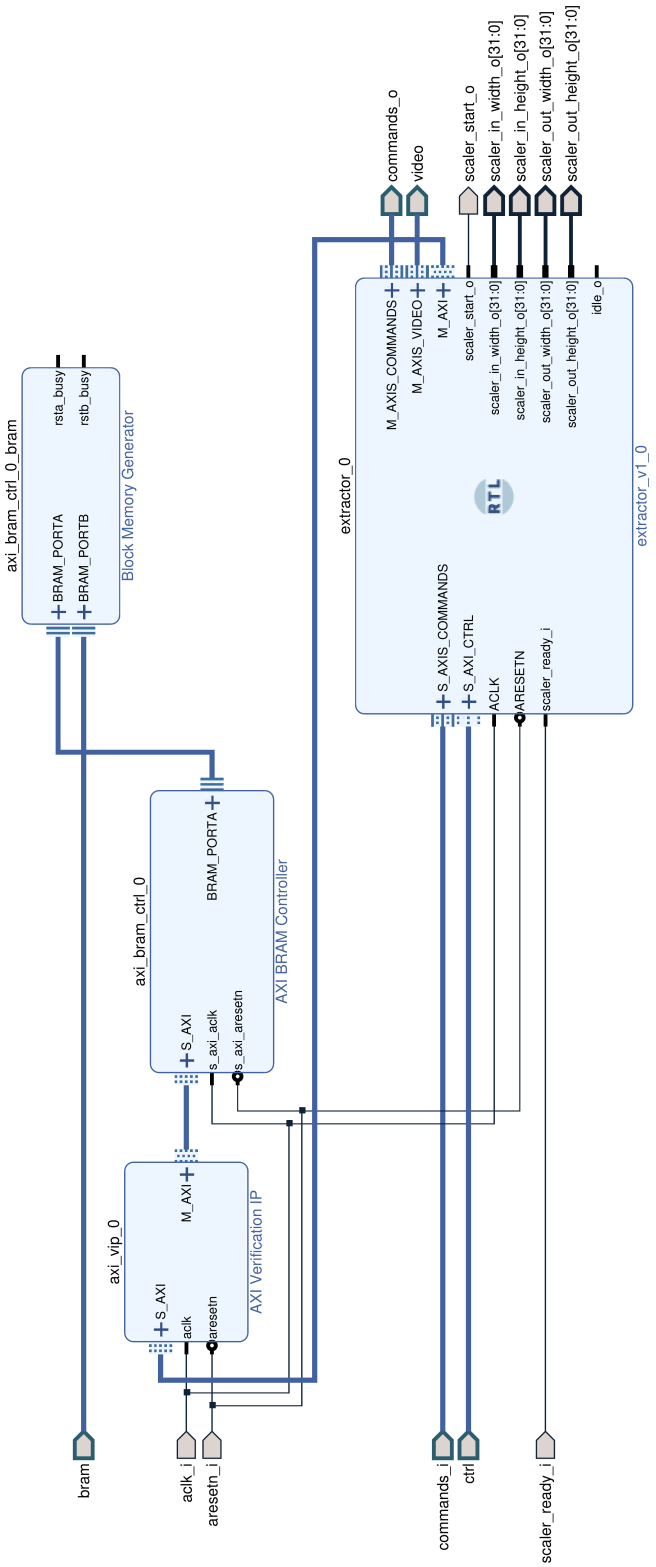


Abbildung 45: Blockdiagramm - Testaufbau für den Extraktor

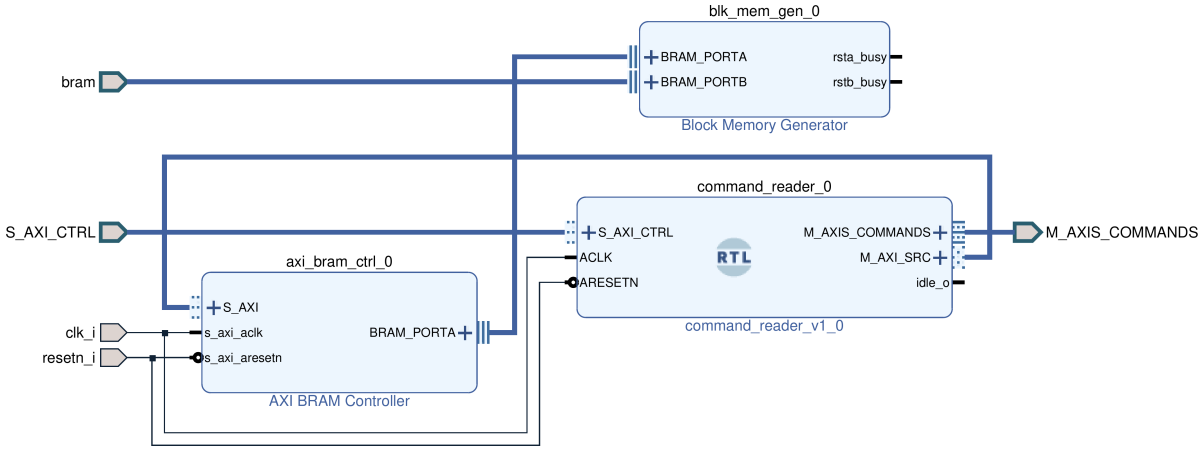


Abbildung 46: Blockdiagramm - Testaufbau für den Befehlsgenerator

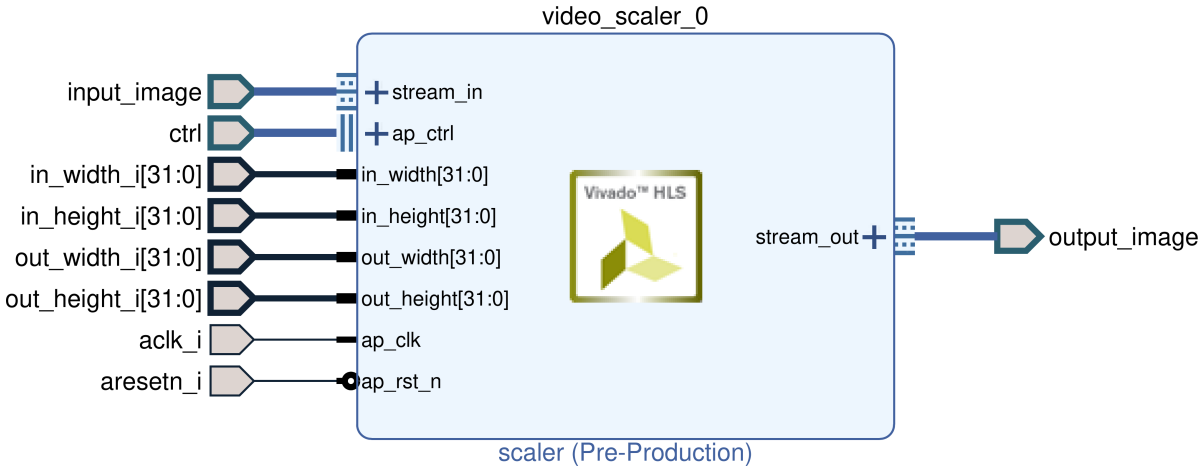


Abbildung 47: Blockdiagramm - Testaufbau für den Skalierer

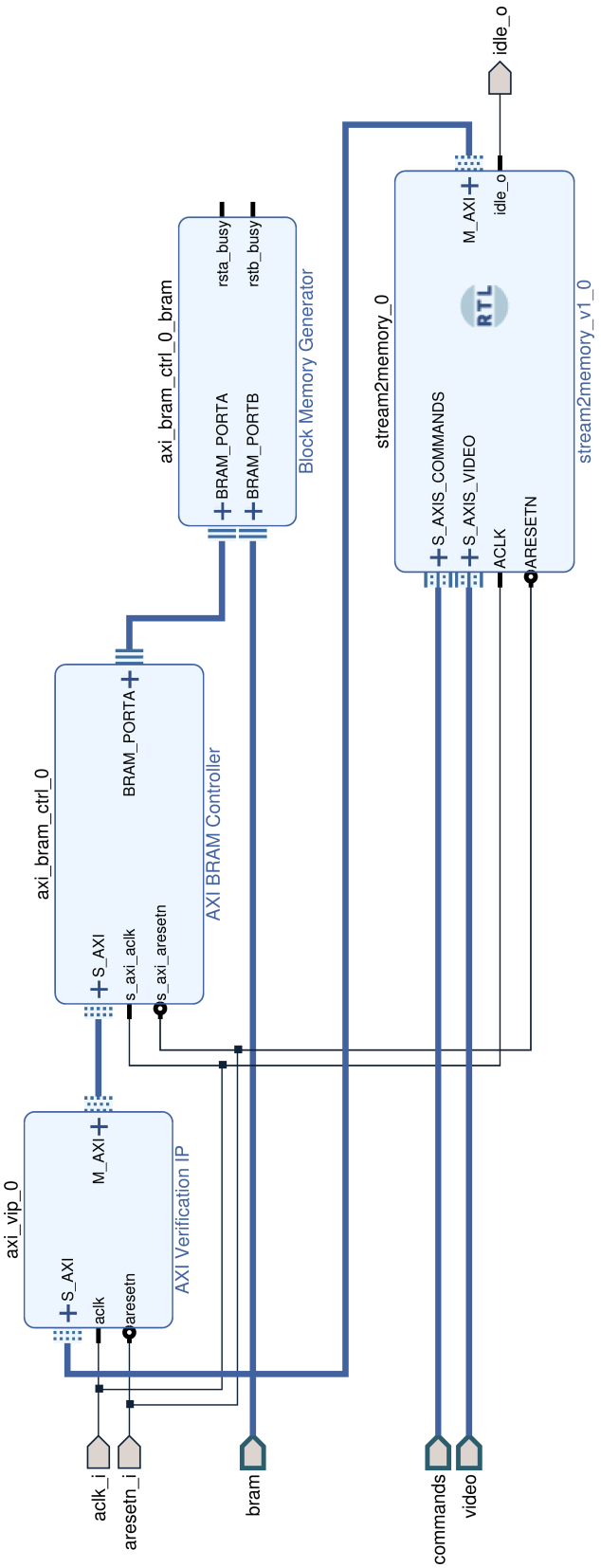


Abbildung 48: Blockdiagramm - Testaufbau für die Ablage

F Testprotokolle

Dieser Anhang enthält die Protokolle der Komponenten- und Integrationstests.

F.1 Komponententests

Es folgen die Protokolle der Komponententests, welche von den Testbenches automatisch generiert werden. Die Ausgaben wurden der TCL Konsole von Vivado entnommen. Ein „Failure“ bei der letzten Ausgabe bedeutet **nicht**, dass die Testbench fehlgeschlagen ist, sondern dient lediglich dazu den Simulator zu beenden. Eine Testbench ist erfolgreich, falls die „===== End =====“ Ausgabe erreicht wird.

Befehlsgenerator
Auf dem Datenträger: Protokolle/Tests/Komponententests/command_ readerTb.txt

```
Note: ===== Start: command_readerTb =====
Time: 0 ps  Iteration: 0  Process: /command_readerTb/stimAndCheck  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/command_readerTb.vhd
Failure: ===== End =====
