

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Fachbereich VI

Technische Informatik - Embedded Systems

Modellbasierter Entwurf (WiSe1920)

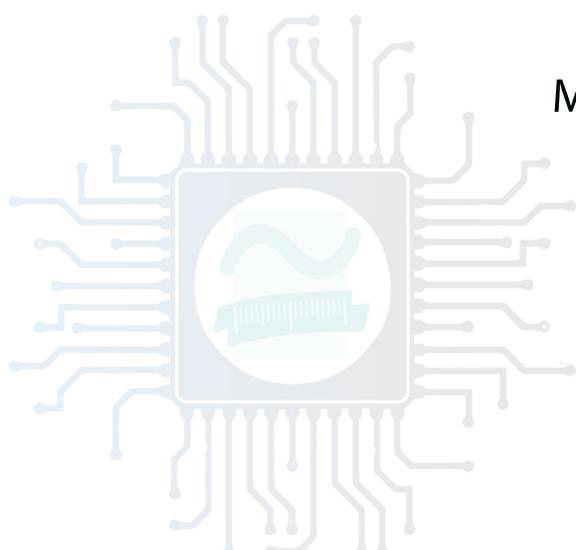
Laborbericht

Prüfer: Prof. Dr. Peter Gregorius

Autoren:

Torsten Michael Schenk s838995

Milad Raeisi s869746



Inhaltsverzeichnis

0.1 Vorwort	3
1 Projektbeschreibung	5
2 Hardware	7
2.1 Xilinx Pynq-Z1 board	8
2.1.1 PYNQ Z1 Core	9
2.1.2 Power Jumper	10
2.1.3 Programming Switching Jumper (Boot Jumper)	10
2.1.4 microSD Slot	11
2.1.5 HDMI IN/OUT	11
2.1.6 USB-HOST	12
2.1.7 PMOD A/B	12
2.1.8 Arduino / chipKIT Shield Header	13
2.1.9 Ethernet	13
2.2 Kamera - Ov7670	14
2.3 VGA- D Adapter	15
3 Software	16
3.1 Einrichtung der SDkarte	16
3.2 PYNQ-Z1 Installationsanleitung	16
3.2.1 Board Setup	17
3.2.2 Verbindung mit einem Rechner	17
3.2.3 Anschließen an Jupyter Notebook	18
4 Implementierung des Projekts	19
4.1 Algorithm	19
4.2 Hardware Implementierung	23
4.2.1 Einstellung von Pmod port	23
4.3 HLS Software Implementierung	23
4.3.1 Mein eigener IP Core	23
4.4 Code, Listings	29
4.4.1 Simulation in OpenCV	29
4.4.2 Simulation Numeric	30

Abbildungsverzeichnis

1.1	Beispiel: binarisertes Bild	5
1.2	Beispiel: Blobs mit Schwerpunkt und Richtung	6
2.1	Entwurf des Projects durch USB Host und HDMI OUT	7
2.2	Entwurf des Projects durch VGA Adapter	8
2.3	Übersicht PYNQ-Z1	9
2.4	PYNQ Z1 Core	9
2.5	Power Jumper JP5	10
2.6	Boot Jumper JP4	10
2.7	microSD-Slot	11
2.8	HDMI IN/OUT	11
2.9	USB Host	12
2.10	PYNQ PMODE Pins und PYNQ PMODE Ports	12
2.11	Shield Pin Diagram	13
2.12	Ethernet	14
2.13	Kamera - Ov7670	14
2.14	VGA D Adapter	15
3.1	Board Setup	17
3.2	Jupyter Notebook	18
4.1	Abbildung von Simulation-Numeric	20
4.2	Explorer	24
4.3	C-Simulation pop-up Fenster	25
4.4	Vivado HLS Directive Editor	26
4.5	Beispiel von Timing Details	27
4.6	Beispiel von Ressourcennutzung des Designs	27
4.7	Beispiel von Co-Simulation	28
4.8	Beispiel von Export RTL	28

0.1 Vorwort

Bei der Recherche zur Bearbeitung der Übungen wurden viele englischsprachige Webseiten zu rate gezogen. Generell kann man sagen, dass englische Fachbegriffe sich im Bereich FPGA und embedded Design etabliert haben, so dass eine Übersetzung eher verwirren als helfen würde. Daher haben wir uns entschieden, die **englischen** Bezeichner und Beschreibungen beizubehalten.

Um Codeabschnitte besser von Beschreibungen unterscheiden zu können, wurde eine eigene Schriftart verwendet:

Kommandozeilen Eingaben und Codesnippets werden wie HIER dargestellt.

Kapitel 1

Projektbeschreibung

In diesem Projekt sollen die Momente von Regionen (Blobs) berechnet werden. Momente sind im englischen Sprachgebrauch etabliert und werden im Deutschen als Massenträgheitsmomente oder einfach Trägheitsmomente bezeichnet. Sie kommen zum Einsatz wo statische Kennwerte die Konturabhängig sind benötigt werden. Eine wichtige Fragestellung ist z.B. in welcher Richtung sich ein L-Profil biegen wird. Die kontinuierliche Darstellung von Momenten als Integral in dem $I(x,y)$ die gesamte Fläche eines Querschnittes darstellt:

$$M = \int \int I(i, j) dx dy$$

Diskret (Bildpixel) können durch Iteration über alle Pixel einer Region die Momente berechnet werden. Die allgemeine Formel kann für alle Kombinationen verwendet werden, z.B. M_{00} , M_{01} , M_{20} . Die Zahlen dienen hierbei gleichzeitig als Exponent p oder q.:

$$M_{p,q} = \sum \sum i^p j^q I(i, j)$$

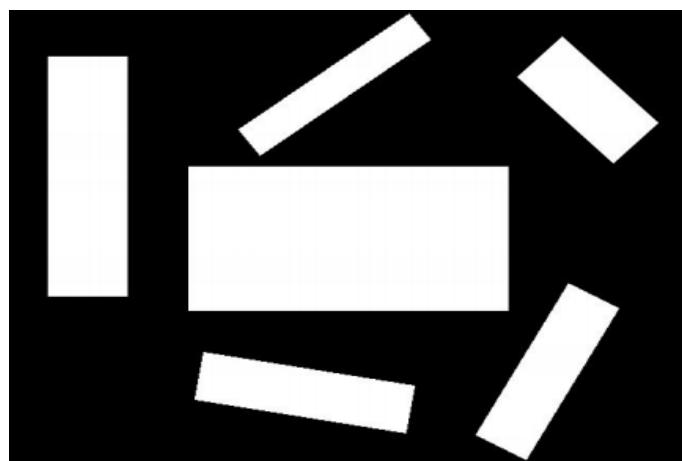


Abbildung 1.1: Beispiel: binarisertes Bild

Im Projekt soll auf Bildern jeweils mindestens eine Region selektiert werden. In diese soll zur Anzeige des Ergebnisses der Schwerpunkt und die Orientierung in Richtung der Achse mit höherem Biegemoment. In der folgenden Abbildung sind die Schwerpunkte lila und die Orientierung grün eingezzeichnet.

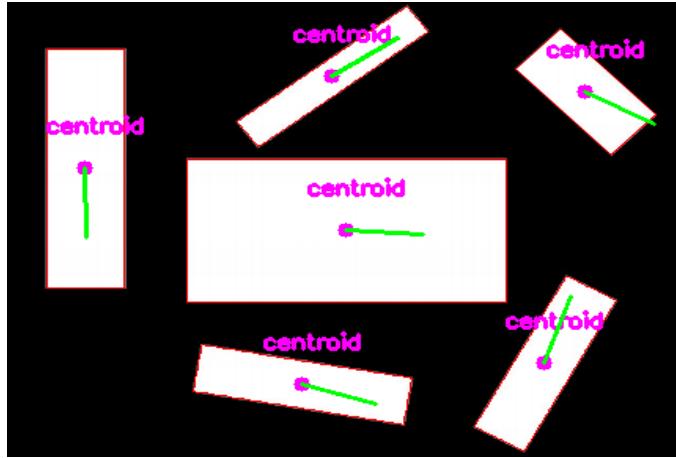


Abbildung 1.2: Beispiel: Blobs mit Schwerpunkt und Richtung

Nach der binarisierung der Region können die Pixel jeder Region gezählt werden und sind in x und y Koordinaten im Bild bekannt. Zuerst werden die Schwerpunkte \bar{x} und \bar{y} berechnet. Danach erfolgt die Berechnung der Orientierung aus den Flächenmomenten mittels der Kovarianzen und der Kovarianzmatrix.

