

Hochschule Rhein-Main
Fachbereich Design Informatik Media
Studiengang Angewandte Informatik

Bachelor-Thesis
zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Entwurf und Entwicklung einer Vektorgrafikeinheit

vorgelegt von Thorsten Knoll

am 27. September 2016

Referent: Prof. Dr. Steffen Reith
Korreferent: Dipl.-Inform.(FH), M.Sc. Marcus Thoss

Erklärung gem. ABPO, Ziff 6.4.3

Ich versichere, dass ich die Bachelor-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 27.09.2016

Thorsten Knoll

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Thesis:

Verbreitungsform	ja	nein
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger	✓	
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	✓	
Veröffentlichung des Titels der Arbeit im Internet	✓	
Veröffentlichung der Arbeit im Internet	✓	

Wiesbaden, 27.09.2016

Thorsten Knoll

Zusammenfassung

High-Level-Synthese (HLS) ist ein wachsender Bereich in der Programmierung von Field-Programmable-Gate-Arrays (FPGA). Durch die Synthese aus höheren Programmiersprachen zu Hardwarebeschreibungssprachen (ähnlich eines Compilers) rückt HLS in den Fokus von Softwareentwicklern. HLS bietet hierzu neue Ansätze, um Methoden und Vorgehensweisen der klassischen Softwareentwicklung zu adaptieren. Die sowohl kommerziellen als auch quelloffen verfügbaren HLS-Werkzeuge gehen dabei teilweise sehr unterschiedliche Wege, um die Abstraktion zwischen Soft- und Hardwareentwicklung zu erreichen. In dieser Arbeit wird das kommerzielle Werkzeug VivadoHLS des Herstellers Xilinx für die High-Level-Synthese zweier Implementierungsaufgaben aus der Sprache C verwendet und dessen Abstraktionsmerkmale besprochen. Eine Vektorgrafikeinheit (VGE) zur Darstellung von 3-dimensionalen Vektorobjekten auf Oszilloskop-Bildschirmen stellt das erste Implementierungsziel dar. Die Grundlagen der VGE über Vektorbildschirme und Computergrafik werden beschrieben und zum Design und der Implementierung der VGE verwendet. Ein Rasterisierer für die Darstellung der Vektorgrafik auf handelsüblichen Computerbildschirmen (Pixelbildschirme, TFT-Monitore) ist das zweite Implementierungsziel. Auch hier wird aus den Grundlagen über Pixelbildschirme und Rasterisierungsmethoden das Design und die Implementierung in VivadoHLS entwickelt. Ein Grundlagenkapitel über FPGAs und deren Programmierung in Hardwarebeschreibungssprachen und High-Level-Synthese ist zum generellen Verständnis des Designs und der Implementierungen enthalten. Anhand der Problemstellungen und Lösungen zur Implementierung der Vektorgrafikeinheit und des Rasterisierers wird eine Bewertung des HLS-Werkzeugs (VivadoHLS) zur Verwendung in den verschiedenen Aufgabenstellungen diskutiert.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Ziele	1
2	Grundlagen	3
2.1	Vektorbildschirme	3
2.2	Vektorgrafik	5
2.3	Pixelbildschirme	8
2.4	Rasterisierer	9
2.5	Field Programmable Gate Arrays (FPGA)	12
2.5.1	Einleitung und Marktentwicklung	12
2.5.2	Aufbau und Funktion	13
2.5.3	Hybrid-FPGAs	15
2.5.4	Developmentboards	15
2.6	FPGA-Programmierung	16
2.6.1	Arbeitsfluss	16
2.6.2	Hardwarebeschreibungssprachen (HDL)	17
2.6.3	High-Level-Synthese	22
2.6.4	Das Konzept von IP-Cores	27
2.6.5	Open-Source Werkzeuge	27
3	Analyse	29
3.1	Einleitung	29
3.2	Vektorgrafikeinheit	29
3.3	Rasterisierer	31
3.4	VivadoHLS-Bewertung	33
3.5	Optionales Spiel oder Grafikdemo	34

4 Design	35
4.1 Auswahl des FPGA-Boards	35
4.2 Vektorgrafikeinheit	36
4.2.1 Datenstrukturen	36
4.2.2 Objektsteuerung	38
4.2.3 Objektinitialisierung	39
4.2.4 Transformationsmodule	39
4.2.5 Backface-Culling und Projektion	40
4.2.6 Zeichnen	41
4.2.7 Grafik-Pipeline	41
4.2.8 XY-Ausgänge	43
4.2.9 Z-Ausgang	45
4.3 Rasterisierer	46
4.3.1 Designansatz	46
4.3.2 VGA-Generator	47
4.3.3 Bildschirmspeicher und Bildschirm-Pufferspeicher	47
4.3.4 Lesemodul	49
4.3.5 Kopiermodul	49
4.3.6 Schreibmodul	50
4.3.7 Synchronisation der Module	50
4.4 Grafikdemo	52
5 Implementierung	55
5.1 Installation der Werkzeuge	55
5.2 Quellcode: GPLv3	55
5.3 Vektorgrafikeinheit	56
5.3.1 Vorgehensweise	56
5.3.2 Top-Level-Funktion: vec_engine	56
5.3.3 Module: C-Unterfunktionen	61
5.4 Rasterisierer	67
5.4.1 VGA-Generator	67
5.4.2 Bildschirmspeicher und Bildschirm-Pufferspeicher	68
5.4.3 Lesemodul: vga_out	68
5.4.4 Kopiermodul: vga_copy	70
5.4.5 Schreibmodul	72
5.5 Grafikdemo	76
5.6 Integration in Vivado Design Suite	77

6 Fazit	79
6.1 Implementierungsziele	79
6.1.1 Vektorgrafikeinheit	79
6.1.2 Rasterisierer	81
6.1.3 Grafikdemo	82
6.2 Vivado High Level Synthese	82
6.3 Ausblick	83
7 Literaturverzeichnis	85
A Anhang	97

X

Kapitel 1

Einleitung

1.1 Motivation

Seit der Markteinführung von Field-Programmable-Gate-Arrays (FPGA) vor über 30 Jahren (1985, Xilinx, [Par13]) haben FPGAs die Nachfolge anderer programmierbarer Logikbausteine (z.Bsp User-programmable-Logic: UPL, Complex-programmable-Logic-Device: CPLD [Wan98]) übernommen. Zu den, aus der Elektrotechnik kommenden, Hardware-Beschreibungssprachen (Verilog, VHDL) ist die Entwicklung von FPGA-Programmierumgebungen für höhere Programmiersprachen (High-Level-Synthese, HLS) hinzugekommen [Ges14]. Zusammen mit FPGA-Hybrid-Plattformen, die FPGAs mit Prozessoren in einem Chip kombinieren, ergeben sich vielseitige Einsatzgebiete und Möglichkeiten der Programmierung von FPGAs. Diese Arbeit bedient sich älteren und aktuellen Bilddarstellungstechniken (Vektorbildschirme, Pixelbildschirme) in Kombination mit aktueller FPGA-Hardware und Programmiermethoden (HLS), um die Evaluierung dieses Teilgebiets der FPGA-Programmierung anschaulich zu erarbeiten. Auch das persönliche Interesse des Autors an Spielekonsolen und Arcadeautomaten des letzten Jahrtausends war für die Auswahl der Implementierungsziele ein Entscheidungsfaktor. Es sind drei festgelegte Ziele und ein optionales Ziel für diese Arbeit definiert. Diese werden im Folgenden beschrieben:

1.2 Ziele

Vektorgrafik:

Es wird eine Vektorgrafikeinheit (VGE) als Implementierung für ein FPGA-Board entworfen und entwickelt. Zur Programmierung des FPGAs wird das High-Level-Synthese-Werkzeug VivadoHLS des Herstellers Xilinx mit der Ausgangssprache C verwendet. Die

VGE wird einfache 3-dimensionale Objekte auf einem Vektorbildschirm darstellen. Es werden Module (C-Funktionen) und Datenstrukturen (C-Structs) für die Modellierung, Transformation, einfache Verdeckungsberechnung, Projektion und Darstellung der Objekte erstellt. Diese Module und Strukturen sind als Grafik-Pipeline zu integrieren. Die elektrotechnischen Schaltungen für die Ansteuerung und Ausgabe auf Vektorbildschirmen sind Bestandteil des Entwurfs und der Entwicklung. Eine geeignete FPGA-Plattform ist auszuwählen.

Rasterisierer:

Auf Basis der VGE wird ein Rasterisierer für die Darstellung der Vektorobjekte auf einem Pixelbildschirm entworfen und entwickelt. Diese Programmierung wird aus der Sprache C mit VivadoHLS entwickelt. Ein Video-Grafics-Array-Ausgang (VGA) wird mit dem zugehörigen Übertragungsprotokoll implementiert. Zum Rasterisieren der Vektoren der VGE wird ein Algorithmus ausgewählt, welcher im Zusammenspiel mit zu definierendem Bildschirmspeicher eine Ausgabe auf Pixelbildschirmen (z.Bsp. TFT-Computermonitor) ermöglicht. Der Rasterisierer wird zusammen mit der VGE auf der selben FPGA-Plattform implementiert. Somit ist die Bildausgabe der VGE gleichzeitig auf einem Vektorbildschirm und auf einem Pixelbildschirm möglich.

Vivado-HLS Bewertung:

Anhand der zwei beschriebenen Implementierungsziele wird eine Bewertung des HLS-Werkzeugs VivadoHLS vorgenommen. Die zwei Implementierungsziele sind so gewählt, dass unterschiedliche Methoden und Techniken aus der High-Level-Synthese Anwendung finden. Hierdurch sollen Aussagen über die Verwendung von VivadoHLS für verschiedene Problemstellungen erarbeitet werden.

Optionales Spiel oder Grafikdemo:

Als beispielhafte Anwendung der Vektorgrafikeinheit und des Rasterisierers könnte ein Spiel oder eine Grafikdemo implementiert werden. Diese würden die Möglichkeiten der beschriebenen Grafik-Pipeline anschaulich darstellen. Die Bewegungslogik und Ansteuerung der VGE könnte, bei Auswahl einer Hybrid-FPGA-Plattform (FPGA mit ARM-CPPUs), aus einem auf der ARM-CPU ausgeführten Linuxsystem erfolgen.

Kapitel 2

Grundlagen

2.1 Vektorbildschirme

Vektorbildschirme sind eine Bauart von Kathodenstrahlröhren, welche nach ihrem Erfinder Ferdinand Braun als Braunsche Röhre bezeichnet werden [Bra97].

Das Grundprinzip einer braunschen Röhre besteht in der Emission von Elektronen aus einer erhitzten Kathode. Die emittierten Elektronen werden, durch Anlegen einer Spannung zwischen Kathode und Anode, zur Anode hin angezogen. Die Anode ist in Form einer Lochblende (Wehnetyylinder) ausgeführt. Diese Lochblende ist damit ein Ausgang für einen gerichteten Elektronenstrahl.

Der Elektronenstrahl wird nach seinem Austritt aus der Lochblende durch magnetische Felder in horizontale und vertikale Richtung (X- und Y-Achse) abgelenkt. Hierfür kamen bei Braun sowohl Elektromagneten als auch drehende Festmagneten zum Einsatz. Den sichtbaren Abschluss der braunschen Röhre stellt eine fluoreszierende Schicht dar, welche durch das Auftreffen der Elektronen zum punktförmigen Leuchten angeregt wird. Die ganze beschriebene Konstruktion ist in einen geschlossenen Vakuum-Glaszyylinder eingebaut. Ferdinand Braun konnte mit dieser Konstruktion die Sinusspannungen des Straßburger Elektrizitätswerks und selbsterzeugte Lissajous-Figuren auf der vorderseitigen Glasscheibe beobachten. Trotz der durch August Karolus schon 1924 vor gestellten elektronischen Übertragung von Bildern, konnte das Prinzip der braunschen Röhre erst durch die Verbesserung der Bildschirm-Phosphore und verbesserten Fokusie-

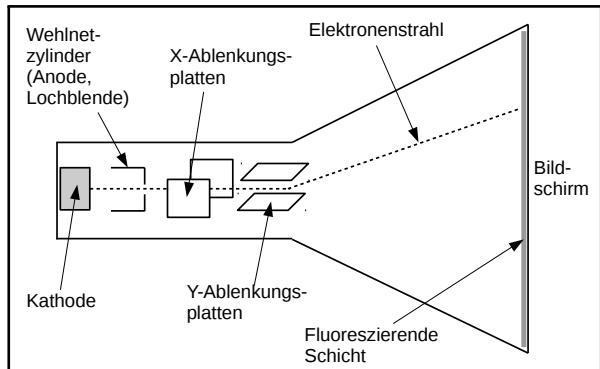


Abbildung 2.1: Oszilloskop-Bildröhre

rung des Elektronenstrahls ab 1930 als Darstellungsmedium Verwendung finden [Kar84]. Es entwickelten sich ab 1930 zwei unterschiedliche Anwendungen aus der Erfindung der braunschen Röhre. Manfred von Ardenne stellte 1931 auf der Berliner Funkausstellung das erste Fernsehübertragungssystem mit einer braunschen Röhre vor [AA88] und das erste Oszilloskop wurde 1932 von der Firma A.C.Cossor (später Raytheon) gebaut [UK11]. Ein wesentlicher Unterschied in der Entwicklung dieser zwei Anwendungen der braunschen Röhre besteht in der Verwendung von elektrischen Feldern (Oszilloskope) oder magnetischen Feldern (Fernseher) zur Ablenkung des Elektronenstrahls [SF78] [Kar84]. Magnetische Felder ermöglichen kurze Röhrenbauformen mit großen Bildschirmflächen in begrenzten Frequenzbereichen der Ablenkspannungen. Im Gegensatz dazu sind mit elektrischen Feldern größere Frequenzbereiche für die Ablenkspannungen möglich, aber die Röhre wird wesentlich länger und die Bildschirmfläche kleiner. Da Oszilloskope als Messinstrumente für Signale in großen Frequenzbereichen funktionieren müssen, werden elektrische Felder zur Ablenkung verwendet [SF78]. In Abbildung 2.1 ist eine Oszilloskop-Kathodenstrahlröhre mit den weiter oben beschriebenen Bestandteilen der braunschen Röhre und den Ablenkplatten für die X- und Y-Achsen dargestellt. Außerdem ist die Position des Wehneltzylinders, welcher die Intensität (Helligkeit, Z-Achse) des Elektronenstrahls regelt, bezeichnet. Oszilloskope haben als Messinstrumente mehrere Funktionsmodi. Im zeitgesteuerten Modus wird die X-Achse von einem internen Dreieck-Funktionsgenerator gesteuert und ist als Zeitbasis der Messung über ein Bedienelement einzustellen. Im X-Y-Modus entfällt die Steuerung durch den Funktionsgenerator und beide Ablenkspannungen sind als Eingänge am Oszilloskop verfügbar. Die Spannungen an den X- und Y-Eingängen entsprechen in diesem Modus linear den Koordinaten auf dem Bildschirm. Zwei feste X- und Y-Spannungen erzeugen somit einen leuchtenden Bildschirmpunkt an den Koordinaten X und Y. Werden die X- und Y-Eingangsspannungen verändert, bewegt sich der Leuchtpunkt auf dem Bildschirm zur neuen Position und hinterlässt dabei eine Leuchtpur, welche den Vektor zwischen den zwei Punkten darstellt. Zusammen mit der Z-Achsen-Steuerung für die Helligkeit sind Oszilloskope damit zur Darstellung von Vektorgrafik geeignet. In diesem X-Y-Modus ist das Oszilloskop somit ein Vektorbildschirm [SF78]. Vektorbildschirme zeichnen sich durch die kontinuierliche, nicht in Rasterschritte unterteilte Darstellung aus. Bis zur Entwicklung und Verbreitung von Flüssigkristall-Bildschirmen (Pixelbildschirme, Kapitel 2.3) waren Vektorbildschirme in vielen Anwendungsbereichen im Einsatz. Beispiele hierfür sind Luftfahrt (Radarbildschirme), Medizin (Röntgengeräte) und Spielautomaten (Anhang A.1, Vectrex Spielekonsole). In dieser Bachelorthesis werden Oszilloskope als Vektorbildschirme für die Bilddarstellung der entwickelten Vektorgrafikeinheit benutzt.

2.2 Vektorgrafik

Die entwickelte Vektorgrafikeinheit basiert auf den mathematischen Grundlagen der Computergrafik und des kartesischen Koordinatensystems. Das 3-dimensionale kartesische Koordinatensystem ist ein Koordinatenraum mit den 3 zueinander orthogonalen und damit linear unabhängigen Einheitsvektoren x_e , y_e und z_e [XP07]. Das System ist rechts-händig orientiert. Analog zur Abbildung 2.2 werden in diesem Koordinatensystem die Strukturen in Tabelle 2.1 definiert [XP07].

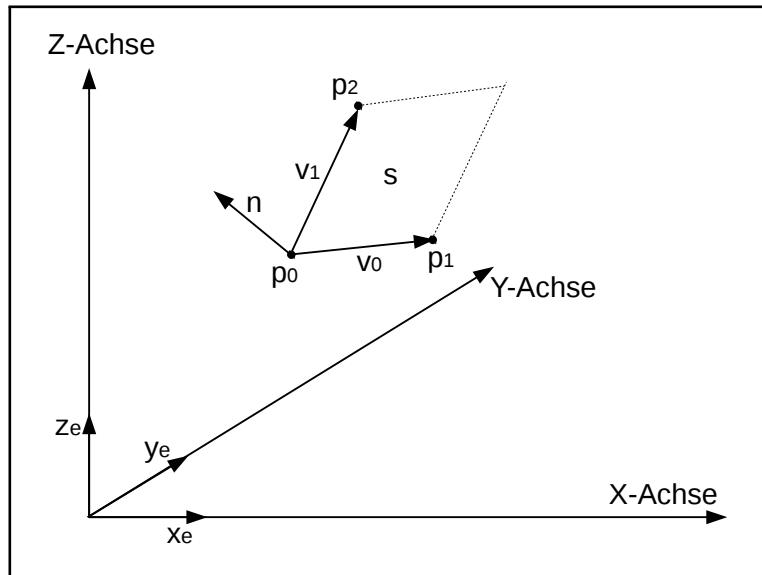


Abbildung 2.2: Koordinatensystem und Strukturen

Struktur	Definition
Punkt p	$p_0 = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$
Vektor v	$v_0 = p_1 - p_0 = \begin{pmatrix} x_1 - x_0 \\ y_1 - y_0 \\ z_1 - z_0 \end{pmatrix}$
Fläche s	$s = (v_0, v_1)$
Skalarprodukt t	$t = \langle v_0, v_1 \rangle = (x_0 \cdot x_1 + y_0 \cdot y_1 + z_0 \cdot z_1)$
Flächennormale n	$n = v_0 \times v_1 = \begin{pmatrix} y_0 \cdot z_1 - z_0 \cdot y_1 \\ z_0 \cdot x_1 - x_0 \cdot z_1 \\ x_0 \cdot y_1 - y_0 \cdot x_1 \end{pmatrix}$

Tabelle 2.1: Definition der Strukturen

Um, aus Punkten und Vektoren modellierte, 3-dimensionale Objekte (z.Bsp Würfel oder Pyramiden) im Raum zu verschieben, zu drehen und zu skalieren, werden Transformationsberechnungen ausgeführt. Diese Berechnungen sind durch die Multiplikation von Transformationsmatrizen mit den Punkten und Vektoren zu erreichen. Vor Ausführung der Transformationsberechnungen sind die kartesischen Koordinaten der Punkte und Vektoren in homogene Koordinaten umzuwandeln (Tabelle 2.2) [XP07].

Kartesische Koordinaten	Homogene Koordinaten
$p_{kar} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$	$p_{hom} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

Tabelle 2.2: Kartesische und homogene Koordinaten

In Tabelle 2.3 sind die Transformationsmatrizen für Verschiebung (Translation), Größenveränderung (Skalierung) und Drehung um die 3 Koordinatenachsen (Rotation) aufgeführt [BB06]. Translation und Skalierung sind hierbei relativ zum Koordinatursprung ($x = 0, y = 0, z = 0$) und die Rotationen relativ zur entsprechenden Koordinatenachse.

Transformation	Transformationsmatrix
Translation	$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Skalierung	$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
X-Rotation	$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Y-Rotation	$R_y(\phi) = \begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Z-Rotation	$R_z(\phi) = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 & 0 \\ \sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
------------	---

Tabelle 2.3: Transformationsmatrizen

Die Abbildung 3-dimensionaler Objekte auf eine 2-dimensionale Fläche ist die Projektion. Es wird zwischen zwei Grundarten der Projektion unterschieden, der Parallelprojektion und der perspektivischen Projektion (Zentralprojektion). Die perspektivische Projektion ist dem menschlichen Sehen ähnlicher als die Parallelprojektion und stellt entfernte Objekte kleiner dar als nahe Objekte. Dies führt zu Abbildungen mit bis zu 3 Fluchtpunkten im Horizont (Abbildung 2.3) [XP07].

Unter den Annahmen, dass die Projektionsfläche von den X- und Y-Koordinatenachsen mit $z = 0$ aufgespannt wird und der Ansichtspunkt (Betrachter) sich mit dem Abstand $-d$ auf der Z-Achse befindet (Ansichtspunkt: $p_{ansicht} = (0, 0, -d)$), ergibt sich für die perspektivische Projektion folgende Projektionsmatrix:

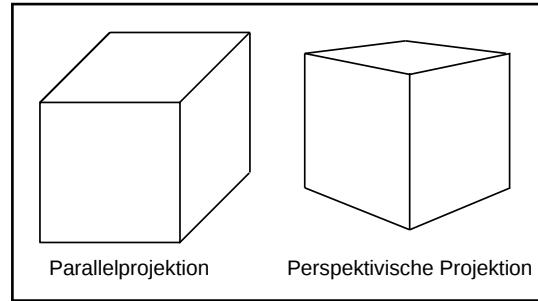


Abbildung 2.3: Projektionsarten mit Flächenverdeckung (Backface Culling)

$$P_{persp} = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & d \end{pmatrix}$$

In Abbildung 2.3 sind die Kanten der nicht sichtbaren Flächen nicht eingezeichnet (verdeckt). Diese Art der Verdeckungsberechnung wird in der Computergrafik als Backface Culling bezeichnet [BB06]. Zur Feststellung ob eine Fläche für den Betrachter aus dem Ansichtspunkt $p_{ansicht}$ sichtbar ist, wird das Skalarprodukt aus dem Sichtvektor auf die Fläche und dem Flächennormalenvektor gebildet. Für die Fläche s aus Abbildung 2.2 ist das Skalarprodukt:

$$t_s = \langle p_0 - p_{ansicht}, n \rangle$$

Die Fläche wird nicht dargestellt, wenn der Winkel zwischen den zwei Vektoren größer als 90° ist (Skalarprodukt ist negativ). Bildlich beschrieben sieht der Betrachter damit die Rückseite der Fläche. Backface Culling löst nicht alle Flächenverdeckungsprobleme, ist aber in der Computergrafik oft der erste Berechnungsschritt der gesamten Verdeckungs-

rechung [BB06]. Aus den bisher beschriebenen Berechnungen wird eine Vektorgrafik-Pipeline erstellt, welche die Berechnungen in einer festgelegten Reihenfolge abarbeitet. Diese Pipeline (Abbildung 2.4) ist die Grundlage für das Design (Kapitel 4.2) und die Implementierung (Kapitel 5.3) der Vektorgrafikeinheit.

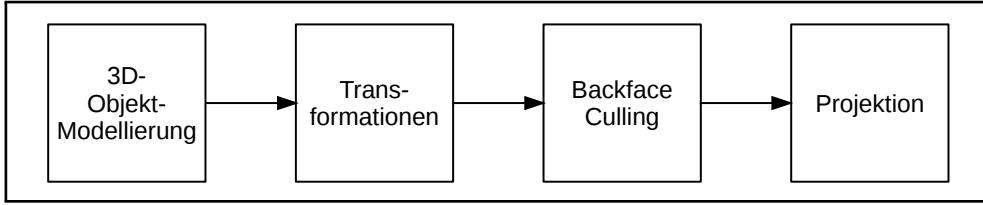


Abbildung 2.4: Vektorgrafik-Pipeline

2.3 Pixelbildschirme

Im Vergleich zur kontinuierlichen Zeichnung von Vektoren in Vektorbildschirmen, werden in Pixelbildschirmen die Bildschirmpunkte (Pixel) einzeln angesteuert. Pixelbildschirme sind damit an die diskrete Anzahl der darstellbaren Pixel gebunden (Bildschirmauflösung) [Gä08]. Moderne Bauarten von Pixelbildschirmen sind Liquid Crystal Displays (LCD), Thin-Film-Transistor-Displays (TFT), und Light-Emitting-Diode-Displays (LED). Wobei die Bezeichnung LED-Bildschirme sich nicht auf die Bilderzeugung (LCD, TFT), sondern auf die Hintergrundbeleuchtung des Bildschirms bezieht [Gä08]. Zur Übertragung der Bildinhalte zu Pixelbildschirmen gibt es mehrere Grafik-Schnittstellen, die an zugehörige Übertragungsprotokolle gebunden sind. Einige der gebräuchlichsten Schnittstellen sind die Video-Grafics-Array-Schnittstelle (VGA), das Digital-Visual-Interface (DVI), das High-Definition-Multimedia-Interface (HDMI) und die Displayport-Schnittstelle [Fis15]. Im Folgenden wird die VGA-Schnittstelle beschrieben. In Abbildung 2.5 ist die VGA-Anschlussbuchse mit den zur Bildübertragung notwendigen Signalen dargestellt. Dies sind drei Signale für die Farübertragung (Rot, Grün, Blau), zwei Signale für horizontale und vertikale Synchronisation und die zugehörigen Masseleitungen. Die Zusammenhänge dieser 5 Signale sind in Abbildung 2.6 dargestellt. Nach jeder Bildzeile findet eine horizontale Synchronisation in der Abfolge H-Front-Porch, H-Sync-Impuls und H-Back-Porch statt und am Ende jedes Bilds findet eine vertikale Synchronisation in der Abfolge V-Front-Porch, V-Sync-Impuls und V-Back-Porch statt. Für die in Kapitel 5 beschriebene Implementierung (1024*768

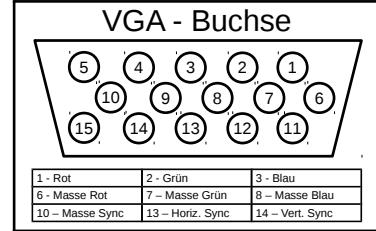


Abbildung 2.5: VGA-Schnittstelle mit Signalen

Pixel bei 60 Hz Bildwiederholungsfrequenz) sind die VGA-Signal-Timings in Tabelle 2.4 zusammengefasst. IBM hat diese Timings im Jahr 1992 als XGA-2-Display-Mode für die IBM PS/2-Computer definiert [IBM92].

Bildauflösung	1024 * 768 Pixel
Bildwiederholungsfrequenz	60 Hz
Bildzeile	1024 Pixel
H-Front-Porch	24 Pixel
H-Sync-Länge	136 Pixel
H-Back-Porch	160 Pixel
Zeilen pro Bild	768 Zeilen
V-Front-Porch	3 Zeilen
V-Sync-Länge	6 Zeilen
V-Back-Porch	29 Zeilen
Pixelfrequenz	65 MHz

Tabelle 2.4: VGA-Signal-Timings

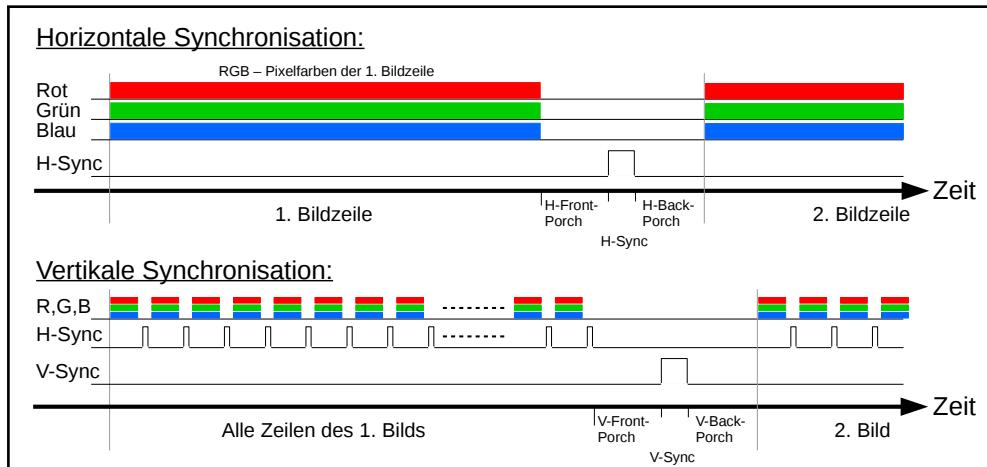


Abbildung 2.6: VGA-Signale: Rot, Grün, Blau, H-Sync, V-Sync

2.4 Rasterisierer

Um die in Kapitel 2.2 beschriebenen Vektor-Objekte auf einem Pixelbildschirm darzustellen, wird ein Rasterisierer verwendet. Die grundlegendsten Objekte der Vektorgrafikeinheit sind Punkte und Linien. Punkte werden durch Skalierung aus dem Koordinatensystem der Vektoreinheit auf das Koordinatensystem eines Pixelbildschirms angepasst. Die Vektoren werden mit einem Berechnungsalgorithmus vom Startpunkt bis zum Endpunkt in pixelbasierte Linien umgewandelt. Ein Algorithmus für diese Linienberechnung ist der

1962 von Jack Bresenham entwickelte und nach ihm benannte Bresenham-Algorithmus [XP07] [BB06]. In einem 2-dimensionalen kartesischen Koordinatensystem (Einheitsvektoren x_{ein} und y_{ein} sind orthogonal zueinander und damit linear unabhängig) werden der Startpunkt $p_s = (x_s, y_s)$ und Endpunkt $p_e = (x_e, y_e)$ einer Linie definiert. Beide Punkte seien im 1. Quadranten des Koordinatensystems mit den Beschränkungen $x_s < x_e$, $y_s < y_e$ und $0 < m = \Delta y / \Delta x = (y_e - y_s) / (x_e - x_s) \leq 1$ angeordnet (siehe Abbildung 2.7).

Zur iterativen Berechnung der Linienpunkte wird die Laufvariable $x_i \in [x_s, x_e]$ mit der Initialisierung $x_0 = x_s$ und den Iterationsschritten $x_{i+1} = x_i + 1$ definiert. Anschaulich erklärt läuft x_i in Pixelschritten über den X-Achsen-Bereich der zu zeichnenden Linie. In jedem X-Schritt wird der zu diesem Linienpunkt gehörige Y-Wert durch die Betrachtung des Abweichungsfehlers von der Geradengleichung ermittelt. Im Folgenden wird die Herleitung des Bresenham-Algorithmus wie in [XP07] beschrieben. Wie in Abbildung 2.8 dargestellt, gibt es für den auf P_i folgenden Linienpunkt P_{i+1} zwei mögliche Kandidaten (Punkt T oder Punkt S). Der Algorithmus soll den jeweils zur Ideallinie näheren Folgepunkt bestimmen. Hierzu wird die Differenz aus den beiden Abweichungsabständen s und t betrachtet. Ist die Differenz ($s - t$) positiv oder gleich null wird der obere Punkt T als Folgepunkt gewählt, bei negativer Differenz der untere Punkt S.

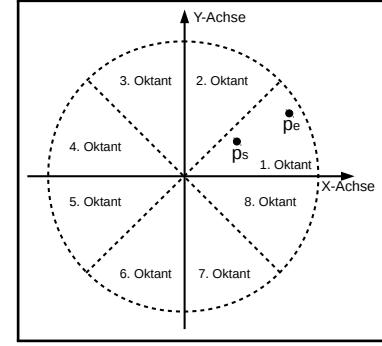


Abbildung 2.7: Oktanten, Linienstart- und Linienendpunkt

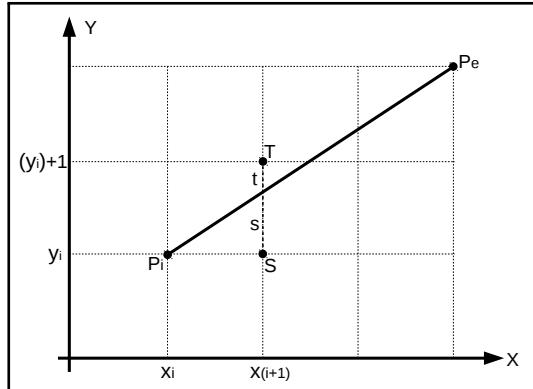


Abbildung 2.8: Folgepunktbestimmung im Bresenham-Algorithmus

Durch Einsetzen der allgemeinen Geradengleichung $y = m * x + b$ mit $m = (\Delta y / \Delta x)$ wird folgende Gleichung für die Differenz erstellt:

$$s - t = (y - y_i) - ((y_i + 1) - y) \quad (2.1)$$

$$= 2y - 2y_i - 1 \quad (2.2)$$

$$= 2(\Delta y / \Delta x)(x_i + 1) + 2b - 2y_i - 1 \quad (2.3)$$

Es wird eine Entscheidungsvariable d_i definiert und die Gleichung nach dieser Variablen aufgelöst:

$$d_i = \Delta x(s - t) \quad (2.4)$$

$$= 2x_i\Delta y - 2y_i\Delta x + 2\Delta y + 2(b\Delta x - 1) \quad (2.5)$$

Für den nächsten Schritt $i + 1$ ergibt sich für die Entscheidungsvariable d_{i+1} :

$$d_{i+1} = 2x_{i+1}\Delta y - 2y_{i+1}\Delta x + 2\Delta y + 2(b\Delta x - 1) \quad (2.6)$$

Durch Subtraktion der Gleichungen 2.6 und 2.5 entsteht die iterative Berechnungsformel für die Entscheidungsvariable:

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i) \quad (2.7)$$

Wie weiter oben beschrieben, wird bei positivem oder gleich null berechnetem $(s - t)$ (Und damit auch positivem d_i , durch die Beschränkung auf den 1.Oktanten) der Punkt T gewählt, sonst der Punkt S . Es entsteht somit eine Fallunterscheidung für d_{i+1} :

$$d_{i+1} = \begin{cases} d_i + 2(\Delta y - \Delta x) & \text{wenn } d_i \geq 0 \\ d_i + 2\Delta y & \text{wenn } d_i < 0 \end{cases} \quad (2.8)$$

Für die Initialisierung der Entscheidungsvariable (d_0) ergibt sich durch Einsetzen von $(\Delta y/\Delta x)x_0 + b - y_0 = 0$ in Gleichung 2.5:

$$d_0 = 2\Delta y - \Delta x \quad (2.9)$$

In Gleichung 2.8 sind nur Additionen und eine Multiplikation mit dem Faktor 2 zu berechnen. Alle Berechnungen finden nur mit ganzen Zahlen statt. Daher ist der Bresenham-Algorithmus für Implementierungen in Hardware (oder in FPGAs) sehr gut geeignet. Die Anwendung des Bresenham-Algorithmus auf Linien in den Oktanten 2-8 ist auf Grund der Symmetrie durch Vorzeichenwechsel und Variablentausch in der Implementierung zu erreichen. Der Pseudocode des Bresenham-Algorithmus ist im Anhang A.1 aufgelistet. Um das vollständige Vektorbild der in Kapitel 2.2 beschriebenen Vektorgrafik auf einem Pixelbildschirm darzustellen, werden alle Objektvektoren mit dem Bresenham-Algorithmus als Linien berechnet. In den Kapiteln 4.3 und 5.4 sind das Design und die Implementierung des Bresenham-Algorithmus als Rasterisierer beschrieben.