Time: 4215 ns  Iteration: 0  Process: /command_readerTb/stimAndCheck  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/command_readerTb.vhd
$finish called at time : 4215 ns : File
↪  "/home/felix/Unison/dynamic_tiling/vhdl/command_readerTb.vhd" Line 280
```

Extraktor
Auf dem Datenträger: Protokolle/Tests/Komponententests/extractorTb.txt

```
Note: ===== Start: extractorTb =====
Time: 0 ps  Iteration: 0  Process: /extractorTb/stim  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
blk_mem_gen_v8_4_4 collision detected at time: 25000, Instance:
↪  extractorTb.dut.extractor_test_design_i.axi_bram_ctrl_0_bram.inst.
\native_mem_mapped_module.blk_mem_gen_v8_4_4_inst , A read address: 0, B write
↪  address: 0
blk_mem_gen_v8_4_4 collision detected at time: 45000, Instance:
↪  extractorTb.dut.extractor_test_design_i.axi_bram_ctrl_0_bram.inst.
\native_mem_mapped_module.blk_mem_gen_v8_4_4_inst , A read address: 0, B write
↪  address: 0
Note: Starting command generation...
Time: 2225 ns  Iteration: 2  Process: /extractorTb/cmd_i_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Note: Address for command 0 ok
Time: 2265 ns  Iteration: 1  Process: /extractorTb/check_output_commands  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Note: Size for command 0 ok
Time: 2265 ns  Iteration: 1  Process: /extractorTb/check_output_commands  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Note: Scaler configuration for command 0 ok
Time: 2275 ns  Iteration: 1  Process: /extractorTb/check_scaler_configuration
↪  File: /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Note: Pixels for command 0 ok
Time: 2315 ns  Iteration: 1  Process: /extractorTb/check_pixels  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
[gekürzt]
Note: Scaler configuration for command 18 ok
Time: 8375 ns  Iteration: 1  Process: /extractorTb/check_scaler_configuration
↪  File: /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Note: Pixels for command 18 ok
Time: 8585 ns  Iteration: 1  Process: /extractorTb/check_pixels  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Note: Pixel check ok
Time: 8585 ns  Iteration: 1  Process: /extractorTb/check_pixels  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
Failure: ===== End =====
Time: 8585 ns  Iteration: 2  Process: /extractorTb/stim  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd
$finish called at time : 8585 ns : File
↪  "/home/felix/Unison/dynamic_tiling/vhdl/extractorTb.vhd" Line 272
```