Schwerpunkt x:

$$\bar{x} = \frac{M_{10}}{M_{00}}$$

Schwerpunkt y:

$$\bar{y} = \frac{M_{01}}{M_{00}}$$

Kovarianzen:

$$\mu_{11} = \frac{M_{11}}{M_{00}} - \bar{x}\bar{y}$$

$$\mu_{02} = \frac{M_{02}}{M_{00}} - \bar{y}^2$$

$$\mu_{20} = \frac{M_{20}}{M_{00}} - \bar{x}^2$$

$$cov(Obj) = \begin{bmatrix} \mu_{20} & \mu_{11} \\ \mu_{11} & \mu_{02} \end{bmatrix}$$

Orientierung einer Region im Wertebereich ($+\pi, -\pi$):

$$\theta = \frac{1}{2} \text{atan}2(2\mu_{11}, (\mu_{20} - \mu_{02}))$$

Bei der Hardware Realisierung sind mehrere Optionen denkbar. Es werden FPGA Implementation für Kamera via PMOD und VGA über SHIELD PINs angestrebt. Sollte sich der mathematische Teil als **semesterfüllend** herausstellen, kann auf USB Kamera und HDMI Display (Anbindung über Pynq-Linux) ausgewichen werden. Im Worse-Case wäre noch das Laden und Speichern von Bildern aus einem Bildordner als Möglichkeit offen. **Im Vordergrund steht die Implementation der mathematischen Funktionen deren Verarbeitung im FPGA erfolgt.**

Kapitel 2

Hardware

In diesem Kapitel wird die für das Projekt erforderliche Hardware dargestellt und erläutert. Optional kann im Projekt die Kamera auf zwei verschiedene Arten angeschlossen werden. Die erste Möglichkeit ist über Verwendung des HDMI-Ports einen Bildschirm anzuschließen:

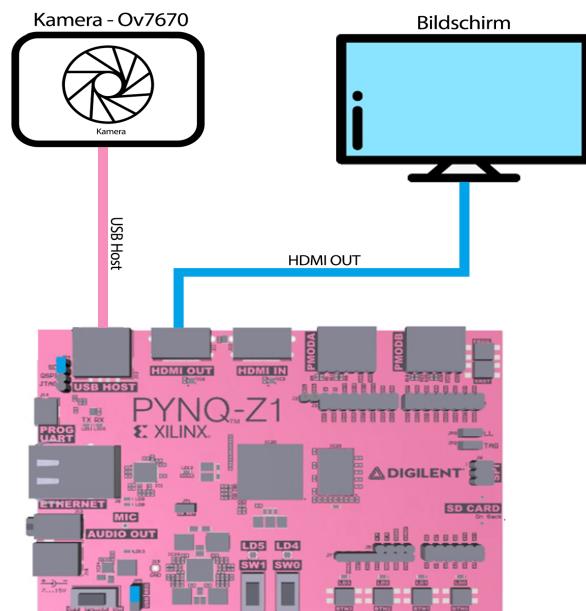


Abbildung 2.1: Entwurf des Projects durch USB Host und HDMI OUT

Die zweite Möglichkeit, wie unten im Entwurf gezeigt, wir über die FPGA Pins mit VGA Adapter ein Bildschirm angeschlossen.

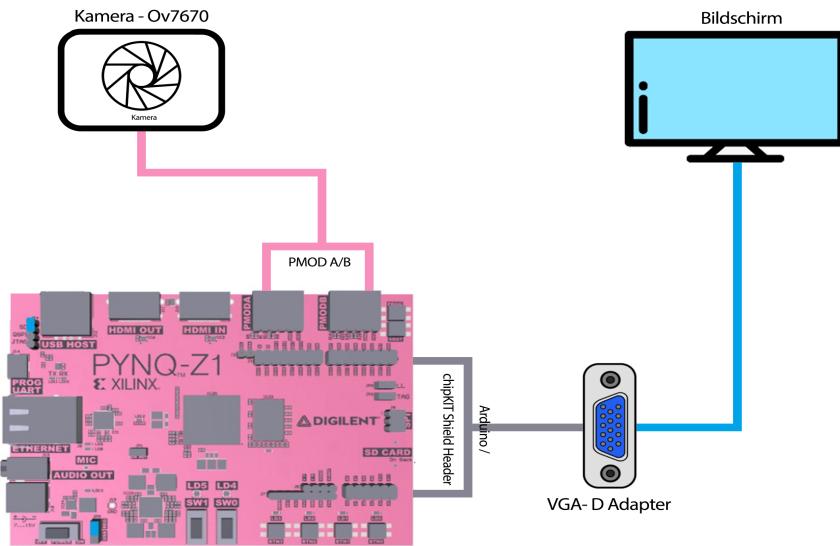


Abbildung 2.2: Entwurf des Projects durch VGA Adapter

Die Kamera kann ebenfalls entweder an USB oder PMOD...

2.1 Xilinx Pynq-Z1 board

Das PYNQ-Z1 Board wurde für die Verwendung mit PYNQ entwickelt, einem neuen Open-Source-Framework, das es Embedded-Programmierern ermöglicht, die Fähigkeiten von Xilinx Zynq All Programmable SoCs¹ (APSoCs) zu nutzen, ohne programmierbare Logikschaltungen entwickeln zu müssen. Die programmierbaren Logikschaltungen werden als Hardwarebibliotheken importiert und ihre APIs² sind im Wesentlichen so programmiert, dass sie wie die Softwarebibliotheken importiert und programmiert werden. Für Designer, die das Basissystem mit neuen Hardware-Bibliotheken erweitern wollen, stehen die Xilinx Vivado WebPACK-Tools kostenlos zur Verfügung.

Das PYNQ-Z1 unterstützt Multimedia-Anwendungen mit integrierten Audio- und Videoschnittstellen und ist so konzipiert, dass es unkompliziert mit Pmod-, Arduino- und Grove-Peripheriegeräten sowie universellen IO-Pins erweiterbar ist. Ebenso kann das PYNQ-Z1 Board auch mit USB-Peripheriegeräten wie WiFi, Bluetooth und Webcams erweitert werden und ist die Hardware-Plattform für das PYNQ Open-Source-Framework.

¹System On Chip

²Application Programming Interface

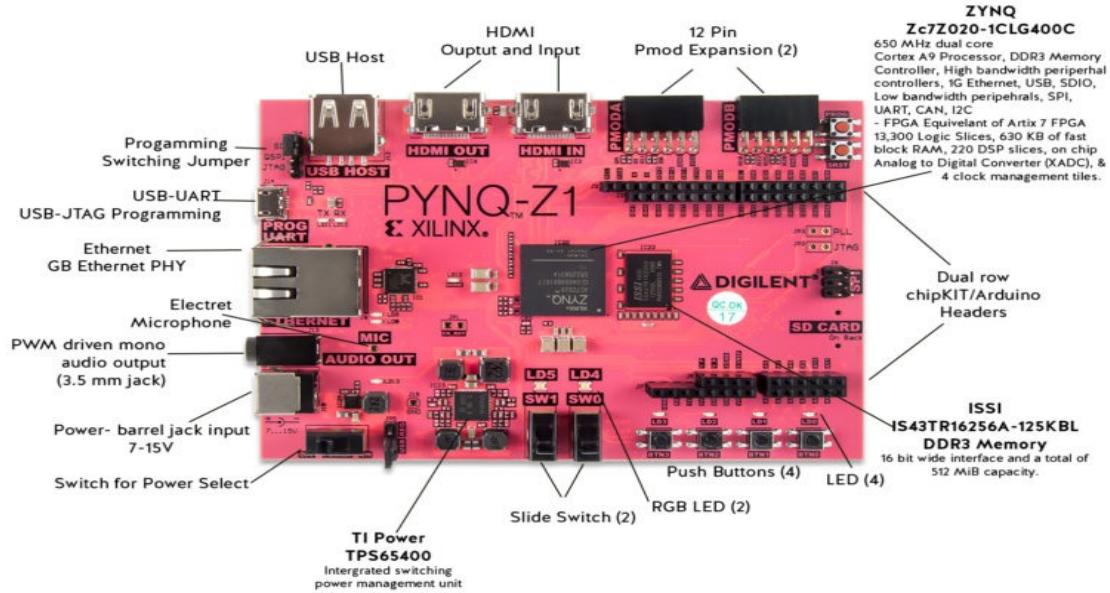


Abbildung 2.3: Übersicht PYNQ-Z1

Hardwarekomponenten auf dem Pynq-Z1 Board:

1. 512 MB DDR3 mit 16-bit Bus bei 1050Mbps.
2. 650 MHz Dual-Core Cortex-A9 Prozessor.
3. 16 MB Quad-SPI³ Flash, werkseitig programmiertem.
4. Peripherie-Controller mit niedriger Bandbreite: SPI, UART, KANN, I2C.
5. 630 KB fast block RAM.
6. 13,300 Logikscheiben mit je vier 6 Eingangs-LUT und 8 Flip-Flops.
7. u.s.w

2.1.1 PYNQ Z1 Core

Das PYNQ-Z1 besteht aus einem Zynq-XC7Z020-1CLG400C (xc7z020dg400-1) SoC, der wiederum einen dual Core ARM Cortex-A9 und einen Artix-7 FPGA enthält:



Abbildung 2.4: PYNQ Z1 Core

³Serial Peripheral Interface

2.1.2 Power Jumper

PYNQ Z1 hat 2 **"Modi"** Formen der Spannungsversorgung. Die Modi werden durch die Änderung des Power Jumpers JP5 gewählt.