2.5 Field Programmable Gate Arrays (FPGA)

2.5.1 Einleitung und Marktentwicklung

Field Programmable Gate Arrays (FPGA) sind frei programmierbare, elektronische Logikbauteile [Sau10]. Der erste kommerzielle FPGA wurde 1985 von den Xilinx-Mitgründern Ross Freeman und Bernard Vonderschmitt unter der Bezeichnung XC2064 hergestellt [Wan98]. Der XC2604 war ausgestattet mit 64 Logikeinheiten, jeweils 2 Look-Up-Tables mit 3 Eingängen pro Logikeinheit und konnte mit bis zu 18 MHz Taktfrequenz betrieben werden [Par13]. Aktuelle FPGAs beeinhalten bis zu mehrere hunderttausend Logikeinheiten, Millionen von Look-Up-Tabellen und sind im Bereich mehrer hundert MHz zu betreiben [Par13]. Die Bedeutung von FPGAs ist, seit ihrer Markteinführung 1985, über die Prototypenanfertigung für Application Specific Integrated Circuits (ASIC) und die Verwendung in Kleinserien für notwendig umkonfigurierbare Hardwaresysteme (z.Bsp. Mobilfunkstationen mit Protokollupdates) in weitere Anwendungsbereiche gewachsen [Sau10].

Lange Produktlebenszyklen und kürzer werdende Technikentwicklungszyklen machen FPGAs als eigenständige Produktplattform immer interessanter [Sau10]. Als Beispiel sei die aktuelle Entwicklung von Auto-zu-Auto-Kommunikation aufgeführt. Welche Informationen in zukünftigen Autogenerationen auf welchen Übertragungskanälen kommuniziert werden, ist in einer Abschätzung auf 10 Jahre Lebenszeit eines Autos nicht exakt prognostizierbar. Hier könnte die einfache, per Softwareupdate einspielbare, Umkonfigurierbarkeit von FPGAs einen Lösungsansatz bereitstellen. Die kontinuierlich fallenden Preise und die Verfügbarkeit günstiger Entwicklungsboards rücken FPGAs mittlerweile auch in den Interessensbereich von Hobby-Entwicklern und der Maker-Szene [Sau10]. High-Level-Synthese bietet die FPGA-Programmierung in höheren Programmiersprachen als die klassischen Hardware-Beschreibungssprachen (HDL) und damit eine wachsende Verbindung zwischen Elektrotechnik und Informatik. In Informatikstudiengängen haben sich Kurse zu FPGA-Programmierung etabliert.

Die von Gordon Moore vorhergesagte Verdopplung der Transistoren pro Fläche alle 18

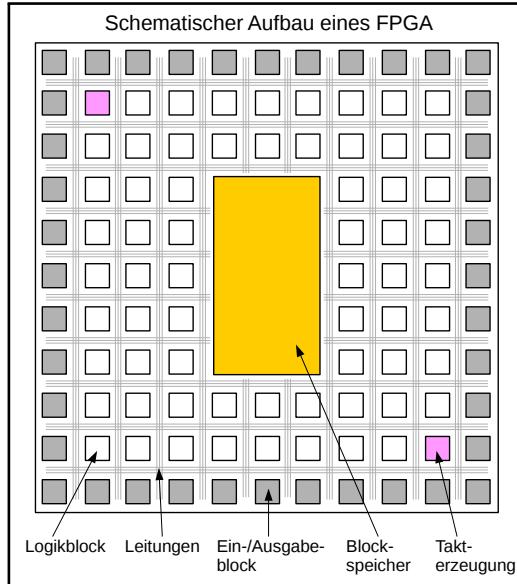


Abbildung 2.9: Schematischer Aufbau eines FPGA-Chips

Monate (Moore's Gesetz) trifft auch für FPGAs zu, scheint aber hier noch länger gültig zu sein als für klassische Prozessorarchitekturen. [Ges14]. Auch die Preisschere zwischen ASIC- und FPGA-Entwicklung, in welcher die ASIC-Entwicklung erst ab einer bestimmten Produktionsmenge günstigere Pro-Stück-Preise als die FPGA-Entwicklung bietet (Non-Recurring-Engineering-Kosten, NRE), entwickelt sich durch die rapide fallenden FPGA-Preise zu Gunsten von FPGAs [Ges14] [Sau10]. Sowohl ASIC-Hersteller arbeiten daher an der Integration von konfigurierbaren Logikteilen in Structured ASICs, als auch FPGA-Hersteller an der Integration von festen Schaltungsbestandteilen in ihre Designs (FPGA-Hybrid-Chips) [Sau10]. Ein solcher FPGA-Hybrid-Chip mit zwei ARM-CPUs, FPGA-Teil, PC-Schnittstellen (VGA, HDMI, Ethernet, Audio) und DDR3-Speicher wird auch als Plattform für das Design und die Implementierung dieser VGE verwendet. Der Preis dieses Entwicklungsboards liegt zum Zeitpunkt der Erstellung dieser Arbeit bei unter 200 Euro.

2.5.2 Aufbau und Funktion

Obwohl FPGAs von verschiedenen Herstellern angeboten werden, sind grundlegende Strukturen und Einheiten gleich. Die Bezeichnungen der FPGA-Grundbestandteile kann bei unterschiedlichen Herstellern abweichen. Es wurde daher versucht, die Bestandteile herstellerunabhängig nach ihrer Funktion und ihrer elektrotechnischen Entsprechung zu benennen. In Abbildung 2.9 sind die Bestandteile Logikblöcke, Speicherblöcke, Ein-/Ausgangsblöcke, Taktzeugung und Verbindungsleitungen in einer schematischen Übersicht dargestellt [Sau10] und im Folgenden beschrieben.

Logikblöcke:

Die grundlegendsten, konfigurierbaren Module in FPGAs sind Logikblöcke. Die Anzahl der Bestandteile eines Logikblocks kann zwischen den verschiedenen FPGA-Herstellern differieren, daher wird hier eine einfache Variante zur Darstellung der Funktionen beschrieben.

Die Lookup-Table (LUT) ist eine frei konfigurierbare Wahrheitstabelle, und kann somit logische Funktionen abbilden. Die LUT wird durch die Programmierung des FPGAs befüllt [Sau10]. LUTs haben mehrere Eingänge, um die Tabelleneinträge anzusteueren und hier im Beispiel einen Ausgang, der den gespeicherten Tabelleneintrag ausgibt. Der

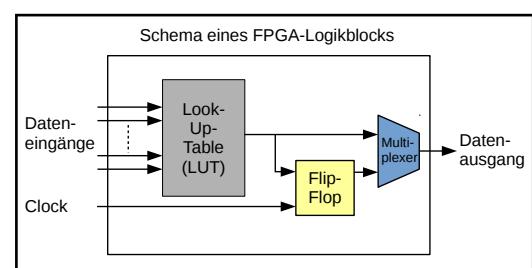


Abbildung 2.10: Schema eines FPGA-Logikblocks

LUT-Ausgang kann mit einem Flip-Flop auf eine eingehende Clock synchronisiert werden, oder direkt (unsynchronisiert) an den Ausgang weitergeleitet werden. Die Entscheidung, ob der Flip-Flop-Ausgang oder der direkte LUT-Ausgang auf den Ausgang des Logikblocks geschaltet wird, ist durch einen Multiplexer als letztem Bestandteil des Logikbausteins realisiert.

Speicherblöcke:

FPGAs enthalten Speichereinheiten direkt auf dem Chip, in der Nähe der Logikblöcke. Es wird zwischen zwei Arten von Speicher im FPGA unterschieden, dem Blockspeicher (Block-RAM) und dem verteilten Speicher (Distributed-RAM) [Sau10]. Block-RAM besteht aus funktional als Speicher festgelegten Einheiten mit mehreren KBit Größe. Distributed-RAM wird durch die FPGA-Programmierung aus Logikblöcken zusammengestellt. Beide Speichervarianten können als Single- oder Dual-Port-Speicher angelegt werden. In der Single-Port-Konfiguration sind die Ein- und Ausgangsdatensignale und die Adressleitungen nur einmal vorhanden, während für die Dual-Port-Konfiguration alle Signale zweifach bereitstehen und parallel betrieben werden können. Schreib- und Lesezugriffe sind somit gleichzeitig möglich. FPGA-Speicher wird mit einem Set aus Steuerleitungen synchronisiert. Im Minimalfall sind dies die Signale Chip-enable (ce), Write-enable (we), Read-enable (re) und der Synchronisationstakt (clk).

Ein-/Ausgangsblöcke:

Für die elektronische Anbindung der FPGAs an externe Komponenten sind Ein-/Ausgänge (I/O-Ports) am Rand der Chipfläche vorhanden. I/O-Ports sind durch die FPGA-Programmierung konfigurierbare Treiberbausteine, die eine Menge von Eigenschaften für jeden einzelnen I/O-Anschluss zuweisen können. Neben der Definition des I/O-Anschlusses als Eingang, als Ausgang oder als Ein- und Ausgang können Signalpegel, Pull-Up oder Pull-Down-Widerstände und der Betrieb im Open-Kollektor-Modus festegelegt werden [Sau10].

Takterzeugung:

In FPGAs sind üblicherweise mehrere Takterzeugungsblöcke auf der Chipfläche verteilt angeordnet. Durch die Verteilung werden die Takt-Signal-Laufzeiten zu den Logikblöcken möglichst gering gehalten. Mit Phase-Locked-Loop Modulen können Taktfrequenzen vervielfacht und mit Taktteilern geteilt werden [Sau10]. So ist die Erzeugung unterschiedlicher Taktfrequenzen für unterschiedliche Takt-Domains (Clockdomains) möglich. Aktuelle FPGAs können mit bis zu mehreren Hundert MHz getaktet werden.

Verbindungsleitungen:

Die beschriebenen FPGA-Bestandteile werden über konfigurierbare Verbindungsleitungen miteinander konnektiert. So werden Logikblöcke zu größeren Schaltungseinheiten zusammengefasst. Die Auswahl und Konfiguration der FPGA-Bestandteile und deren Verbindung untereinander wird durch die Programmierung des FPGAs festgelegt und stellt somit die gewünschte, programmierte Funktionalität des FPGAs her.

Digitale Signalprozessoren(DSP):

Je nach Hersteller des FPGAs gibt es weitere (nicht in Abbildung 2.9 dargestellte) Bausteine. In Xilinx-FPGAs sind zum Beispiel zusätzlich digitale Signal-Prozessoren (DSP) enthalten. Diese können bestimmte Zahlenarithmetik-Funktionen (z.Bsp. Multiplikationen) sehr schnell ausführen, ohne dafür Logikblöcke zu verwenden.

2.5.3 Hybrid-FPGAs

Seit mehreren Jahren sind von verschiedenen FPGA-Herstellern (z.Bsp. Xilinx und Altera) Hybrid-Chips erhältlich. Wurden anfänglich nur 1-2 Prozessoren zusammen mit Memory-Management-Units (MMU), Bussystem-Einheiten (AXI, AMBA) mit auf dem FPGA-Chip untergebracht, sind in aktuellen Architekturen viele verschiedene Prozessorsysteme mit dedizierten Aufgaben als integrierte Lösung mit FPGAs erhältlich. Als Beispiel sei hier die Xilinx UltraScale-MPSoC-Architektur erwähnt. Diese hat 7, in ihrer Funktion unterschiedliche, ARM-Cortex-CPUs mit dem FPGA vereint und baut auf einer Virtualisierungsumgebung mit Hypervisor auf [Inc].

2.5.4 Developmentboards

Entwicklungsboards (Developmentboards) gibt es in einer Vielzahl von Ausführungen und Größen. Die FPGA-Hersteller bieten zu ihren FPGA-Chips ein Referenzdesign, welches von anderen Firmen zur Entwicklung eigener Development-Boards genutzt wird. Außer dem verwendeten FPGA-Chip sind hier die Größe und Art des zusätzlichen RAM-Speichers (z.Bsp. DDR3 oder DDR4) und die Schnittstellen (z.Bsp. VGA, HDMI, UART, I/O-Pins, PCI-Express, Ethernet) als Auswahlkriterien relevant. Viele Developmentboards werden für die Forschung und Lehre zu günstigeren Preisen (Akademische Preise) und inklusive Softwarelizenzen für die Hersteller-Werkzeuge angeboten.

2.6 FPGA-Programmierung

2.6.1 Arbeitsfluss

Der Arbeitsfluss einer FPGA-Entwicklung beinhaltet die in Abbildung 2.11 dargestellten Schritte. Ausgehend von einer Programmiersprache wird der Quellcode zuerst synthetisiert. Hierbei findet die Syntaxanalyse und eine Übersetzung in die weiter oben beschriebenen Strukturen und Bestandteile des FPGAs statt. In der Synthese wird, ähnlich einem Kompiliervorgang, der Quellcode für die Verwendung auf FPGAs optimiert. Das Ergebnis ist die Netzliste als ein (noch) FPGA-unabhängiges Zwischenprodukt, aus dem die erste Zuteilung der FPGA-Resourcen ersichtlich ist. Im nächsten Bearbeitungsschritt, dem Platzieren und Verbinden (Place and Route, oder auch Implementierung), wird die Platzierung der benötigten FPGA-Resourcen (Kapitel 2.5.2) auf dem FPGA-Chip festgelegt und werden die benötigten Verbindungsleitungen dazu definiert. Hierzu benötigt das Softwarewerkzeug die genauen Angaben des verwendeten FPGA-Chips und FPGA-Boards. Diese Informationen werden von den FPGA- und Board-Herstellern in Form von Board-Definition-Files bereitgestellt.

Nach allen bisher beschriebenen Schritten (Quellcode, Synthese, Place and Route) kann eine Simulation der Funktion oder des Timingverhaltens erfolgen. Die Simulation kann zur Fehleranalyse und Überprüfung des korrekten Verhaltens ohne die Verwendung der FPGA-Hardware benutzt werden. Da Place and Route für größere FPGA-Projekte mehrere Stunden bis Tage in Anspruch nehmen kann, ist die Simulation der Synthesee Ergebnisse eine Maßnahme, die den Arbeitsfluss beschleunigen kann.

Erst im letzten Schritt wird, nach erfolgreicher Synthese, Place and Route und Simulation, die Bitstream-Datei generiert, welche auf den FPGA geladen wird und die programmierte Funktionalität herstellt. FPGAs verlieren bei jedem Ausschalten ihre Programmierung und die Bitstream-Datei muss neu geladen werden. Hierzu werden SD-Karten oder interner Speicher im FPGA verwendet. Auf den FPGA-Boards sind Hardware-Steckbrücken (Jumper) zur Auswahl des Boot-Mediums verfügbar. Bei dem meisten FPGA-Herstellern ist das Bitstream-Format proprietär und nicht einsehbar, was die Entwicklung von Open-Source-Werkzeugen erschwert. Ein kleine Auswahl aktuell verfügbarer Open-Source-Werkzeuge ist in Kapitel 2.6.5 beschrieben.

Auf Hybrid-FPGAs kann zusätzlich zum Bitstream ein Linuxsystem installiert werden. Obwohl prinzipiell jede, für die Prozessor-Architektur passende, Linux-Distribution verwendet werden kann, bieten FPGA-Hersteller eigene Distributionen an. Diese sind mit ei-

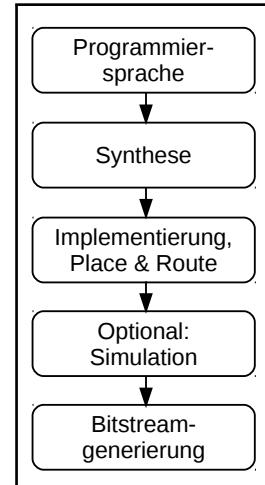


Abbildung 2.11:
FPGA-Workflow

genen Konfigurations- und Komplizier-Werkzeugen ausgestattet und bieten einen einfachen Einstieg in die Programmierung von FPGA-Hybrid-Plattformen. In diesen hersteller-eigenen Distributionen sind Kernelmodule und Treiber für die Kommunikation zwischen FPGA und Prozessor schon enthalten oder Referenzbeispiele zur Entwicklung angegeben.

2.6.2 Hardwarebeschreibungssprachen (HDL)

Die Entwicklung von FPGA-Quellcode findet in Hardware-Beschreibungs-Sprachen (Hardware-Description-Language, HDL) statt. Es haben sich die zwei Sprachen VHDL (Very High Speed Integrated Circuit Hardware Description Language) und Verilog etabliert. Diese Sprachen sind nicht durch die Entwicklung von FPGAs entstanden, sondern sind in der Entwicklung von ASICs schon lange im Einsatz. Mit HDLs werden das Verhalten und die Eigenschaften von Hardware-Bauteilen beschrieben. Daher ist, in HDL geschriebener, FPGA-Quellcode unter bestimmten Einschränkungen auch zum Prototyping und der Entwicklung von ASICs geeignet. Im Folgenden wird der Aufbau eines einfachen VHDL-Beispiels (Quellcode 2.1) besprochen. Die Sprache Verilog wird in dieser Arbeit nicht verwendet oder beschrieben.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4
5 entity Addierer is
6     Port ( a : in std_logic_vector (7 downto 0);
7             b : in std_logic_vector (7 downto 0);
8             result : out std_logic_vector (7 downto 0));
9 end Addierer;
10
11 architecture Behavioral of Addierer is
12 begin
13     result <= a + b;
14 end Behavioral;
```

Quellcode 2.1: VHDL-Quellcode eines Addierers

Nach dem Import zweier Standard-Bibliotheken für die Datentypdefinitionen und Zahlenaarithmetik (Zeilen 1-3) folgen die zwei, in VHDL als Grundgerüst mindestens notwendigen, Blöcke **entity** (Zeilen 5-9) und **architecture** (Zeilen 11-14). Im **entity**-Block sind die Ein- und Ausgangssignale (Ports) des Moduls definiert. Die Angaben **in** und **out** definieren die Signale als Ein- oder Ausgänge. Der Datentyp **std_logic_vector** (**7 downto 0**) stellt 8 * 1-Bit breite Datenleitungen dar, zusammengefasst zu einem Si-

gnal und mit dem Most-Significant-Bit (MSB) zuerst. Im `architecture`-Block ist die funktionale Definition des Moduls enthalten. In diesem Beispiel wird dem Ausgangssignal `result` das Ergebnis der Addition der zwei Eingangssignale `a` und `b` zugewiesen. Das Modul enthält keine Takt-Synchronisation oder Reset-Möglichkeit, da die Addition in einem Taktzyklus ausgeführt wird und weder Speicher noch Register enthalten sind. Im Anhang A.2 ist das aus der Synthese entstandene Logikschema des Addierers abgebildet. Es sind 8 LUTs für die bitweise Addition und zwei 4-fach Carry-Logikeinheiten zu erkennen. Die mit IBUF gekennzeichneten Einheiten sind Leitungstreiber für die Signale. In Abbildung 2.12 ist die in Kapitel 2.6.1 beschriebene Simulation (Erstellt mit Xilinx Vivado) und das Blockbild des Addierers mit den Ein- und Ausgangsports dargestellt. Die Simulation erfolgte auf Basis der Synthese des weiter oben beschriebenen VHDL-Quellcodes (Quellcode 2.1). Das Ergebnis der Addition liegt direkt nach Anlegen der Eingangssignale vor und ändert sich mit Änderung der Eingangssignale. Das Addierer-Modul ist somit ein asynchroner Schaltkreis und unabhängig vom Systemtakt.

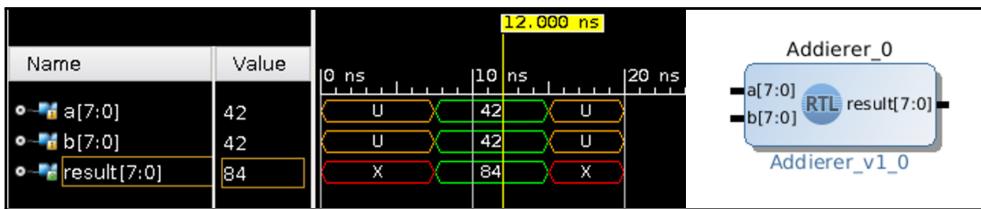


Abbildung 2.12: Simulation und Blockbild des synthetisierten Addierers in Xilinx Vivado

Es folgt eine grundlegende Auswahl von Programmierstrukturen, -methoden, -techniken und Besonderheiten in der Programmierung von VHDL für FPGAs:

Bibliotheken:

Das Institute of Electrical and Electronics Engineers (IEEE) hat Standard Bibliotheken für VHDL entwickelt, die in VHDL-Werkzeugen ohne weitere Installation zum Import zur Verfügung stehen [Ash08]. Der Import dieser Bibliotheken findet wie in Quellcode 2.1 (Zeilen 1-3) statt. Erst wird die zu verwendende Bibliothek referenziert, danach die Pakete und Elemente der Bibliothek eingebunden. Die Hierarchie von VHDL-Bibliotheken ist in drei Ebenen nach dem Schema `library_name.package_name.item_name` unterteilt. Der `item_name` kann durch die Angabe `all` ersetzt werden, um alle Elemente zu nutzen [Chu08]. In VHDL können eigene Packages erstellt werden.

Prozesse:

Prozesse sind, innerhalb von Architekturen angelegte, Blöcke und werden zueinander parallel ausgeführt. Die Ausführung innerhalb von Prozessen ist sequentiell und ist damit der Prozessor-basierten Softwareentwicklung ähnlich [MR13]. Anfang und Ende eines

Prozesses sind wie in Quellcode 2.2 zu definieren. Mit dem **begin**-Statement wird eine Sensitivitätsliste von Eingangssignalen angegeben, die im Prozess zu lesenden (z.Bsp. in bedingten Verzweigungen) Signale definiert. Es ist unabdingbar Sensitivitätslisten für die Simulation von VHDL-Code anzugeben [Ash08].

```

1 process (<Sensitivitätsliste>)
2 begin
3   # Prozess-Code mit Sensitivitätssignalen
4 end process;
```

Quellcode 2.2: VHDL-Quellcode eines Prozesses

Bedingte Verzweigungen:

Bedingte Verzweigungen sind in VHDL, wie in anderen Programmiersprachen auch, mit den Schlüsselwörtern **if**, **then**, **elsif**, **else** und **end if** definiert [MR13]. Ein Beispiel mit allen Schlüsselwörtern ist in Quellcode 2.3 angegeben.

```

1 if a < 10 then
2   b <= 0;
3 elsif a > 20 then
4   b <= 1;
5 else
6   b <= 2;
7 end if;
```

Quellcode 2.3: VHDL-Quellcode einer bedingten Verzweigung

Um unnötige Latenzen zu vermeiden, sollten die Verzweigungspfade vollständig abgedeckt sein. Bei nicht-vollständiger Abdeckung können Latches (Zwischenspeicher) entstehen und die Ausführung um bis zu mehrere Taktzyklen verzögern. Switch-Cases sind in VHDL möglich und sind wie in Quellcode 2.4 definiert [MR13]. Hier ist der, als letztes genannte, **when others**-Zweig zu beachten und wieder zur Vermeidung von Latches zu implementieren.

```

1 case a is
2   when 10 => b <= 0;
3   when 20 => b <= 1;
4   when others => b <= 2;
5 end case;
```

Quellcode 2.4: VHDL-Quellcode: Switch-Case

Schleifen:

Die Syntax für For-Schleifen ist in Quellcode 2.5 dargestellt [Ash08]. Die Start- und Stop-Werte der Schleife müssen zum Zeitpunkt der Synthese feststehen und dürfen nicht dynamisch sein. Auch While- und Loop/Exit-Schleifen sind in der VHDL-Syntax enthalten, werden aber hier nicht beschrieben.

```

1 for <variable> in <start> to <stop> loop
2   # Schleifen-code
3 end loop;
```

Quellcode 2.5: VHDL-Quellcode einer For-Schleife

Signale, Variablen und Zuweisungen:

Signale in VHDL sind wie Verbindungsleitungen zu verstehen. Die Port-Definitionen im **entity**-Block sind die Ein- und Ausgangsleitungen des VHDL-Moduls. Auch innerhalb eines VHDL-Moduls werden Signale wie Leitungen behandelt. Die Zuweisung von Werten auf Signale erfolgt in Prozessen erst am Ende des Prozess-Blocks [MR13]. Wenn also mehrere Zuweisungen innerhalb eines Prozess-Blocks auf dasselbe Signal erfolgen, wird nur die letzte Zuweisung aktiv. Dies kann bei Nichtbeachtung zu unerwarteten und fehlerhaften Ergebnissen führen. In VHDL können aber auch Variablen definiert werden. Variablen werden als Speicher ausgeführt, Zuweisungen auf Variablen sind sofort aktiv und können daher für sequentielle Strukturen in Prozessen verwendet werden [MR13]. Es ist zu beachten, dass die Verwendung von Variablen (und damit Speicher) zur Verzögerung des Ergebnisses (Taktzyklen) führen kann und Variablen nicht außerhalb des Blocks verwendet werden sollten, in welchem sie definiert wurden. Die Syntax für Definition und Zuweisung von Signalen und Variablen unterscheidet sich wie in Quellcode 2.6 beschrieben. Auch die Werte-Initialisierung in der Definition ist dargestellt.

```

1 signal a : std_logic := '1';    # Definition und Initialisierung eines
                                  Signals
2 variable b : std_logic := '1';  # Definition und Initialisierung einer
                                  Variablen
3 a <= '0';                      # Zuweisung in ein Signal
4 b := '0';                       # Zuweisung in eine Variable
```

Quellcode 2.6: VHDL-Quellcode: Zuweisungen zu Signalen und Variablen

Datentypen und Datenstrukturen:

In den Quellcode-Beispielen 2.1 und 2.6 sind bereits zwei Datentypen aus dem Package **std_logic_1164** der IEEE-Bibliothek aufgeführt. Der Typ **std_logic** stellt die kleinste logische Dateneinheit dar und kann außer den Werten '0' oder '1' (ein Bit) noch 7

weitere Werte (Zustände der Elektrotechnik) annehmen. Diese sind mit den Schlüsselsymbolen L, H, U, W, X, Z und - anzugeben und haben die Bedeutungen Low, High, Nicht initialisiert, Schwach, Nicht bestimmbar, Hochohmig und Dont care [Ash08]. Der Datentyp `std_logic_vector(<x> downto <y>)` stellt einen Vektor der Breite $|x - y|$ aus `std_logic`-Daten dar. Für VHDL-Datentypen können auch Wertebereiche mit dem Schlüsselwort `RANGE` angegeben werden. Dies hat die Vorteile der besseren Fehlererkennung in der Synthese (bei Zuweisungen außerhalb des Wertebereichs) und der Einsparung von Ressourcen durch Vermeidung nicht benötigter Leitungen und Logik. Datenstrukturen in Form von Arrays oder Records (ähnlich C-Structs) können in VHDL selbst definiert und verwendet werden.

Modularität (Komponenten):

Einmal definierte Module (wie z.Bsp. Quellcode 2.1) können in VHDL in anderen Modulen durch Instanziierung als Komponenten wiederverwendet werden [Ash08]. Auch die generische Erstellung und Verwendung von VHDL-Modulen ist möglich. Somit lassen sich VHDL-Projekte in hierarchische Strukturen einteilen und eigene Modul-Bibliotheken erstellen.

Takt-Synchronisation und Modul-Reset:

In Quellcode 2.1 ist ein asynchrones VHDL-Modul gezeigt und beschrieben. Üblicherweise sind FPGA-Schaltungen mit einem Taktsignal synchronisiert. Eine einfache VHDL-Umsetzung der Takt-Synchronisation von Prozessen ist in Quellcode 2.7 dargestellt [MR13].

```

1 process (clk, rst)
2 begin
3     if rising_edge(clk) then
4         if rst = '1' then
5             # Reset des Moduls
6         else
7             # Prozess-code
8         end if;
9     end if;
10 end process;
```

Quellcode 2.7: VHDL-Quellcode: Takt-Synchronisation

Der Eingangstakt (Signal `clk`) wird in der Prozess-Sensitivitätsliste aufgeführt und mit einer bedingten Verzweigung auf die steigende Signal-Flanke abgefragt (`rising_edge (clk)`). Immer wenn eine steigende Flanke erkannt wird, wird der Prozess getriggert.

Durch Synchronisation lassen sich Glitches (Fehler) in der Schaltung verhindern. Nach der `if`-Verzweigung der Takt-Synchronisation ist die Reset-Funktionalität für das Modul integriert. Auch das `rst`-Signal ist in der Sensitivitätsliste des Prozesses aufgeführt. In der Implementierung des Rasterisierers (Kapitel 5.4) ist ein VHDL-Modul für die Synchronisation der VGA-Schnittstelle enthalten, welches einige der beschriebenen VHDL-Strukturen verwendet.

2.6.3 High-Level-Synthese

Die im Folgenden verwendeten Begriffe zur Quellcode-Synthese und den entsprechenden Werkzeugen sind vom Hersteller Xilinx definiert, damit nicht herstellerunabhängig und teilweise als Marken von Xilinx geschützt. Die Grundlagen und Prinzipien sind bei anderen Herstellern ähnlich. Im Folgenden wird die Abkürzung HLS gleichbedeutend mit dem Xilinx Produkt VivadoHLS benutzt.

Die Synthese aus höheren Programmiersprachen (High-Level-Synthese, HLS) schafft eine weitere Abstraktionsschicht in der Programmierung von FPGAs. Die Abstraktionsidee von HLS ist die Trennung zwischen Software- und Hardwareprogrammierung für FPGAs schon in der Synthese. Die Funktionalität des Quellcodes wird mit einer höheren Programmiersprache abgebildet und die Definition der daraus entstehenden Hardware-Strukturen mit optionalen Parametern (Direktiven) festgelegt. Dies ermöglicht die Fokussierung auf die zu lösende Problemstellung in Strukturen der Softwareentwicklung für prozessorbasierte Systeme. Eine Bewertung, in welchen Problem-Szenarien diese Abstraktion zu einer Vereinfachung oder kürzeren Entwicklungszeit führt, ist Bestandteil der in Kapitel 1.2 definierten Ziele.