Skalierer
Auf dem Datenträger: Protokolle/Tests/Komponententests/scalerTb.txt

```
Note: ===== Start: scalerTb =====
Time: 0 ps  Iteration: 0  Process: /scalerTb/stim  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Starting image generation...
Time: 10 ns  Iteration: 1  Process: /scalerTb/image_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Starting config generation...
Time: 10 ns  Iteration: 1  Process: /scalerTb/config_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Image 0 ok.
Time: 4505 ns  Iteration: 1  Process: /scalerTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Image 1 ok.
Time: 8475 ns  Iteration: 1  Process: /scalerTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Config generation finished.
Time: 18165 ns  Iteration: 1  Process: /scalerTb/config_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Image 2 ok.
Time: 19385 ns  Iteration: 1  Process: /scalerTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Image generation finished.
Time: 40625 ns  Iteration: 1  Process: /scalerTb/image_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Note: Image 3 ok.
Time: 44355 ns  Iteration: 1  Process: /scalerTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
Failure: ===== End =====
Time: 44355 ns  Iteration: 2  Process: /scalerTb/stim  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd
$finish called at time : 44355 ns : File
↪  "/home/felix/Unison/dynamic_tiling/vhdl/scalerTb.vhd" Line 123
```

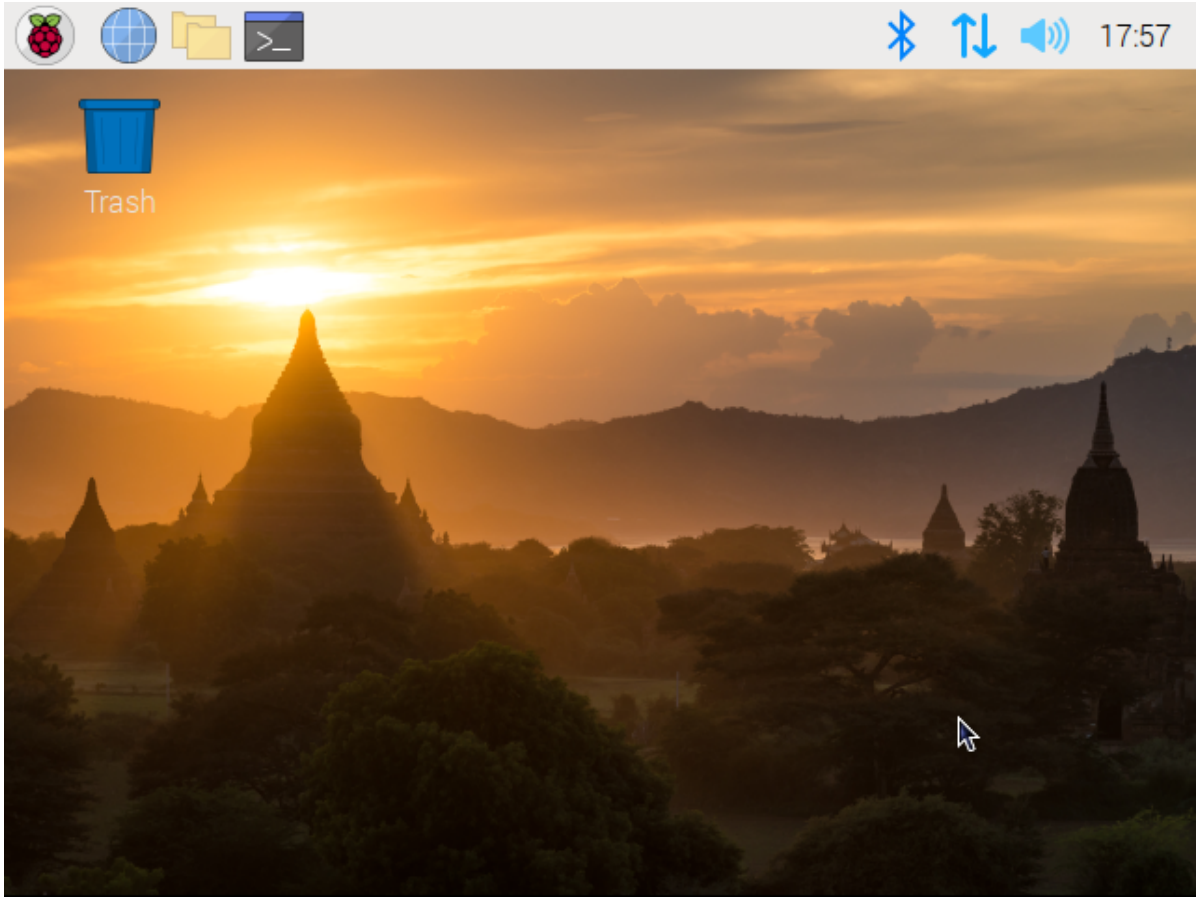
Ablage
Auf dem Datenträger: Protokolle/Tests/Komponententests/stream2memoryTb.txt

```
Note: ===== Start: stream2memoryTb =====
Time: 0 ps  Iteration: 0  Process: /stream2memoryTb/stim  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Starting address generation...
Time: 17500 ps  Iteration: 0  Process: /stream2memoryTb/address_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Starting data generation...
Time: 17500 ps  Iteration: 0  Process: /stream2memoryTb/data_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Address generation finished.
Time: 1377500 ps  Iteration: 0  Process: /stream2memoryTb/address_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Data generation finished.
Time: 15877500 ps  Iteration: 0  Process: /stream2memoryTb/data_gen  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Checking image 0
Time: 16047500 ps  Iteration: 0  Process: /stream2memoryTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Checking image 1
[gekürzt]
Time: 20127500 ps  Iteration: 0  Process: /stream2memoryTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Checking image 8
Time: 20607500 ps  Iteration: 0  Process: /stream2memoryTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Checking image 9
Time: 20637500 ps  Iteration: 0  Process: /stream2memoryTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Checking image 10
Time: 22557500 ps  Iteration: 0  Process: /stream2memoryTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Note: Checking image 11
Time: 22587500 ps  Iteration: 0  Process: /stream2memoryTb/check  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
Failure: ===== End =====
Time: 53307500 ps  Iteration: 1  Process: /stream2memoryTb/stim  File:
↪  /home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd
$finish called at time : 53307500 ps : File
↪  "/home/felix/Unison/dynamic_tiling/vhdl/stream2memoryTb.vhd" Line 124
```