1. **USB** über Anschluss J14 (PROG oder UART)
2. **REG** über Anschluss J18 (External Power)

Hinweis1 : Die Ausgangsspannung **AC** muss zwischen 7VDC und 15VDC sein.

Hinweis2 : Wenn die Eingangsspannungen **>15VDC** sein, wird das Board beschädigt.

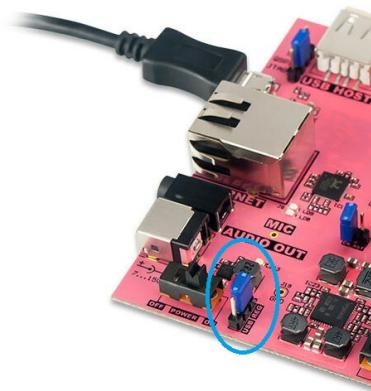


Abbildung 2.5: Power Jumper JP5

2.1.3 Programming Switching Jumper (Boot Jumper)

Der Bootmodus wird mittels Jumper JP4 gewählt. Das Board kann über SD-Karte, Quad SPI oder JTAG gebootet werden.

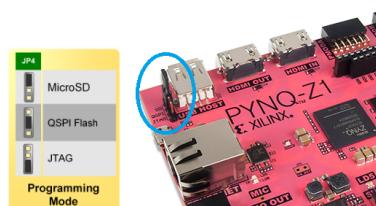


Abbildung 2.6: Boot Jumper JP4

Die drei Startmodi werden im folgenden Text beschrieben:

1. *microSD Boot Mode*

Der PYNQ-Z1 unterstützt **Booting** von einer microSD-Karte, die in Anschluss **J9** eingesetzt ist.

2. *Quad SPI Boot Mode*

Der PYNQ-Z1 verfügt über einen integrierten 16-MB-Quad-SPI-Flash, von dem der Zynq booten kann.

3. JTAG Boot Mode

Wenn der Prozessor in den JTAG-Startmodus versetzt wird, wartet er, bis die Software mithilfe der Xilinx-Tools von einem Host-Computer geladen wird.

Nach dem Laden der Software kann die Software entweder mit der Ausführung beginnen oder mit dem Xilinx SDK zeilenweise durchlaufen werden.

2.1.4 microSD Slot

Der PYNQ-Z1 bietet einen microSD-Slot⁴ (J9) für „non-volatile“ externen Speicher sowie zum Booten des Zynq. Der Steckplatz ist mit Bank 1/501 MIO [40-47] auch einschließlich Card Detect verbunden.

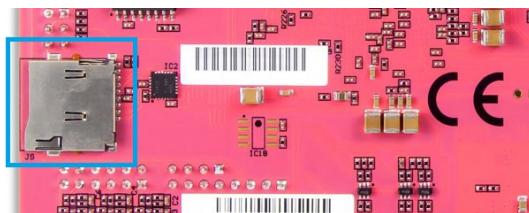


Abbildung 2.7: microSD-Slot

Hinweis: Für das Betriebssystem wird mindestens eine **Klasse-4 Karte mit 8 GB Speicherplatz** empfohlen.

Informationen zum Einrichten und Verwendung der Speicherkarte finden Sie im Abschnitt Software und im Bereich **??**.

2.1.5 HDMI IN/OUT

Die PYNQ-Z1-Karte enthält einen HDMI-Eingangsanschluss und einen HDMI-Ausgangsanschluss, die mit dem FPGA-Fabric des Zynq Chips verbunden sind. Dies bedeutet, dass zur Verwendung der HDMI-Anschlüsse die HDMI-Controller in einer Hardware-Bibliothek oder einem Overlay enthalten müssen.



Abbildung 2.8: HDMI IN/OUT

Das Basis-Overlay enthält einen HDMI-Eingangscontroller und einen HDMI-Ausgangscontroller, die beide an die entsprechenden HDMI-Anschlüsse angeschlossen sind.

Am Zynq PS ist ein USB-Controller angeschlossen. Eine Webcam kann auch zum Aufnehmen von Bildern oder Videoeingaben verwendet werden, die über den HDMI-Ausgang verarbeitet und angezeigt werden können.

⁴Steckplatz

2.1.6 USB-HOST

Der PYNQ-Z1 implementiert eine der beiden verfügbaren PS-USB-OTG⁵-Schnittstellen auf dem Zynq-Gerät. Als PHY wird ein Microchip USB3320 USB 2.0 Transceiver-Chip mit einer 8-Bit-ULPI-Schnittstelle verwendet.



Abbildung 2.9: USB Host

Das PHY verfügt über ein komplettes HSUSB Physical Front-End, das Geschwindigkeiten mehr als 480 MBit / s unterstützt.

Der PHY ist an die MIO Bank 1/501 angeschlossen, die mit 1,8 V versorgt wird. Das USBO-Peripheriegerät wird an die PS verwendet, die über MIO [28-39] angeschlossen ist.

Hinweis: USB OTG ist eine Variante des Universal Serial Bus (USB), die es einem USB-Gerät ermöglicht, eingeschränkte USB-Host-Aufgaben zu übernehmen.

2.1.7 PMOD A/B

Ein Pmod-Port ist eine offene 12-polige Schnittstelle, die von einer Reihe von Pmod-Peripheriegeräten unterstützt wird.

Typische Pmod-Peripheriegeräte sind:

- Sensoren (Spannung, Licht, Temperatur)
- Kamera
- Kommunikationsschnittstellen (Ethernet, seriell, WLAN, Bluetooth)
- Eingangs- und Ausgangsschnittstellen (Tasten, Schalter, LEDs)



Abbildung 2.10: PYNQ PMODE Pins und PYNQ PMODE Ports

High-Speed Pmod ports, jeder 12 Pins Pmod-Port liefert zwei 3,3 VCC Signale (Pins 6 und 12), zwei Groundssignale (Pins 5 und 11) und 8 Logiksignale.

⁵On-The-Go

Hinweis: Da die Pins nicht gegen Kurzschluss oder Überspannung (>3.3V) geschützt sind, muss beim Verdrahten besonders aufgepasst werden.

Aufgrund der Verwendung dieser Komponente im Projekt werden wird in den folgenden Kapiteln und der ??-Abschnitt vollständig erklärt.

2.1.8 Arduino / chipKIT Shield Header

Der PYNQ-Z1 kann an Standard-Arduino und ChipKIT Shields angeschlossen werden, um die Funktionalität zu erweitern. Der Shield Connector verfügt über 49 Pins, die für allgemeine digitale E / A mit dem Zynq PL verbunden sind.

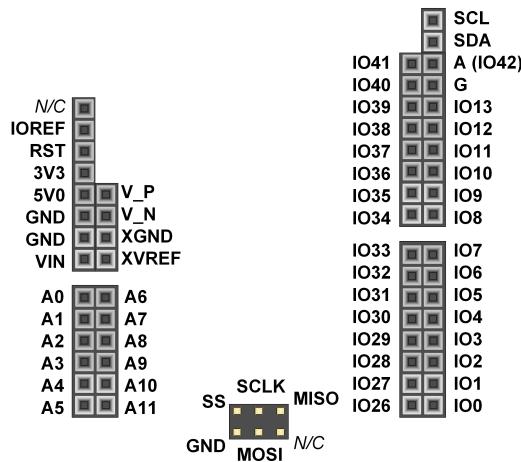


Abbildung 2.11: Shield Pin Diagram

Aufgrund der Flexibilität von FPGAs können diese Pins für nahezu alles verwendet werden, einschließlich digitalem Lesen / Schreiben, SPI-Verbindungen, UART-Verbindungen, I²C⁶-Verbindungen, VGA Controller und PWM.