```

1 #include <ap_cint.h>
2 void addierer(uint8 a, uint8 b, uint8 *result) {
3     #pragma HLS INTERFACE ap_vld port=result
4     #pragma HLS INTERFACE ap_ctrl_none port=return
5     *result = a + b;
6 }
```

Quellcode 2.8: C-Quellcode eines Addierers (HLS synthesisierbar)

Zum Einstieg in die HLS-Thematik ist der synthesefähige C-Code (Quellcode 2.8) des Addierer-Beispiels aus Kapitel 2.6.2 gelistet. Als Erstes fällt auf, dass der C-Code kompakter als der VHDL-Code ist. Ohne die Pragma-Angaben besteht der C-Code nur aus einem Include, dem Funktionsprototypen und einer Ausführungscodezeile. Diese wenigen Zeilen weisen aber einige Besonderheiten auf, die im Folgenden beschrieben werden.

Die Funktion ist ohne Rückgabetyp als void-Funktion beschrieben. Die Rückgabe des Additionsergebnisses wird über den, als Pointer angelegten, Parameter `uint8 *result` realisiert. Die Rückgabe muss in C für VivadoHLS auf diese Art erfolgen. Der Datentyp `uint8` ist Teil der inkludierten `ap_cint.h`-Bibliothek für arbiträre Datentypen. Diese Bibliothek stellt Bit-genaue Integer-Datentypen bereit. Die Pragma-Angaben im C-Code sind HLS-spezifische Angaben und definieren die Umsetzung der Übergabeparameter in VHDL-Port-Signale und die Steuerleitungen des VHDL-Moduls. Der synthetisierte VHDL-Code ist im Anhang [A.1](#) gelistet. Wie in Kapitel [2.6.2](#) folgt eine grundlegende Auswahl von Programmierstrukturen, -methoden, -techniken und Besonderheiten in der Programmierung von C für die High-Level-Synthese:

Grundlegende Betrachtungen:

In der prozessorbasierten Ausführung von kompiliertem C-Code werden entweder die Inhalte (z.Bsp. Variablenwerte) oder Speicheradressen (Pointer) bei Aufruf einer Funktion auf den Stack kopiert. In FPGAs gibts es keinen Stackspeicher und auch keinen Heap-Speicher. Das ist einer der ersten Punkte, die ein Umdenken in der Programmierung von HLS-synthetisierbarem C-Code erfordern. Weiterhin ist die Größe der FPGA-Implementierung fest und nur durch erneutes Programmieren des FPGAs zu verändern. Es ist vorstellbar, das bestimmte Datenstrukturen (z.Bsp Verkettete Listen oder Bäume) in veränderlichen Größen (dynamisch) nicht für die Implementierung in FPGAs geeignet sind oder anders umgesetzt werden müssen, als bisher gewohnt. Daher stellt sich die Frage, wie diese prozessor- und speicherbasierten Paradigmen aus C-Code in Hardbeschreibungen umgewandelt werden können. Im Folgenden werden einige dieser Probleme und ihre HLS-spezifischen Lösungen beschrieben.

Keine dynamischen Strukturen oder Rekursionen:

HLS kann keine dynamischen Datenstrukturen oder Rekursionen synthetisieren [[Xil16b](#)]. Dynamische Datenstrukturen müssen in HLS als nicht-dynamische Datenstrukturen mit festen Größen angegeben werden und sind nicht zur Laufzeit veränderbar. Bäume, Listen oder eigene Strukturen mit veränderlichen Größen müssen, wenn keine nicht-dynamische Struktur verwendet werden kann, mit ihrer maximalen Größe angegeben werden und verbrauchen die FPGA-Ressourcen auf jeden Fall, unabhängig davon ob sie vollständig befüllt werden. So sind auch Arrays mit ihrer maximalen Größe zu definieren und können nicht darüber hinauswachsen [[Xil16b](#)]. Für Rekursionen muss die maximale Tiefe vorberechnet werden und die Rekursion mit dieser Tiefe in einer funktional identischen, iterativen Form implementiert werden. In Kapitel [2.5.4](#) wurde RAM (z.Bsp. DDR3) beschrieben, welcher auf FPGA-Development-Boards zusätzlich vorhanden sein kann. Es könnte die Idee entstehen, diesen RAM-Speicher für die oben beschriebenen Zwecke (dyn.

Strukturen) zu verwenden. Aber auch dieser Speicher ist nicht in einfacher Weise für dynamische Datenstrukturen im FPGA zu verwenden. Es fehlt hierfür eine Ressourcenverwaltung, welche in prozessorbasierten Systemen durch das Betriebssystem übernommen wird.

Keine Systemcalls:

Systemaufrufe (Systemcalls) können von HLS nicht synthetisiert werden [Xil16b]. Systemcalls (z.Bsp. `getc()`, `time()`, `sleep()`, `printf()`) sind an ein Betriebssystem gebunden und daher in FPGAs nicht möglich. Deshalb steht auch keine Speicherallokierung oder -freigabe (`malloc`, `free`) zur Verfügung, was die weiter oben beschriebenen Beschränkungen für Datenstrukturen begründet. Die Verwendung von Pointern ist in HLS möglich und wird in einem der folgenden Absätze beschrieben.

Toplevel-Funktion, Unterfunktionen:

Für die HLS-Synthese ist eine C-Top-Level-Funktion zu deklarieren [Xil16b]. Diese Top-Level-Funktion und ihre Funktionsparameter stellen die Definition für die Port-Ein- und Ausgangssignale des gesamten Moduls (wie das Blockbild des Addierers in Abbildung 2.12) dar. Für die Erstellung der Port-Signale sind weitere Angaben in Form von Direktiven möglich. Diese Direktiven werden in einem der folgenden Absätze beschrieben. Top-Level-Funktionen sind immer ohne Rückgabetyp als void-Funktionen auszuführen. Die Rückgabe von Parametern wird über Funktionsparameter realisiert. Das C-Projekt kann Unterfunktionen enthalten, für welche diese Einschränkung der Top-Level-Funktion nicht gilt. In Unterfunktionen können Rückgabetypen definiert werden. Unterfunktionen werden als eigene VHDL-Module synthetisiert und daher parallel ausgeführt. Wird eine sequentielle Ausführung der Unterfunktionen gewünscht (Ergebnis aus Unterfunktion 1 muss vorliegen, damit Unterfunktion 2 startet), ist dies nur über Direktiven zu steuern. Dieses Verhalten der Unterfunktionen ist in der Implementierung der Grafikpipeline und des Rasterisierers zu beachten und wird im Kapitel 5 mit den notwendigen Direktiven beschrieben.

Funktionsparameter:

HLS unterscheidet die Funktionsparameter in die drei Kategorien skalare Parameter, Arrays und Pointer. Diese Parameter können durch Direktiven in ihrer Umsetzung zu Ports signalen konfiguriert werden und sind an Steuerungsprotokolle und/oder AXI-Bussysteme gekoppelt. Eine Übersicht der möglichen Kombinationen ist im HLS-User-Guide [Xil16b] auf Seite 88 angegeben. Zum vollständigen Verständnis der Tabelle sind Kenntnisse über die verschiedenen AXI-Bus-Varianten notwendig, die nicht in dieser Arbeit beschrieben werden.

Die Funktionsrückgabewerte können über Pointer auf skalare Parameter oder Arrays realisiert werden. Die Standard-Einstellung von HLS, ohne weitere Direktiven, erstellt Eingabeports aus skalaren Funktionsparametern. Für die Bestimmung ob aus einem Pointer ein Ein- und/oder Ausgangsport erstellt werden soll, erkennt HLS Schreib- und Lesezugriffe auf den Pointer in der Funktion und erstellt die Ports nach den erkannten Zugriffen. Für jeden Parameter werden die zu erstellenden Steuersignale (enable, valid, ready) mit Direktiven angegeben. Eine Liste der in der Implementierung verwendeten Direktiven mit Beschreibung der Funktion ist im Anhang [A.6](#) gelistet.

Pointer sind in HLS nicht als Speicheradressen zu betrachten. Vielmehr sind Pointer in HLS nur eine Referenz auf Datenstrukturen, die außerhalb der Funktion erstellt und vorgehalten werden. Selbstdefinierte Datenstrukturen (C-Structs) werden von HLS in VHDL-Records umgewandelt und unterliegen nur den weiter oben beschriebenen Beschränkungen (nicht dynamisch), sind aber ansonsten wie C-Structs zu verwenden.

HLS-Bibliotheken: Datentypen und Arithmetik

HLS stellt eigene C-Bibliotheken für Datentypen und Arithmetik auf diese Datentypen bereit [[Xil16b](#)]. Die Bibliothek `ap_cint.h` ermöglicht die Verwendung von arbiträren Datentypen im Format `<u>int<1..1023>`. In der Synthese werden aus diesen arbiträren Datentypen genau die Anzahl der benötigten Leitungen erstellt und damit keine unnötigen FPGA-Resourcen verbraucht. Zu den arbiträren Datentypen bietet HLS auch entsprechende Arithmetik-Bibliotheken. Diese Bibliotheken haben von Standard-C-Bibliotheken abweichende Ergebnistypen-Eigenschaften. Für jede in den Bibliotheken enthaltene Berechnung ist der Ergebnistyp daher in der VivadoHLS-User-Guide [[Xil16b](#)] eigens angegeben. Ohne Beachtung dieser Angaben können Berechnungen in VivadoHLS unerwartete und andere Ergebnisse als mit Standard-C-Bibliotheken verursachen.

HLS-Direktiven:

HLS-Direktiven sind zusätzliche Angaben zum C-Quellcode, welche die Umsetzung des Quellcodes in FPGA-Hardware mitbestimmen. Direktiven werden in HLS über eine GUI (Graphical User Interface) eingegeben und in einer vom Quellcode separierten Datei gespeichert oder als Pragmas direkt in den C-Quellcode geschrieben. In HLS sind 20 verschiedene Direktiven verfügbar und viele davon sind parametrisierbar. In den Xilinx-Dokumentationen sind die Beschreibungen der Direktiven über mehrere Usertutorials verteilt und nicht einheitlich beschrieben. Die Suche und Anwendung der für das gewünschte Synthese-Ergebnis passenden Direktiven ist daher zeitaufwändig. Im Anhang dieser Arbeit ist eine Liste der in der Implementierung verwendeten Direktiven enthalten ([A.6](#)).

Zur Kontrolle der Wirkungsweise der Direktiven sind Resourcen- und Timing-Übersichten nach der Synthese in HLS einsehbar. Mit Hilfe dieser Übersichten können verschiedene

Direktiven schnell und innerhalb der HLS-Arbeitsumgebung überprüft und geändert werden.

Die Direktiven können in die zwei Gruppen Port-Direktiven und Struktur-Direktiven eingeteilt werden. Port-Direktiven bestimmen das Verhalten der Ein- und Ausgangssignale und Struktur-Direktiven bestimmen die Struktur des synthetisierten funktionalen VHDL-Codes. Direktiven können auch für beide Gruppen anwendbar sein und unterscheiden sich dann nur in ihrer Parametrisierung. Es folgen jeweils zwei kleine Beispiele für Port- und Struktur-Direktiven als Pragmas:

```
1 #pragma HLS INTERFACE ap_ctrl_none port=return
2 #pragma HLS RESOURCE variable=in core=RAM_1P_BRAM latency=2
```

Quellcode 2.9: HLS-Port-Direktiven

In Zeile 1 des Quellcodes 2.9 wirkt die Direktive **HLS_INTERFACE** durch den Parameter **port=return** auf die Ports des gesamten Moduls. Der Parameter **ap_ctrl_none** unterdrückt die Erstellung von Steuerungssignalen. Diese Pragma-Zeile gibt an, dass keine Steuersignale (enable, ready, valid) für das VHDL-Modul als Ganzes erstellt werden.

In Zeile 2 wirkt die Direktive **HLS_RESOURCE** auf das Portsignal **in** (welches als Funktionsparameter ein Array bezeichnet, z.Bsp. **uint8 in[10]**). Die Parameter **core=RAM_1P_BRAM** und **latency=2** geben die Art des zum Array zugehörigen Speichers (Single-Port BlockRAM) und die Latenz des Speichers (2) an. Es werden die entsprechenden Steuerleitungen für Speicheransteuerung und das Timingverhalten im Modul an diese Werte angepasst.

```
1 for (i = 0; i < 20; i++) {
2 #pragma HLS PIPELINE II=2
3     // Schleifencode
4 }
5 for (i = 0; i < 50; i++) {
6 #pragma HLS UNROLL
7     // Schleifencode
8 }
```

Quellcode 2.10: HLS-Struktur-Direktiven

In Zeile 2 des Quellcodes 2.10 wirkt die Direktive **HLS_PIPELINE** auf die For-Schleife aus Zeile 1 und HLS wird versuchen, diese Schleife zu einer Pipeline mit einem Iterationsintervall von 2 Taktzyklen zu synthetisieren.

In Zeile 6 wirkt die Direktive **HLS_UNROLL** auf die For-Schleife aus Zeile 5 und HLS wird versuchen, diese Schleife vollständig auszurollen und damit zu parallelisieren.

Ergbnis als gepackter IP-Core:

Das Ergebnis der High-Level-Synthese sind sowohl die VHDL- und Verilog-Dateien des synthetisierten C-Codes, als auch ein als zip-komprimiertes Archiv im Format eines IP-Cores. Das Konzept von IP-Cores wird im folgenden Kapitel beschrieben.

2.6.4 Das Konzept von IP-Cores

Intellectual Property Cores (IP-Cores) sind vorgefertigte, fertig-zur-Anwendung erstellte Hardware-Design-Blöcke. Die Namensgebung zeigt das Hauptmerkmal, das geistige Eigentum des Herstellers, an. IP-Cores sind als Softcores mit Quellcode oder als Hardcores ohne Quellcode (nur die synthetisierte Netzliste) erhältlich. FPGA-Hersteller haben eigene IP-Core-Bibliotheken, die mit herstellerspezifischen Lizenzmodellen belegt sind. Xilinx gibt mit der Lizenz der Vivado-Software-Suite eine kleine Zahl an IP-Cores zur Verwendung mit, hat aber weitere IP-Cores zur gesonderten Lizensierung im Angebot. Es gibt Open-Source IP-Core-Bibliotheken. Diese unterscheiden sich aber stark in der Qualität der Cores und deren Kontrolle auf korrekte Funktionalität. Anwender können sich eigene IP-Core-Bibliotheken ihrer Implementierungen erstellen und damit die Wiederverwendung in zukünftigen Projekten vereinfachen. Die mit VivadoHLS synthetisierten IP-Cores sind nicht lizenziert und es werden daher mit dieser Arbeit nur die C-Quellcodes oder selbstverfassten VHDL-Quellcodes der Implementierungen veröffentlicht.

2.6.5 Open-Source Werkzeuge

Aus den im letzten Absatz beschriebenen Gründen für die Nicht-Weitergabe der in dieser Arbeit mit HLS erstellen IP-Cores stellt sich die Frage nach Open-Source Werkzeugen für die Programmierung von FPGAs. Es gibt nur wenig FPGA-Open-Source-Werkzeuge. Die Entwicklung von FPGAs ist mit Patentrechten geschützt und FPGA-Hersteller halten die Formate ihrer Bitstream-Dateien geschlossen. Es gibt einzelne Projekte, die die Synthese von höheren Programmiersprachen zu VHDL anbieten, aber die Implementierung auf die FPGA-Chips als letztes Glied im Arbeitsfluss ist immer noch herstellerabhängig. Zum Zeitpunkt der Erstellung dieser Arbeit ist nur eine vollständige Open-Source-Werkzeugkette für FPGAs bekannt und im Folgenden kurz beschrieben:

Yosys: Die Yosys-Werkzeuge bestehen aus den Einzel-Produkten Yosys, Arachne-PnR und IceStorm für FPGAs des Herstellers Lattice (Yosys: [Wol]). Yosys ist das Synthesewerkzeug und verarbeitet Verilog zur Synthese in Netzlisten. Mit Arachne-PnR werden

die Netzlisten platziert und verbunden (Place and Route) und im letzten Schritt wird mit IceStorm die Bitstream-Datei generiert. Für die vollständige Implementierung und Erstellung des Bitstreams wurde hierfür die iCE40-FPGA-Serie des Herstellers Lattice reverse-engineered.

Die Yosys-Werkzeuge stellen keine High-Level-Synthese bereit. Es gibt aber auch Open-Source-HLS-Werkzeuge. Ein, zum Zeitpunkt der Erstellung dieser Arbeit, in Entwicklung befindliches Werkzeug ist SpinalHDL [Pap]. Damit könnte eine vollständig quelloffene Werkzeugkette von HLS bis zum Bitstream auf den FPGA genutzt werden. Diese könnte Bestandteil weiterer Arbeiten im Bereich der FPGA-Programmierung werden.

Kapitel 3

Analyse

3.1 Einleitung

Vor dem Design und der Implementierung steht die Analyse der Ziele, um die zu lösenden Problemstellungen zu spezifizieren. Im Folgenden werden die Ziele aus Kapitel 1.2 einzeln betrachtet, deren Problemstellungen herausgestellt und als Arbeitspakete mit Angabe der entsprechenden Lösungskapitel dargestellt. Die Festlegung auf die Entwicklung für FPGAs im Allgemeinen wird hierbei nicht weiter betrachtet, sondern genauso wie die Verwendung von VivadoHLS als Faktum angenommen.

3.2 Vektorgrafikeinheit

Für die zu entwickelnde Vektorgrafikeinheit sind Datenstrukturen (C-Structs) für einfache 3-dimensionale Objekte (Würfel, Pyramide) zu definieren. Diese Datenstrukturen bilden die Objekte mit ihren Eckpunkten, Kanten und Flächen ab. Es sind daher mehrere, aufeinander aufbauende Strukturen notwendig (Punkte, Vektoren, Flächen). Auf Basis dieser Objekt-Datenstrukturen werden Berechnungsmodule (C-Funktionen) erstellt, um die Objekte zu skalieren, zu verschieben und zu drehen. Diese Module ermöglichen den Aufbau von 3-dimensionalen Szenen als Abbildung in Datenstrukturen und erlauben die Animation dieser Objekte. In Kapitel 2.2 sind die mathematischen Grundlagen für das Design und die Implementierung dieser Datenstrukturen und Module beschrieben. Die Darstellung der 3-dimensionalen Szene soll auf einem 2-dimensionalen Bildschirm (hier ein Vektorbildschirm) erfolgen. Für die 2-dimensionale Darstellung gelten die Prinzipien der Projektion. Von den zwei gebräuchlichsten Projektionsarten, Parallel- und Zentralprojektion, kommt die Zentralprojektion die dem menschlichen Sehen am Nächsten. In

Zentralprojektionen werden entfernte 3-D-Objekte als 2-D-Objekte kleiner dargestellt als nahe Objekte. Dies führt zu einer perspektivischen 2-D-Darstellungen der Szene (Abbildung 2.3). Für die Umwandlung der 3-D-Szene in eine 2-D-Szene, unter Verwendung der Zentralprojektion, ist ein weiteres Modul zu erstellen (Projektionsmodul). In Abbildung 2.3 sind nur die vom Betrachter sichtbaren Flächen mit ihren Ecken und Kanten sichtbar. Die von den sichtbaren Flächen verdeckten Flächen sind nicht dargestellt. Um diese Art der Darstellung für die Vektorgrafikeinheit (Backface Culling, Kapitel 2.2) verwenden zu können, wird ein weiteres Modul (Backface-Culling-Modul) erstellt. Zusammen bilden diese Datenstrukturen und Module eine Vektorgrafik-Pipeline in welcher die beschriebenen Berechnungen in einer festgelegten Reihenfolge ausgeführt und am Ende der Pipeline als Ausgabedaten in 2-D zur Verfügung stehen. Diese Ausgabedaten werden von einer Zeichen-Funktion (eigenes Modul) auf die im Folgenden beschriebenen FPGA-Ausgänge geschrieben.

Die Darstellung der 2-D-Ausgabe der Vektorgrafikeinheit (VGE) bedarf eines Darstellungsmediums (Bildschirm). Um die Haupteigenschaft der Vektorgrafikeinheit, die homogene und kontinuierliche Darstellung von Vektoren (Linien zwischen zwei Punkten), nutzen zu können, wird ein Oszilloskop (Kapitel 2.1) als Vektorbildschirm verwendet. Wie im Grundlagenkapitel beschrieben, hat ein Oszilloskop im X-Y-Z-Betriebsmodus die Spannungseingänge X, Y und Z. Das FPGA-Board hat I/O-Pins (Input/Output) als Ausgänge. Es sind geeignete Schaltungen für die Verbindung zwischen den I/O-Ausgängen und den X-Y-Z-Eingängen zu entwickeln. Die Datenbreite der I/O-Ausgänge wird hierbei für die X- und Y-Verarbeitung auf 8 Bit beschränkt. Diese 8-Bit-Datenbreite bestimmt auch die Datentypen der Datenstrukturen und die Berechnungen innerhalb der Grafikmodule. Somit können Punkte auf dem Bildschirm in vertikaler und horizontaler Richtung (X-/Y-Achsen) 256 unterschiedliche Werte annehmen. Die Vektoren (Linien) zwischen den Punkten werden durch die in Kapitel 2.1 beschriebenen Eigenschaften von Vektorbildschirmen kontinuierlich gezeichnet. Da zwischen jeweils zwei, auf den X-Y-Ausgängen ausgegebenen, Punkten eine Linie gezogen wird, sind zwischen den getrennten Objekten einer Szene unerwünschte Linien zu sehen. Um die Zeichnung dieser unerwünschten Linien zu verhindern, wird der Helligkeitseingang (Z-Achse) des Oszilloskops mitbenutzt. Hierfür wird ein gesonderter I/O-Pin des FPGA-Boards verwendet. Die Ausgabe dieses Pins wird im letzten Teil der Grafik-Pipeline, dem Zeichen-Modul, mit angesteuert. Auch für den Z-Ausgang ist eine geeignete Schaltung für die Verbindung zwischen I/O-Pin und Z-Eingang zu entwickeln.

Die Gesamtanzahl der, für die Vektorgrafikeinheit benötigten, I/O-Pins ist für die Auswahl des FPGA-Boards mit entscheidend. Aus den obigen Beschreibungen stellt sich der Bedarf von 8 * I/O-Pins für den X-Ausgang, 8 * I/O-Pins für den Y-Ausgang und einem weiteren I/O-Pin für den Z-Ausgang, also insgesamt 17 I/O-Pins, dar. Diese 17 I/O-Pins

müssen auf dem FPGA-Board als Anschlüsse verfügbar sein. Weitere Auswahlkriterien sind in den folgenden Abschnitten definiert und führen zur Auswahl des FPGA-Boards in Kapitel 4.1.

Nr.	Aufgabenpaket	Lösung
A-V-1	Definition der Datenstrukturen für Würfel und Pyramiden	Kapitel 4.2.1
A-V-2	Design der Module Skalierung, Translation und Rotation	Kapitel 4.2.4
A-V-3	Design des Moduls Zentralprojektion	Kapitel 4.2.5
A-V-4	Design des Moduls Backface-Culling	Kapitel 4.2.5
A-V-5	Design des Moduls Zeichnen	Kapitel 4.2.6
A-V-6	Design der Vektorgrafik-Pipeline	Kapitel 4.2.7
A-V-7	Design der Schaltungen für I/O-zu-X-Y-Z-Ausgängen	Kapitel 4.2.8 und Kapitel 4.2.9

Tabelle 3.1: Aufgabenpakete der Vektorgrafikeinheit

3.3 Rasterisierer

Die Darstellung der Vektorgrafik aus der weiter oben beschriebenen Vektorgrafikeinheit basiert auf der Eigenschaft von Vektorbildschirmen, die Linien zwischen den X-Y-Punkten durch die Bewegung des Elektronenstrahls selbstständig zu zeichnen. Die Grafikeinheit gibt hierbei nur die X-Y-Punkte in einer vorgegebenen, zeitgesteuerten Reihenfolge auf die X-Y-Ausgänge aus und blendet die nicht erwünschten Linien mit dem Helligkeits-Ausgang (Z-Ausgang) aus. In der Darstellung der selben Grafik auf Pixelbildschirmen müssen diese Punkt-zu-Punkt-Linien mit ihren einzelnen Bildpunkten berechnet und ausgegeben werden. Nach dem in Kapitel 2.3 beschriebenen VGA-Protokoll ist jedem Bildpunkt ein genauer Zeitpunkt seiner RGB-Ausgabe in Abfolge eines Pixeltakts (Pixelclock) zugewiesen. Es müsste für die Echtzeit-Berechnung der Pixelinhalte bestimmt werden, ob dieser spezielle Pixel auf einer der Linien aller Objekte enthalten ist und diese Berechnung müsste wiederum in einem einzelnen Taktzyklus der Pixelclock erfolgen. Die Berechnung jedes Pixels genau zum Zeitpunkt seiner Ausgabe ist daher in dieser Implementierung nicht möglich. Für die Lösung dieses Problems gibt es das Konzept des Bildschirmspeichers. Bildschirmspeicher bildet jeden Bildpunkt in einer Speichereinheit ab und hält somit ein vollständiges Bild im Speicher vor. Dieser Bildschirmspeicher wird im Takt der Pixelclock ausgelesen und darauf synchronisiert auf dem VGA-Ausgang ausgegeben. Für diese Ausgabe auf Pixelbildschirme (über den VGA-Anschluss) sind Module

für die VGA-Pixelsynchronisation (VGA-Sync), der Bildschirmspeicher und das Auslesen und Ausgeben des Speichers zu designen und implementieren. Das Beschreiben des Bildschirmspeichers kann nicht zeitgleich mit dem Auslesen des selbigen erfolgen. Würden Schreiben und Lesen zum gleichen Zeitpunkt erfolgen, könnten Linien, welche noch nicht fertig gezeichnet sind, schon ausgelesen werden und zu falscher Darstellung des Bildes führen. Daher wird der Bildaufbau (Schreiben des Bildes) in einem zweiten Speicher (Bildschirm-Puffer-Speicher) vorgenommen. Während aus dem Bildschirmspeicher das Bild ausgelesen wird, kann im Puffer-Speicher das nächste Bild geschrieben werden. Wenn das gleichzeitige Lesen und Schreiben beendet ist, wird der Inhalt aus dem Pufferspeicher in den Bildschirmspeicher kopiert und der Puffer danach gelöscht. So entsteht ein Schreib-, Kopier- und Lesezyklus mit zwei Bildspeichern. Dieser zusätzliche Bildschirm-Puffer-Speicher und die Synchronisation der Schreib-, Kopier und Lesevorgänge ist zu designen und implementieren.

Für das Beschreiben des Bildschirm-Puffer-Speichers mit den Linien aller darzustellenden Objekte ist die Datenübergabe zwischen der Vektorgrafik- und der Rasterisierereinheit zu definieren. Die Objekte liegen in Datenstrukturen der Vektorgrafikeinheit vor und werden durch einen Zeichenalgorithmus in den Pufferspeicher geschrieben. Für das Zeichnen (Rasterisieren) der Linien wurde der Bresenham-Algorithmus ausgewählt. Dieser ist im Kapitel 2.4 beschrieben und wird im Design und der Implementierung des Rasterisierers verwendet. In der Auswahl des FPGA-Boards (Kapitel 4.1) wird der benötigte Bildschirm- und Pufferspeicher als Kriterium mit betrachtet.

Nr.	Aufgabenpaket	Lösung
A-R-1	Design des Moduls VGA-Sync	Kapitel 4.3.2
A-R-2	Design des Moduls Bildschirmspeicher	Kapitel 4.3.3
A-R-3	Design des Moduls Bildschirm-Puffer	Kapitel 4.3.3
A-R-4	Design des Moduls VGA-Lesen	Kapitel 4.3.4
A-R-5	Design des Moduls VGA-Kopieren	Kapitel 4.16
A-R-5	Design des Moduls VGA-Schreiben	Kapitel 4.3.6
A-R-6	Design der Synchronisation aller Module	Kapitel 4.3.7
A-R-7	Design der Datenübergabe von Vektor-einheit zu Rasterisierer	Kapitel 5.4.5

Tabelle 3.2: Aufgabenpakete des Rasterisierers

3.4 VivadoHLS-Bewertung

VivadoHLS führt eine weitere Abstraktionsebene zwischen Software- und Hardwareentwicklung für die Programmierung von FPGAs ein. Der funktionale C-Code bildet die Softwareentwicklung ab und die Applizierung von Direktiven (in Form von Pragmas) die Hardwareentwicklung (Kapitel 2.6.3). Die zwei gewählten Implementierungsziele stellen unterschiedliche Anforderungen an diese, von VivadoHLS vorgegebene, Abstraktion. Diese Unterschiede werden im Folgenden beschrieben und in den Kapiteln Implementierung (5) und Fazit (6) wieder aufgegriffen. Die Vektorgrafikeinheit ist durch die Verkettung von nacheinander abfolgenden Berechnungsmodulen ein in C-Unterfunktionen unterteiltes Design. Hier steht die Ein- und Ausgabe der Berechnungsergebnisse in einem voneinander abhängigen Datenpfad (Pipeline) im Vordergrund. Die Implementierung des C-Codes ist daher ähnlich zu reiner Softwareentwicklung und die anzuwendenden Direktiven bestimmen die Abhängigkeiten der C-Unterfunktionen, da diese (wie in Kapitel 2.6.3 beschrieben) in parallel ausführbare VHDL-Module synthetisiert werden. Außerdem sind die Ausgangssignale der Vektorgrafikeinheit auf I/O-Pins des FPGA-Boards herauszuführen.

Die Implementierung des Rasterisierers stellt ein anderes Anwendungsszenario für die Verwendung von VivadoHLS dar. Hier steht die taktgenaue Synchronisation von zwei Speichereinheiten (Bildschirm- und Puffer-Speicher) im Vordergrund. Die Entwicklung eines VGA-Synchronisations-Moduls für die taktgenaue Ausgabe nach der VGA-Protokoll-Definition (Kapitel 2.3) und die Synchronisation der Schreib-, Lese- und Kopiervorgänge zwischen den Speichern werden durch die Direktiven mitbestimmt. Die Entwicklung des C-Codes in Verbindung mit der Anwendung von Direktiven wird in den jeweiligen Implementierungskapiteln für die Bewertung in den Kriterien Zeitaufwand, Machbarkeit des gewünschten Ergebnisses und Qualität der Hersteller-Dokumentation betrachtet. Hier nicht vorhersehbare Problemstellungen werden gesondert aufgeführt.

Nr.	Aufgabenpaket	Lösung
A-H-1	Bewertung der Implementierung nach den beschriebenen Kriterien	Kapitel 6.2
A-H-2	Nicht in den Kriterien enthaltene Besonderheiten aufführen	Kapitel 6.2

Tabelle 3.3: Aufgabenpakete der Bewertung

3.5 Optionales Spiel oder Grafikdemo

Der Unterschied in der Implementierung eines Spiels oder einer Grafikdemo liegt in der Interaktivität eines Benutzers mit der Vektorgrafikeinheit. Eine Grafikdemo stellt eine nicht-interaktive Präsentation der Fähigkeiten der Vektorgrafikeinheit dar. Ein Spiel stellt auch eine solche Präsentation dar, bietet aber zusätzlich die Möglichkeit der Benutzerinteraktion, reagiert auf die Benutzereingaben und hat ein Spielziel (Gewinnen ist möglich). Es wird eine Grafikdemo implementiert. Die Entscheidung für die Grafikdemo ist erst am Ende der Implementierung der weiter oben beschriebenen Ziele getroffen worden. Wie in Kapitel 6 beschrieben, hat die Implementierung des Rasterisierers einige unerwartete Problemstellungen und Lösungsansätze erfordert. Die zeitliche Begrenzung dieser Arbeit stellt damit die Begründung für die Auswahl der Grafikdemo dar.

Für die Erstellung der Grafikdemo ist eine Ansteuerungs-Implementierung für die Vektorgrafikeinheit zu entwickeln. Für die Objekte und Berechnungsmodule der Vektorgrafikeinheit sind Steuerungsparameter zu definieren. Mit diesen Parametern werden die Objekte initialisiert und im 3-D-Raum verschoben, skaliert und gedreht. Zur Ansteuerung der Parameter ist eine Steuerungs-Programmierung zu designen. Diese Steuerung könnte mit VivadoHLS für den FPGA erfolgen oder auf einem, in Kapitel 2.5.3 beschriebenen, Hybrid-FPGA unter Linux implementiert werden. Die erste Variante würde voraussetzen, dass jede Änderung in der Steuerung den gesamten Arbeitsfluss der HLS-Programmierung mit allen Implementierungen durchläuft. Die zweite Variante (Linux auf Hybrid-FPGA) löst die Steuerungs-Implementierung von der FPGA-Programmierung, kann in C für das Linux-Hostsystem kompiliert werden und beschleunigt dadurch den Erstellungsprozess. Damit ist die Benutzung eines Hybrid-FPGA-Systems ein weiteres Kriterium für die in Kapitel 4.1 beschriebene Auswahl des FPGA-Boards.

Nr.	Aufgabenpaket	Lösung
A-D-1	Definition der Steuerungsparameter für die Vektorgrafik-Module	Kapitel 4.2.2
A-D-2	Design der Steuerungsprogrammierung und Inhalte der Grafikdemo	Kapitel 4.4
A-D-3	Auswahl des Linux-Hostsystems	Kapitel 4.4

Tabelle 3.4: Aufgabenpakte der Grafikdemo

Kapitel 4

Design

4.1 Auswahl des FPGA-Boards

Aus den Zielen dieser Arbeit und deren Analyse im Kapitel 3 ergeben sich Anforderungen an das auszuwählende FPGA-Development-Board. Die Anforderungen sind in der folgenden Liste zusammengefasst:

- I/O-Pins, Anzahl: mindestens 17 Stück
- Schnittstellen: VGA, SD-Karte
- Bildschirmspeicher (Im FPGA-Teil als BlockRAM)
- Speicher für Instanzen der 3-D-Objekte (Im FPGA-Teil als BlockRAM)
- Hybrid-FPGA-Board mit ARM-CPU (Linuxhost) für die Steuerung der Grafikdemo

Außerdem ist die Verwendung des VivadoHLS-Werkzeugs als HLS-Programmierumgebung festgelegt und die benötigten Xilinx-Werkzeuge (VivadoHLS, Vivado-Design-Suite) sind über die Hochschule RheinMain lizenziert und für diese Arbeit verfügbar. Daher findet die Auswahl des Hybrid-FPGA-Boards aus dem Sortiment des Herstellers Xilinx statt. Xilinx hat die Zynq-Chip-Serie, mit jeweils einem FPGA-Teil und zwei ARM-CPUs in einem Chip, im Produktpotfolio verfügbar. Die Zynq-Chips sind in unterschiedlichen Größen erhältlich und unterscheiden sich hierbei nur durch die Größe (Anzahl der in Kapitel 2.5) beschriebenen Ressourcen) des FPGA-Teils. Hiervon wird der kleinste Zynq-Chip ausgewählt. Die Herausforderung der Implementierung auf den kleinsten Zynq-Chip regt zur Sparsamkeit in der Datenhaltung an und verspricht eine intensive Auseinandersetzung mit der zu bewertenden Abstraktion zwischen Soft- und Hardwareprogrammierung in VivadoHLS.

Für die Zynq-Serie ist von Xilinx die Linux-Distribution Petalinux erhältlich und wird

zur Programmierung der Grafikdemo-Steuerung verwendet. Petalinux hat Treiber für den Datenaustausch zwischen ARM-CPUs und FPGA-Teil als einfach zu konfigurierende Bestandteile enthalten und eigene C-Applikationen können diese Treiber benutzen. Das gesamte Petalinux-System besitzt eine grafische Konfigurations-Oberfläche und Konsolen-Bauwerkzeuge.

Das ausgewählte Hybrid-FPGA-Board ZyBo des Herstellers Digilent beherbergt den Zynq-7010-Chip und besitzt die weiter oben genannten Anforderungen an Schnittstellen, I/O-Pins, Block-RAM-Speicher und ARM-CPUs. In Abbildung 4.1 ist ein ZyBo-Board mit Benennung der Bauteile und Angaben über die verfügbaren FPGA-Ressourcen dargestellt.

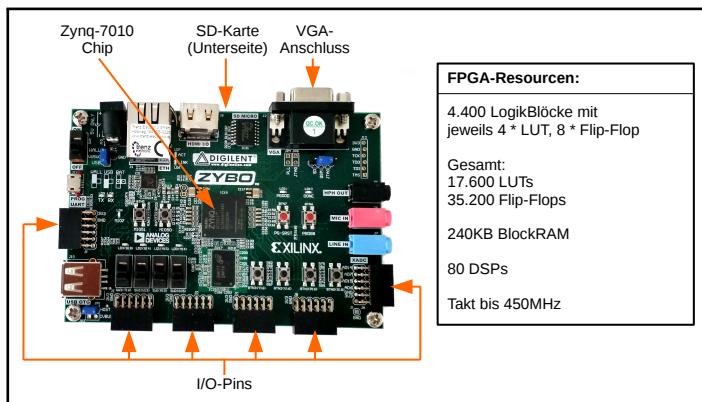


Abbildung 4.1: ZyBo-Board: Schnittstellen und Ressourcen

4.2 Vektorgrafikeinheit

4.2.1 Datenstrukturen

Für den Aufbau der Vektorgrafikeinheit sind die darzustellenden Objekte in Datenstrukturen abzubilden. Hierfür wird eine C-Header-Datei (Quellcode 4.1) angelegt, welche auf den in Tabelle 2.1 gelisteten mathematischen Strukturen aufbaut und C-Structs daraus definiert. Die Definitionen für Punkte (Zeilen 3-5, **Point**) und Vektoren (Zeilen 7-9, **vector**) enthalten jeweils ein Array mit 3 Werten für x,y und z. Die Definition einer Fläche (Zeile 11-13, **surface**) ist eine Andere als in der Tabelle 2.1. Flächen werden hier durch 4 Punkte und nicht durch ihre 2 aufspannenden Vektoren definiert. Diese Definition hat den Vorteil, dass alle Kanten der Fläche (die in der Darstellungsberechnung benötigt werden) direkt berechnet werden können. Sie hat aber auch die Einschränkung, dass die Definition der 3-D-Objekte nur planare (ebene) Flächen erstellen darf. 4 Punkte im Raum könnten auch eine nicht-planare Fläche darstellen. Diese Beschränkung wird in der Initialisierung der 3-D-Objekte berücksichtigt. Unter Verwendung dieser C-Structs wer-

den die 3-D-Objekte in mehreren Versionen definiert. Diese verschiedenen Versionen der Objekte sind für die zu erstellenden Pipeline-Module als Zwischenstufen der Berechnung notwendig. Es hätte auch eine einzelne Objektstruktur für alle Schritte der Berechnung erstellt werden können, aber die hier verwendeten Definitionen benötigen in der gesamten Implementierung weniger FPGA-Ressourcen.