F.2 Integrationstests

Es folgen die Protokolle der Integrationstests auf echter Hardware.

Auslesen eines über HDMI/DVI aufgenommenen Videosignals
Auf dem Datenträger: Protokolle/Tests/Integrationstests/test1.png



Kopieren eines Bildes mittels des Tilings auf dem FPGA
Auf dem Datenträger: Protokolle/Tests/Integrationstests/test2.txt

```
Starting integration test...
Clearing buffers...
Generating mock data...
Configuring command reader and extractor...
Starting tiling...
Checking data...
Integration test completed successfully.
```

G Inhalt des beiliegenden Datenträgers

- Protokolle/ - *Protokolle*
  - Blockdiagramme/ - *Hochauflösende Bilder der erstellen Blockdiagramme*
  - Evaluation/ - *Protokolle der Evaluation*
  - Tests - *Protokolle der Testdurchläufe*
- Skripte/ - *Verwendete Skripte zu Testzwecken*
- Thesis.pdf - *Digitale Version der Thesis*
- Thesis_Symmetrisch.pdf - *Digitale Version der Thesis mit symmetrischem Rand*
- Vivado.zip - *Quellcode für das FPGA (erstellt mit Vivado 2020.2)*
  - constraints/ - *Constraints für die physischen Anschlüsse am PYNQ-Z1*
  - ip_repo/ - *IP-Core Bibliothek*
    - bnn/ - *Verwendete Version des BNN-PYNQ*
    - digilent-library/ - *IP-Cores von Digilent (z.B. für DVI-Anbindung)*
    - finn/ - *Verwendetes und mit FINN erstelltes KNN*
    - scaler/ - *Quellcode des angepassten Skalierers*
  - project/dynamic_tiling.xpr - *Projektdatei zum Öffnen in Vivado*
  - project/dynamic_tiling.srcs/*/bd/ - *Blockdiagramme*
  - vhdl/ - *VHDL Quellcodedateien & Testbenches*
  - waveforms/ - *Waveforms für den Vivado Simulator*
- Vitis.zip - *Quellcode für die CPU (erstellt mit Vitis 2020.2)*