6 dieser Pins (mit AN0-AN5 bezeichnet) können auch als unsymmetrische Analogeingänge mit einem Eingangsbereich von 0 V bis 3,3 V verwendet werden, und weitere 6 (mit AN6-11 bezeichnet) können als differentielle Analogeingänge verwendet werden.

Hinweis1 : Der PYNQ-Z1 ist **nicht** mit Shield kompatibel, die 5-V-Digital- oder Analogsignale ausgeben.

Hinweis2 : Mehr als **5 V** kann die Driving pins am PYNQ-Z1- Shield Connector beschädigen.

2.1.9 Ethernet

Der PYNQ-Z1 verwendet einen Realtek RTL8211E-VL PHY, um einen 10/100/1000 Ethernet-Port für die Netzwerkverbindung zu implementieren.

⁶I-squared-C protocol



Abbildung 2.12: Ethernet

Das Board versucht nach dem Anschalten über DHCP eine IP-Adresse zu erhalten und fällt auf 192.168.3.99/24 192.168.3.99:9090 zurück, falls kein DHCP Server verfügbar ist.

Hinweis: IP-Adressen Port hängt von der Version des Image Files **pynq_z1_v2.1.img**, z.B. bei Version 2.0 lautet 192.168.2.99:9090 und es ist möglich, dass bei einer anderen Version als 2.0 die Ip-Adresse angepasst werden muss.

Der PYNQ-Z1 enthält auch andere Komponenten wie MIC/AUDIO OUT, Schalter, Taster, RGB LEDs, u.s.w.

Weitere Informationen zu diesen Teilen finden Sie im PYNQ-Z1-Handbuch.

2.2 Kamera - Ov7670

Der OV7670 verwendet den RGB565-Modus mit 30 Bildern pro Sekunde. Die Frames können im internen BRAM⁷ des ZU200 des ZYNQ7-Chips gespeichert werden. Der VGA-Controller liest die Daten aus dem Block Ram und verwendet eine 12-Bit-Farbtiefe auf dem VGA-Schnittstellenbildschirm.

Die VGA-Auflösung beträgt ebenfalls 640x480. Die Kamera wird mit +3,3V versorgt.

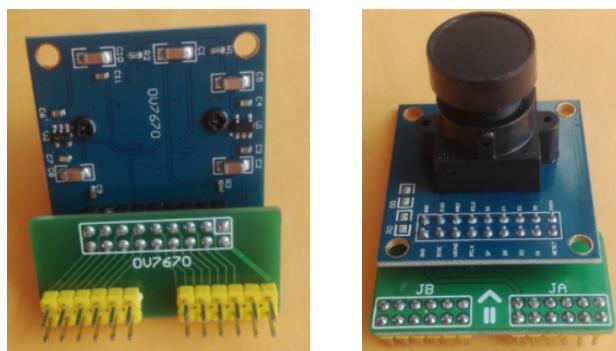


Abbildung 2.13: Kamera - Ov7670

Hinweis: ZU200 verfügt über einen umfangreichen BLOCK RAM-Block, sodass Sie 640X480 = 307200 12-Bit-Nummern direkt speichern können, sodass kein externer Speicher für die Videopufferung verwendet wird.

Bemerkenswerte Eigenschaften dieser Kamera sind:

- Steuerung den SCB⁸-Busteil

⁷BLOCK RAM

⁸Szcz Communication Bus

- Broad stopband performance bis to 8 GHz
- Fast roll-off
- Connectorized package

2.3 VGA- D Adapter

Es ist Kleine Platine mit 16 poligem Terminalblock (Klemmleisten 3,81mm Raster) auf eine 3-reihige High Density Buchse für VGA⁹ Anwendungen. VGA- D Adapter umfasst die Spezifikation einer analogen elektronischen Schnittstelle zur Übertragung von Bildern oder Videos zwischen unseren Board und Bildschirm sowie Spezifikationen für hierzu geeignete Stecker und Kabel.



Abbildung 2.14: VGA D Adapter

⁹Video Graphics Array

Kapitel 3

Software

3.1 Einrichtung der SDkarte

Mit dem folgenden Verfahren können Sie den Zynq von microSD mit einem Standard-**Zynq-Boot-Image** starten, das mit den Xilinx-Tools erstellt wurde:

1. Formatieren Sie die microSD-Karte mit einem FAT32-Dateisystem.
2. Kopieren Sie das mit Xilinx SDK erstellte **Zynq Boot Image** auf die microSD-Karte.
3. Benennen Sie das Zynq-Boot-Image auf der microSD-Karte in **BOOT.bin** um.
4. Entnehmen Sie die microSD-Karte aus Ihrem Computer und stecken Sie sie in den Anschluss **J9** des PYNQ-Z1.
5. Schließen Sie eine **Stromquelle** an den PYNQ-Z1 an und wählen Sie sie mit **JP5** aus.
6. Stecken Sie einen einzelnen **Jumper** auf **JP4** und schließen Sie die beiden oberen Stifte (mit der Bezeichnung „SD“) kurz.
7. Schalten Sie die Karte ein. Das Board bootet nun das Image von der microSD-Karte.

3.2 PYNQ-Z1 Installationsanleitung

Für PYNQ-Z1-Board können Sie der Kurzanleitung folgen:

3.2.1 Board Setup

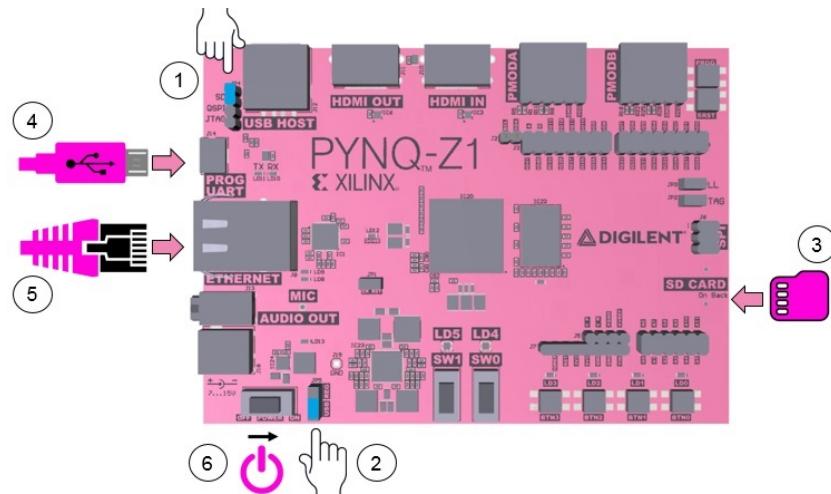


Abbildung 3.1: Board Setup

1. Setzen Sie den **JP4 / Boot-Jumper** auf die **SD-Position**, indem Sie den Jumper wie in der Abbildung gezeigt über die beiden oberen Stifte von JP4 stecken.
(Dadurch wird das Board so eingestellt, dass es von der Micro-SD-Karte bootet.)
2. Um den PYNQ-Z1 über das Micro-USB-Kabel mit Strom zu versorgen, setzen Sie den **JP5 / Power**-Jumper auf die USB-Position.
(Sie können die Karte auch über einen externen 12-V-Spannungsregler mit Strom versorgen, indem Sie den Jumper auf REG setzen.)
3. Setzen Sie die mit dem PYNQ-Z1-Image geladene Micro-SD-Karte in den **Micro-SD**-Kartensteckplatz unter der Karte ein.
4. Setzen Sie die mit dem PYNQ-Z1-Image geladene Micro-SD-Karte in den Micro-SD-Kartensteckplatz unter der Karte ein.
5. Schließen Sie das USB-Kabel an Ihren PC / Laptop und an den **PROG-UART / J14** MicroUSB-Anschluss auf der Karte an.
6. Schließen Sie die Karte an Ethernet an, indem Sie die folgenden Anweisungen befolgen.
7. Schalten Sie den PYNQ-Z1 ein.

3.2.2 Verbindung mit einem Rechner

Auf Ihrem Computer muss ein Ethernet-Anschluss verfügbar sein, und Sie müssen über Berechtigungen zum Konfigurieren Ihrer Netzwerkschnittstelle verfügen. Ohne Internetzugang können Sie keine neuen Pakete aktualisieren oder laden.