```

1 #include <stdbool.h>
2
3 typedef struct point {
4     int xyz[3];
5 } Point;
6
7 typedef struct vector {
8     int xyz[3];
9 } Vector;
10
11 typedef struct surface {
12     Point points[4];
13 } Surface;
14
15 typedef struct obj_1 {
16     Point points[8];
17     Point middle;
18 } Obj_1;
19
20 typedef struct obj_s {
21     Surface surfaces[6];
22     Vector normals[6];
23 } Obj_s;
24
25 typedef struct obj_p {
26     Surface surfaces[6];
27     bool visible[6];
28 } Obj_p;
```

Quellcode 4.1: C-Header mit Datenstrukturen (C-Structs) für die Vektorgrafikeinheit

In den Zeilen 15-18 aus Quellcode 4.1 ist die erste Struktur (**obj_1**) für 3-D-Objekte definiert. Diese Struktur bildet ein würfelnähliches Objekt mit 8 Eckpunkten und einem Mittelpunkt ab und wird für die Transformations-Module als Ein- und Ausgabedaten verwendet. Das Objekt muss nicht genau einen Würfel abbilden, aber es sind die folgenden Beschränkungen für das Objekt einzuhalten:

- 8 Ecken, 6 Flächen und 12 Kanten

- Vollständig konvex
- Flächennormal-Vektoren zeigen nach außen
- Alle Flächen planar

Mit diesen Beschränkungen können zum Beispiel auch Spate oder Pyramidenstümpfe als Objekte modelliert werden. Die Indizierung der Ecken (Punkte), Kanten (Vektoren) und Flächen ist in Abbildung 4.2 dargestellt. Die Kanten-Vektoren werden erst innerhalb der Module berechnet, existieren nur temporär und sind zur Berechnung der Flächennormale definiert. In den Zeilen 20-23 ist das gleiche Objekt mit seinen 6 Flächen und den Flächennormal-Vektoren definiert (`obj_s`). Diese Struktur wird für die Bestimmung der Flächennormal-Vektoren verwendet und bildet damit die Basis für die letzte Strukturdefinition (`obj_p`, Zeilen 25-28), welche die 2-D-Koordinaten des Objekts enthält, das im Projektions-Modul berechnet wird.

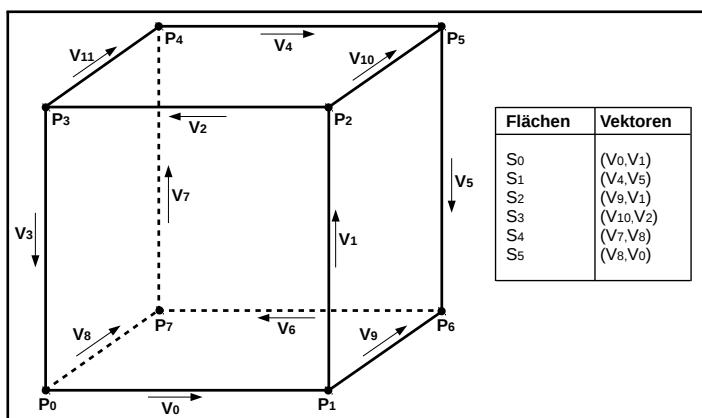


Abbildung 4.2: Index-Definitionen der Punkte, Vektoren und Flächen eines 3-D-Objekts

4.2.2 Objektsteuerung

Damit die 3-D-Objekte nicht statisch im Koordinatenraum stehen, ist eine Ansteuerung mit veränderlichen Parametern für jedes einzelne Objekt notwendig. Unabhängig von der noch zu wählenden Steuerungs-Programmierung werden die in Tabelle 4.1 gelisteten Parameter zur Ansteuerung definiert. Diese Parameter sind in den, im Folgenden beschriebenen, Modulen als Eingänge wiederzufinden.

Beschreibung	Parameter	Datenbreite
Mittelpunkt:	m_x, m_y, m_z	jeweils 8 Bit
Rotation:	r_x, r_y, r_z	jeweils 8 Bit
Größe:	<i>size</i>	8 Bit
Sichtbarkeit:	<i>visible</i>	1 Bit
Objektart	<i>category</i>	4 Bit

Tabelle 4.1: Parameter zur Objektsteuerung

Die Parameter für die Definition des Mittelpunkts, der Rotation und der Größe wirken auf die Transformationen der Objekte in den entsprechenden Berechnungsmodulen. Die Sichtbarkeits- und Objektkategorie-Parameter sind dazu Ausnahmen. Da die Anzahl der maximal darstellbaren Objekte durch die Programmierung des FPGAs vordefiniert sein muss (siehe Kapitel 2.6.2, dynamische Strukturen), wird durch den Parameter *visible* die Möglichkeit der Nicht-Darstellung eines Objekts erreicht. Für die Erstellung unterschiedlicher, würfelähnlicher Objekte (z.Bsp. Spate oder Pyramidenstümpfe) ist der Objektart-Parameter vorgesehen.

4.2.3 Objektinitialisierung

Aus den weiter oben beschriebenen Definitionen der Datenstrukturen werden 3-D-Objekte initialisiert. Die Objekt-Initialisierungen sind die ersten Module in der Grafik-Pipeline und erhalten die Parameter für Mittelpunkt, Größe, Sichtbarkeit als Eingänge. Der Objektart-Parameter bestimmt die Auswahl des Initialisierung-Moduls. Es werden für die verschiedenen Objektarten unterschiedliche Initialisierungsmodule erstellt. Die in Kapitel 4.2.1 beschriebenen Datenstrukturen sind für alle Objektarten zu verwenden. In Abbildung 4.3 sind zwei Initialisierungsmodule mit ihren Ein- und Ausgängen dargestellt. Außerhalb der Module instanzierte `obj_1`-Strukturen werden an die Module übergeben und von diesen unter Verwendung der Objekt-Parameter befüllt.

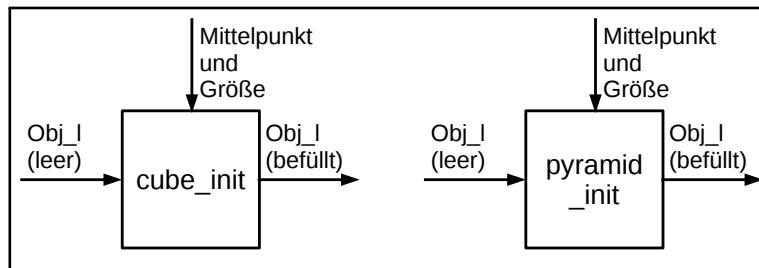


Abbildung 4.3: Module zur Initialisierung von Würfeln und Pyramidenstümpfen

4.2.4 Transformationsmodule

Die in Tabelle 2.3 angegebenen Transformations-Matrizen sind die Basis für die zu entwickelnden Transformations-Module. Die drei Rotationsberechnungen (X-, Y- und Z-Achsen-Rotationen) werden in einem Modul zusammengefasst. Für die Rotationen ist zu beachten, dass die Berechnungen relativ zu den entsprechenden Achsen des Koordinatensystems stattfinden. Ein zu rotierendes Objekt ist daher erst zum Koordinatenursprung zu verschieben (Translation), dann zu rotieren und danach wieder an seine vorherige Position zu verschieben. Hierfür wird die, in den Objektstrukturen `obj_1` enthaltene, Angabe

des Objekt-Mittelpunkts verwendet. In Abbildung 4.4 sind die Module und die Aufrufe der Untermodule dargestellt.

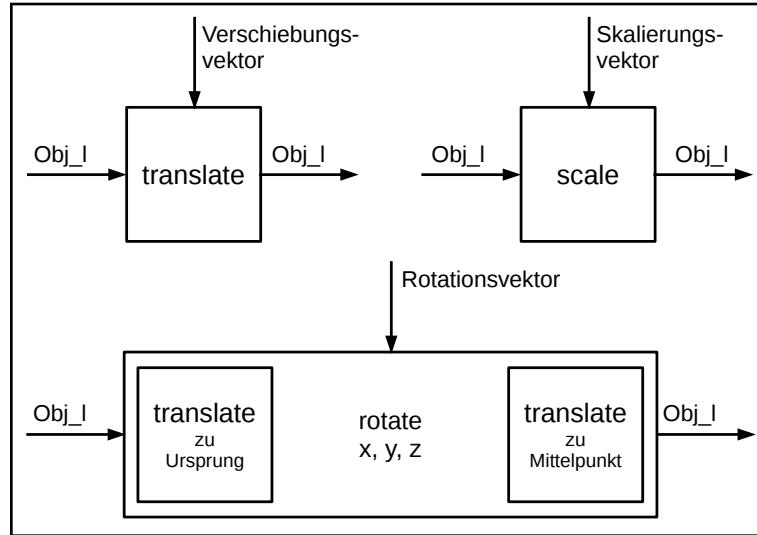


Abbildung 4.4: Module zur Transformation von Objekten

4.2.5 Backface-Culling und Projektion

In der Analyse in Kapitel 3.2 hat sich der Bedarf für zwei Module zur Berechnung von Backface-Culling und Projektion gezeigt. Diese Unterteilung ist bei Betrachtung der zu berechnenden Daten anders zu arrangieren. Für die Backface-Culling-Berechnung wird das Kreuzprodukt aus dem Flächennormal-Vektor und dem Aufsichtsvektor des Betrachters auf die Fläche gebildet und ausgewertet. Die Auswertung gibt dann an, ob die Fläche für den Betrachter sichtbar ist. Da in der Struktur `obj_1` keine Vektoren, sondern die Eckpunkte, enthalten sind, wird die Berechnung der Flächenvektoren (spannen die Fläche auf und geben eine Vorder- und Rückseite vor, Abbildung 4.2) zusammen mit der Berechnung der Flächennormal-Vektoren (zeigen aus der Vorderseite der Flächen heraus) in ein Modul zusammengeführt. Dieses Modul (`toSurface`) hat als Eingabe die Struktur `obj_1` und als Ausgabe die Struktur `obj_s`. Die als Zwischenprodukt notwendigen Flächenvektoren werden hierbei nur temporär gespeichert und nach der Berechnung und Speicherung der Flächennormal-Vektoren wieder verworfen.

Das zweite Modul wird als Projektions-Modul (`projection`) benannt und erhält die Struktur `obj_s` als Eingabe. Daraus wird das immer noch 3-dimensionale Objekt in ein 2-dimensionales Objekt mit Angabe der Sichtbarkeit gewandelt. Für die Berechnung der Sichtbarkeit ist der Flächenvektor jeder Fläche aus dem vorherigen Modul vorhanden und der Aufsichtsvektor wird aus dem Eingabe-Parameter `d` (Betrachterabstand, Kapitel 2.2) berechnet. Die Projektion ins 2-Dimensionale findet, wie in den vorherigen Modulen auch, auf Basis der Eckpunkte und der Projektionsmatrix statt. Die Ausgabe des

Projektions-Moduls besteht aus der, für die 2-D-Darstellung berechneten, Struktur `obj_p`. In Abbildung 4.5 sind die beiden Module mit ihren Ein- und Ausgaben dargestellt.

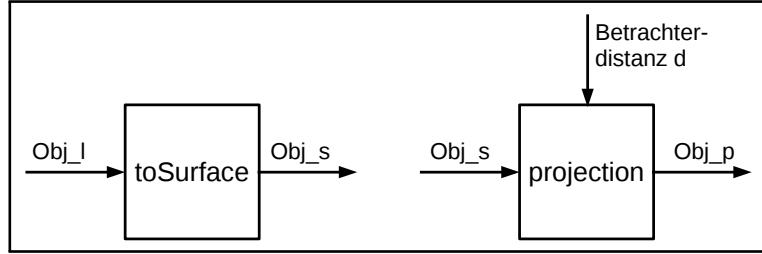


Abbildung 4.5: Module: toSurface und projection

4.2.6 Zeichnen

Das Zeichnen-Modul (draw) ist der letzte Schritt der Grafik-Pipeline. Es erhält als Eingabe die im Projektions-Modul berechneten 2-dimensionalen Objektdaten und generiert daraus die X-, Y- und Z-Werte zur Ausgabe auf die entsprechenden I/O-Pins. In diesem Modul werden die zu zeichnenden Objekt-Kanten mit ihren Start- und Endpunkten in Schleifen durchlaufen. Über die Sichtbarkeitsangabe der Flächen wird bestimmt, ob die Objektkante gezeichnet wird. Für jede Kante wird erst der Startpunkt (x_s, y_s) für eine festgelegte Zeitspanne (ca. 500 Taktzyklen bei 50MHz Taktfrequenz) ausgegeben und danach der Endpunkt (x_e, y_e) für die gleiche Dauer. Diese Zeitspanne zur konstanten Ausgabe der Punkte ist für die korrekte Darstellung der Linien notwendig. Werden die Punkte zu schnell nacheinander ausgegeben, kommt der Elektronenstrahl des Oszilloskops nicht schnell genug zum nächsten Punkt und es entstehen Darstellungsfehler (Rundungen an Ecken). Zwischen den zu zeichnenden Linien (Objektkanten) wird der Elektronenstrahl durch Ausgabe auf den Z-Ausgang in seiner Helligkeit reduziert und ist damit nicht sichtbar. Es wird ein zusätzlicher Eingang in das Modul definiert. Dieser Eingang kann die Z-Ausgabe invertieren und wird im Design des Z-Ausgangs weiter beschrieben. In Abbildung 4.6 ist das Schema dieses Moduls dargestellt.

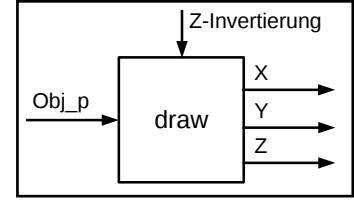


Abbildung 4.6: Modul: draw

4.2.7 Grafik-Pipeline

Aus den beschriebenen Berechnungs-Modulen wird durch Aufreihung der Module eine Grafik-Pipeline erstellt. Bisher wurden die Berechnungsschritte für jeweils ein einzelnes Grafik-Objekt betrachtet. Die Vektorgrafikeinheit soll aber mehrere Objekte einer 3-D-Szene berechnen und darstellen können. Für das Design der Grafik-Pipeline wurden daher zwei Ansätze entwickelt, die ganze 3-D-Szenen mit mehreren Objekten bearbeiten

können:

Designansatz 1:

Alle Objekte werden in Objekt-Listen instanziert. Zu den in Kapitel 4.2.1 definierten Strukturen kommen weitere Strukturen für die Objektlisten hinzu. Die Pipeline-Module werden auf die Verarbeitung von Objektlisten, statt von einzelnen Objekten umgestellt. Die Ein- und Ausgaben der Module sind dann Objektlisten und werden in jedem Schritt vollständig berechnet und ans nächste Modul weitergegeben. Am Ende der Pipeline wird die gesamte, berechnete 2-D-Objektliste im letzten Schritt dargestellt (draw).

Designansatz 2:

Es werden, wie im Modul-Design beschrieben, nur einzelne Objekte durch die Pipeline verarbeitet. Die ganze Pipeline wird in einer Schleife über die Anzahl der möglichen Objekte iteriert. Dabei wird am Anfang der Pipeline dieselbe, einzeln instanzierte Objekt-Datentruktur in jedem Schleifendurchlauf mit dem nächsten zu berechnenden und darzustellenden Objekt befüllt. Jedes Objekt durchläuft einzeln die gesamte Pipeline bis zur Darstellung (draw). Ist ein Objekt gezeichnet, startet die Pipeline mit dem nächsten Objekt.

Für die Auswahl des Designansatzes ist der Speicherbedarf im FPGA das Hauptkriterium. In Ansatz 1 wird Speicher (BlockRAM) für alle Objekte in allen Bearbeitungszuständen gleichzeitig benötigt. Für den Rasterisierer (Kapitel 4.3) werden auch noch BlockRAM-Ressourcen in der Größe von zwei vollständigen Bildschirmspeichern benötigt. Für die Entscheidung des Pipeline-Designs wurden Tests zur Zeichengeschwindigkeit (500 Taktzyklen an einem Punkt), die maximale Objektanzahl (6 Linien pro Objekt, ca. 600 Objekte ohne Pipeline) und den Ressourcenaufwand ($15 * \text{BlockRAM-18Kbit-Einheiten pro Objekt in der Pipeline}$) durchgeführt. Für jedes Objekt benötigt die Pipeline 15 BlockRAM-Einheiten mit jeweils 18Kbit Größe. Die Ressourcen-Übersicht aus VivadoHLS ist im Anhang A.3 abgebildet. Die Gesamtanzahl der im Zynq-7010 verfügbaren BlockRAM-Einheiten beträgt 120 Einheiten á 18Kbit. Mit weniger als 10 Objekten in den Objektlisten aus Ansatz 1 wäre der verfügbare BlockRAM-Speicher aufgebraucht. Damit würden keine weiteren BlockRAM-Einheiten für den Rasterisierer zur Verfügung stehen. Der Bedarf des Rasterisierers ist minimal mit $1024 * 768 \text{ Pixel} * 2 \text{ Speicher} = 1.573.864 \text{ Bit anzunehmen}$. Das sind über 80 benötigte BlockRAM-Einheiten der im Zynq-7010 verfügbaren 120 Einheiten. Daher ist für die Grafik-Pipeline der Designansatz 2 zu wählen. Weiterhin ergab sich aus den Tests für die Zeichengeschwindigkeit und maximaler Objektanzahl, dass die Anzahl der benötigten Taktzyklen für die fehlerfreie Darstellung eines Objekts hinreichend wenig ist, um die Objekte nacheinander durch die Pipeline zu berechnen und

trotzdem ein homogenes Bild (kein Flackern oder Bildzittern) dargestellt wird. Also sind Geschwindigkeit und Objektanzahl keine Ausschlussgründe für den Designansatz 2. Im Anhang ist ein Bild des Tests auf die maximal darstellbare Objektanzahl (ohne Pipeline, nur gleiche Objekte mit Mittelpunkt-Verschiebung) dargestellt (Anhang A.4). Im Folgenden wird die erweiterte Grafik-Pipeline unter Verwendung von Designansatz 2 beschrieben.

Vor Beginn der Pipeline-Schleife werden die Objektsteuerungs-Daten (Kapitel 4.2.2) in Arrays kopiert und aus diesen Arrays in jedem Schleifen-Iterationsschritt für die Initialisierung der erstellten Objekt-Instanz (Kapitel 4.3) verwendet. Abbildung 4.7 stellt diese, aus Abbildung 2.4 modifizierte, Grafik-Pipeline dar und ist im Anhang A.5 nochmals etwas größer abgebildet.

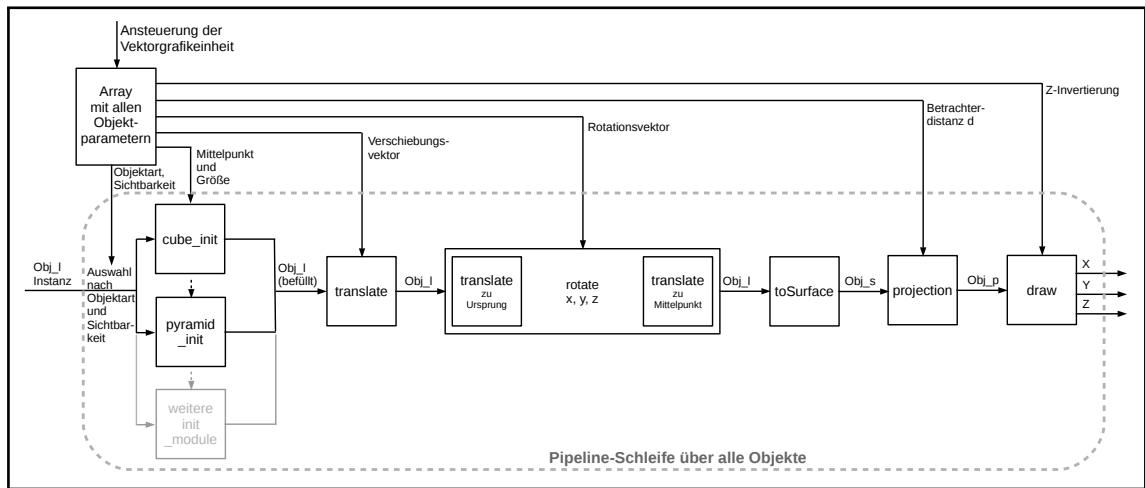


Abbildung 4.7: Grafik-Pipeline mit Objekt-Schleife

4.2.8 XY-Ausgänge

Die zwei 8-Bit breiten Datenausgänge X und Y werden durch die FPGA-Programmierung auf 2 * I/O-Pins geleitet und sind am Rand des FPGA-Boards als Anschlüsse herausgeführt. Die I/O-Pins bilden eine logische 0 als Groundpegel (0V) und eine logische 1 als Highpegel (3,3V) ab. Für die Ansteuerung der X- und Y-Eingänge des Oszilloskops werden analoge Spannungen benötigt. Die zu entwickelnden Schaltungen sind für beide Ausgänge identisch, daher wird im Folgenden nur die Schaltung für einen (X oder Y) Ausgang beschrieben. Die Schaltung besteht aus 2 nacheinander gereihten Stufen und ist in Abbildung 4.8 dargestellt. Die erste Stufe ist ein Digital-Analog-Wandler (DAC). Der DAC besteht hier aus einem vorgefertigten Modul der Firma Digilent, welches für die Verwendung mit Digilent FPGA-Boards (also auch dem ZyBo) vorgesehen ist (Abbildung 4.10).

Der Steck-Anschluss des DAC-Moduls ist im Format eines Pmod-Anschlusses gefertigt und passt daher ohne weitere Modifikation direkt auf einen 8-Bit breiten Pmod-Anschluss des ZyBo. Der DAC ist durch ein R2R-Widerstandsnetzwerk ausgeführt. Die Widerstandsschaltung des R2R-Netzwerks ist in Abbildung 4.8 gezeigt. Das R2R-Netzwerk ist ein rein passives elektronisches Bauteil, benötigt keine eigene Spannungsversorgung zum Betrieb und bietet eine sehr schnelle Wandlung. Die High-pegl (3,3V) der I/O-Pins werden durch kaskadierte Spannungsteiler zur analogen Ausgangsspannung aufaddiert. Der Ausgang des DAC ist an eine RC-Schaltung (Widerstands-Kondensator-Glied) gelegt. Das RC-Glied stellt eine einfache Variante eines Signalintegrators dar [Pau99]. In Abbildung 4.9 ist ein Vergleich zwischen

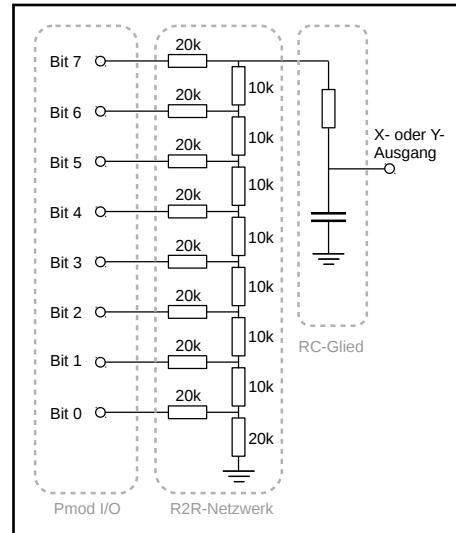


Abbildung 4.8: X- oder Y-Ausgang

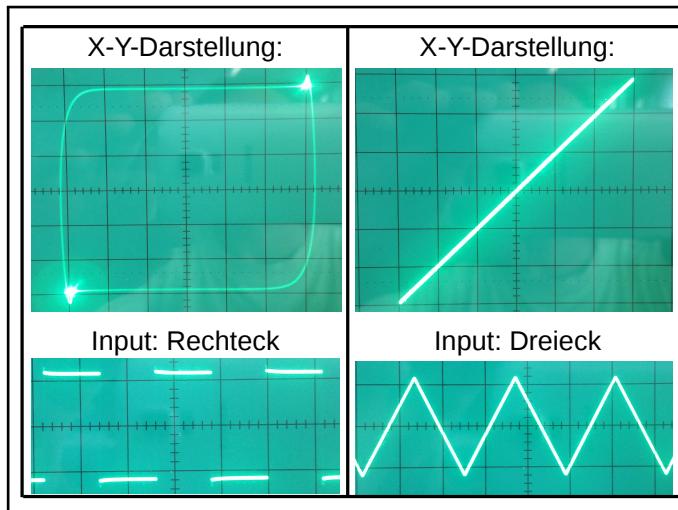


Abbildung 4.9: Vergleich: Rechteck- und Dreieckspannungen als X-Y-Eingänge

zwei verschiedenen Eingangssignalen für die X-Y-Darstellung auf einem Oszilloskop dargestellt. Im linken Teil der Abbildung ist ein Rechtecksignal an die X- und Y-Eingänge angelegt, im rechten Bildteil eine Dreieckspannung. Die Dreieckspannung führt zu einer geraden, schmalen Linie (Vektor) zwischen den zwei Endpunkten und ist die Darstellung, die erreicht werden soll. Die Rechteckspannung führt zu einer bauchigen, nicht diagonalen Darstellung zwischen den zwei Endpunkten. Die linke Darstellung hat ihren Grund in den schnellen Übergängen zwischen den zwei Signalpegeln. Der schnelle Pegelwechsel liegt über der Grenzfrequenz in welcher das Oszilloskop korrekte Darstellungen liefert. Die I/O-Pins des FPGA-Ausgangs werden genau wie im linken Bildteil sehr schnell geschaltet. Durch diesen Effekt ist die Darstellung diagonaler Vektoren ohne

eine Integrations-Schaltung fehlerhaft. Im zu verwendenden RC-Glied lädt sich der Kondensator bei Pegeländerung von Low zu High langsam auf und bei Pegeländerung von High zu Low entlädt sich der Kondensator wieder (Exponentielle Funktion). Diese Auf- und Entladung lässt die Spannungspegel langsamer steigen und sinken. Damit wandelt das RC-Glied die Rechteckspannungen der I/O-Pins (und damit auch des DAC) in Signale mit weniger steilen Signalflanken und die Darstellung von Diagonalen Vektoren sieht aus wie im rechten Bildteil. Zur Bestimmung der Werte für den Widerstand und den Kondensator im RC-Glied wurde eine Steck-Schaltung (Abbildung 4.10) gebaut und mittels Widerstands- und Kondensatorsortimenten passende Werte experimentell gefunden. Der Widerstand hat den Wert 47 Ohm und der Kondensator 220 pF. Um die Signale an das Oszilloskop zu übertragen, wurden BNC-Einbau-Steckverbinder verwendet. An Oszilloskopen sind die Eingänge auch mit BNC-Steckverbindern ausgeführt. Somit werden zum Anschluss der Vektorgrafikeinheit nur BNC-Kabel benötigt.

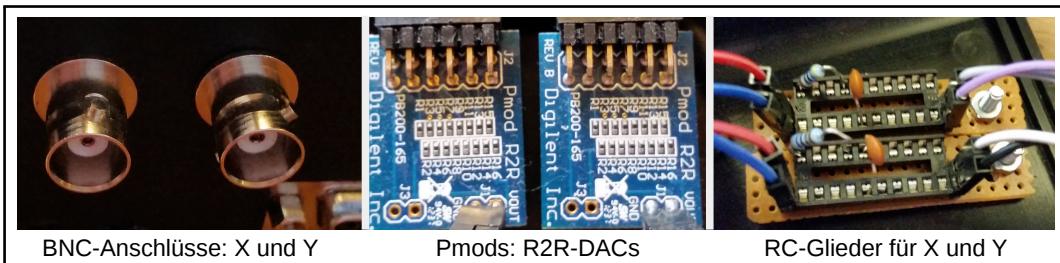


Abbildung 4.10: Schaltungsbestandteile der X- und Y-Ausgänge

4.2.9 Z-Ausgang

Der Z-Ausgang ist, wie die X- und Y-Ausgänge, durch eine zu entwickelnde Schaltung zum Oszilloskop zu führen. Die Bestandteile der Schaltung sind in Abbildung 4.11 abgebildet. Die Helligkeitssteuerung von Oszilloskopen findet durch angelegte Spannungen im Bereich 0-20V statt. Je nach Hersteller bedeuten 20V die vollständige Abdunklung oder maximale Helligkeit des Elektronenstrahls. Für diese Oszilloskop-abhängige Eigenschaft ist der in Kapitel 4.2.6 beschriebene Z-Invertierungs-Parameter vorgesehen. Die I/O-Pins des FPGA-Boards geben eine maximale Highpegel-Spannung von 3,3V aus. Die Schaltung für den Z-Ausgang muss diese Spannung bei möglichst schnellen Schaltzeiten (MHz-Bereich) auf 20V heben. Hierfür ist ein Komparator auf Basis eines Operationsverstärkers vorgesehen [KSW04]. Der Komparator wird mit einer Betriebsspannung von 20V versorgt. Diese wird durch ein vorgefertigtes StepUp-Modul bereitgestellt (Abbildung 4.12). Dieses StepUp-Modul wandelt die auf dem Board als Betriebsspannung vorhandenen 3,3V in 20V bei begrenzter Stromaufnahme (<50mA) um. Da der Oszilloskop-Z-Eingang hochohmig (47kOhm) ausgeführt ist, wird diese Stromaufnahme nicht über-

schritten. Der Komparator hat zwei Signaleingänge. An Eingang 2 wird durch einen Spannungsteiler ($2 * 10\text{k}\Omega$) eine Referenzspannung von 1,65V eingestellt. Eingang 1 ist mit dem Z-I/O-Pin des FPGA-Boards belegt. Liegt am Eingang 1 eine niedrigere Spannung (Pin ist auf Lowpegel) als die Referenzspannung an, wird der Ausgang des Komparators auf Ground (0V) gelegt. Wenn Eingang 1 auf Highpegel (3,3V) liegt, wird der Ausgang des Komparators mit seiner Versorgungsspannung von 20V beschaltet. Somit vergleicht der Komparator die Spannung an Eingang 1 mit der Referenzspannung an Eingang 2 und schaltet den Ausgang entsprechend zwischen 0V und 20V um.

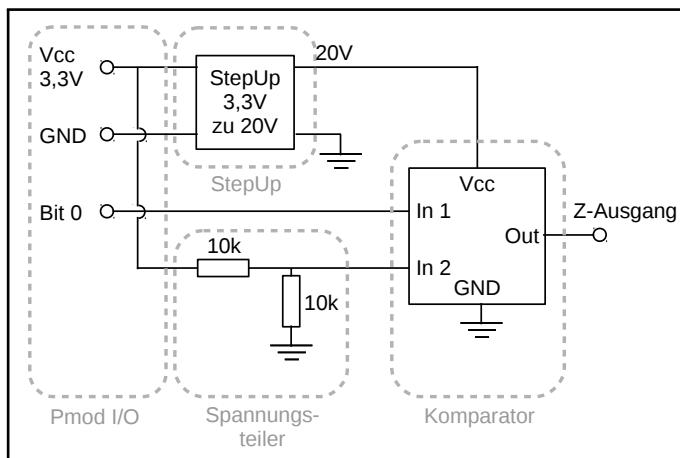


Abbildung 4.11: Schaltungsbestandteile des Z-Ausgangs

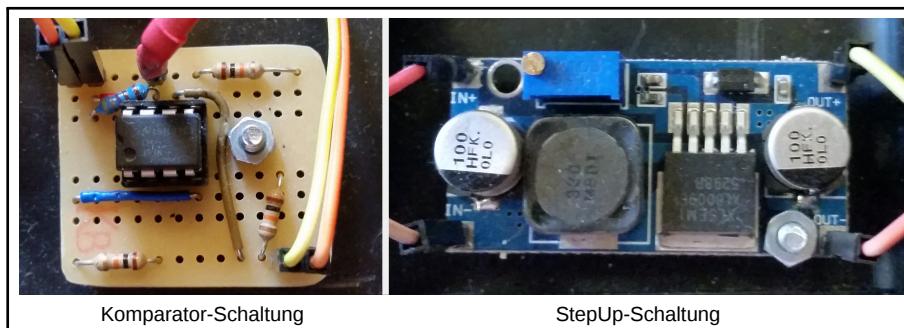


Abbildung 4.12: Komparator und StepUp-Modul des Z-Ausgangs

4.3 Rasterisierer

4.3.1 Designansatz

Während der Fokus des Vektorgrafik-Designs auf der Berechnung über Datenstrukturen und deren voneinander abhängige Abfolge als Grafik-Pipeline lag, steht für das Design des Rasterisierers die Synchronisation im Vordergrund. In der Ansteuerung von Pixelbildschirmen über die VGA-Schnittstelle hat jeder Pixel einen eigenen Zeitslot in

der Übertragung der RGB-Werte. Die VGA-Quelle (hier das FPGA-Board) gibt die Synchronisations-Signale (Tabelle 2.4) vor und der Bildschirm erkennt diese Timings und ordnet danach die übertragenen RGB-Werte dem dazu gehörigen Pixel zu. Als Basis dieser Synchronisation dient ein, im FPGA generierter, Pixeltakt (Pixelclock: 65MHz für 1024*768 Pixel mit 60Hz Wiederholfrequenz). Alle für den Rasterisierer zu entwickelnden Module sind auf diesen Pixeltakt und daraus generierte Horizontal- und Vertikalzähler synchronisiert. Wenn die Ausgabe der RGB-Werte nicht genau dem Pixeltakt folgt, führt dies zu fehlerhaften Darstellungen auf dem Bildschirm oder sogar zu Synchronisationsverlust des Bildschirms (Bild bleibt Schwarz). Hardwarebeschreibungssprachen (VHDL oder Verilog) sind für die Entwicklung taktgenau synchronisierter FPGA-implementierung sehr gut geeignet. Die Verwendung von VivadoHLS für diese Aufgabe wird zeigen, wie die HLS-Abstraktion mit C-Code und Direktiven hierfür geeignet ist. Im Folgenden werden erst die Designs der Module beschrieben und danach die Synchronisation dieser Module betrachtet.

4.3.2 VGA-Generator

Das VGA-Generator-Modul (Abbildung 4.13) erzeugt aus dem eingehenden Pixeltakt (65MHz) zwei Zähler (h-count und v-count) für die Bestimmung, an welchem Bildschirmpixel sich die VGA-Übertragung gerade befindet. Auf Basis dieser Zähler werden die H- und V-Synchronisations-Signale für den VGA-Ausgang generiert. Das VGA-Protokoll durchläuft mit den h- und v-Zählern immer eine komplette Bildzeile, springt dann an den Anfang der nächsten Zeile (Abbildung 4.14). Ist die letzte Zeile erreicht, werden beide Zähler zurückgesetzt und der Aufbau des nächsten Bildes beginnt. Wie in der Abbildung zu erkennen, gibt es sichtbare und nicht sichtbare Bildbereiche. Am Ende jeder Zeile und am Ende des Bildes befinden sich nicht sichtbare Bereiche. Für die weiter unten (Kapitel 4.3.7) beschriebene Synchronisation der Module werden unter anderem die nicht sichtbaren Bildbereiche für das Schreiben und das Kopieren von Bilddaten genutzt.

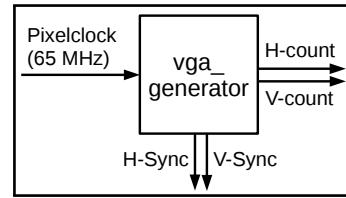


Abbildung 4.13: Modul: VGA-Generator

4.3.3 Bildschirmspeicher und Bildschirm-Pufferspeicher

Wie in Kapitel 3.2 beschrieben, werden zwei Bildschirmspeicher verwendet. Der eine Bildschirmspeicher ist zum Auslesen und Ausgeben der RGB-Werte für die VGA-Übertragung vorgesehen. Der anderer Speicher wird als Bildschirm-Pufferspeicher zum

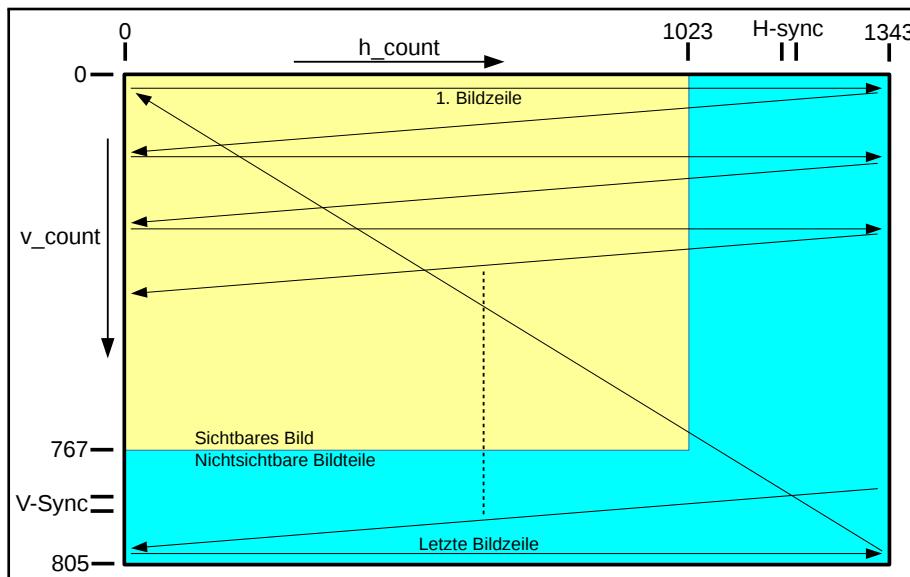


Abbildung 4.14: VGA-Generator: Horizontale und vertikale Bildzähler

Schreiben des nächsten Bildes verwendet. Beide Speicher sind in der Größe so zu erstellen, das jeweils ein vollständiger Bildinhalt darin abgebildet werden kann. Da die Darstellung der Vektorgrafikeinheit keine Farben beinhaltet, ist ein Bit als Speicher für einen Pixel ausreichend. Die Größe berechnet sich wie folgt:

$$1.024 * 768 \text{ Pixel} = 786.432 \text{ Pixel} = 786.432 \text{ Bit}$$

In VivadoHLS wird für den Bildschirmspeicher wieder eine passende Datenstruktur in C definiert. In VivadoHLS ist die Zuordnung von Arrays als Ein- und Ausgabe-Funktionsparameter zu BlockRAM als Standardeinstellung vorgegeben. Als Datentyp für die Bildschirmspeicher-Arrays wird **uint32** festgelegt. Damit hat ein Array die Größe:

$$786.432 \text{ Bit}/32 \text{ Bit} = 24576$$

Die vollständige Definition eines Bildschirmspeicher-Arrays ist:

```
1 uint32 vgaram[24576];
```

Quellcode 4.2: Bildschirmspeicher-Array in C

Die Größe in 18Kbit-BlockRAM-Einheiten ist:

$$786.432 \text{ Bit}/18 \text{ Kbit} = 43 \text{ BlockRAM-Einheiten (aufgerundet)}$$

Somit werden für die beiden Bildschirmspeicher (vgaram, vgabuffer) insgesamt 86 BlockRAM-Einheiten verwendet. Diese sind als Dual-Port-BlockRAM zu erstellen, da-

mit die Lese- und Schreibzugriffe aus unterschiedlichen Modulen gleichzeitig erfolgen können. Im FPGA-Teil des ZyBo sind insgesamt 120 BlockRAM-Einheiten verfügbar. Und wie im Kapitel 4.2.7 beschrieben, werden für die Vektorgrafikeinheit mindestens 15 BlockRAM-Einheiten benötigt. Damit ergibt sich der rechnerische Bedarf an BlockRAM insgesamt mit 101 Einheiten.

4.3.4 Lesemodul

Das Lesemodul ist die Verbindung zwischen dem Bildschirmspeicher und dem VGA-Ausgang. Das Modul hat die Zähler H-count, V-count und einen Pointer auf das Bildschirmspeicherarray als Eingänge (Abbildung 4.15). Die Rot-, Grün- und Blausignale für den VGA-Anschluss sind die Ausgänge. Sind die Zähler in den Bereichen des sichtbaren Bildes ($H\text{-count} < 1024$, $V\text{-count} < 768$) werden die dem jeweiligen Pixel entsprechenden RGB-Werte ausgegeben. Da die Darstellung keine Farben enthält, ist die Ausgabe auf zwei RGB-Werte-Tripel beschränkt. Ein schwarzer Pixel wird mit $R = 0, G = 0, B = 0$ und ein weißer Pixel mit $R = \text{max}, G = \text{max}, B = \text{max}$ ausgegeben. Der im ZyBo eingebaute VGA-Anschluss hat Datenbreiten von 5 Bit für Rot, Blau (max. Integer-Wert = 31) und 6 Bit für Grün (max. Integer-Wert = 63). Der Wert eines Pixels im Speicher ist in einem 32 Bit-breiten Datenworten durch Bitmasken zu ermitteln. Da der Lesevorgang eines `uint32`-Wertes aus dem Bildschirmspeicher nicht latenzfrei erfolgen kann (Mind. 1 Taktzyklus Latenz), wird die jeweils nächste Bildzeile vollständig in ein Register geladen und von dort zur Ausgabe der RGB-Werte gelesen. Der Zeitpunkt des Zeilenkopievorgangs ist innerhalb des nicht sichtbaren Bereichs, am Ende der aktuellen Zeile (Abbildung 4.14), definiert. In Kapitel 4.3.7 sind die Schreib-, Kopier- und Lesevorgänge in einer Übersicht zusammengefasst.

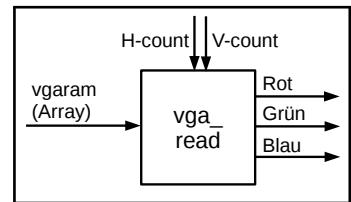


Abbildung 4.15: Modul: VGA-Lesen

4.3.5 Kopiermodul

Das Kopiermodul kopiert den Bildschirm-Pufferspeicher in den Bildschirmspeicher. Die Eingänge des Moduls sind wieder die Zähler H-count und V-count um den Zeitpunkt des Kopievorgangs im Modul zu bestimmen. Das Bildschirm-Pufferarray `vgabuf` ist ein weiterer Eingang und das Bildschirm-Speicherarray `vgaram` ein Ausgang. Beide Arrays werden als Pointer übergeben und referenzieren damit auf die in Kapitel 4.3.3 beschriebenen BlockRAM-Speicher. In Abbildung 4.16 sind

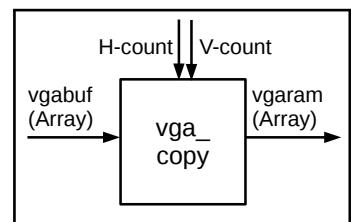


Abbildung 4.16: Modul: VGA-Kopieren

die Ein- und Ausgänge des Moduls dargestellt. Der Kopierzeitpunkt wird zum Ende der Bilddarstellung im nichtsichtbaren Bildbereich definiert ($V\text{-count} > 767$) und ist mit der gesamten Rasterisierer-Synchronisation in Kapitel 4.3.7 beschrieben.

4.3.6 Schreibmodul

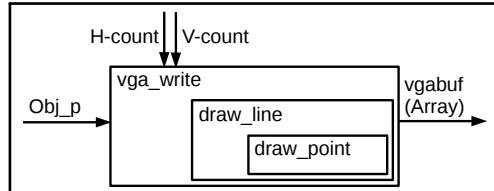


Abbildung 4.17: Modul: VGA-Schreiben

Das Schreibmodul hat die, aus der Vektorgrafik-Pipeline ausgehenden, 2-D-Objektstrukturen (`obj_p`) als Eingang und schreibt auf das Bildschirm-Pufferspeicherarray. Wie die anderen Rasterisierermodule, erhält das Schreibmodul die Zähler H-count und V-count zur Bestimmung des Schreib-Zeitpunkts. Aus der Objektstruktur werden alle zu zeichnenden Linien (Objektkanten) in einer Schleife iteriert und mit einer Implementierung des Bresenham-Algorithmus in den Pufferspeicher geschrieben. Vor dem Beschreiben des Pufferspeichers mit dem aktuellen Bild, wird der Pufferspeicher (noch mit dem letzten Bild befüllt) gelöscht. Es werden Untermodule entwickelt und aus dem Schreibmodul aufgerufen. Ein Untermodul schreibt einzelne Bildpunkte (`write_point`) in den Pufferspeicher, ein anderes Untermodul schreibt unter Verwendung des ersten Untermoduls ganze Linien (Bresenham-Algorithmus) in den Pufferspeicher (`draw_line`). Das Schreibmodul hat somit zwei Untermodule in einer, wie in Abbildung 4.17 dargestellten, hierarchischen Struktur.

4.3.7 Synchronisation der Module

Wie im dem Design der einzelnen Module zu erkennen ist, wird das VGA-Generator-Modul mit der Ausgabe der Zähler H-count und V-count als Basis für die Synchronisation des gesamten Rasterisierers verwendet. Die beiden Zähler sind in jedem Modul als Eingang definiert und jedem Modul werden Zeitfenster zur Bearbeitung der definierten Aufgaben zugewiesen. Abbildung 4.18 ist eine Modifikation von Abbildung 4.14 und zeigt die zeitlichen Bezüge der Module und der definierten Aufgaben zueinander und zu den Zählern H-count und V-count.

Lesemodul (vgaread):

Das Lesemodul liest am Ende jeder sichtbaren Zeile ($V\text{-count} \leq 767$ und $1024 \leq H\text{-count} \leq 1343$) die nächste Bildzeile ($V\text{-count} + 1$) in ein Zeilenregister ein. Die erste Bildzeile wird am Ende des Bildes in das Zeilenregister geschrieben. Für jede Zeile werden 32 Werte vom Typ `uint32` gelesen und im Register abgelegt. Aus dem Zeilenregister

wird der Wert des aktuellen Pixels ($V\text{-count} \leq 767$ und $H\text{-count} \leq 1023$) bestimmt und auf die RGB-Signale ausgegeben.

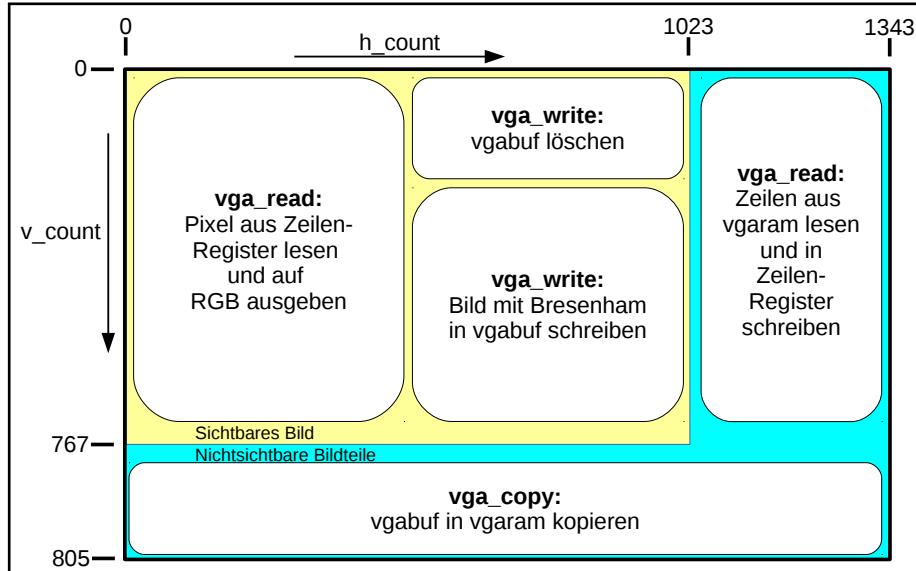


Abbildung 4.18: Synchronisation der Rasterisierer-Module auf die Zähler H-count und V-count

Kopiermodul (vga_copy):

Das Kopiermodul kopiert am Ende des Bildes im nichtsichtbaren Bildbereich ($768 \leq V\text{-count} \leq 805$) die Inhalte vom Bildschirm-Pufferspeicher in den Bildschirmspeicher. Beide Arrays sind 24.576 Werte groß. Der Kopiervorgang eines einzelnen Wertes dauert einen Taktzyklus und der gesamte Kopiervorgang 24.576 Taktzyklen. Das entspricht weniger als 19 Zeilen ($24576/1344 = 18,28$). Der nichtsichtbare Bereich unterhalb des Bildes enthält 38 Zeilen ($805 - 767 = 38$). Das Zeitfenster ist damit ausreichend groß für den gesamten Kopiervorgang.

Schreibmodul (vga_write):

Bevor das nächste Bild in den Bildschirm-Pufferspeicher (vgabuf) geschrieben wird, löscht das Schreibmodul diesen Pufferspeicher. Die Bilder würden sich sonst überschreiben, jeder jemals geschriebene Bildpunkt erhalten bleiben und die Darstellung animierter Szenen unmöglich werden. Der Löschkvorgang des Pufferspeichers findet an Anfang jedes Bildes statt ($V\text{-count} \leq 30$) und dauert genauso lange wie das Kopieren der Speicher (19 Bildzeilen). Erst danach wird das nächste Bild in den leeren Bildschirm-Pufferspeicher geschrieben ($100 \leq V\text{-count} \leq 767$).

Alle Module:

In Abbildung 4.19 sind alle beschriebenen Module in einem Blockdiagramm zusammen-

gefasst und die Beziehungen der Ein- und Ausgänge dargestellt. Die Eingänge des Rasterisierers sind die Pixelclock und die Objekt-Datenstrukturen aus der Vektorgrafikeinheit. Alle Ausgänge sind direkt mit den Signalen des VGA-Anschlusses verbunden. Der Rasterisierer hat keine weiteren Steuersignale als Eingänge.

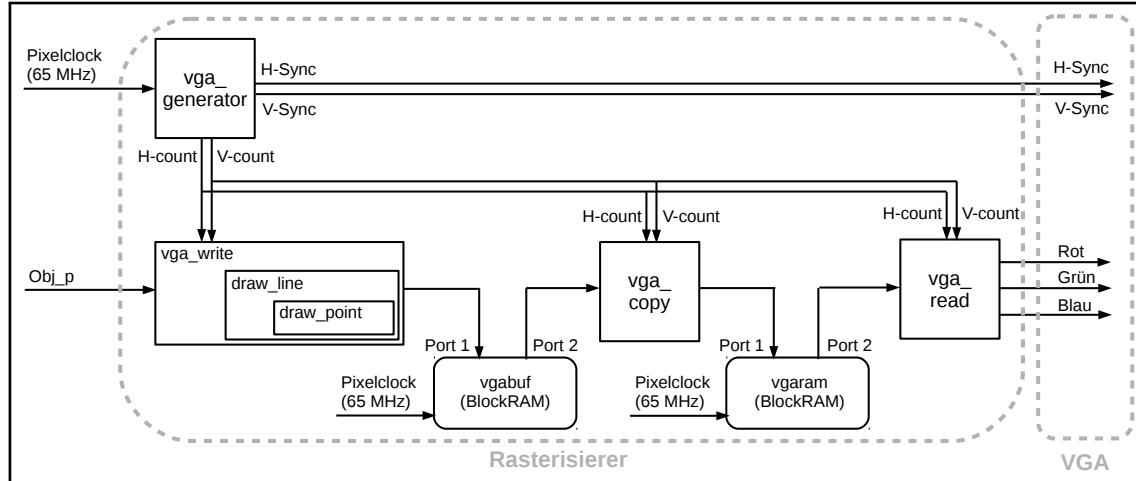


Abbildung 4.19: Rasterisierer: Alle Module

4.4 Grafikdemo

Die Grafikdemo zeigt die Möglichkeiten der Vektorgrafikeinheit (VGE) und des Rasterisierers. In Abbildung 4.20 ist das Schema eines Hybrid-FPGA-Chips mit den Implementierungsteilen dargestellt. Die Grafikdemo wird in einem Linux-System auf der ARM-CPU ausgeführt und steuert über einen AXI-Bus die Objektparameter der VGE.

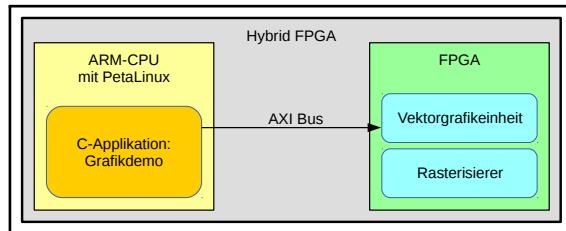


Abbildung 4.20: Schema des Hybrid-FPGA mit Grafikdemo-Applikation

Der Hersteller Xilinx stellt neben den Werkzeugen VivadoHLS und Vivado Design Suite die Linuxdistribution PetaLinux zur Verwendung auf den Xilinx Hybrid-FPGAs bereit. PetaLinux hat Konfigurations-Werkzeuge und spezielle Linux-Treiber für die Kommunikation zwischen CPU und FPGA integriert. Die

Erstellung einer Anwendungs-Applikation ist durch die Verwendung von Templates auf die Entwicklung des C-Quellcodes reduziert. Da weder die Anpassung eines Linuxsystems für Hybrid-FPGAs, noch die Entwicklung von Kommunikationstreibern Bestandteil dieser Arbeit sind, wurde die Verwendung von PetaLinux für die Grafikdemo ausgewählt. Im Folgenden werden die Inhalte der Grafikdemo, die Übertragung der Ansteuerungsparameter und die Kompilierung in PetaLinux beschrieben.

Demo-Inhalte:

Die Grafikdemo steuert in einer Endlosschleife mit festgelegtem, zeitlichen Ablauf die Funktionalität der VGE und des Rasterisierers. Im ersten Teil der Demo werden die Objekte einzeln mit kleiner Größe und ohne Rotation in der Mitte des Bildschirms angezeigt. Während sie größer werden, drehen sie sich in den 3 Rotationsachsen um dann nach Erreichen einer Maximalgröße wieder zu schrumpfen und in der Ausgangsposition kurz still zu stehen. Dies wird für alle möglichen Objektarten durchlaufen. Der zweiten Teil der Demo beinhaltet die Darstellung der maximal gleichzeitig darstellbaren Objekte. Ausgehend von einem einzelnen Objekt kommen mehr Objekte dazu, verschieben und drehen sich. Dann werden die Sichtbarkeiten der Objekte verändert und die Objekte damit ein- und ausgeschaltet. Zum Schluss der Demo sind alle Objekte unsichtbar und der nächste Durchlauf der Endlosschleife startet. Die beiden Teile der Demo sind jeweils ca. 10 Sekunden lang und die gesamte Demo ca. 20 Sekunden.

Übertragung der Ansteuerungsparameter:

Zur Übertragung der Steuerungsparameter werden die in PetaLinux integrierten AXI-Bus-Treiber verwendet. Diese Treiber sind im Linux-User-Bereich als beschreibbare, ein Bit breite Stubs (**GPIO<Nummer>**) unter dem Pfad **/sys/class/** ins Dateisystem eingehängt. Die Objektparameeter werden mit zu implementierenden Funktionen auf diese Stubs geschrieben. Die gesamte Grafikdemo-Applikation wird in zwei C-Quellcodedateien mit den Namen **demo1.c** und **gpio_lib.c** und der Header-Datei **gpio_lib.h** implementiert.

Kompilierung in PetaLinux

Wie weiter oben beschrieben, wird in PetaLinux ein Template für die Erstellung der Demo-Applikation verwendet. Mit der Verwendung dieses Templates wird ein Makefile mitgeneriert und die Applikation ins Root-Filesystem von PetaLinux integriert. PetaLinux und die Applikation werden dann gemeinsam mit einem Konsolenbefehl (**petalinux-build**) kompiliert. Die Erstellung des bootfähigen Images für die im FPGA-Board verwendete SD-Karte erfolgt mit einem weiteren Konsolenbefehl (**petalinux-package**) und der Angabe von Pfaden zum automatisch generiertem First-Stage-Bootloader (FSBL) und der FPGA-Bitstream-Datei. Weiter Informationen über die Verwendung von PetaLinux sind im Xilinx Dokument [[Xil16a](#)] beschrieben.

Kapitel 5

Implementierung

5.1 Installation der Werkzeuge

Zur Verwendung des, in dieser Arbeit erstellen, Quellcodes in High-Level-Synthese, Synthese, Place and Route, Bitstream-generierung und Anwendung in einem FPGA-Board (ZyBo) werden kommerzielle Werkzeuge des Herstellers Xilinx mit ihren zugehörigen Lizenzen benötigt. Es wurden die folgenden Werkzeuge in den angegebenen Versionen auf dem Betriebssystem Ubuntu 14.04 LTS installiert und verwendet:

- VivadoHLS 2016.1
- Vivado Design Suite 2016.1
- Petalinux 2016.1 Eine selbstgeschriebene Installationsanleitung mit Nennung der notwendigen System-Bibliotheken ist im Anhang [A.2](#) und auf dem beigefügten Datenträger (als Textdatei) enthalten. Diese Anleitung erhebt keinen Anspruch auf Vollständigkeit und ist an die genannten Versionen der Werkzeuge und des Betriebssystems gebunden. Ausführlichere Informationen zu den Installationen sind in den Xilinx-Userguides zu finden.

5.2 Quellcode: GPLv3

Auf dem beigefügten Datenträger sind die, in dieser Arbeit entwickelten, Quellcodes enthalten. Die mit den Xilinx-Werkzeugen generierten Netzlisten, IP-Cores, Blockdesigns und Bitstream-Dateien sind nicht enthalten. Die Quellcodes dieser Arbeit werden unter der GNU General Public License Version 3 veröffentlicht (GPLv3). GPLv3 ist ein Open-Source Lizenzmodell und beinhaltet als Hauptmerkmal das Prinzip Copyleft. Copyleft gibt vor, dass der Quellcode weiterverteilt, kopiert und verändert werden darf, dabei aber immer Open-Source bleibt. Weitere Informationen zum GPLv3-Lizenzmodell sind auf der Internetseite des GNU-Projekts zu finden [[GNU](#)].

5.3 Vektorgrafikeinheit

5.3.1 Vorgehensweise

Das Design der Vektorgrafikeinheit wurde mit einem „Von-Unten-nach-Oben“-Ansatz erstellt. Es wurden erst die einzelnen Module und danach deren Zusammenspiel als Grafik-Pipeline entworfen. In der Implementierung wird dieser Ansatz umgekehrt und zuerst die äußere Schleife für die Pipeline entwickelt. Aus dem Design ging die Notwendigkeit für die sequentielle Berechnung der Pipeline-Schritte hervor. Diese Voraussetzung wird durch die Definition der Datenübergabe zwischen den Modulen und den dafür zu beachtenden Vivado-HLS-Umsetzungen in der Entwicklung der Pipeline-Schleife erreicht. Diese Schleife ist Hauptbestandteil der Top-Level-Funktion `vec_engine()`, welche im folgenden Kapitel als erster Teil der Implementierung beschrieben wird.

5.3.2 Top-Level-Funktion: `vec_engine`

In VivadoHLS ist eine einzelne Funktion des C-Quellcodes als Top-Level-Funktion (TLF) zu bestimmen. Diese darf keinen Funktions-Rückgabetyp besitzen (`void`). Die Funktionsparameter der TLF werden, unter Verwendung von Direktiven, in die Ports signale des synthetisierten IP-Cores gewandelt. In der TLF instanzierte Datenstrukturen werden durch die Synthese zu Speicher. Der funktionale Block der TLF (Pipeline-Schleife) stellt die höchste Hierarchie-Ebene im IP-Core dar und die Unterfunktionen werden zu Untermodulen. Die Bestandteile der TLF werden in dieser Reihenfolge (Funktionsparameter, Instanzen, Pipeline-Schleife) beschrieben.

Im Design der Vektorgrafikeinheit wurden Eingangsparameter für die Objekte definiert (Tabelle 4.1). Jedes darstellbare Objekt benötigt ein Set dieser Parameter. Die Anzahl der gleichzeitig darstellbaren Objekte wird auf maximal 10 Objekte festgelegt, da diese Anzahl nicht dynamisch sein darf (Kapitel 2.6.3). Daher werden 10 Sets aus Objektparametern als Funktionsparameter für die TLF definiert. Jedes Set ist in zwei Kanäle mit 32 Bit Datenbreite eingeteilt. Daraus ergibt sich der Datentyp `uint32` für die Kanäle. Die Objekt-Parameter sind, wie in Abbildung 5.1 dargestellt, in den zwei Kanälen enthalten. Skalare Funktionsparameter werden von VivadoHLS in den Standard-Einstellungen (Default, keine Direktiven) als Eingänge erkannt und zu Eingangssignalen in der Port-Definition gewandelt. Damit keine weiteren Steuersignale für diese Objektparameter angelegt werden, wird auf jeden Kanal (skalarer Funktionsparameter) die Direktive `INTERFACE ap_none` gesetzt. Ein weiterer Eingangs-Funktionsparameter wird auf die gleiche Art für die Invertierung der Z-Achse definiert (Kapitel 4.2.9). Für diesen Parameter ist der Datentyp `uint1` gewählt, da nur ein Bit Information (invertiert oder nicht-

invertiert) übertragen wird. Auch auf diesen Parameter wird die Direktive **INTERFACE ap_none** gesetzt und VivadoHLS erkennt den skalaren Funktionsparameter als Eingang.

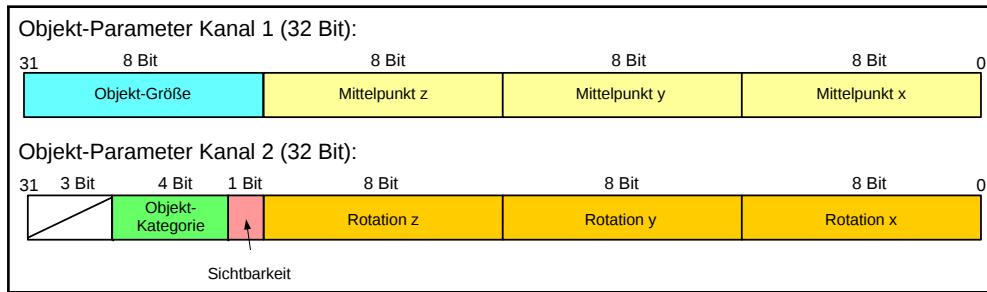


Abbildung 5.1: Eingangskanäle der Objekt-Parameter

In der Vektorgrafikeinheit sind die Signale X,Y und Z als Ausgänge zu definieren. Die im Design festgelegte Datenbreite von jeweils 8 Bit bestimmt den Datentyp **uint8** für diese Funktionsparameter (Kapitel 4.2.8). Diese Funktionsparameter werden als Pointer definiert und damit von VivadoHLS (wieder in den Standard-Einstellungen) als Ausgänge erkannt und als Ausgangssignale synthetisiert. Die Direktive **INTERFACE ap_none** wird zur Unterdrückung von Steuerungssignalen auch hier gesetzt. Der Funktionsprototyp der TLF ist in Quellcode 5.1 gelistet. In der Implementierung des Rasterisierers wird sich die Notwendigkeit für weitere TLF-Funktionsparameter zeigen. Die Erweiterung um diese Parameter wird zusammen mit der Implementierung des Rasterisierers in Kapitel 5.4 beschrieben.

```

1 void vec_engine (uint32 obj1ch1,
2                  uint32 obj1ch2,
3                  ... // Objektparameter der Objekte 2 bis 9
4                  uint32 obj10ch1,
5                  uint32 obj10ch2,
6                  uint8 *x,
7                  uint8 *y,
8                  uint1 *z,
9                  uint1 z_invert);

```

Quellcode 5.1: Funktionsprototyp der Top-Level-Funktion vec_engine

Für alle Berechnungsschritte der Grafik-Pipeline (Unterfunktionen) werden Instanzen der Objekt-Datenstrukturen **obj_1**, **obj_s** und **obj_p** angelegt (Quellcode 5.2, Zeilen 2-6). Für die Objektparameter werden Arrays der Größe 10 angelegt (Zeilen 8-12). Auch für das Signal zur Invertierung der Z-Achse wird ein Zwischenspeicher angelegt (Zeile 14).

```

1 // Instanziierung der Objekte als Datenstrukturen
2 Obj_l list_init;
3 Obj_l list_trans;
4 Obj_l list_rot;
5 Obj_s list_surfaces;
6 Obj_p list_proj;
7 // Instanziierung der Objekt-Parameter-Strukturen:
8 Point m_objects[10];
9 Vector r_objects[10];
10 uint8 size_objects[10];
11 uint1 vis_objects[10];
12 uint4 cat_objects[10];
13 // Z-Invertierung
14 uint1 z_inv;

```

Quellcode 5.2: Instanzen der Objekt- und Parameter-datenstrukturen für die Grafik-Pipeline

Diese Arrays werden mit den Objekt-Parametern befüllt. Für diese Zuweisung werden die zwei Kanäle pro Objekt (Abbildung 5.1) durch Bitshifts und Bitmaskierung in die Parameter aufgetrennt. In Quellcode 5.3 ist dies für das erste Objekt gelistet.

```

1 m_objects[0].xyz[0] = (int)((obj1ch1 & ((uint32) 255))-127);
2 m_objects[0].xyz[1] = (int)((((obj1ch1 >> 8) & ((uint32) 255))-127);
3 m_objects[0].xyz[2] = (int)((obj1ch1 >> 16) & ((uint32) 255));
4 size_objects[0] = (uint8)((obj1ch1 >> 24) & ((uint32) 255));
5 r_objects[0].xyz[0] = (int)(obj1ch2 & ((uint32) 255));
6 r_objects[0].xyz[1] = (int)((obj1ch2 >> 8) & ((uint32) 255));
7 r_objects[0].xyz[2] = (int)((obj1ch2 >> 16) & ((uint32) 255));
8 vis_objects[0] = (uint1)((obj1ch2 >> 24) & ((uint32) 1));
9 cat_objects[0] = (uint4)((obj1ch2 >> 25) & ((uint32) 15));

```

Quellcode 5.3: Befüllen der Objekt-Parameter-Arrays

Um die Implementierung der Pipeline-Schleife zu beschreiben, wird die Verwendung und Bedeutung von C-Pointern nochmals thematisiert und deren Unterschiede in CPU-ausführbarem und synthesefähigen VivadoHLS-C-Quellcode verglichen. In Quellcode 5.4 ist das Grundgerüst der Pipeline-Schleife mit den für den Vergleich notwendigen Bestandteilen gelistet (teilweise in Pseudocode). Der vollständige Quellcode ist auf dem beigefügten Datenträger (dort mit englischen Kommentaren) enthalten.

```

1 void vec_engine(<Funktionsparameter>) {
2
3     // Quellcode 5.2
4     // Quellcode 5.3
5
6     // Pipeline-Schleife ueber 10 Objekte
7     for (i = 0; i < 10; i++) {
8
9         // Initialisierung des i-ten Objekts
10        object_init (&list_init, &m__objects[i], size_objects[i]);
11
12        // Verschiebung des i-ten Objekts
13        translate (&list_trans, &list_init, t_objects);
14
15        // Rotation des i-ten Objekts
16        rotate (&list_rot, &list_trans, r_objects[j].xyz[0],
17                 r_objects[j].xyz[1], r_objects[j].xyz[2]);
18
19        // Berechnung der Flaechen des i-ten Objekts
20        to_surfaces (&list_surfaces, &list_rot);
21
22        // Projektion in 2-D des i-ten Objekts
23        projection (&list_proj, &list_surfaces, 255);
24
25        // Ausgabe des i-ten Objekts auf X,Y,Z
26        draw (&list_proj, x, y, z, z_inv);
27    }
28}

```

Quellcode 5.4: Grundgerüst der Pipeline-Schleife

Das Ausführen dieses Quellcodes auf CPU-basierten Systemen (kompilieren und im Betriebssystem starten) würde die Objekt-Instanzen im Speicher des Systems (Heap-Speicher) anlegen und deren Speicheradressen (Pointer) innerhalb der Schleife an die Funktionen übergeben (call by reference). Die Funktionen würden sequentiell aufgerufen und ihre Berechnungen nacheinander in den referenzierten Speicher schreiben. Am Ende jedes Schleifen-Durchlaufs würde das in 2-D berechnete Objekt ausgegeben werden. Die sequentielle Ausführung der Schleife mit ihren Datenabhängigkeiten zwischen den Modulen ergäbe sich damit direkt aus der Struktur des Quellcodes. Die nicht durch Pointer übergebenen Objekt-Parameter (call by value) würden über den Kellerspeicher des Systems (Stack) an die Funktionen übergeben.

In der Synthese des Quellcodes mit VivadoHLS wird durch das Instanziieren der Objekte und deren Parameter Speicher im FPGA zugewiesen. Dieser Speicher (verteilter

Speicher oder BlockRAM) hat festgelegte Daten- und Steuerungssports, mit den Signalen Adresse, Daten, Read-/Write-enable und Chip-enable. Die Übergabe eines Pointers auf eine solche Instanz ist damit nicht die Übergabe einer Speicheradresse, sondern die Speicherport-Signale werden als Leitungen in die Funktion (das spätere HDL-Modul) verbunden. Die C-Funktionen der Pipeline werden durch VivadoHLS zu parallel arbeitenden VHDL-Modulen synthetisiert. Ohne weitere Maßnahmen (z.Bsp. Direktiven) schreiben und lesen die Module parallel auf den Speicher der Objektinstanzen und korrumptieren die Objektdaten. Diese Parallelität kann für FPGA-Implementierungen, deren Datenabhängigkeiten in speziellen Beziehungen stehen (z.Bsp. Videostream-Daten oder Matrixberechnungen), zu höheren Datendurchsatzraten führen als in CPU-basierten Implementierungen und wird als ein Vorteil für die Verwendung von FPGAs angesehen. Für die Implementierung der Grafikpipeline sind die Datenabhängigkeiten nicht streamfähig und die Berechnungen der Unterfunktionen müssen streng sequentiell erfolgen. Es wurden zwei Implementierungsansätze entworfen, teilweise implementiert und auf korrekte Darstellung auf einem Oszilloskop getestet. Im Folgenden sind diese Ansätze beschrieben und die Auswahl für einen dieser Ansätze begründet.

Implementierungs-Ansatz: Direktiven

Aus der, in den Grundlagen beschriebenen, VivadoHLS-Abstraktion zwischen Software- und Hardwareprogrammierung ist die Verwendung von Direktiven ein Ansatz zur Bestimmung der Hardware-Eigenschaften dieser Implementierung. Die Funktionen werden in parallele Hardware-Module synthetisiert und die Spezifizierung dieser Module durch Direktiven bestimmt. Es sind die notwendigen VivadoHLS-Direktiven und deren Parameter für die sequentielle Ausführung der Module zu ermitteln und anzuwenden. Die sequentielle Ausführung könnte damit durch, von VivadoHLS erstellten, Zustandsautomaten erreicht werden. Die Direktive **INTERFACE ap_ctrl_chain** fügt die Signale **ap_start**, **ap_ready**, **ap_idle**, **ap_done** und **ap_continue** zum Port eines Moduls hinzu. Diese Steuerungssignale geben den Zustand des Moduls und seiner Ein- und Ausgangssignale an. Erst wenn ein Modul seine Daten mit **ap_ready** freigibt, erhält das nachfolgende Modul mit **ap_start** die Freigabe zum Start der Berechnungen. So kann eine sequentielle Kette aus Modulen erreicht werden. Ein weiterer Direktiven-Kandidat ist **DATAFLOW**. Auch diese Direktive erlaubt die Steuerung von Datenabhängigkeiten zwischen Modulen, ist aber für streamfähige Datenabhängigkeiten entwickelt. Nach mehreren Test-Implementierungen stellte sich heraus, dass die Direktive **INTERFACE ap_ctrl_chain** nur auf die Synthese der Top-Level-Funktion anzuwenden ist und für Unterfunktionen keinerlei Änderung des Synthese-Verhaltens bewirkt. Die Direktive **DATAFLOW** ist auch auf Unterfunktionen anzuwenden, stellt aber keine Parameter zur Konfiguration bereit und kann damit nicht die sequentielle Abfolge der Pipeline erwirken.

Implementierungs-Ansatz: Lokaler Speicher

Der zweite Implementierungs-Ansatz arbeitet nicht mit Direktiven, sondern mit lokalem Speicher innerhalb der Unterfunktionen. Die Berechnungsergebnisse auf die, als Pointer übergebenen, Objektinstanzen werden in funktions-lokalen Instanzen gespeichert. Erst am Ende der Funktion wird das lokale Ergebnis in die, auch als Pointer übergebene, Ergebnisinstanz kopiert. Durch die Verwendung von lokal definiertem Speicher erkennt VivadoHLS die Abhängigkeiten unter den Modulen selbstständig und setzt diese in sequentiell berechnende Module um. Dieser Ansatz ist damit an die Verwendung von zusätzlichem Speicher verknüpft. Jede Objekt-Datenstruktur ist damit zweimal als Speicher vorhanden, einmal in der Top-Level-Funktion und noch einmal in einer der Unterfunktionen. Dieser Ansatz wird für die Implementierung der Grafik-Pipeline verwendet und in den, in Kapitel 5.3.3 beschriebenen, Unterfunktionen umgesetzt.

5.3.3 Module: C-Unterfunktionen

Um in allen Unterfunktionen gleich implementierte Bestandteile nicht wiederholend zu beschreiben, werden diese Teile vor der Beschreibung der Funktionen hier zusammen aufgeführt. Die Funktionsprototypen sind nach dem in Quellcode 5.5 gelisteten Schema entwickelt.

```

1 void funktionsname (<Ausgabeparameter>,
2                     <Eingabeparameter>,
3                     <Objektparameter>);
```

Quellcode 5.5: Prototypen der Unterfunktionen

In den Funktionsparametern wird zuerst der Pointer auf die zu beschreibende Datenstruktur und danach der Pointer auf die zu lesende Datenstruktur angegeben. Die für die jeweilige Berechnung notwendigen Objekt-Parameter werden als letzter Funktionsparameter (call by value) übergeben. Die einzelnen Prototypen sind durch die Funktionsaufrufe in Quellcode 5.4 zu erkennen.

Im Funktionsblock wird eine Ergebnisvariable deklariert. Diese hat den gleichen Datentyp wie der Ausgabeparameter und wird als Zwischenspeicher für die Berechnungsergebnisse verwendet. Nach den, in jeder Funktion unterschiedlichen, Berechnungen werden die Ergebnisse am Ende der Funktion in die Datenstruktur des Ausgabeparameters kopiert. Zum Kopieren der Daten werden Schleifen verwendet, wo es möglich ist.

object_init:

Die Funktion `object_init` ist in zwei verschiedenen Versionen implementiert (`cube_init` und `pyramid_init`). Mit den zwei Versionen werden unterschiedliche 3-D-Objekte in die gleiche Datenstruktur `Obj_1` geschrieben: Mit `cube_init` ein Würfel und mit `pyramid_init` ein Pyramidenstumpf. Als Objektparameter werden der Mittelpunkt und die Größe des Objekts an die Funktion übergeben. Der Würfel wird mit dem doppelten Größenparameter als Kantenlänge erstellt. Der Pyramidenstumpf hat die doppelte Höhe des Größenparameters. Die Mittelpunkte werden mit ihren x,y und z-Parametern in die Datentruktur übernommen. Die Reihenfolge der Eckpunkt-Zuweisungen erfolgt nach den Definitionen aus Abbildung 4.2 und ist für den ersten Eckpunkt (P_0) in Quellcode 5.6 gelistet.

```

1 result.points[0].xyz[0] = m.xyz[0] - size;
2 result.points[0].xyz[1] = m.xyz[1] - size;
3 result.points[0].xyz[2] = m.xyz[2] + size;
```

Quellcode 5.6: Funktion `cube_init`: Zuweisung des ersten Würfeleckpunkts

Die Auswahl der Initialisierungsfunktion ist in der Pipeline-Schleife durch eine bedingte Verzweigung (`if`) auf den Parameter `cat_objects` implementiert. Es können weitere Initialisierungsfunktionen für weitere Objekte erstellt werden und die `if`-Verzweigung um diese Funktionen erweitert werden. Der Parameter `cat_object` ist mit 4 Bit Datenbreite definiert und kann bis zu 16 verschiedene Objekt-Initialisierungsfunktionen ansteuern. Diese Option ist für zukünftige Erweiterungen der Vektorgrafikeinheit vorgesehen.

translate:

In der Funktion `translate` werden der Mittelpunkt und alle Eckpunkte des Objekts mit dem Eingabeparameter `t_objects` addiert. Hierfür werden Schleifen für die drei Koordinatenachsen und 8 Eckpunkte verwendet. Diese Schleifen sind in Quellcode 5.7 gelistet. In der Pipeline-Schleife wird diese Funktion mit einem Null-Vektor als Parameter aufgerufen und berechnet daher keine Verschiebung. Die Verschiebung der Objekte in der Vektorgrafikeinheit wird durch den Mittelpunkt-Parameter der Top-Level-Funktion erreicht. `translate` ist als eigene Funktion implementiert, da sie in der Funktion `rotation` benötigt wird. In der Pipeline-Schleife ist sie enthalten, um die Vollständigkeit der Vektorgrafikeinheit zu erhalten und stellt eine Grundlage für weitere Entwicklungen an der Einheit dar.

```

1 for (i = 0; i < 3; i++) {
2     result.middle.xyz[i] = list_in->middle.xyz[i] + vtrans.xyz[i];
```

```

3 }
4 for (i = 0; i < 8; i++) {
5     for (j = 0; j < 3; j++) {
6         result.points[i].xyz[j] = list_in->points[i].xyz[j] +
7             vtrans.xyz[j];
8 }

```

Quellcode 5.7: Funktion translate: Verschiebung des Mittelpunkts und der Eckpunkte

rotate:

Die Funktion **rotate** ist als Sub-Pipeline implementiert. Da die in Kapitel 2.3 genannten Rotationsmatrizen die Rotationen relativ zu den Koordinatenachsen berechnen, ist jedes Objekt vor der Rotation an den Koordinatenursprung zu verschieben, dann in allen Achsen zu rotieren und danach wieder an seine ursprüngliche Position zurück zu verschieben. Das ergibt die folgende Reihenfolge der Rotations-Pipeline:

1. Objektmittelpunkt zwischenspeichern
2. Objekt zum Koordinatenursprung verschieben
3. Objekt um die X-, Y- und Z-Achsen rotieren
4. Objekt zum gespeicherten Mittelpunkt verschieben

Die Objekt-Verschiebung wird durch Aufrufe der **translate**-Funktion erreicht. Die Rotationensberechnungen sind den entsprechenden Matrizen folgend implementiert. Für diese Berechnungen werden die trigonometrischen Funktionen sinus und cosinus verwendet. Diese Funktionen werden durch vorberechnete Tabellen implementiert. Diese Tabellen bilden die 8-Bit Rotationsparameter auf den Wertebereich $[-1, 1]$ ab und verwenden weniger FPGA-Ressourcen als eine Implementierung mit Digitalen Signal Prozessoren (DSP). Die Tabellen sind in der C-Header-Datei (Quellcode 4.1) nach den Datenstrukturen definiert. Die vollständige Header-Datei ist auf dem beigefügten Datenträger enthalten. In Quellcode 5.8 ist die Rotation des Objekts um die X-Achse gelistet.

```

1 co = cos_table[a];
2 si = sin_table[a];
3 for (i = 0; i < 8; i++) {
4     y_temp = (float)result2.points[i].xyz[1];
5     z_temp = (float)result2.points[i].xyz[2];
6     result2.points[i].xyz[1] = (int)((co * y_temp) - (si * z_temp));
7     result2.points[i].xyz[2] = (int)((si * y_temp) + (co * z_temp));
8 }

```

Quellcode 5.8: Funktion rotate: Rotation des Objekts um die X-Achse

to_surfaces:

In der Funktion **to_surfaces** werden die 8 Eckpunkte des Objekts den Flächeneckpunkten der Datenstruktur **obj_s** zugewiesen. Aus den Flächeneckpunkten werden die, die Flächen aufspannenden, Vektoren berechnet. Aus diesen Flächenvektoren werden dann die Flächennormal-Vektoren berechnet. Die Flächennormal-Vektoren sind für die Verdeckungsberechnung notwendig und werden in der Ausgabe-Datenstruktur **obj_s** gespeichert. Die in den Zwischenberechnungen verwendeten Flächenvektoren werden nach der Berechnung nicht weiterverwendet und deren Instanzen sind erst für die Berechnung des nächsten Objekts in der Grafik-Pipeline wieder notwendig. Die Implementierung dieser gesamten Funktion besteht zu großen Teilen aus der Zuweisung der Eckpunkte und den sich daraus ergebenden Zwischenvektoren und ist daher nicht hier als Quellcode aufgelistet. Für die Berechnung der Flächennormal-Vektoren wurde eine weitere Unterfunktion **crossproduct** implementiert. Diese Funktion ist in Quellcode 5.9 gelistet.

```

1 void crossproduct(Vector *vout, Vector *v1, Vector *v2) {
2     Vector result;
3     Vector vin1;
4     Vector vin2;
5     int i;
6     for(i = 0; i < 3; i++) {
7         vin1.xyz[i] = v1->xyz[i];
8         vin2.xyz[i] = v2->xyz[i];
9     }
10    result.xyz[0] =
11        ((vin1.xyz[1]*vin2.xyz[2])-(vin1.xyz[2]*vin2.xyz[1]));
12    result.xyz[1] =
13        ((vin1.xyz[2]*vin2.xyz[0])-(vin1.xyz[0]*vin2.xyz[2]));
14    result.xyz[2] =
15        ((vin1.xyz[0]*vin2.xyz[1])-(vin1.xyz[1]*vin2.xyz[0]));
16    for (i = 0; i < 3; i++) {
17        vout->xyz[i] = result.xyz[i];
18    }
19 }
```

Quellcode 5.9: Funktion crossproduct: Berechnung der Flächennormal-Vektoren (Vektor-Kreuzprodukt)

In diesem Quellcode ist auch die, in allen Unterfunktionen implementierte, Verwendung von Zwischenspeicher-Instanzen für die Berechnungsergebnisse zu erkennen. Am Anfang der Funktion wird eine Ergebnisvariable instanziert, mit den berechneten Ergebnissen befüllt und am Ende der Funktion in die, als Pointer übergebene, Ausgabestruktur kopiert.

projection:

Die Funktion `projection` führt zwei Berechnungen auf das Objekt aus. Erst wird die Sichtbarkeit jeder Objektfläche berechnet und gespeichert. Danach wird die Berechnung zur Projektion jedes 3-D-Flächeneckpunkts auf die 2-dimensionale Projektionsfläche berechnet. Die Berechnung der Sichtbarkeit setzt zwei Vektoren voraus, den Flächennormalenvektor aus der Funktion `to_surfaces` und den Aufsichtsvektor des Betrachters auf die Fläche. Dieser Aufsichtsvektor wird als lokale Variable berechnet und aus den beiden Vektoren das Skalarprodukt bestimmt. Das Vorzeichen des Skalarprodukts gibt die Sichtbarkeit der Fläche an. Diese Sichtbarkeitsberechnung ist in Quellcode 5.10 gelistet und wird in einer Schleife über die 6 Flächen des Objekts ausgeführt.

```

1 for (i = 0; i < 6; i++) {
2     x_view = list_in->surfaces[i].points[0].xyz[0];
3     y_view = list_in->surfaces[i].points[0].xyz[1];
4     z_view = list_in->surfaces[i].points[0].xyz[2] + d;
5     x_norm = list_in->normals[i].xyz[0];
6     y_norm = list_in->normals[i].xyz[1];
7     z_norm = list_in->normals[i].xyz[2];
8     if (((x_view * x_norm) + (y_view * y_norm) + (z_view * z_norm)) <=
9         0) {
10        result.visible[i] = true;
11    } else {
12        result.visible[i] = false;
13    }
14}

```

Quellcode 5.10: Funktion `projection`: Berechnung der Sichtbarkeit der Objektflächen

Die Projektion der Flächeneckpunkte ins 2-Dimensionale ist in Quellcode 5.11 gelistet und verwendet die Projektionsmatrix aus Kapitel 2.2. In dieser Matrix sind Additionen, Multiplikationen und Divisionen enthalten. Diese Berechnungen werden auf `uint8`-Datentypen ausgeführt und auch wieder in solche gespeichert. Die entstehenden Rundungsfehler sind in der Darstellung auf dem Oszilloskop-Bildschirm als Ruckeln der Objekte zu bemerken und können durch Verwendung breiterer Datentypen in zukünftigen Erweiterungen der Vektorgrafikeinheit verbessert werden.

```

1 for (i = 0; i < 6; i++) {
2     for (j = 0; j < 4; j++) {
3         result.surfaces[i].points[j].xyz[0] = ((d *
4             result.surfaces[i].points[j].xyz[0]) /
5             (result.surfaces[i].points[j].xyz[2] + d)) ;

```

```

4     result.surfaces[i].points[j].xyz[1] = ((d *
5         result.surfaces[i].points[j].xyz[1]) /
6             (result.surfaces[i].points[j].xyz[2] + d));
7     result.surfaces[i].points[j].xyz[2] = 0;
}
}

```

Quellcode 5.11: Funktion `projection`: Projektion des Objekts auf die 2-D-Projektionsfläche

draw:

Die Funktion `draw` ist keine Berechnungsfunktion wie die bisher beschriebenen Funktionen. Sie erhält das Ergebnis der Projektion als Pointer auf die Datenstruktur `obj_p` und das `z_invert`-Signal als Eingabeparameter und hat keine Ausgabedatenstruktur für berechnete Ergebnisse. Die Ausgaben dieser Funktion sind die 8 Bit breiten X,Y und Z-Ausgangssignale. Diese sind als Pointer angelegt und werden durch die Top-Level-Funktion an die I/O-Pins weitergeleitet. Der Funktionsprototyp ist in Quellcode 5.12 gelistet.

```

1 void draw(Obj_p *list_in, uint8 *x, uint8 *y, uint1 *z, uint1 z_inv);

```

Quellcode 5.12: Funktionsprototyp: `draw`

Die Ausgabe der X,Y und Z-Werte erfolgt nach einer zeitlichen, taktgesteuerten Reihenfolge. In einer Schleife über die 6 Flächen des darzustellenden Objekts wird das in Abbildung 5.2 dargestellte Struktogramm als Algorithmus durchlaufen. Ausgehend vom ersten

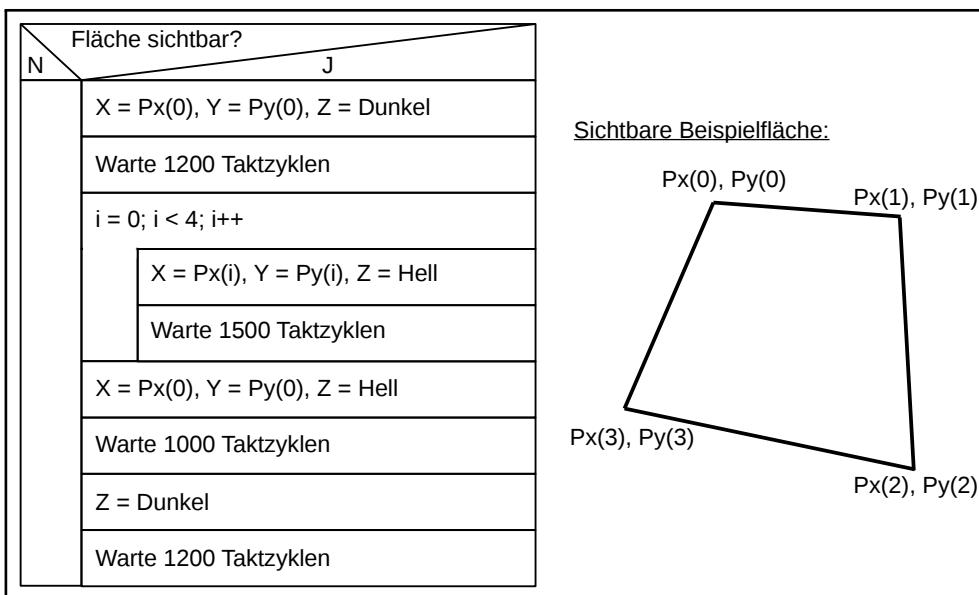


Abbildung 5.2: Struktogramm des Schleifen-Ausführungsblocks in der `draw`-Funktion

Eckpunkt der Fläche wird dieser ausgegeben, aber die Helligkeit (Z-Achse, Werte: Hell-/Dunkel) auf Dunkel gesetzt. So bewegt sich der Elektronenstrahl im Oszilloskop zum Punkt, bleibt aber für 1200 Taktzyklen dunkel. Erst dann wird der Z-Ausgang auf hell gesetzt und die restlichen Eckpunkte der Fläche werden mit einer jeweiligen Wartezeit von 1500 Taktzyklen pro Punkt angesteuert. Zum Ende wird der Elektronenstrahl zurück zum Anfangspunkt der Fläche bewegt und wieder auf Dunkel gesetzt. Somit sind die 4 Eckpunkte und Verbindungslien (Vektoren) als sichtbare Leuchtspur auf dem Oszilloskop-Bildschirm zu sehen. Das Zeichen einer Fläche benötigt 9400 Taktzyklen.

5.4 Rasterisierer

Der Rasterisierer wird nach den Design-Entscheidungen aus Kapitel 4.3 implementiert. Die Implementierungen des Rasterisierers sind in der gleichen Reihenfolge wie im Design beschrieben. Die Pixelclock (65MHz) wird in der Vivado Design Suite innerhalb eines IP-Cores generiert, steht damit für die Module VGA-Generator und BlockRAM-Speicher zur Verfügung und wird hier nicht weiter beschrieben. Alle im Folgenden beschriebenen Module sind eigenständige Projekte in VivadoHLS und aus jedem dieser Projekte kann ein einzelner IP-Core generiert werden. Diese IP-Cores können in der Vivado Design Suite zu einem Blockdesign zusammengefügt und verbunden werden. Eine Gesamtübersicht der Vektorgrafikeinheit und des Rasterisierers als Blockdesign ist im Anhang A.7 dargestellt.

5.4.1 VGA-Generator

Der VGA-Generator ist das einzige Modul dieser Arbeit, das in nativem VHDL implementiert wurde. Die taktgenaue, synchronisierte Ausgabe der Zähler H- und V-count und den daraus resultierenden H- und V-Synchronisationssignalen (H- und V-Sync für die VGA-Schnittstelle) konnte nach mehreren Implementierungsansätzen nicht mit VivadoHLS erreicht werden. Aus der Zähllogik und der Synchronisation auf festgelegte Werte dieser Zähler (siehe Abbildung 4.14) wurden in VivadoHLS Zustandsautomaten mit Speicher synthetisiert, wodurch die Ausgabe um teilweise mehrere Taktzyklen verzögert wurde. Die Taktverzögerung ergab einen Versatz zwischen den Zählern und Synchronisationssignalen. Da diese für die folgenden Rasterisierer-Module aber synchron benötigt werden, konnte keine fehlerfreie Implementierung erreicht werden. Im Anhang in Quellcode A.3 ist die Implementierung in VHDL gelistet. In der Portdefinition ist die Pixelclock als einziger Eingang und die H-/V-count-, H-/V-sync-Signale als Ausgänge bestimmt. H-

und V-count sind mit 11 Bit und 10 Bit Datenbreite angelegt, können damit die notwendigen Integer-Werte abbilden und belegen nicht mehr Leitungsressourcen im FPGA als benötigt werden. Die Synchronisationssignale sind mit einem Bit Datenbreite angelegt, da sie nur die Zustände High- und Low-Pegel annehmen. Für die Zähler werden Variablen definiert und am Ende des Prozesses auf die Portausgänge zugewiesen. Im Prozess-Block werden mit jeder steigenden Flanke des Taktsignals (`clk_in`) die Zähler bis zu ihren Maximalwerten inkrementiert und bei Erreichen des Maximalwerts auf 0 zurückgesetzt. Mittels dieser Zähler werden die Ausgabe-Zeitpunkte der Synchronisationssignale durch bedingte Verzweigungen (`if`-Bedingungen) ermittelt und den Signalen die High- oder Low-Pegel zugewiesen. Der Prozess läuft dauerhaft und hat keine Reset-Funktionalität.

5.4.2 Bildschirmspeicher und Bildschirm-Pufferspeicher

Die benötigten zwei BlockRAM-Speicher sind durch IP-Cores aus der Vivado Design Suite Bibliothek erstellt. Diese IP-Cores sind als True-Dual-Port Speicher mit 32 Bit Datenbreite und 24.576 Einheiten Größe angelegt. Diese Größenangabe ist gleich den Array-Größen in den Funktionsparametern der Rasterisierer-Module und findet sich in den Funktionsprototypen wieder (`uint32 <Arrayname>[24576]`). Beide BlockRAM-Module können durch ihre True-Dual-Port-Eigenschaft auf den zwei Ports gleichzeitig gelesen und beschrieben werden. Da die Ports in unterschiedliche Module verbunden sind (Abbildung 4.19), ist diese Eigenschaft notwendig. Die in den Ports vorhandenen Signale sind in Abbildung 5.3 dargestellt. Für jeden Port werden die Adresssignale, write- und chip-enable-Signale, der Takt (Clock) und die Datensignale (ein- und ausgehend) generiert. Die Konfiguration des Adressraums und der Datenbreite ist in der Vivado Design Suite über Grafische Oberflächen möglich.

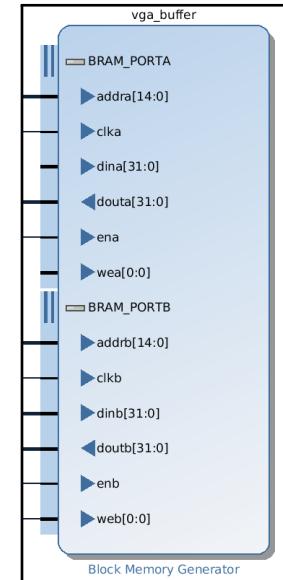


Abbildung 5.3: BlockRAM mit Portsignalen: vga-buffer

Quelle: Xilinx Vivado Design Suite 2016.1

5.4.3 Lesemodul: vga_out

Das Lesemodul `vga_out` ist in Quellcode 5.13 gelistet und im Folgenden beschrieben. Im Funktionsprototyp sind das Array `vgaram_in` und die zwei Zähler `hcount` und `vcount` als

Eingänge und die Werte für Rot, Grün und Blau als Ausgänge (Pointer auf skalare Typen) enthalten. Für das Array ermittelt VivadoHLS die Zugriffsarten im Funktions-Codeblock (Lesen, Schreiben oder Beides) und erstellt danach die Ports signale. Durch die Direktiven (Quellcode 5.14) wird die Erstellung der Signale weiter spezifiziert. Durch die erste Direktive werden die Ports signale auf die Signale des BlockRAMs aus Abbildung 5.3 angepasst. Ohne diese Direktive würden die Signale im Port zu einem einzigen Signalstrang zusammen gebündelt und könnten dann nicht mit dem BlockRAM verbunden werden. Die zweite Direktive gibt die Art des Speichers für das Array (BlockRAM, ein Port wird benutzt) an und gibt eine Schreib-/Lese-Latenz von 2 Taktzyklen vor. Die restlichen Funktionsparameter werden, durch die, schon in der Vektorgrafikeinheit verwendete, Direktive **INTERFACE ap_none** als Ports signale ohne Steuersignale erstellt.

```

1 void vga_out(uint32 vgaram_in[24576],
2               uint11 hcount,
3               uint10 vcount,
4               uint5 *red_out,
5               uint6 *green_out,
6               uint5 *blue_out) {
7     uint32 linebuffer[32];
8     int i, j;
9
10    if ((hcount > (uint11) 1024) &&
11        (hcount < (uint11) 1058) &&
12        (vcount >= (uint10) 0) &&
13        (vcount < (uint10) 767))
14    {
15        // Nächste Bildzeile in Zeilenpuffer kopieren
16        for (i = 0; i < 32; i++) {
17            linebuffer[i] = vgaram_in[((vcount + 1) * 32) + i];
18        }
19    } else if ((hcount > (uint11) 1024) &&
20               (hcount < (uint11) 1058) &&
21               (vcount > (uint10) 766) &&
22               (vcount < (uint10) 768))
23    {
24        // Erste Bildzeile am Ende des Bildes in Puffer kopieren
25        for (j = 0; j < 32; i++) {
26            linebuffer[j] = vgaram_in[j];
27        }
28    } else if ((hcount < (uint11) 1024) &&
29               (vcount < (uint10) 768) &&
30               ((linebuffer[((hcount) / 32)] >> (hcount % 32)) & 0x01))
31    {

```

```

32     // Pixel an
33     *red_out = 31;
34     *green_out = 63;
35     *blue_out = 31;
36 } else {
37     // Pixel aus
38     *red_out = 0;
39     *green_out = 0;
40     *blue_out = 0;
41 }
42 }
```

Quellcode 5.13: Rasterisierer-Funktion: vga_out

```

1 HLS INTERFACE ap_memory port=vgaram_in
2 HLS RESOURCE variable=vgaram_in core=RAM_1P_BRAM latency=2
```

Quellcode 5.14: Direktiven für die Portssignale des Arrays vgaram_in

Im Funktionsausführungsblock folgen drei verzweigte Bedingungsblöcke aufeinander. Jede Bedingung ist nach der Abbildung 4.18 aus den Zählern H- und V-count erstellt. Im ersten Verzweigungsblock wird die jeweils nächste Bildzeile in den Zeilenpuffer kopiert. Im zweiten Verzweigungsblock wird die erste Bildzeile am Ende des Bildes in den Zeilenpuffer kopiert. Der dritte Verzweigungsblock hat neben den Zählerständen noch die zusätzliche Bedingung (Zeile 30), dass der aktuelle Pixel im Zeilenpuffer gesetzt sein muss. Ist dieses Bit gesetzt, werden die R, G und B-Ausgänge mit ihren maximalen Werten (Farbe Weiß) ausgegeben, sonst die minimalen Werte (Farbe Schwarz).

5.4.4 Kopiermodul: vga_copy

Das Kopiermodul ist, im Vergleich zu den anderen Modulen des Rasterierers, ein kleines Modul mit wenig Quellcode-Zeilen (Quellcode 5.15). Die Direktiven sind hier als Pragmas mit im Quellcode aufgelistet. Die Anzahl der Direktiven ist größer als die Anzahl der Codezeilen. Die ersten drei Direktiven sind aus anderen Modulen bekannt und verhindern die Erstellung von Steuersignalen auf den gesamten Port (Zeile 6) und die Signale **vcount** und **hcount** (Zeilen 7-8).

```

1 void ram_copy(uint32 vgabuf_in[24576],
2               uint32 vgaram_out[24576],
3               uint11 hcount,
4               uint10 vcount)
```

```

5 {
6 #pragma HLS INTERFACE ap_ctrl_none port=return
7 #pragma HLS INTERFACE ap_none port=vcount
8 #pragma HLS INTERFACE ap_none port=hcount
9 #pragma HLS RESOURCE variable=vgabuf_in core=RAM_1P_BRAM latency=2
10 #pragma HLS INTERFACE ap_memory port=vgabuf_in
11 #pragma HLS RESOURCE variable=vgaram_out core=RAM_1P_BRAM
12 #pragma HLS INTERFACE ap_memory port=vgaram_out
13
14 if ((vcount > 767) && (vcount < 780)) {
15     int i;
16     for (i = 0; i < 24576; i++) {
17         vgaram_out[j] = vgabuf_in[i];
18     }
19 }
20 }
```

Quellcode 5.15: Rasterisierer-Funktion: vga_copy

Die Direktiven in den Zeilen 9 bis 12 bewirken die Erstellung der Array-Signale zur Verbindung mit den BlockRAM-Modulen, geben BlockRAM als Speicherart für die Arrays an und legen die Schreib-/Lese-Latenzen fest. Für das Array `vgabuf_in` ist eine Latenz von 2 Taktzyklen und für das Array `vgaram_out` keine Angabe (Latenz = 1 Taktzyklus) angegeben. Die Ermittlung dieser Werte war der zeitintensivste Teil der Implementierung des Rasterisierers und wird daher im Folgenden beschrieben.

In Abbildung 5.4 sind zwei Simulationsdiagramme der Implementierung des Kopiermoduls dargestellt. Beide Diagramme zeigen den Kopiervorgang der Arrayzelle 2082 (Schleifenvariable $i = 2082$). FLCK ist die Pixelclock und für die Simulation auf einen Taktzyklus von 10ns pro Zyklus gesetzt. Die Zeilen darunter geben die aktuellen hcount und vcount-Werte als Integers wieder. addra und douta sind die Adress- und Datensignale des zu lesenden BlockRAMs (`vgabuf_in[i]`). addrb und doutb sind die Adress- und Datensignale von `vgaram_out[i]`. enb und web sind die Enable- und Write-enable-Signale des zu schreibenden BlockRAMs (`vgaram_out[i]`). Das linke Diagramm zeigt die fehlerhafte Implementierung ohne Latenz-Angaben in den Direktiven (Standardeinstellung: Latenz = 1) und das rechte Diagramm zeigt die korrekte Implementierung, wie in Quellcode 5.15 gelistet. Ohne Latenzangabe (links) folgen die Schreib- und Lesevorgänge mit jeweils zwei Taktzyklen aufeinander. Der korrekte Wert für die Arrayzelle 2082 (Wert: 0x00800000) liegt zum Schreibvorgang noch nicht vor und der falsche Wert (0x00000000) wird geschrieben. Im rechten Diagramm dauert der Schreib-/Lesezyklus drei Taktzyklen und der richtige Wert liegt während des Schreibvorgangs vor.

Die, aus der fehlerhaften Implementierung resultierenden Darstellungsfehler bestanden

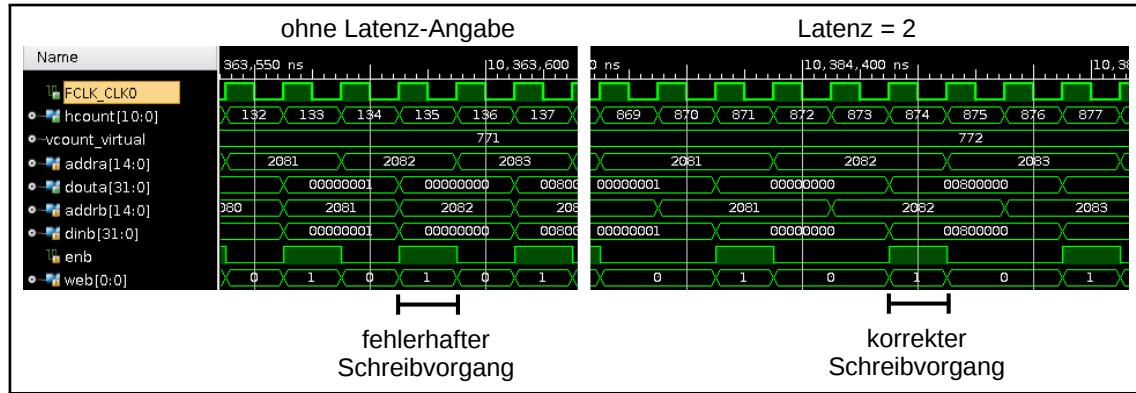


Abbildung 5.4: Kopiermodul: Fehlerhafte und korrekte Direktiven in der Simulation

Quelle: Erstellt mit Xilinx Vivado Design Suite 2016.1

aus Linien mit einzelnen, an falscher Stelle dargestellten Bildpunkten. Als Fehlermuster war zu erkennen, dass immer das Most-Significant-Bit (MSB) oder Least-Significant-Bit (LSB) eines 32 Bit-Wortes im Speicher betroffen war. Ein ähnlicher Fehler hat sich im Schreibmodul ergeben und wird im Kapitel 5.4.5 beschrieben. Weitere Betrachtungen zur aufgewendeten Zeit für die Fehlersuche und resultierenden Betrachtungen über VivadoHLS sind im Fazit der Arbeit enthalten.

5.4.5 Schreibmodul

Für das Schreibmodul sind mehrere Aufgaben zu implementieren. Die Datenstruktur `obj_p` aus der Vektorgrafikeinheit enthält die darzustellenden 2-D-Objekte und wird in der Grafik-Pipeline mit jedem Schleifendurchlauf neu berechnet. Es liegen also nicht alle darzustellenden Objekt-Daten zu einem gemeinsamen Zeitpunkt vor. Die Datenübergabe von der Vektorgrafikeinheit zum Rasterisierer ist damit eine zu lösende Aufgabe. Das Beschreiben des Bildschirm-Pufferspeichers erfolgt durch eine Implementierung des Bresenham-Algorithmus. Hierfür ist eine Unterfunktion zum Beschreiben eines einzelnen Pixels notwendig. Da der Bildschirmspeicher in 32 Bit breiten Dateneinheiten organisiert ist (eine Arrayzelle bildet 32 Pixel ab), wird hierfür eine lesende und schreibende Funktion unter Verwendung von Bitshifts und Bitmaskierungen benötigt. Die Implementierung dieser Aufgaben in mehreren Funktionen wird im Folgenden beschrieben.

Datenübergabe:

Die Funktionen des Schreibmoduls sind in die Grafik-Pipeline integriert. Zu den Funktionsaufrufen innerhalb der Pipeline wird vor Aufruf der Vektorzeichen-Funktion `draw` die Funktion `vga_draw` aufgerufen. Das gesamte Schreibmodul ist in die Implementierung der Vektorgrafikeinheit integriert. Eine synchronisierte Datenübergabe zwischen den bei-

den Implementierungen ist machbar, wurde aber in dieser Arbeit nicht entwickelt. Die Liste der Funktionsparameter der Top-Level-Funktion `vec_engine` enthält durch diese Integration die in Quellcode 5.16 gelisteten zusätzlichen Parameter.