Direkt an einen Rechner anschließen (Statische IP):

1. Geben Sie Ihren Rechner eine statische IP-Adresse
2. Verbinden Sie die Karte mit dem Ethernet-Anschluss Ihres Computers

3. Navigieren Sie zu **http://192.168.2.99**

Hinweis: Wenn obere Methode nicht klappt sein, können Sie Ihren Rechner durch "Verbindung zu einem Netzwerk-Router" mit dem Board verbinden.

3.2.3 Anschließen an Jupyter Notebook

Sobald Ihr Board eingerichtet ist, öffnen Sie zum Herstellen einer Verbindung zu Jupyter Notebooks einen Webbrower und navigieren Sie zu:

- **http://192.168.2.99** If your board is connected to a computer via a static IP address.
- Wenn Ihr Board richtig konfiguriert ist, wird eine Anmeldung-Webseite angezeigt. Geben Sie hier den Benutzernamen und Passwort.

Benutzername: **xilinx**
Passwort: **xilinx**

Nach der Anmeldung wird der folgende Bildschirm angezeigt:

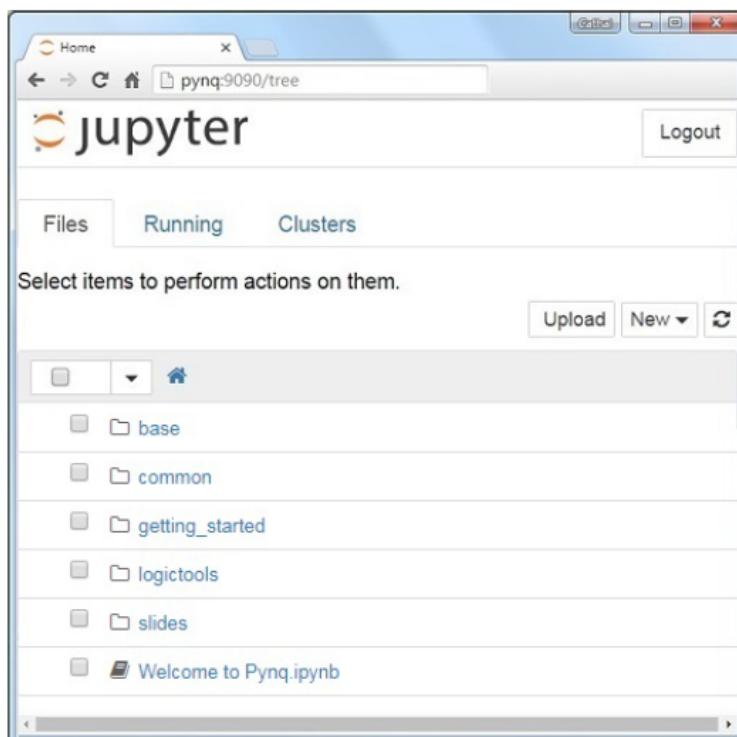


Abbildung 3.2: Jupyter Notebook

Der Standard-Hostname lautet **pynq** und die statische Standard-IP-Adresse lautet **192.168.2.99**.

Hinweis: Wenn Sie die statische IP-Adresse der Karte geändert haben, müssen Sie die Adresse ändern.

Kapitel 4

Implementierung des Projekts

4.1 Algorithm

Zur Einschätzung des Zahlenraumes wird mit Bildgröße 640x480 gearbeitet. Bei einem Objekt von ca. 1/4 Größe des Kamerabildes muss mit $640*480 / 4 = 76800$ Pixlen im Objekt gerechnet werden. Wenn das Objekt mittig liegt sind x oder y Koordinaten im Bereich $\{100, 500\}$. Bei der Berechnung von m_{02} wird die Spaltennummer mit sich selbst multipliziert und aufaddiert. Dies ergibt eine Summe von $41.463.400 = \{500 * 500 + 499 * 499 + \dots + 102 * 102 + 101 * 101 + 100 * 100\}$ und würde in den Wertebereich in (nicht Vorzeichen behaftete) 26 Bits $2^{26} = 67.108.864$. Bei Betrachtung der gesamten Bildgröße: $87.586.240 = \{640 * 640 + 639 * 639 + \dots + 2 * 2 + 1 * 1\}$ benötigt mindestens 27 Bits $2^{27} = 134.217.728$. Der folgende Algorithmus zur Bestimmung der Flächenmomente bis zum Grad 2 wird vollständig in einen Int32 Datentyp passen.

Algorithm 1 Momente

Require: Ein Bild I mit Breite b , Höhe h und Pixelintensitäten $px \in \{I(u, v) | 0, \dots, 255\}$ (8Bit).
Die Laufindizes $u, v, i, j, r, c \in \mathbb{N}_0$. I wird als zeilenweise serialisierter Stream mit der Pixelanzahl $L = b * h$ verarbeitet. Jedes px wird als 8 separate Signalleitungen ausgelegt, die Adressierung erfolgt über $n \in \{n | 0, \dots, 7\}$ und wird als px_n indiziert. Sechs Ausgabewerte $m_{ij} \in \{m_{ij} | 0, \dots, 2^{32} - 1\}$ werden mit 0 initialisiert.

```
1: for jedes  $i$  in  $L$  do
2:    $px = L_i$ 
3:   if  $px_7 == 1$  then
4:      $r = i \% b$  (Zeile)
5:      $c = i \% b$  (Spalte)
6:      $m_{00}+ = 1$ 
7:      $m_{01}+ = c$ 
8:      $m_{10}+ = r$ 
9:      $m_{11}+ = r * c$ 
10:     $m_{02}+ = c * c$ 
11:     $m_{20}+ = r * r$ 
12:   end if
13: end for
```

Algorithm 2 Zentrum und Winkel der Hauptachse

Require: Sechs Eingabewerte $m_{ij} \in \{m|0, \dots, 2^{32} - 1\}, ij \in \{00, 01, 10, 11, 02, 20\}$ enthalten die Momente. Temporäre Variablen $\bar{x}, \bar{y}, \bar{xy}, \bar{x^2}, \bar{y^2}, \mu_{11}, \mu_{02}, \mu_{20} \in \{0, \dots, 2^{32} - 1\}$ und $sub, mult \in \{-2^{16}, \dots, 2^{16} - 1\}$ werden mit 0 initialisiert. Ergebnisse werden als Festkommazahl $\theta \in \{+\pi, -\pi\}$ und \bar{x}, \bar{y} als Zentrumkoordinaten ausgegeben.

```

1: if  $m_{00} > 0$  then
2:    $\bar{x} = \frac{m_{10}}{m_{00}}$ 
3:    $\bar{y} = \frac{m_{01}}{m_{00}}$ 
4:    $\bar{xy} = \bar{x} * \bar{y}$ 
5:    $\bar{x^2} = \bar{x} * \bar{x}$ 
6:    $\bar{y^2} = \bar{y} * \bar{y}$ 
7:    $\mu_{11} = \frac{m_{11}}{m_{00}} - \bar{xy}$ 
8:    $\mu_{02} = \frac{m_{02}}{m_{00}} - \bar{y^2}$ 
9:    $\mu_{20} = \frac{m_{20}}{m_{00}} - \bar{x^2}$ 
10:   $mult = \mu_{11} * \mu_{11}$ 
11:   $sub = \mu_{20} - \mu_{02}$ 
12:   $\theta = \frac{1}{2} \text{atan2}(mult, sub)$ 
13: end if

```

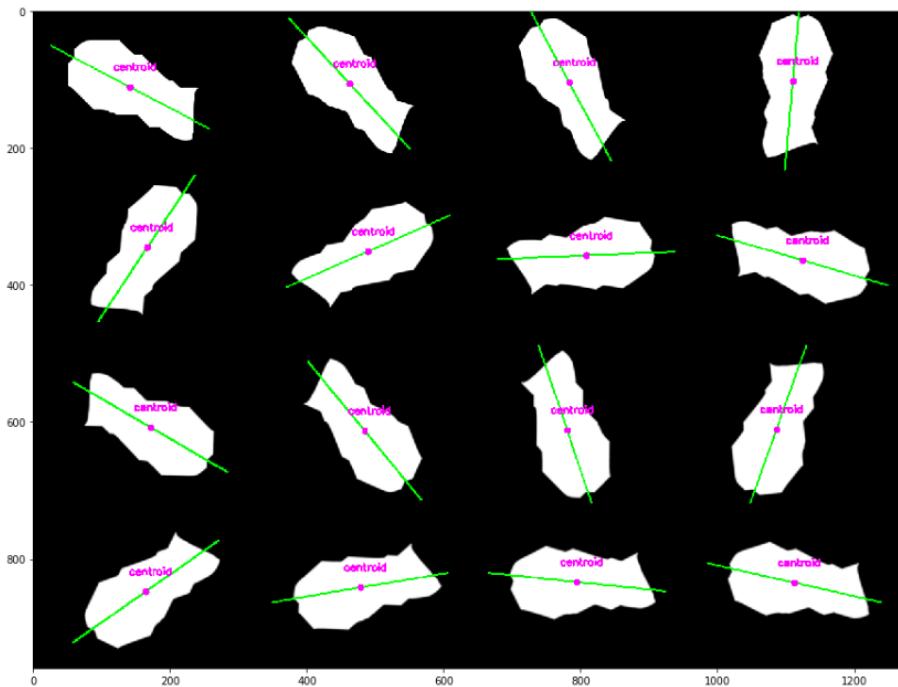


Abbildung 4.1: Abbildung von Simulation-Numeric

Ergebnis von Simulation-Numeric:

Moments: m00: 15964 m01: 1773113 m10: 2281256 m11: 269670230 m20: 365332132
 m02 215154207 Center X 142.00, Center Y 111.00, Angle clockwise 27.66°

Moments: m00: 15979 m01: 1704601 m10: 2337778 m11: 268749999 m20: 369586230
 m02 211883899 Center X 146.00, Center Y 106.00, Angle clockwise 47.24 °

Moments: m00: 15975 m01: 1662657 m10: 2398584 m11: 265651241 m20: 377952942
 m02 212836887 Center X 150.00, Center Y 104.00, Angle clockwise 61.61 °

Anpassung des Winkelergebnisses, atan2 mit positiven Argumenten: Es erfolgt eine Gegenüberstellung von dem Ergebnis der atan2 Funktion mit Argumenten, die einmal mit korrektem Vorzeichen verwendet werden und einmal mit ihrem Absolutwert. Bei Verwendung des Absolutwertes muss das Ergebnis nochmal korrigiert werden.

$mult = \mu_{11} * \mu_{11}$ versus $mult = abs(\mu_{11} * \mu_{11})$ Ergebnis mit abs gekennzeichnet

$sub = \mu_{20} - \mu_{02}$ versus $sub = abs(\mu_{20} - \mu_{02})$ Ergebnis mit abs gekennzeichnet

$$\theta = \frac{1}{2} \text{atan2}(mult, sub)$$

sub: -1348.09 mul: 2058.37 Center X 150.00, Center Y 104.00, Angle atan2 61.61°

korrekt: $-s+m=61.61^\circ$, abs: $+s+m: 28.39^\circ$, Anpassung: $(90^\circ-x)$

sub: -887.58 mul: -2094.80 Center X 167.00, Center Y 105.00, Angle atan2 -56.48°

korrekt: $-s-m=-56.48^\circ$, abs: $+s+m: 33.52^\circ$, Anpassung: $(x+90^\circ)$

sub: 1435.64 mul: -1549.62 Center X 173.00, Center Y 111.00, Angle atan2 -23.59°

korrekt: $+s-m=-23.59^\circ$, abs: $+s+m: 23.59^\circ$, Anpassung: $(x*(-1))$

Algorithm 3 Zentrum und Winkel der Hauptachse mit abs(mult) und abs(sub)

Require: Sechs Eingabewerte $m_{ij} \in \{m|0, \dots, 2^{32} - 1\}, ij \in \{00, 01, 10, 11, 02, 20\}$ enthalten die Momente. Temporäre Variablen $\bar{x}, \bar{y}, \bar{xy}, \bar{x^2}, \bar{y^2}, \mu_{11}, \mu_{02}, \mu_{20}, sub, mult \in \{0, \dots, 2^{32} - 1\}$ werden mit 0 initialisiert. Ergebnisse werden als Festkommazahl $\theta \in \{+\pi, -\pi\}$ und \bar{x}, \bar{y} als Zentrumkoordinaten. Flags s, m für Korrektur des Ergebnisses θ . $s, m \in \{0, 1\}$ ausgegeben.

```

1:  $m = 0, s = 0$ 
2: if  $m_{00} > 0$  then
3:    $\bar{x} = \frac{m_{10}}{m_{00}}$ 
4:    $\bar{y} = \frac{m_{01}}{m_{00}}$ 
5:    $\bar{xy} = \bar{x} * \bar{y}$ 
6:    $\bar{x^2} = \bar{x} * \bar{x}$ 
7:    $\bar{y^2} = \bar{y} * \bar{y}$ 
8:    $\mu_{11} = \frac{m_{11}}{m_{00}} - \bar{xy}$ 
9:    $\mu_{02} = \frac{m_{02}}{m_{00}} - \bar{y^2}$ 
10:   $\mu_{20} = \frac{m_{20}}{m_{00}} - \bar{x^2}$ 
11:  if  $\mu_{11} < 0$  then
12:     $m = 1$ 
13:     $\mu_{11} = -\mu_{11}$ 
14:  end if
15:   $mult = \mu_{11} * \mu_{11}$ 
16:  if  $\mu_{02} < \mu_{20}$  then
17:     $sub = \mu_{20} - \mu_{02}$ 
18:     $s = 1$ 
19:  else
20:     $sub = \mu_{02} - \mu_{20}$ 
21:  end if
22:   $\theta = \frac{1}{2} \text{atan2}(mult, sub)$ 
23: end if
24: if  $m = 1 \&& s = 1$  then
25:    $\theta = \theta + \frac{\pi}{2}$ 
26: else if  $m = 0 \&& s = 1$  then
27:    $\theta = \frac{\pi}{2} - \theta$ 
28: else if  $m = 1 \&& s = 0$  then
29:    $\theta = \theta * (-1)$ 
30: end if
```

4.2 Hardware Implementierung

4.2.1 Einstellung von Pmod port

Wie im Abschnitt **PMOD A/B** schon erwähnt wurde,

4.3 HLS Software Implementierung

In dieser Sektion wird eine grundlegende Einführung in die High Level Synthesis erläutert.
(HLS)

Was bedeutet HLS?

High Level Synthesis wurde eingeführt, um die Elektronikkenntnisse zu reduzieren, die zum Entwerfen von Hardware erforderlich sind. Dies erleichtert auch den Ablauf des Hardware-Designs, wenn es darum geht, ein bestimmtes Verhaltensmodell zu erreichen.

Unter Verwendung einer Programmiersprache wie C / C ++ implementieren wir das Verhaltensmodell in Hardware.

Das HLS-Tool bildet unser Verhaltensmodell in Hardware ab und generiert den HDL-Code dafür und dann wird synthetisiert und danach im letzten Schritt wird auf einem FPGA implementiert.

Wenn wir mit HLS vertraut sind, fahren wir mit dem Beginn der Schritte fort und erzeugen unseren eigenen **IP Core**.

4.3.1 Mein eigener IP Core

Im ersten Schritt werden wir dies Thema nachvollziehen, dass wie ich meinen eigenen IP Core in HLS erstellen kann!

Schritt 1: Erstellung eines neuen Projekts

- Öffnen wir Vivado HLS und legen wir ein neues Projekt mit dem eigenen Funktionsnamen z.B. „XXXX“ an.

- Wählen wir ein Bauteil oder eine Entwicklungsplatine aus (wir verwenden PYNQ-Z1) und beenden wir die Erstellung des neuen Projekts.

- Behalten wir vorerst den Standardzeitraum (10ns) bei. (Wir können ihn später ändern, wenn wir wollen.)

- Auf der rechten Seite des Fensters sehen wir den "Explorer", wie in der folgenden Abbildung.

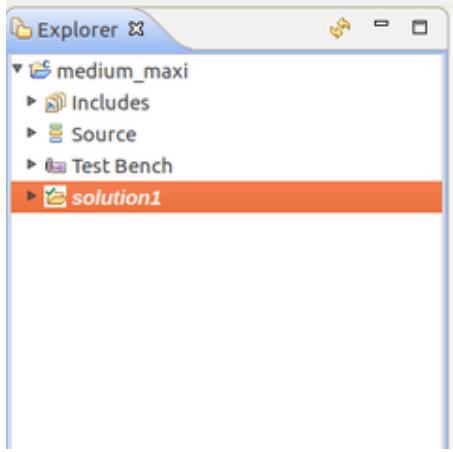


Abbildung 4.2: Explorer

- Klicken wir mit der rechten Maustaste auf die Quelle, erstellen eine neue Datei, nennen wir sie mit dem eigenen Name z.B. "**core.cpp**" und speichern wir sie im Projektordner.
- Nach dem Speichern öffnet Vivado HLS automatisch die leere neue Datei.

Beginnen wir mit dem Entwerfen des IP-Cores

Schritt 2: Entwurf des IP-Cores

In diesem Schritt wird der IP-Core nur für Funktionstests entworfen. Diese Funktion muss mit Ihrem Top-Funktionsnamen benannt werden. Damit der Entwurf des IP-Kerns abgeschlossen wird, benötigen wir jedoch zum Test einen Test Bench, ob die Funktionalität des IP-Cores unseren Erwartungen entspricht.

Schritt 3: Entwurf des Test-Benches

- Erstellen wir eine neue Datei unter Test bench. Wir können die Top-Funktion (bei uns z.B. „XXXX“) in der Hauptfunktion des Test Benchs aufrufen und Ein- und Ausgänge weitergeben und prüfen.

- In der Hauptfunktion `test_bench` ist die Rückgabe von **0** sehr wichtig, weil Vivado HLS damit prüft, ob die Funktionssimulation erfolgreich war.

Schritt 4: Ausführung des Test Benchs für Verhaltenssimulation

- Führen wir unseren Test Bench aus. Klicken wir in der oberen Toolbar auf "**Run C Simulation**".

- Dann wird "Popup-Window" wie in der folgenden Abbildung angezeigt:

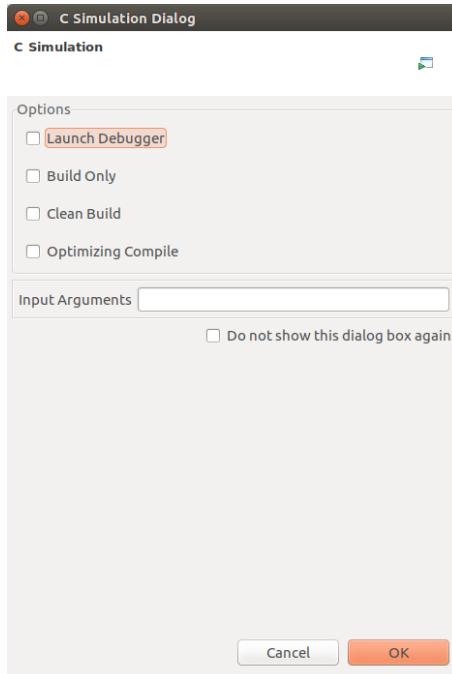


Abbildung 4.3: C-Simulation pop-up Fenster

- Wählen wir hier einfach "OK" und die *C-Simulation* wird gestartet. Dies wird einige Zeit (abhängig von Ihren Ressourcen) nehmen.

Schritt 5: Vorbereitung des IP-Cores für die Synthese

Wir können *Compiler-Directives* verwenden, um den Compiler dazu zu bringen, den IP-Core so zu synthetisieren.

Wenn wir keine *Compiler-Directives* verwenden, analysiert Vivado *HLS* den Code und verwendet Standard-Directives an den entsprechenden Positionen des Codes. Wir können diese Standardanweisungen im rechten Fenster unter der Registerkarte "**Directive**" sehen.

Mit Doppelklick auf "Ein- oder Ausgänge" wird Ein Fenster angezeigt, dass wir die Optionen wie in der folgenden Abbildung ausfüllen können und dass klicken wir auf "OK".

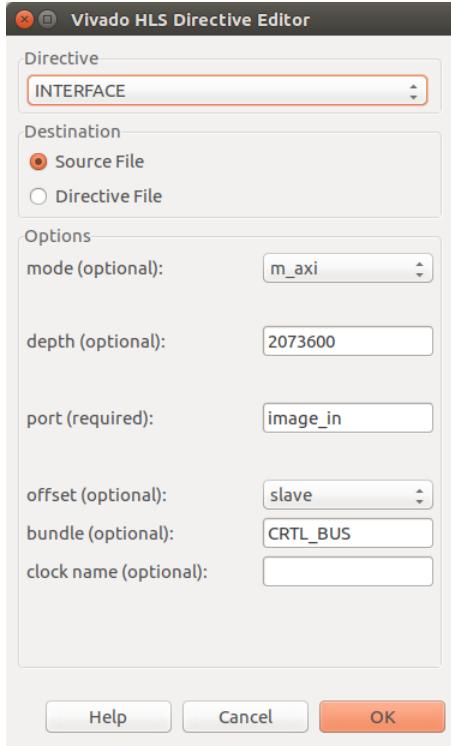


Abbildung 4.4: Vivado HLS Directive Editor

- **directive**

Interface, da wir Eingabe-Ausgabeschnittstelle für den IP-Core verwenden wollen.

- **mode**

Es steht für einen AXI oder AXI Master Full-Port.

- **depth**

Größe des Arrays. Dies ist optional. Wenn wir dieses Feld leer lassen, verwendet Vivado HLS die Standardtiefe. Es ist besser, wenn wir hier die Tiefe angeben können. Durch Auswahl von "Slave" können wir die Speicheradresse des Ports zum *Run-time* über einen AXI Lite-Port festlegen.

- **offset**

Hier wird "*mapped memory location*" des angeschlossenen Eingangs oder Ausgangs gesteuert.

- **bundle**

Über AXI Lite wird der Port mit dem Namen "CRTL_BUS" gebündelt.

Wählen wir dann **OK**.

Zum Synthetisieren des unseren IP-Cores klicken wir auf "Run C Synthesis".

Wenn das Design ordnungsgemäß synthetisiert wurde, wird ein ähnlicher Synthesebericht angezeigt.

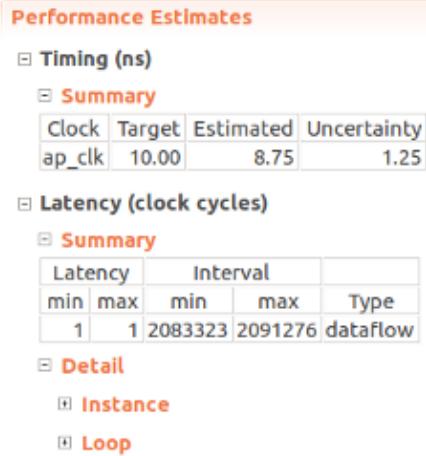


Abbildung 4.5: Beispiel von Timing Details

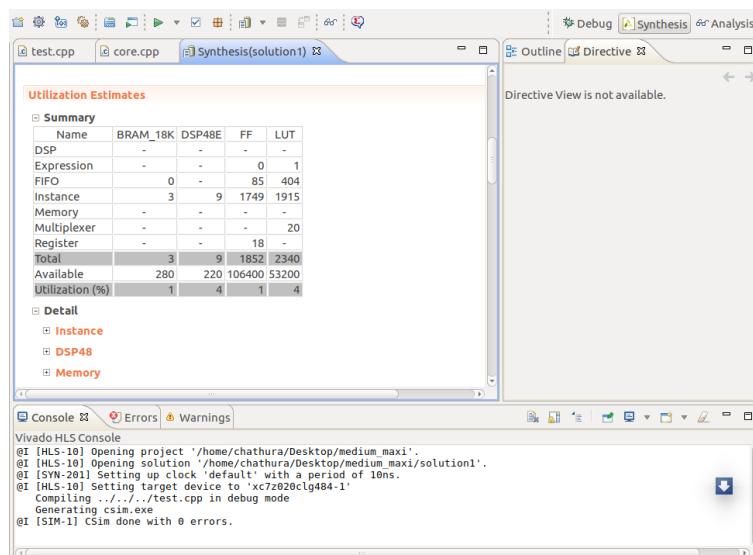


Abbildung 4.6: Beispiel von Ressourcennutzung des Designs

Bevor wir unseren IP-Core exportieren, müssen wir eine **RTL-Co-Simulation** durchführen, um die Funktionsunterschiede zwischen der **C-Simulation** und der **RTL-Simulation** zu überprüfen.

Klicken wir auf **"Run C/RTL co-simulation"** und dann klicken im Popup-Fenster auf **"OK"**, wenn **"setup only"** aktiviert ist.