```

1 void vec_engine  (...  
2           // Bisher definierte Funktionsparameter  
3           ...  
4           // Zusätzliche Funktionsparameter:  
5           uint32 vgabuf[24576],  
6           uint11 hcount,  
7           uint10 vcount);
```

Quellcode 5.16: Zusätzliche Funktionsparameter in der Top-Level-Funktion der Vektorgrafikeinheit

Der Aufruf der Rasterisierer-Schreibfunktion in der Pipeline wird mit dem Löschen des Bildschirm-Pufferspeichers zusammengefasst. Der, in die Pipeline eingefügte, Quellcode 5.17 entscheidet mittels des Zählers `vcount`, ob die Funktion `vga_draw` aufgerufen oder der Bildschirm-Pufferspeicher gelöscht wird.

```

1 // Zeichne Objekt in Bildschirm-Pufferspeicher  
2 if ((vcount > (uint11) 100) &&  
3     (vcount < (uint11) 500))  
4 {  
5     vga_draw(&list_proj, vgabuf);  
6     // oder loesche den Bildschirm-Pufferspeicher  
7 } else if ((vcount > (uint10) 0) &&  
8             (vcount < (uint10) 80))  
9 {  
10    for (k = 0; k < 24576; k++) {  
11        vgabuf[k] = 0;  
12    }  
13}
```

Quellcode 5.17: Zusätzlicher Quellcode in der Grafik-Pipeline

Unterfunktion `vga_draw`:

Die Funktion `vga_draw` kopiert die Objektstruktur in einen lokal erstellten Zwischenspeicher (`Obj_p result`) und durchläuft in zwei geschachtelten Schleifen die 6 Flächen und jeweils 4 Flächenkanten des Objekts (Quellcode 5.18). Hierfür wird das Wertepaar des letzten Eckpunkts zwischengespeichert und die Funktion `vga_draw_line` mit dem Start- und Endpunkt der zu zeichnenden Linie aufgerufen.

```

1 for (i = 0; i < 6; i++) {
2     if (result.visible[i]) {
3         x_0 = ((result.surfaces[i].points[3].xyz[0])+127)*4;
4         y_0 = 767-(((result.surfaces[i].points[3].xyz[1])+127)*3);
5         for (j = 0; j < 4; j++) {
6             x_1 = ((result.surfaces[i].points[j].xyz[0])+127)*4;
7             y_1 = 767-(((result.surfaces[i].points[j].xyz[1])+127)*3);
8             vga_draw_line(vgabuf, x_0, y_0, x_1, y_1);
9             x_0 = x_1;
10            y_0 = y_1;
11        }
12    }
13 }
```

Quellcode 5.18: Geschachtelte Schleifen in der Rasterisierer-Funktion **vga_draw**

In den Zuweisungen der temporär gespeicherten x/y-Wertepaare (Zeilen 3,4,6,7) sind Additionen, Subtraktionen und Multiplikationen enthalten. Auf die Werte wird eine Verschiebung zum Bildmittelpunkt (+127) addiert. Die Erstellung der 3-D-Objekte erfolgt zentriert zum Ursprung des Koordinatensystems mit positiven und negativen Werten im Bereich [-127, 127]. Die Pixelbildschirm-Darstellung hat den Ursprung (0,0) an der linken, oberen Bildschirmecke. Dieser Unterschied wird durch die Addition ausgeglichen. Die Multiplikation der x-Werte mit dem Faktor 4 (und y-Werte mit Faktor 3) skaliert die gesamte Darstellung von den 8 Bit-Werten der Vektorgrafikeinheit auf die Auflösung des Pixelbildschirms (1024 * 768 Pixel). Die Subtraktion für die y-Werte (767 - Wert) bildet die Umkehrung der Y-Achse ab. In der Vektorgrafikeinheit sind die kleinen Y-Werte unten im Bild und auf einem Pixelbildschirm sind diese oben im Bild.

Unterfunktion vga_draw_line:

Die Unterfunktion **vga_draw_line** enthält die Implementierung des Bresenham-Algorithmus und erhält neben dem Pointer auf das Array des Bildschirm-Pufferspeichers noch die Wertepaare (x,y) des Start- und Endpunkts der Linie. Die Implementierung tauscht vor dem Start der Zeichen-Schleife die Vorzeichen und x/y-Werte für die Anpassung an alle Oktanten. In der Schleife wird der aktuelle Pixel durch Aufruf der Funktion **vga_draw_point** gezeichnet. Der Quellcode ist dem Pseudocode (Anhang A.1) sehr ähnlich, daher hier nicht nochmals gelistet und auf dem beigefügten Datenträger enthalten.

Unterfunktion vga_draw_point:

Die Unterfunktion **vga_draw_point** ist in Quellcode 5.19 gelistet. Sie erhält den Pointer auf das Array des Bildschirm-Pufferspeichers und das x/y-Wertepaar des zu schreibenden Pixels als Funktionsparameter.

```

1 void vga_draw_point(uint32 vgabuf[24576],
2                     int x,
3                     int y) {
4     int index, index2;
5     uint6 bit;
6     uint32 buf, buf2;
7
8     index = (y * 32) + (x / 32);
9     index2 = (y * 32) + (x / 32);
10    bit = (x % 32);
11    buf = vgabuf[index];
12    buf2 = buf | (((uint32)(0x01)) << bit);
13    vgabuf[index2] = buf2;
14 }
```

Quellcode 5.19: Unterfunktion vga_draw_point

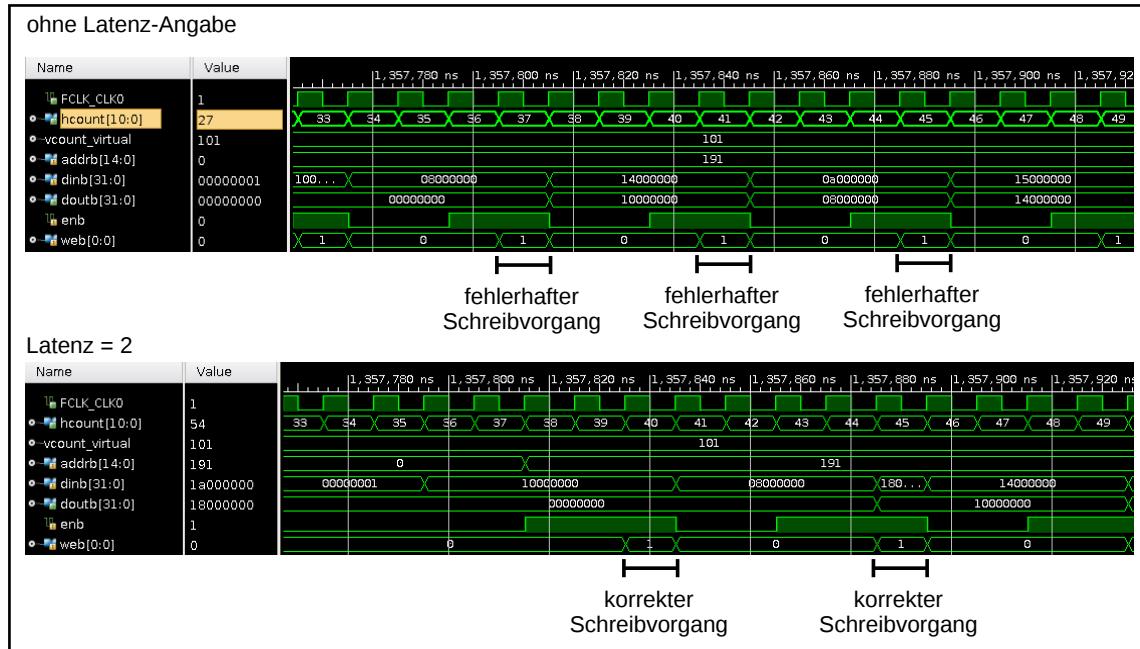


Abbildung 5.5: Schreibmodul: Fehlerhafte und korrekte Direktiven in der Simulation

Quelle: Erstellt mit Xilinx Vivado Design Suite 2016.1

Nach Berechnung des Array-Index in welchem der zu schreibende Pixel enthalten ist, wird dieser 32 Bit-Speicherwert gelesen und zwischengespeichert. Durch Bitshift wird

eine Maske für diesen 32 Bit-Wert erstellt, damit das Bit gesetzt und danach der Wert in das Array zurückgeschrieben. Die Verwendung von zwei Index-Variablen ist experimentell entstanden und behebt einen Darstellungsfehler, dessen Ursache in der Entwicklung dieser Arbeit nicht geklärt wurde.

Wie in Kapitel 5.4.5 beschrieben, ist zusammen mit der Integration des Schreibmoduls in die Top-Level-Funktion `vec_engine` auch die Direktive `HLS RESOURCE variable=vgabuf core=RAM_1P_BRAM latency=2` für die korrekte Funktionalität zu setzen. In Abbildung 5.5 sind die zwei Simulationen des Schreibmoduls dargestellt. Oben ist die fehlerhafte und unten die korrekte Implementierung gezeigt. In der fehlerhaften Implementierung passen der gelesene Array-Wert und die berechnete Bitmaske nicht zusammen. In der Bilddarstellung ist hierdurch jede Linie breiter gezeichnet. Die Pixel eines 32 Bit-Wertes addieren sich durch die falsche ODER-Operation auf. Es werden mehr Pixel geschrieben, als der Bresenham-Algorithmus berechnet. Die genannte Direktive löst diesen Implementierungsfehler. Die Fehler sind im Kopiermodul und Schreibmodul getrennt beschrieben. In der Implementierung sind diese Fehler gleichzeitig aufgetreten und haben die Analyse dadurch wesentlich erschwert. Erst durch Trennung der Module mit Entwicklung getrennter Test- und Simulationsumgebungen konnte der Fehler eingegrenzt und behoben werden, sowie die dargestellten Simulationsergebnisse erarbeitet werden.

5.5 Grafikdemo

Die Implementierung der Grafikdemo erfolgt nach den, im Design festgelegten, Kriterien. Es wurden drei Dateien im PetaLinux-Template angelegt und bearbeitet:

- demo1.c
- gpio_lib.h
- gpio_lib.c

Die erste C-Datei (demo1.c) kompiliert zur ausführbaren Anwendung und durchläuft in ihrer Ausführung eine Endlosschleife (`while(1){};`) mit weiteren inneren Schleifen zur Animation der Objekte. PetaLinux bietet in seiner Standard-Konfiguration die Verbindung zum Linuxsystem über eine USB-UART-Konsole. Mit einem, über USB an das ZyBo angeschlossenen, PC und der Verwendung eines Terminal-Programms (z.Bsp. gtkterm) wird die Anwendung im PetaLinux gestartet. Die C-Datei gpio_lib.c enthält die Übergabe der Objektparameter auf die beschriebenen GPIO-Treiber. Der Quellcode ist auf dem beigefügten Datenträger enthalten.

5.6 Integration in Vivado Design Suite

Die implementierten Bestandteile dieser Arbeit können in der Vivado Design Suite zu einem Blockdesign zusammengefügt und zu einem Bitstream für das ZyBo generiert werden. Die Benutzung der Vivado Design Suite ist nicht Bestandteil dieser Arbeit. Im Anhang A.7 ist das vollständige Blockdesign abgebildet. Die VivadoHLS-Bestandteile sind als IP-Cores aus VivadoHLS exportiert und als solche ins Blockdesign importiert. Das VGA-Generator-Modul ist als VHDL-Modul in der Vivado Design Suite erstellt und direkt ins Blockdesign übernommen. Die BlockRAM-Cores, AXI-GPIO-Cores, der Processor-System-Reset-Core, der Zynq-Processing-System-Core und der AXI-Interconnect-Core sind im IP-Katalog der Design Suite enthalten und daraus in das Blockdesign eingefügt. Die Funktion und Verwendung dieser IP-Cores ist in den entsprechenden Xilinx-Dokumentationen beschrieben.

Die I/O-Pins sind im Blockdesign ganz links (Eingänge) und ganz rechts (Ausgänge) dargestellt. Es ist eine Zuordnung dieser Pins aus dem Blockdesign zu den am Board vorhandenen Anschlüssen (Pmods) notwendig. Diese Zuordnung erfolgt in der Vivado Design Suite durch Constraint-Dateien. Die Constraint-Datei (constraint.xdc) für dieses Projekt ist auf dem beigefügten Datenträger enthalten.

Kapitel 6

Fazit

6.1 Implementierungsziele

6.1.1 Vektorgrafikeinheit

Das Ziel der Entwicklung der Vektorgrafikeinheit (VGE) mit den definierten Eigenschaften wurde erreicht und der gewählte Designansatz vollständig implementiert. Die bemerkenswerten Punkte über die Verwendung von VivadoHLS für diese Implementierung sind im Folgenden zusammengefasst.

Grafik-Pipeline:

Das Design des Grundgerüsts der Grafik-Pipeline ist nach softwaretechnischen Aspekten entwickelt. Die Definition der Pipeline-Schleife und der darin enthaltenen Funktionsaufrufe zur Berechnung der Datenstrukturen wäre auf einem prozessorbasierten System ähnlich. Die Entscheidung für Designansatz 2 (Kapitel 4.2.7) ist aus der Analyse der dafür benötigten FPGA-Ressourcen hervorgegangen. Diese Entscheidung ist als Hardware-Design-Entscheidung zu betrachten und ist daher ein Beispiel wie stark Software- und Hardware-Design trotz der Verwendung von VivadoHLS weiterhin miteinander verknüpft sind. Dies ist ein Indikator dafür, dass ausführbarer C-Code nicht ohne Modifikationen für die Verwendung in FPGAs geeignet ist. Die Verwendung von Pointern in synthetisierbarem C-Code erfordert ein Umdenken von den klassischen Programmierparadigmen für sequentielle Ausführung hin zu ausführlicheren Betrachtungen über die Speicherverwendung. Während sequentieller Code (Threads oder Shared-Memory ausgenommen) die Datenkonsistenz über die gesamte Grafik-Pipeline selbstständig gewährleistet, ist in synthetisierbarem C-Code der zeitliche Ablauf über alle Lese- und Schreibvorgänge gesondert zu designen. Ohne das Design dieser zeitlichen Steuerung korrumptieren die Daten und

fehlerhafte Ergebnisse sind die Folge. Die Synthese der Unterfunktionen als parallel berechnende Module macht dieses Design notwendig.

Berechnungsfunktionen:

Die Berechnungen innerhalb der Unterfunktionen sind mit VivadoHLS einfach zu implementieren. Eingabedaten durch Berechnung in Ausgabedaten zu wandeln ist eine Standard-Anwendung von VivadoHLS und mit vielen Beispielen in den Hersteller-Dokumentationen beschrieben. Die korrekte Implementierung der Datenabhängigkeiten für die rein sequentielle Ausführung der Grafik-Pipeline ist in den Dokumentationen weniger ausführlich dargestellt und wurde durch Testimplementierungen gelöst. Hierfür waren Recherche in Internet-Foren und Sekundärquellen notwendig. Der gewählte Implementierungsansatz mit lokalem Speicher innerhalb der Unterfunktionen ist nicht als optimale Lösung anzusehen und könnte durch weitere Entwicklung eventuell ganz entfallen.

Ausgabe auf X,Y und Z:

Die Ausgabe der Werte für X,Y und Z aus der VGE zu den I/O-Pins war ein einfach zu implementierender Bestandteil. VivadoHLS erkennt die Definition von Pointern auf skalare Funktionsparameter automatisch als Portausgänge. Mit der Angabe einer Direktive wird die Erstellung von Steuerungssignalen zu diesen Portausgängen unterdrückt. Die in der Unterfunktion `draw` berechneten Werte für X,Y und Z werden direkt an die I/O-Pins weitergeleitet. Die zeitlichen Vorgaben für die Ausgabe werden durch die Anzahl der Schleifendurchläufe erreicht. Die Darstellungszeit eines Punkts auf dem Oszilloskop-Bildschirm verhält sich linear zu dieser Anzahl. Die VivadoHLS-Abstraktion zwischen Soft- und Hardwareentwicklung auf C-Code und Direktiven ist für diesen Teil der Implementierung klar getrennt und einfach nachvollziehbar.

Datenübergabe an Rasterisierer:

Die Datenübergabe von der VGE zum Rasterisierer ist durch die Integration des Rasterisierer-Schreibmoduls in die VGE gelöst. Die Lösung ist ein Kompromiss zwischen Entwicklungszeit und Trennung der Implementierungsziele. Eine andere Lösung zur Datenübertragung wäre die Verwendung von DDR3-RAM Speicher, welcher auf dem ZyBo enthalten ist. Dafür wäre ein Design der Speicheranbindung, Synchronisation und Datenkonsistenz notwendig. Dieser Ansatz könnte in zukünftigen Entwicklungen der VGE und des Rasterisierers implementiert werden.

6.1.2 Rasterisierer

Das Design des Rasterisierers mit den Bestandteilen Schreiben, Kopieren und Lesen gibt eine klare Struktur für die Implementierung vor. Die geplante Zeitspanne für die Implementierung (ca. 1/3 der gesamten Implementierungszeit) wurde durch, sich während der Implementierung ergebende, Problemstellungen überschritten. Diese Problemstellungen werden im Folgenden beschrieben.

VGA-Generator:

Die Implementierung des VGA-Generators sollte mit VivadoHLS in C-Code erfolgen. Dieses Modul stellt die Zähler und Synchronisations-Signale für alle Bestandteile des Rasterisierers bereit. Ein zeitlicher Versatz zwischen diesen Signalen führt zu Fehlfunktionen der Module. Es wurden mehrere Implementierungen in VivadoHLS entwickelt. Keine dieser Implementierungen konnte das benötigte Ergebnis erreichen. Daher wurde dieses Modul in VHDL implementiert. Aus allen implementierten Bestandteilen hat dieses Modul den größten Hardware-Bezug und ist eventuell daher nicht für die Implementierung in VivadoHLS geeignet. Es besteht aus dieser Arbeit kein Ansatz, wie diese Problemstellung in VivadoHLS gelöst werden könnte.

BlockRAM-Synchronisation:

Die in Kapitel 5.4.4 und 5.4.5 beschriebenen Simulationen waren zeitaufwändig. Der Arbeitsfluss für diese Simulationen beginnt mit der Synthese der IP-Cores in VivadoHLS. Diese IP-Cores werden in der Vivado Design Suite ins Blockdesign eingefügt und dann gemeinsam in eine Netzliste synthetisiert. Nach dieser Synthese kann eine Simulation erfolgen. Jede Änderung einer Direktive oder C-Codezeile erfordert diese Schritte erneut. Die Timing-Übersichten in VivadoHLS können keine Aussagen über die Integration der IP-Cores liefern, sondern nur über jeden einzelnen IP-Core. Der Zeitaufwand für die beschriebenen Schritte bis zum Ergebnis der Simulation beträgt auf dem verwendeten PC-System (Intel i5, 16Gb RAM) ca. 1 Stunde. In der Simulation müssen mindestens zwei vollständige Bildzyklen berechnet werden, um alle Lese-, Kopier- und Schreibvorgänge analysieren zu können. Das ergibt mehr als eine Million Taktzyklen in der Simulation. Das Auffinden und Bewerten der neuralgisch wichtigen Stellen ist daher auch zeitaufwändig. Die gefundene Lösung (Direktive mit dem Parameter `latency = 2`) erscheint in der Retroperspektive im Vergleich zum beschriebenen Lösungsweg sehr einfach. Es sei angemerkt, dass die Schleife im Kopiermodul (Quellcode 5.15, Zeilen 16-18) trotz ihrer Einfachheit nicht von VivadoHLS in den Standard-Einstellungen als Kopiervorgang zwischen zwei BlockRAM-Speichern erkannt wird. Selbst der Direktiven-Parameter `core=RAM_1P-BRAM` reicht nicht aus, um den Kopiervorgang korrekt zu synthetisieren.

Die Verwendung des Direktiven-Parameters `latency = 2` ist ausschließlich experimentell bestimmt. Für zukünftige VivadoHLS-Implementierungen mit Speichersynchronisation sind andere Vorgehensweisen zu entwickeln und anzuwenden.

6.1.3 Grafikdemo

Auf Grund der zeitaufwändigen Implementierungen der VGE und des Rasterisierers wurde das optionale Ziel auf die nicht-interaktive Grafikdemo festgelegt. Die Verwendung von PetaLinux für die Implementierung der Grafikdemo nutzt die Vorteile der herstellereigenen Distribution (Treiber, App-Templates). Die Übertragung der Objektparameter über das AXI-GPIO-Bussystem ist einfach und schnell zu implementieren, hat aber mit der Anzahl der Objekte (10 Objekte) ihre maximale Übertragungskapazität erreicht. Es werden 640 einzelne GPIOs mit jeweils eigenem Filedescriptor in Linux geöffnet und beschrieben. Eine andere Übertragung unter Verwendung eines seriellen Protokolls oder Datenstrukturen in DDR3-RAM würde die Möglichkeiten der VGE erweitern und weitere Parameter ermöglichen.

6.2 Vivado High Level Synthese

Die Bewertung von VivadoHLS als Werkzeug zur Implementierung der gestellten Ziele wird anhand der Kriterien Zeitaufwand, Machbarkeit und Dokumentation vorgenommen. Die Betrachtungen dieser Kriterien sind in den Implementierungskapiteln und im Fazit der Ziele verteilt und werden im Folgenden zusammengefasst.

Zeitaufwand:

Der Zeitaufwand für Implementierungen in VivadoHLS ist von der Problemstellung abhängig. Berechnungen auf Datenstrukturen nutzen die Abstraktion von VivadoHLS in Software- und Hardware-Entwicklung am Besten aus. Hier kann prozessor-ausführbarer C-Code mit wenig Änderungen in VivadoHLS übernommen und das Hardware-Verhalten mit Direktiven auf den FPGA angepasst werden. Dieser Anwendungsfall ist vom Hersteller Xilinx als Hardware-Beschleunigungs-Feature vorgesehen und entspricht in dieser Arbeit der Implementierung der VGE-Unterfunktionen. Weicht die Implementierung von diesem Ideal-Anwendungsfall ab, kann der Zeitaufwand steigen. FPGA-Programmierungen mit starkem Hardware-Bezug und taktgenauen Synchronisationen sind zeitaufwändige Implementierungen in VivadoHLS. Der Implementierungsaufwand des Rasterisierers wurde in der Zielsetzung und im Design unterschätzt. Die Timing- und Ressourcen-Übersichten

in VivadoHLS sind für die Erstellung einzelner IP-Cores ausreichend. Muss das Zusammenspiel vieler Komponenten analysiert werden, muss die Integration in der Vivado Design Suite und alle daraus folgenden Zeitaufwände mit berücksichtigt werden. Der Aufwand der Implementierung mit VivadoHLS ist dann ähnlich dem einer Implementierung in Hardware-Beschreibungssprachen.

Machbarkeit:

Außer dem VGA-Generator-Modul sind die Bestandteile der VGE und des Rasterisierers in VivadoHLS implementiert. Das Ziel der Verwendung von VivadoHLS als Werkzeug für diese Arbeit ist damit fast vollständig erreicht. Die Probleme in der Synchronisation der Rasterisierer-Module haben zu erheblichem Zeitaufwand geführt, aber nicht die Machbarkeit in Frage gestellt. Ob jedes denkbare Implementierungsziel mit VivadoHLS erreicht werden kann, ist nicht durch diese Arbeit zu bestimmen. Es ist anzunehmen, dass für einige Problemstellungen einfachere Lösungswege existieren, aber nicht gefunden wurden. Um die Machbarkeit und den Zeitaufwand einer Implementierung abschätzen zu können, ist ausführliches Wissen über VivadoHLS und FPGA-Programmierung in Hardware-Beschreibungssprachen notwendig. Die Abstraktion von VivadoHLS reicht nicht aus, um eine Implementierung ausschließlich von der Software-Seite zu betrachten und die Hardware-Seite nachträglich durch Direktiven zu applizieren.

Dokumentation:

Der Hersteller Xilinx stellt sehr umfangreiche Dokumentationen bereit. Dies ist grundsätzlich als positiv für den Entwickler zu betrachten. Durch die große Menge an Dokumentationen sind die gesuchten Informationen oft auf mehrere Dokumente verteilt. Trotz intensiver Recherche wurden manche Sachverhalte nicht in den Dokumentationen gefunden. Die Beschreibung der Direktiven ist über die gesamte VivadoHLS-Dokumentation verteilt und nicht zentral an einer Stelle beschrieben. Die in den Dokumentationen enthaltenen Beispiele sind an Ideal-Szenarien (z.Bsp. Matrizen-Berechnungen) orientiert und helfen in speziellen Problemfällen (Rasterisierer) nicht weiter.

6.3 Ausblick

In der Recherche zu dieser Arbeit hat sich gezeigt, dass Vektorgrafik auf FPGAs bisher nur in sehr grundlegenden Implementierungen als Open-Source-Projekte verfügbar sind. Eine mehrfach beschriebene Implementierungsaufgabe ist die Analog-Uhr auf einem Oszilloskop-Bildschirm. Aber selbst in diesen 2-dimensionalen Vektorgrafiken sind

die meisten Uhrzeiger durch punktweise Ansteuerung einer Linie dargestellt. Eine Implementierung, welche die Eigenschaft von Vektorbildschirmen zum kontinuierlichen Zeichen vollständig ausnutzt, wurde nicht gefunden. Der Quellcode dieser Arbeit wird unter der GPLv3-Lizenz veröffentlicht und der Autor trägt die Hoffnung auf Benutzung und Weiterentwicklung durch die FPGA-Entwickler-Gemeinschaft.

Folgende Punkte könnten in zukünftigen Erweiterungen oder Modifikationen aufgegriffen und entwickelt werden:

Echtes Datenpipelining:

Die Berechnungen innerhalb der Grafik-Pipeline sind größtenteils sequentiell und könnten als echte Daten-Pipeline designed und implementiert werden. Die Berechnungsergebnisse müssten dazu von Modul zu Modul als Datenstrom (Stream) angelegt werden.

Maximale Objektanzahl:

Durch Optimierungen der Datentypen und Darstellungszeiten könnten mehr als 10 Objekte von der VGE dargestellt werden. Hierzu müsste auch die Übertragung der Objektparameter aus der PetaLinux-Anwendung neu gestaltet werden. Die Verwendung von DDR3-Speicher zur Übertragung ist eine solche Option.

Mehr verschiedene Objekte:

Die zwei enthaltenen Objekt-Initialisierungsfunktionen sind Beispiele für die Erstellung von Objekten in der VGE. Es könnten weitere Initialisierungs-Funktionen geschrieben werden. Weitere Datenstrukturen für nicht-würfelähnliche Objekte (Andere Anzahl von Eckpunkten, Kanten und Flächen) könnten erstellt werden. Der Objektparameter `cat_object` hat 4 Bit Datenbreite und könnte 16 verschiedene Objekte unterscheiden.

Datenübergabe von der VGE zum Rasterisierer:

Die Trennung der VGE und des Rasterisierers wäre eine weitere Zielsetzung für zukünftige Arbeiten. Eine Datenübergabe in BlockRAM oder DDR3-Speicher ist machbar und wäre der Übersichtlichkeit und Struktur des Projekts zuträglich.

Spiel:

Das nicht implementierte Spiel (Optionales Ziel) ist nur wenige Implementierungs-Schritte entfernt. Die Grafikdemo mit ihrer GPIO-Ansteuerung ist eine verwendbare Vorlage hierfür. Die Benutzereingaben könnten durch einen an I/O-Pins angeschlossenen Joystick (und Button) erfolgen. An die ARM-CPUs sind eigene I/O-Ports verbunden, diese sind von PetaLinux aus erreichbar und könnten hierfür benutzt werden.

Kapitel 7

Literaturverzeichnis

- [AA88] ARDENNE, M. ; ARDENNE, M. von: *Sechzig Jahre für Forschung und Fortschritt: Autobiographie.* Verlag der Nation, 1988 <https://books.google.de/books?id=WBEfAQAAIAAJ>
- [Ash08] ASHENDEN, Peter J.: *The student's guide to VHDL.* 2. ed. Amsterdam [u.a.] : Elsevier, 2008 <http://www.gbv.de/dms/bowker/toc/9781558608658.pdf>. – ISBN 1558608656
- [BB06] BENDER, Michael ; BRILL, Manfred: *Computergrafik : ein anwendungsorientiertes Lehrbuch.* 2., überarb. Aufl. München [u.a.] : Hanser, 2006 http://scans.hebis.de/HEBCGI/show.pl?13466683_toc.pdf. – ISBN 9783446404342
- [Bra97] BRAUN, Ferdinand: Ueber ein Verfahren zur Demonstration und zum Studium des zeitlichen Verlaufs variabler Ströme. In: *Annalen der Physik und Chemie* 60 (1897), S. 552–559
- [Chu08] CHU, Pong P.: *FPGA prototyping by VHDL examples : Xilinx Spartan-3 version.* [Nachdr.]. Hoboken, NJ : Wiley, 2008. – ISBN 9780470185315
- [Edw12] EDWARDS, Stehen A.: *Bresenhams Line Algorithm in Hardware.* Präsentationsfolien, 2012. – Online erhältlich unter <http://www.cs.columbia.edu/~sedwards/classes/2012/4840/lines.pdf>; abgerufen am 24. September 2016.
- [Fis15] FISCHER, Martin: Anschluss-Kuddelmuddel entwirren: DVI, HDMI, DisplayPort. In: *c't 2* (2015), S. 148–150

- [Ges14] GESSLER, Ralf: *Entwicklung Eingebetteter Systeme : Vergleich von Entwicklungsprozessen für FPGA- und Mikroprozessor-Systeme*. Wiesbaden : Springer Vieweg, 2014 (Lehrbuch). http://scans.hebis.de/HEBCGI/show.pl?34247102_toc.pdf. – ISBN 9783834813176
- [GNU] GNU, Free Software F.: *GNU General Public License*. <https://www.gnu.org/licenses/licenses.html>
- [Gä08] GÄRTNER, Armin: LCD-Monitore -Teil 1: Grundlagen und Technologie. In: *mt-Medizintechnik* 2 (2008), S. 54–66
- [IBM92] IBM ; IBM (Hrsg.): *VGA XGA Technical Reference Manual*. IBM Corporation, 1 New Orchard Road, New York, United States: IBM, 1992
- [Inc] <https://www.xilinx.com/products/silicon-devices/soc.html>
- [Kar84] KAROLUS, Hildegard (Hrsg.): *August Karolus : die Anfänge des Fernsehens in Deutschland in Briefen, Dokumenten und Veröffentlichungen aus seiner Zusammenarbeit mit der Telefunken GmbH, Berlin 1923 - 1930*. Berlin [u.a.] : VDE-Verlag, 1984. – ISBN 3800713721
- [KSW04] KORIES, Ralf ; SCHMIDT-WALTER, Heinz: *Taschenbuch der Elektrotechnik : Grundlagen und Elektronik*. 6., erw. Aufl. Frankfurt am Main : Deutsch, 2004 http://scans.hebis.de/HEBCGI/show.pl?12302444_toc.pdf. – ISBN 3817117345
- [MR13] MOLITOR, Paul ; RITTER, Jörg: *Kompaktkurs VHDL : mit vielen anschaulichen Beispielen*. München : Oldenbourg, 2013 <http://d-nb.info/1018815252/04>. – ISBN 9783486712926
- [Pap] PAPON, Charles: *SpinalHDL - An Open Source HDL*. <http://spinalhdl.github.io/SpinalDoc/>
- [Par13] PAROBEK, Lucas: *Research, Development and Testing of a Fault-tolerant FPGA-based Sequencer for CubeSAT Launching Applications*, Naval Postgraduate School, Monterey, Diplomarbeit, 2013
- [Pau99] PAUL, Reinhold: *Elektrotechnik und Elektronik für Informatiker : 1. Grundgebiete der Elektrotechnik*. 2., durchges. Aufl. Stuttgart : Teubner, 1999 http://scans.hebis.de/HEBCGI/show.pl?08199400_toc.pdf. – ISBN 3519121263

- [Sau10] SAUER, P.: *Hardware-Design mit FPGA : Eine Einführung in den Schaltungsentwurf mit FPGA-Bausteinen.* Aachen : Elektor-Verl., 2010 http://deposit.d-nb.de/cgi-bin/dokserv?id=3462341&prov=M&dok_var=1&dok_ext=htm. – ISBN 3895762091
- [SF78] SESSIONS, Kendall W. ; FISCHER, Walter A.: *Understanding oscilloscopes and display waveforms.* New York [u.a.] : Wiley, 1978. – ISBN 0471026255
- [UK11] UK, Raytheon C.: *Celebrating 100 years of Raytheon in the UK.* Website, 2011. – Online erhältlich unter http://www.raytheon.co.uk/rtnwcm/groups/rsl/documents/content/rsl_business_brochure.pdf; abgerufen am 25. August 2016.
- [Wan98] WANNEMACHER, Markus: *Das FPGA-Kochbuch.* 1. Aufl. Bonn [u.a.] : International Thomson Publishing Company, 1998 http://scans.hebis.de/HEBCGI/show.pl?09339144_toc.pdf. – ISBN 3826627121
- [Wol] WOLF, Clifford: *Yosys - An Open Source HDL Toolchain.* <http://www.clifford.at/yosys>
- [Xil16a] XILINX ; XILINX INC. (Hrsg.): *PetaLinux Tools Documentation: Reference Guide, UG1144.* Xilinx Inc., 2100 Logic Drive, San Jose, CA 95124-3400: Xilinx Inc., 2016
- [Xil16b] XILINX ; XILINX INC. (Hrsg.): *Vivado Design Suite User Guide: High Level Synthesis, UG902.* Xilinx Inc., 2100 Logic Drive, San Jose, CA 95124-3400: Xilinx Inc., 2016
- [XP07] XIANG, Zhigang ; PLASTOCK, Roy A.: *UTB.* Bd. 8378 : Schaum's Reptitorien: *Computergrafik : Einführung in die theoretischen Grundlagen.* [Nachdr. der 1. Aufl. 2003]. [Bonn] : mitp, 2007 http://deposit.d-nb.de/cgi-bin/dokserv?id=2967396&prov=M&dok_var=1&dok_ext=htm. – ISBN 9783825283780

Abbildungsverzeichnis

2.1	Oszilloskop-Bildröhre	3
2.2	Koordinatensystem und Strukturen	5
2.3	Projektionsarten mit Flächenverdeckung (Backface Culling)	7
2.4	Vektorgrafik-Pipeline	8
2.5	VGA-Schnittstelle mit Signalen	8
2.6	VGA-Signale: Rot, Grün, Blau, H-Sync, V-Sync	9
2.7	Oktanten, Linienstart- und Linienendpunkt	10
2.8	Folgepunktbestimmung im Bresenham-Algorithmus	10
2.9	Schematischer Aufbau eines FPGA-Chips	12
2.10	Schema eines FPGA-Logikblocks	13
2.11	FPGA-Workflow	16
2.12	Simulation und Blockbild des synthetisierten Addierers in Xilinx Vivado .	18
4.1	ZyBo-Board: Schnittstellen und Ressourcen	36
4.2	Index-Definitionen der Punkte, Vektoren und Flächen eines 3-D-Objekts .	38
4.3	Module zur Initialisierung von Würfeln und Pyramidenstümpfen	39
4.4	Module zur Transformation von Objekten	40
4.5	Module: toSurface und projection	41
4.6	Modul: draw	41
4.7	Grafik-Pipeline mit Objekt-Schleife	43
4.8	X- oder Y-Ausgang	44
4.9	Vergleich: Rechteck- und Dreieckspannungen als X-Y-Eingänge	44
4.10	Schaltungsbestandteile der X- und Y-Ausgänge	45

4.11 Schaltungsbestandteile des Z-Ausgangs	46
4.12 Komparator und StepUp-Modul des Z-Ausgangs	46
4.13 Modul: VGA-Generator	47
4.14 VGA-Generator: Horizontale und vertikale Bildzähler	48
4.15 Modul: VGA-Lesen	49
4.16 Modul: VGA-Kopieren	49
4.17 Modul: VGA-Schreiben	50
4.18 Synchronisation der Rasterisierer-Module auf die Zähler H-count und V-count	51
4.19 Rasterisierer: Alle Module	52
4.20 Schema des Hybrid-FGPA mit Grafikdemo-Applikation	52
 5.1 Eingangskanäle der Objekt-Parameter	57
5.2 Struktogramm des Schleifen-Ausführungsblocks in der draw-Funktion	66
5.3 BlockRAM mit Portsignalen: vga-buffer	68
5.4 Kopiermodul: Fehlerhafte und korrekte Direktiven in der Simulation	72
5.5 Schreibmodul: Fehlerhafte und korrekte Direktiven in der Simulation	75
 A.1 Vectrex-Spielkonsole, 1982, Hersteller MB, Foto auf der Gamescom-Messe 2016 von T.Knoll	97
A.2 Implementierungs-Schema des Addierers in Xilinx Vivado	99
A.3 Anzahl der Ressourcen eines Objekts in der Grafik-Pipeline	101
A.4 Test auf die maximal darstellbare Objektanzahl ohne Grafikpipeline	101
A.5 Design der gesamten Grafik-Pipeline	102
A.6 Verwendete Direktiven	103
A.7 Blockdesign der kompletten Implementierung	110

Tabellenverzeichnis

2.1	Definition der Strukturen	5
2.2	Kartesische und homogene Koordinaten	6
2.3	Transformationsmatrizen	7
2.4	VGA-Signal-Timings	9
3.1	Aufgabenpakete der Vektorgrafikeinheit	31
3.2	Aufgabenpakete des Rasterisierers	32
3.3	Aufgabenpakete der Bewertung	33
3.4	Aufgabenpakete der Grafikdemo	34
4.1	Parameter zur Objektsteuerung	38

Verzeichnis der Quellcodes

2.1	VHDL-Quellcode eines Addierers	17
2.2	VHDL-Quellcode eines Prozesses	19
2.3	VHDL-Quellcode einer bedingten Verzweigung	19
2.4	VHDL-Quellcode: Switch-Case	19
2.5	VHDL-Quellcode einer For-Schleife	20
2.6	VHDL-Quellcode: Zuweisungen zu Signalen und Variablen	20
2.7	VHDL-Quellcode: Takt-Synchronisation	21
2.8	C-Quellcode eines Addierers (HLS synthesisierbar)	22
2.9	HLS-Port-Direktiven	26
2.10	HLS-Struktur-Direktiven	26
4.1	C-Header mit Datenstrukturen (C-Structs) für die Vektorgrafikeinheit	37
4.2	Bildschirmspeicher-Array in C	48
5.1	Funktionsprototyp der Top-Level-Funktion vec_engine	57
5.2	Instanzen der Objekt- und Parameter-datenstrukturen für die Grafik-Pipeline	57
5.3	Befüllen der Objekt-Parameter-Arrays	58
5.4	Grundgerüst der Pipeline-Schleife	59
5.5	Prototypen der Unterfunktionen	61
5.6	Funktion cube_init: Zuweisung des ersten Würfeleckpunkts	62
5.7	Funktion translate: Verschiebung des Mittelpunkts und der Eckpunkte	62
5.8	Funktion rotate: Rotation des Objekts um die X-Achse	63
5.9	Funktion crossproduct: Berechnung der Flächennormal-Vektoren (Vektor-Kreuzprodukt)	64
5.10	Funktion projection: Berechnung der Sichtbarkeit der Objektflächen	65
5.11	Funktion projection: Projektion des Objekts auf die 2-D-Projektionsfläche	65
5.12	Funktionsprototyp: draw	66
5.13	Rasterisierer-Funktion: vga_out	69
5.14	Direktiven für die Portsignale des Arrays vgaram_in	70
5.15	Rasterisierer-Funktion: vga_copy	70

5.16 Zusätzliche Funktionsparameter in der Top-Level-Funktion der Vektorgrafikeinheit	73
5.17 Zusätzlicher Quellcode in der Grafik-Pipeline	73
5.18 Geschachtelte Schleifen in der Rasterisierer-Funktion vga_draw	74
5.19 Unterfunktion vga_draw_point	75
A.1 VHDL-Code des Addierers (Quelle: Xilinx VivadoHLS)	100
A.2 Installationsanleitung Xilinx-Werkzeuge	104
A.3 VGA-Generator: VHDL-Implementierung	108

Verzeichnis der Algorithmen

A.1 Bresenham-Pseudocode	98
------------------------------------	----

Anhang A

Anhang



Abbildung A.1: Vectrex-Spielkonsole, 1982, Hersteller MB, Foto auf der Gamescom-Messe 2016 von T.Knoll

Algorithmus A.1 Bresenham-Pseudocode

Require: Startpoint x_0, y_0 . Endpoint x_1, y_1