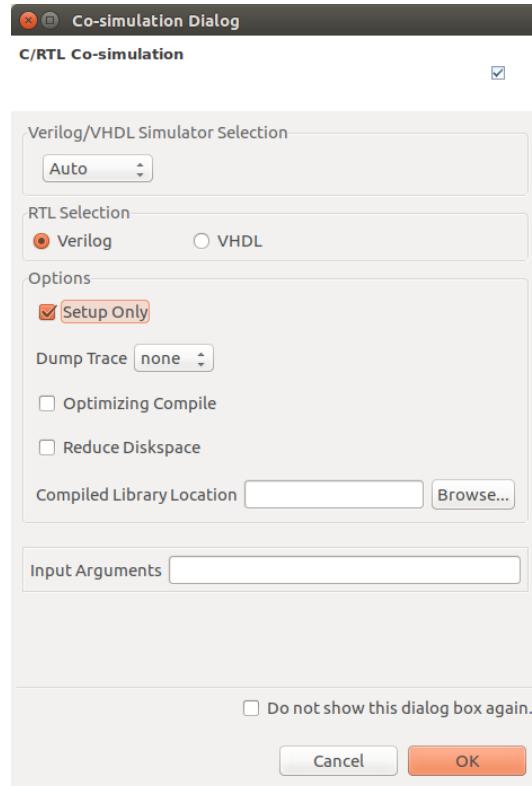


Abbildung 4.7: Beispiel von Co-Simulation

Nachdem wir diese Simulation erfolgreich geschafft haben, können wir mit dem Export des IP-Cores vorgehen. Klicken wir auf **"Export RTL"** und dann auf **"OK"**.

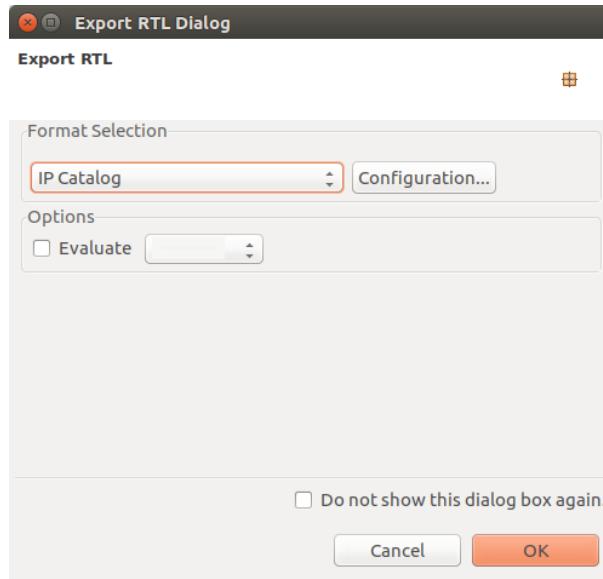


Abbildung 4.8: Beispiel von Export RTL

Im obigen Fenster können wir auswählen, ob wir unser Design mithilfe einer HDL oder Verilog bewerten möchten. Nach dem Export unseres IP-Cores sind wir mit dem IP-Core-Design mit Vivado HLS fertig.

Der nächste Schritt besteht darin, die gesamte Hardware-Architektur einschließlich unseres IP-Cores mit Vivado zu entwerfen.

4.4 Code, Listings

4.4.1 Simulation in OpenCV

Listing 4.1: Simulation in OpenCV

```

import cv2
import numpy as np
import sys
import math

img = cv2.imread("img.png", 0)
# convert the grayscale image to binary image
ret, thresh = cv2.threshold(img, 127, 255, 0)
# find contours in the binary image
im2, contours, hierarchy = cv2.findContours(\ 
    thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

print('NumberContours:', len(contours))
for c in contours:
    area = cv2.contourArea(c)
    print(area)
    if (area < 100.0):
        contours.remove(c)
    print('NumberContours>100:', len(contours))

cv2.namedWindow("main", cv2.WINDOW_NORMAL)
color_image = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
cv2.drawContours(color_image, contours, -1, (0, 0, 255), 1)

for c in contours:
    # calculate moments for each contour
    M = cv2.moments(c)

    # calculate x,y coordinate of center
    cX = int(M["m10"] / M["m00"])
    cY = int(M["m01"] / M["m00"])

    # cv2.circle(img, (cX, cY), 5, (255, 0, 255), -1)
    cv2.circle(color_image, (cX, cY), 5, (255, 0, 255), -1)
    cv2.putText(color_image, "centroid", (cX - 25, cY - 25), \
        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 255), 2)

    # display the image

```

```

cv2.imshow( "main" , color_image)
cv2.waitKey(0)

for c in contours:
    M = cv2.moments(c)
    cX = int(M[ "m10" ] / M[ "m00" ])
    cY = int(M[ "m01" ] / M[ "m00" ])
    mu11 = (M[ "m11" ]/M[ "m00" ])-(cX*cY)
    mu02 = (M[ "m02" ]/M[ "m00" ])-(cY**2)
    mu20 = (M[ "m20" ]/M[ "m00" ])-(cX**2)
    theta = 0.5 * math.atan2( (2*mu11) , (mu20-mu02) )

    length = 50
    x2 = cX + length * math.cos(theta)
    y2 = cY + length * math.sin(theta)
    lineThickness = 2
    cv2.line(color_image, (int(cX) , int(cY)) , (int(x2) , \
        int(y2)) , (0,255,0) , lineThickness)

# display the image
cv2.imshow( "main" , color_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

Die Abbildung vom Beispiel "Blobs mit Schwerpunkt und Richtung" wurde im Abschnitt **Projektbeschreibung** gezeigt.

4.4.2 Simulation Numeric

Listing 4.2: Simulation Numeric

```

import cv2
import numpy as np
import math # for tan
import matplotlib.pyplot as plt

def calcmoments(img):
    # function to calculate moments
    moments = np.zeros(6, dtype=int) # Moments :[m00,m01,m10,m11,m20,m02]
    for y in range(img.shape[0]):
        for x in range(img.shape[1]):
            if (img[y,x] & 128): \
                # check if 7th bit is '1', it is set for gray values >=128
                moments[0] += 1
                moments[1] += y
                moments[2] += x
                moments[3] += x*y
                moments[4] += y*y

```

```

        moments[5] += x*x
    return moments

def calcthetaCXY(m):
    # calculate cX,cY,theta of blob
    cxytheta = np.zeros(3, dtype=float) # cxytheta : [ cX, cY, theta ]
    cxytheta[0] = int(m[2]/m[0])
    cxytheta[1] = int(m[1]/m[0])
    mu20 = (m[4]/m[0]) - cxytheta[0]**2
    mu11 = (m[3]/m[0]) - cxytheta[0]*cxytheta[1]
    mu02 = (m[5]/m[0]) - cxytheta[1]**2
    cxytheta[2] = 0.5 * math.atan2((2*mu11),(mu20-mu02))
    return cxytheta

test = cv2.imread('pics/Block0.png',0)
tiles = 4
rows = test.shape[0]
cols = test.shape[1]
hugelmg = np.zeros( (4*rows, 4*cols, 3), np.uint8)

for i in np.arange(16):
    img = cv2.imread('pics/Block'+str(i)+'.png',0)
    m = calcmoments(img) # Moments : [ m00, m01, m10, m11, m02, m20 ]
    cxytheta = calcthetaCXY(m) # cxytheta : [ cX, cY, theta ]

    # visu, create line through center point
    length = 130
    x1 = cxytheta[0] + length * math.cos(cxytheta[2])
    y1 = cxytheta[1] + length * math.sin(cxytheta[2])
    x2 = cxytheta[0] - length * math.cos(cxytheta[2])
    y2 = cxytheta[1] - length * math.sin(cxytheta[2])

    color_image = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
    lineThickness = 2
    cv2.line(color_image, (int(x1), int(y1)), (int(x2), int(y2)), \
              (0,255,0), lineThickness)
    cv2.circle(color_image, (int(cxytheta[0]), int(cxytheta[1])), \
               5, (255, 0, 255), -1)
    cv2.putText(color_image, "centroid", (int(cxytheta[0])-25, \
                                           int(cxytheta[1])-25), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 255), 2)

    # copy images into large result image
    rpos = int(i/4)*rows
    cpos = i%4*cols
    hugelmg[rpos:rpos+rows, cpos:cpos+cols] = color_image

# display plot in notebook

```

```
%matplotlib inline  
plt.axis("off")  
plt.figure(figsize=(15,15))  
plt.imshow(hugelmg)  
plt.show()
```

Literaturverzeichnis

[1] Xilinx PYNQ Z1 Board,

<https://blog.digilentinc.com/python-zynq-pynq-introducing-our-latest-collaboration/>

[2] PYNQ Z1-Komponenten,

<https://buildmedia.readthedocs.org/media/pdf/pynq/v1.4/pynq.pdf>

[3] PYNQ Z1-Einrichtung,

Pynq_Labor_Doku_Jaschko_Stolle.pdf

[4] Ov7670camera,

<http://www.alselectro.com/arduino-camera-ov7670.html>

[5] Board Setup,

pynqZ1_v205.pdf