```
dx ← abs( $x_1 - x_0$ )
dy ← abs( $y_1 - y_0$ )
if  $x_0 < x_1$  then
    sx ← 1
else
    sx ← -1
end if
if  $y_0 < y_1$  then
    sy ← 1
else
    sy ← -1
end if
error ← dx - dy
loop
    setPixel( $x_0, y_0$ )
    if  $x_0 = x_1$  and  $y_0 = y_1$  then
        exit loop
    end if
    error2 ← 2 * error
    if error2 > -dy then
        error ← error - dy
         $x_0 \leftarrow x_0 + sx$ 
    end if
    if error2 < dx then
        error ← error + dx
         $y_0 \leftarrow y_0 + sy$ 
    end if
end loop
```

Quelle: [Edw12]

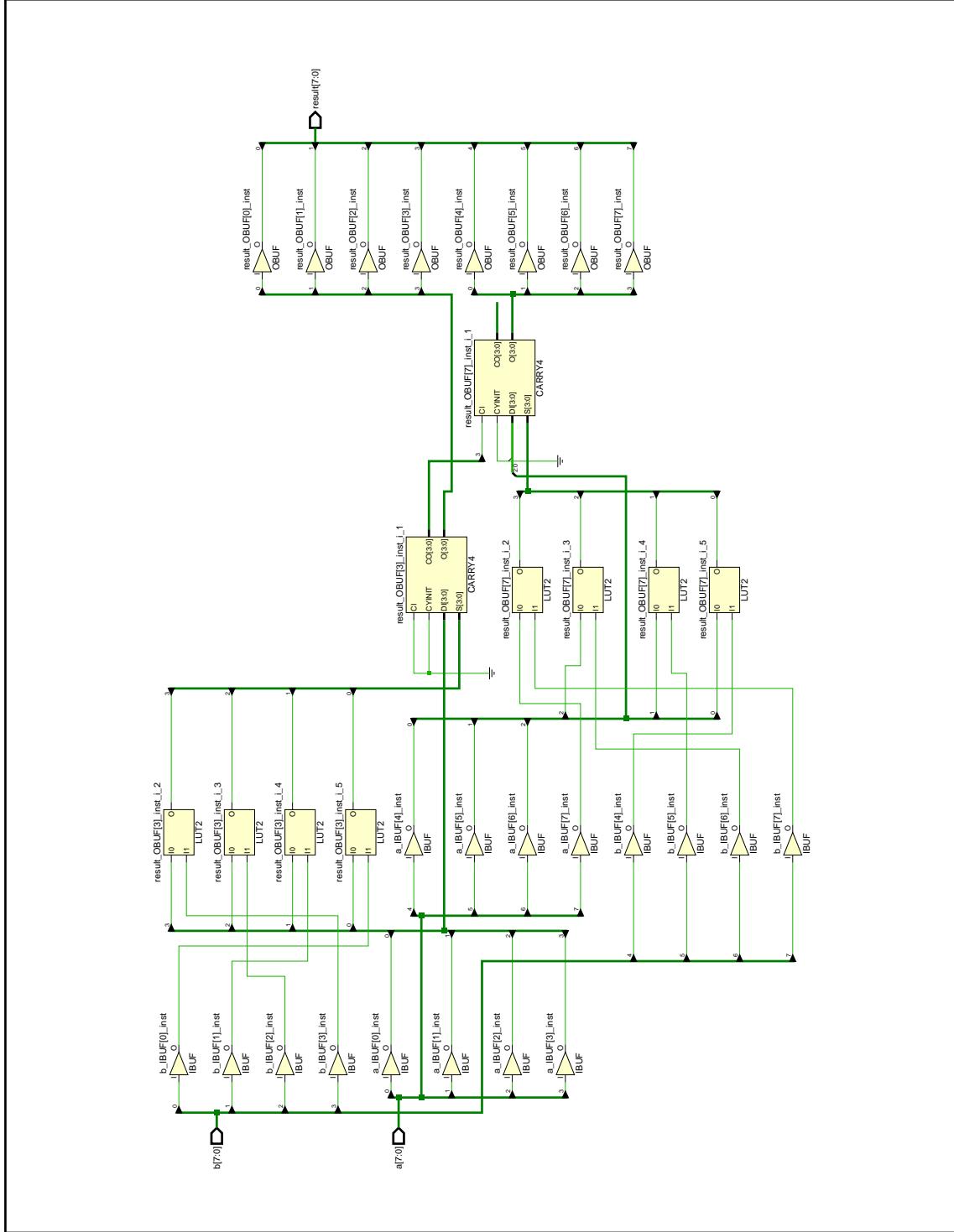


Abbildung A.2: Implementierungs-Schema des Addierers in Xilinx Vivado

Quelle: Xilinx Vivado Design Suite 2016.1

```
1 -- =====
2 -- RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++
3 -- and SystemC
4 -- Version: 2016.1
5 --
6 -- =====
7
8 library IEEE;
9 use IEEE.std_logic_1164.all;
10 use IEEE.numeric_std.all;
11
12 entity addierer is
13 port (
14     a : IN STD_LOGIC_VECTOR (7 downto 0);
15     b : IN STD_LOGIC_VECTOR (7 downto 0);
16     result : OUT STD_LOGIC_VECTOR (7 downto 0);
17     result_ap_vld : OUT STD_LOGIC );
18 end;
19
20 architecture behav of addierer is
21     attribute CORE_GENERATION_INFO : STRING;
22     attribute CORE_GENERATION_INFO of behav : architecture is
23     "addierer, hls_ip_2016_1, {HLS_INPUT_TYPE=c, HLS_INPUT_FLOAT=0,
24     HLS_INPUT_FIXED=1,
25     HLS_INPUT_PART=xc7z010clg400-1, HLS_INPUT_CLOCK=10.000000,
26     HLS_INPUT_ARCH=others, HLS_SYN_CLOCK=1.720000, HLS_SYN_LAT=0,
27     HLS_SYN_TPT=none,
28     HLS_SYN_MEM=0, HLS_SYN_DSP=0, HLS_SYN_FF=0, HLS_SYN_LUT=8 } ";
29     constant ap_true : BOOLEAN := true;
30     constant ap_const_logic_0 : STD_LOGIC := '0';
31     constant ap_const_logic_1 : STD_LOGIC := '1';
32
33 begin
34     result <= std_logic_vector(unsigned(b) + unsigned(a));
35     result_ap_vld <= ap_const_logic_1;
36 end behav;
```

Quellcode A.1: VHDL-Code des Addierers (Quelle: Xilinx VivadoHLS)

Anhang A. Anhang

Instance						
Instance	Module	BRAM_18K	DSP48E	FF	LUT	
grp_vec_engine_cube_init_fu_472	vec_engine_cube_init	1	0	476	676	
grp_vec_engine_draw_fu_500	vec_engine_draw	2	0	154	291	
grp_vec_engine_projection_fu_435	vec_engine_projection	2	16	1139	1146	
grp_vec_engine_pyramid_init_fu_487	vec_engine_pyramid_init	0	0	303	508	
grp_vec_engine_rotate_fu_404	vec_engine_rotate	4	30	4738	9849	
grp_vec_engine_to_surfaces_fu_428	vec_engine_to_surfaces	2	24	2206	2162	
grp_vec_engine_translate_fu_452	vec_engine_translate	0	0	468	706	
grp_vec_engine_vga_draw_fu_444	vec_engine_vga_draw	1	0	597	1498	
Total		8	12	70 10081	16836	

DSP48						
Memory						
Memory	Module	BRAM_18K	FF	LUT	Words	Bits
list_4a_surfaces_points_xyz_U	vec_engine_draw_result_surfaces_points_xyz	2	0	0	72	32
list_4a_normals_xyz_U	vec_engine_list_4a_normals_xyz	0	64	9	18	32
list_5a_visible_U	vec_engine_projection_result_visible	0	2	1	6	1
list_1a_points_xyz_U	vec_engine_translate_result_points_xyz	0	64	12	24	32
list_2a_points_xyz_U	vec_engine_translate_result_points_xyz	0	64	12	24	32
list_3a_points_xyz_U	vec_engine_translate_result_points_xyz	0	64	12	24	32
list_5a_surfaces_points_xyz_U	vec_engine_vga_draw_result_surfaces_points_xyz	1	0	0	72	32
Total		7	3 258	46	240 193	7
						7494

Abbildung A.3: Anzahl der Ressourcen eines Objekts in der Grafik-Pipeline

Quelle: Xilinx VivadoHLS 2016.1

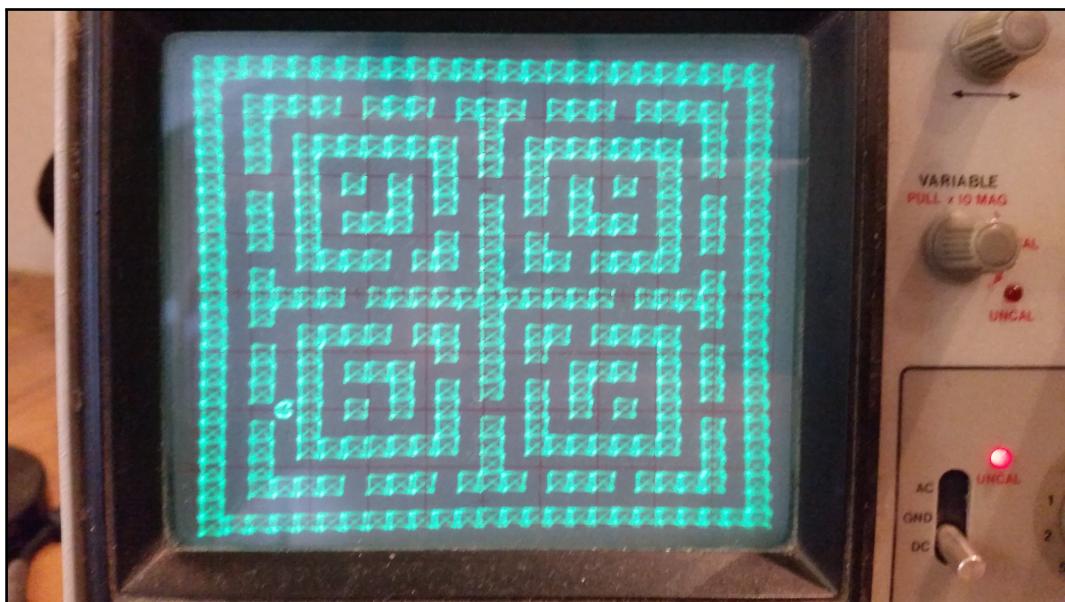


Abbildung A.4: Test auf die maximal darstellbare Objektanzahl ohne Grafikpipeline

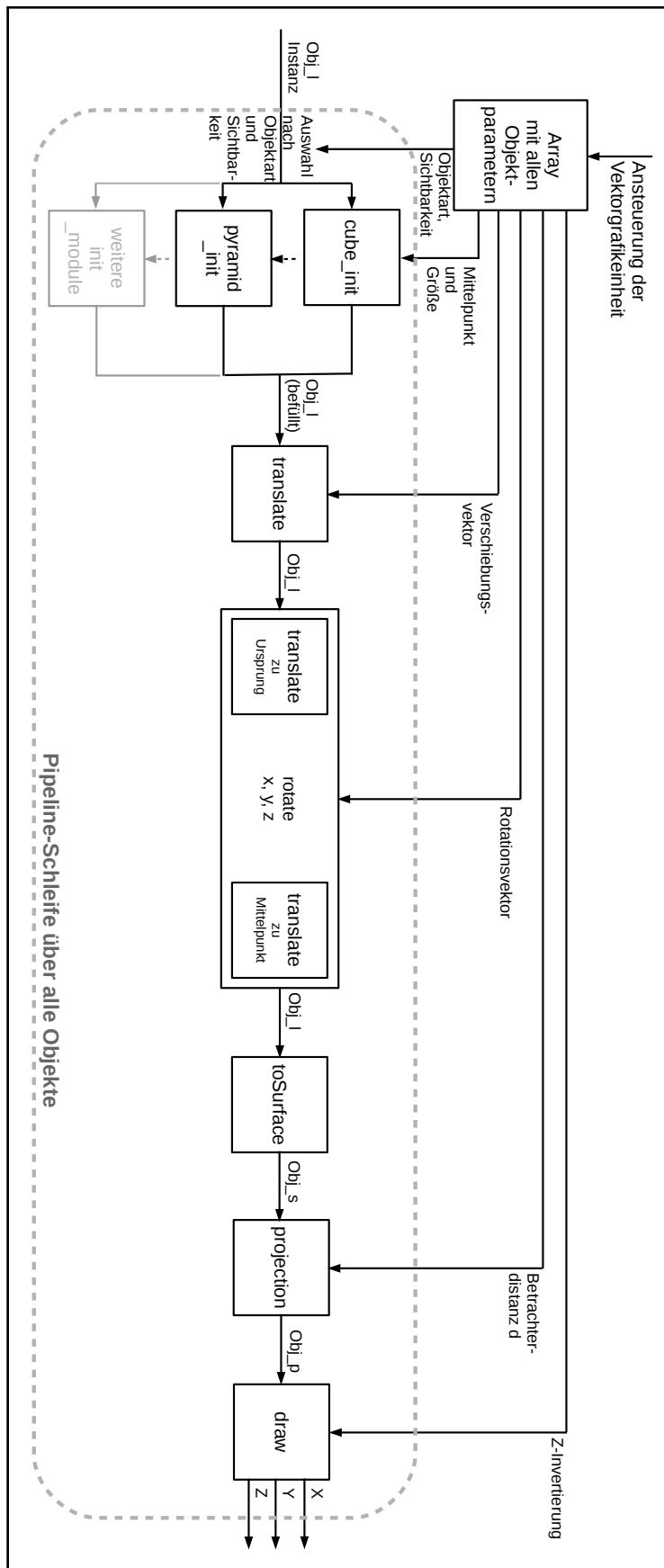


Abbildung A.5: Design der gesamten Grafik-Pipeline

Verwendete VivadoHLS Direktiven:

Direktive:	INTERFACE
Parameter:	port = return ap_ctrl_none
Beschreibung:	Wirkt auf den ganzen Port der Funktion (port=return). Die Erstellung von Steuerungssignalen (valid, ready, enable) wird unterdrückt.
Direktive:	INTERFACE
Parameter:	port = <Funktionsparameter> ap_ctrl_none
Beschreibung:	Wirkt auf den Funktionsparameter eines Funktionsprototypen. Die Erstellung von Steuerungssignalen (valid, ready, enable) wird unterdrückt.
Direktive:	INTERFACE
Parameter:	port = <Funktionsparameter> ap_memory
Beschreibung:	Wirkt auf den Funktionsparameter eines Funktionsprototypen. Die erstellten Ports signale werden nicht zu einem Strang zusammengefasst und zur Verwendung mit BlockRAM-IP-Cores erstellt.
Direktive:	RESOURCE
Parameter:	variable = <Variablenname> oder <Funktionsparameter> core = RAM_1P_LUTRAM
Beschreibung:	Wirkt auf die angegebene Variable. Der Parameter core = RAM_1P_LUTRAM bestimmt die Erstellung von verteiltem Speicher für diese Variable. Oder: Wirkt auf den angegebenen Funktionsparameter. Die Ports signale werden zur Verwendung mit verteiltem Speicher erstellt.
Direktive:	RESOURCE
Parameter:	variable = <Variablenname> oder <Funktionsparameter> core = RAM_1P_BRAM latency = 2
Beschreibung:	Wirkt auf die angegebene Variable. Der Parameter core = RAM_1P_BRAM bestimmt die Erstellung von BlockRAM für diese Variable. Oder: Wirkt auf den angegebenen Funktionsparameter. Die Ports signale werden zur Verwendung mit der Speicherart BlockRAM erstellt. Zusätzlich gibt der Parameter latency = 2 die Latenz des zu erstellenden oder zu verwendenden Speichers an.

Abbildung A.6: Verwendete Direktiven

```
1 INSTALLATION OF THE XILINX TOOLCHAIN
2
3 IMPORTANT:
4 The steps need to be done in the given order!
5
6 0. PREREQUISITES:
7 -----
8 OS: Ubuntu >= 14.04.3
9
10 Installation Files for:
11 - Xilinx SDSoc 2016.1
12 - Xilinx Vivado Suite 2016.1
13 - Xilinx Petalinux-Tools 2016.1
14
15 Licenses for
16 - SDSoc 2016.1
17 - Vivado 2016.1
18
19
20 1. CHANGE SHELL FROM 'DASH' to 'BASH':
21 -----
22 a) Change Shell:
23 chsh -s /bin/bash <user>
24 sudo dpkg-reconfigure dash
25 Configure Screen -> NO
26
27 b) Logout the user and login again to make the change.
28
29 2. SDSOC INSTALLATION (UG1028 version:2016.1):
30 -----
31 a) Install i386 Libraries:
32 sudo dpkg --add-architecture i386
33 sudo apt-get update
34 sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
35 sudo apt-get install libgtk2.0-0:i386 dpkg-dev:i386
36 sudo ln -s /usr/bin/make /usr/bin/gmake
37
38 b) Install lib32 Libraries:
39 sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0
40
41 c) Install SDSoc:
42 chmod 777 -R <install-directory>
43 ./xsetup
44
45 d) Install License or Point to License-Server
```

```
46
47 e) Open SDSoc and check for 'no-errors':
48 In SDSOC Log (Window right lower corner) should be no error.
49 The log should look like this:
50 10:37:46 INFO    : Launching XSDB server: xsdb -n -interactive <path to
      workspace>
51 10:37:47 INFO    : XSDB server has started successfully.
52 10:37:52 INFO    : Validating SDSoc License
53 10:37:52 INFO    : License available for SDSoc
54
55 f) Try to create a new SDSoc-Project and check
56 if the build-in platforms are available (zybo, zedboard, ...).
57
58 2. VIVADO SUITE INSTALLATION (UG973 version:2016.1):
59 -----
60 a) Install Vivado:
61 chmod 777 -R <install directory>
62 ./xsetup
63
64 b) Settings in the Install-Screens:
65 - Include SDK to install
66 - If DocNav was already installed with SDSoc, then leave it out
67 - No needs for Ultra-Scale+MPSoC-Installations
68 - Zynq-7000 + 7 Series is needed
69 - Change the default-Install-Directory
70
71 c) Install License or Point to License-Server
72
73 d) Install the Board-Definition-Files from the Board-Vendor:
74 Digilent Boards:
    https://github.com/Digilent/vivado-boards/archive/master.zip
75 Copy the directories to
    <vivado-install-dir>/2016.1/data/boards/board_files
76
77 e) A good point to restart Linux !! (The Boards didn't show up for me
      till reboot)
78
79 f) Open Vivado and check for the new boards. Do this by creating a new
      project.
80 The last create-screen shows the devices and boards.
81
82 3. PETALINUX INSTALLATION (UG1144 version:2016.1):
83 -----
84 a) Install Libraries:
85 sudo apt-get install tofrodos iproute2 gawk gcc git make
86 sudo apt-get install net-tools libncurses5-dev zlib1g-dev libssl-dev
```

```

87 sudo apt-get install flex bison libselinux1
88
89 b) Get an tftp-server running:
90 sudo apt-get install xinetd tftpd tftp
91
92 Create a file: /etc/xinetd.d/tftp with the folling content:
93
94 service tftp
95 {
96 protocol          = udp
97 port              = 69
98 socket_type      = dgram
99 wait              = yes
100 user             = nobody
101 server           = /usr/sbin/in.tftpd
102 server_args      = /tftpboot
103 disable          = no
104 }
105
106 Create a folder /tftpboot (path matching to the server_args above):
107
108 sudo mkdir /tftpboot
109 sudo chmod -R 777 /tftpboot
110 sudo chown -R nobody /tftpboot
111
112 Restart the xinetd service:
113
114 sudo service xinetd restart
115
116 c) Change Shell to bash, if not already done. (See Step 1.)
117
118 d) Install Petalinux Tools:
119 chmod 777 petalinux-v2016.1-final-installer.run
120 ./petalinux_v2016.1-final-installer.run <path-to-install-directory>
121
122 e) Source Vivado and Petalinux-Tools:
123 source <path-to-vivado-settings64.sh>
124 source <path-to-petalinux-settings.sh>
125
126 With sourcing petalinux, the following should show up:
127
128 PetaLinux environment set to '<path-to-petalinux>'
129 INFO: Checking free disk space
130 INFO: Checking installed tools
131 INFO: Checking installed development libraries
132 INFO: Checking network and other services

```

```
133
134 f) Check the version of the arm-toolchain:
135 arm-linux-gnueabihf-g++ -v
136
137 The last line should look like this:
138 gcc version 4.9.2 20140904 (prerelease) (crosstool-NG
139     linaro-1.13.1-4.9-2014.09 -
140 Linaro GCC 4.9-2014.09)
141
142 g) Check the environment-path:
143 echo $PETALINUX
144 -----
145 --END OF FILE--
146 -----
```

Quellcode A.2: Installationsanleitung Xilinx-Werkzeuge

```

1 entity vga_sync is
2   Port ( clk_in : in STD_LOGIC;
3         hcount : out STD_LOGIC_VECTOR(10 downto 0);
4         vcount : out STD_LOGIC_VECTOR(9 downto 0);
5         hsync : out STD_LOGIC;
6         vsync : out STD_LOGIC );
7 end vga_sync;
8
9 architecture Behavioral of vga_sync is
10
11 signal h_count: integer range 0 to 1344:=0;
12 signal v_count: integer range 0 to 806:=0;
13
14 begin
15
16   process (clk_in, h_count, v_count)
17   begin
18
19     -- Start at the rising edge of the clock
20     if (clk_in'event AND clk_in='1') then
21
22       -- Count h_count and v_count
23       if (h_count < 1343) then
24         h_count <= h_count + 1;
25       else
26         h_count <= 0;
27         if (v_count < 805) then
28           v_count <= v_count + 1;
29         else
30           v_count <= 0;
31         end if;
32       end if;
33
34       -- Generate h_sync
35       if (h_count > 1047 AND h_count < 1084) then
36         hsync <= '1';
37       else
38         hsync <= '0';
39       end if;
40
41       -- Generate v_sync
42       if (v_count > 770 AND v_count < 777) then
43         vsync <= '1';
44       else
45         vsync <= '0';

```

```
46      end if;
47
48      end if;
49
50      hcount <= STD_LOGIC_VECTOR(to_unsigned(h_count, 11));
51      vcount <= STD_LOGIC_VECTOR(to_unsigned(v_count, 10));
52
53  end process;
54
55 end Behavioral;
```

Quellcode A.3: VGA-Generator: VHDL-Implementierung

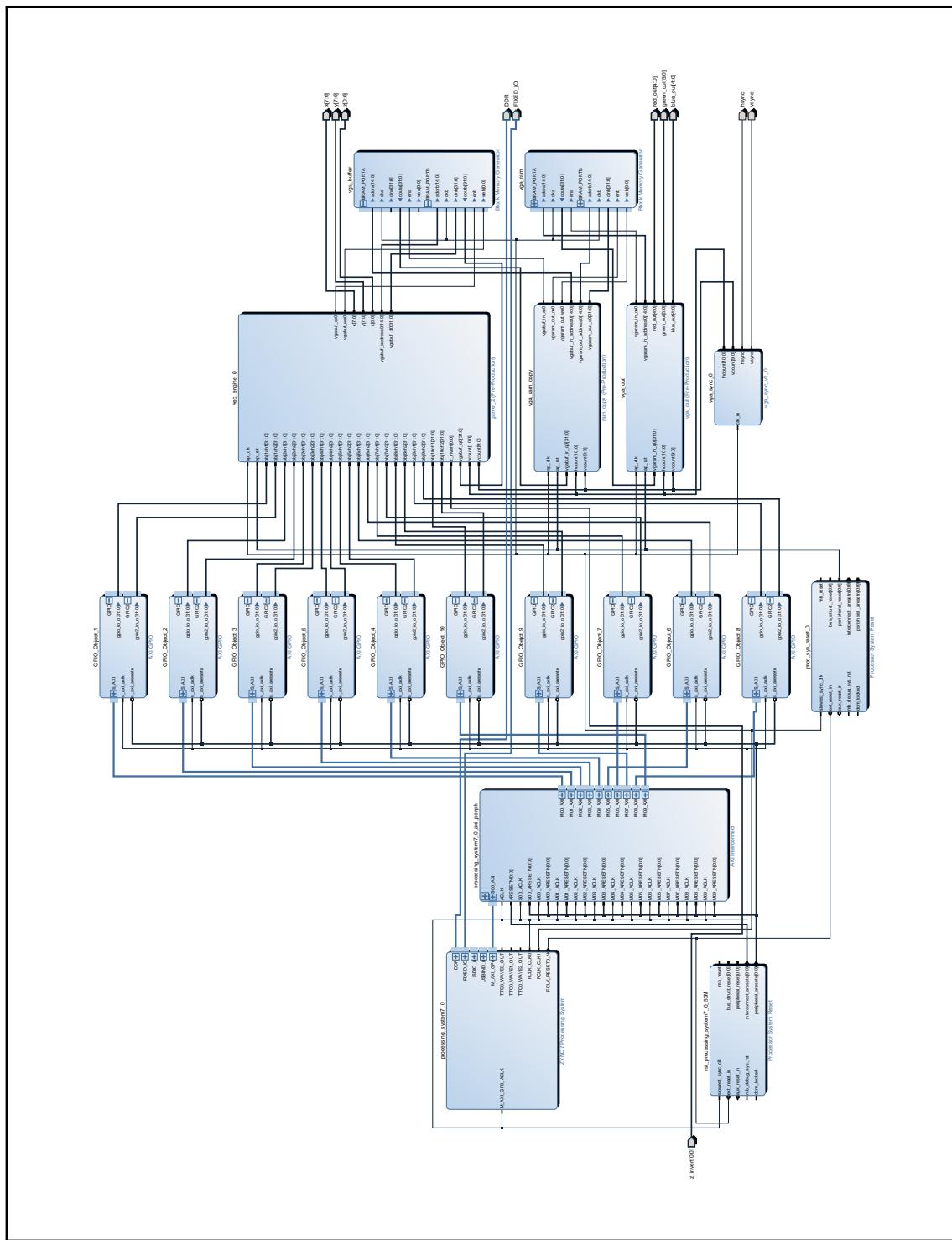


Abbildung A.7: Blockdesign der kompletten Implementierung

Quelle: Xilinx Vivado Design Suite 2016.1