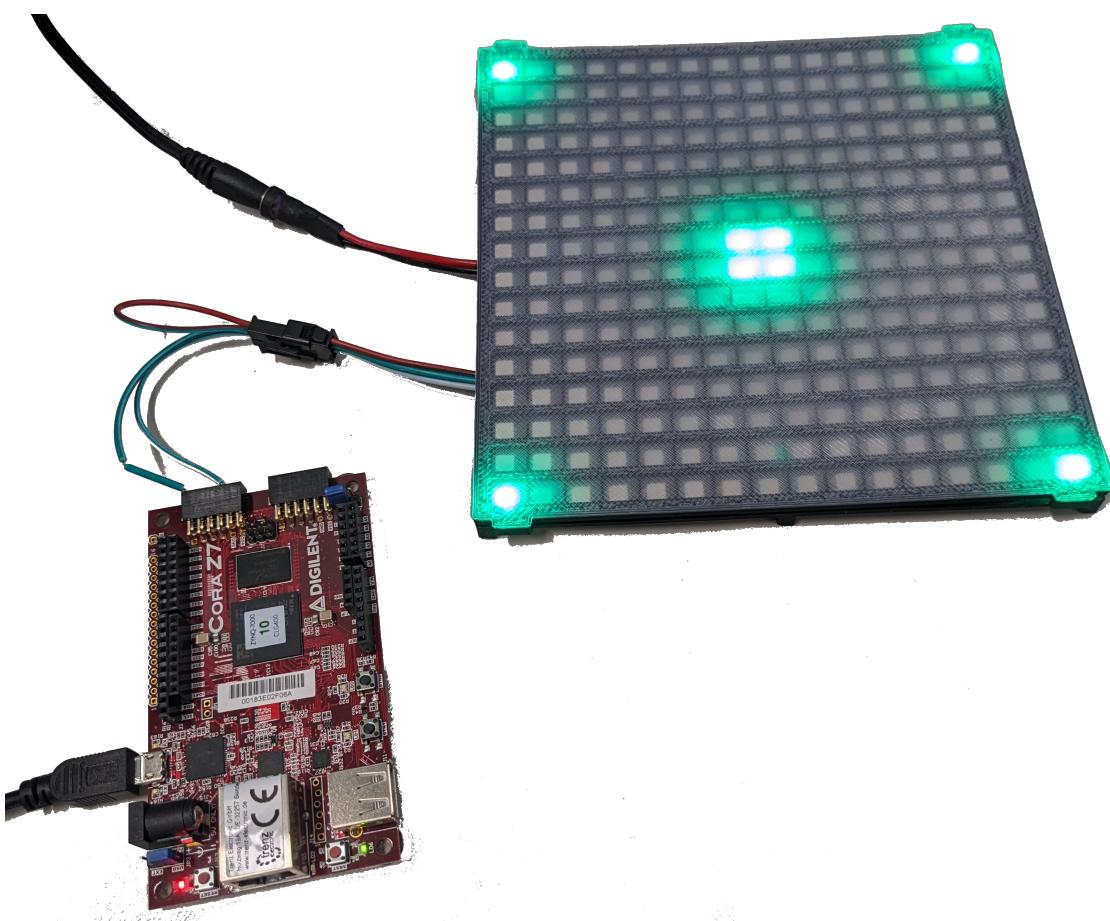


Ansteuerung von adressierbaren LED Vierkanal mit je 256 LED

Benedikt Gareis



Name:

Benedikt Gareis

Matrikelnummer:

3324411

Datum:

16. April 2024

Inhaltsverzeichnis

1 Einleitung	1
2 Informationsbeschaffung	1
2.1 CORA Z7 Development Board	1
2.2 WS2812B LEDs	1
3 Entwurf und Konzept	3
3.1 Hierarchie	3
3.2 FPGA - Physical Layer	3
3.3 IP-Core	3
3.4 Timing	5
3.5 Datentransfer	6
3.6 Blockdesign	7
3.7 Direct Memory Access (DMA) - Direct Memory Access	7
3.8 Microcontroller	7
3.9 Fehlersuche	9
4 Fazit	10
5 Anhang	I
5.1 Blockschaltbild Intellectual Property (IP)-Core	I
5.2 Blockschaltbild Blockdesign	II
6 Quellen und Verzeichnisse	III

1 Einleitung

In diesem Projekt wird die Aufgabe gelöst eine Ansteuerung für WS2812B Light Emitting Diodes (LEDs) zu realisieren. Diese LEDs besitzen einen integrierten Microcontroller und sind über eine einfache serielle Schnittstelle einzeln adressierbar. Um die gewünschte Farbe einzustellen, müssen die Helligkeitswerte der drei Farben, rot, grün und blau, gesendet werden.

Es wurde zusätzlich die Anforderung festgelegt vier Kanäle mit einer maximalen, aber variablen, Anzahl von 256 LEDs, anzusteuern. Die Bildwiederholrate wurde auf mindestens 60 Hz angelegt um kein Flackern oder ähnliche Bildartefakte zu erzeugen. Die Kanäle sollen unabhängig voneinander funktionieren.

2 Informationsbeschaffung

2.1 CORA Z7 Development Board

Das Cora Z7 besitzt einen Zynq-7000 von Xilinx welcher als All-Programmable System-on-Chip (APSoC) beschrieben wird. Es stand das Entwicklungsboard mit dem APSoC XC7Z010-1CLG400C zur Verfügung. Dieses bietet einen Cortex-A9 Prozessor sowie einen Artix-7 Field Programmable Gate Array (FPGA) mit 17.600 lookup tables und 35.200 Flip-Flops was mehr als ausreichend sein wird.

Da die Datenübergabe vom Microcontroller zum FPGA über DMA gedacht ist, wird auch der DDR3L Memory Baustein verwendet. Es stehen bis zu acht DMA Kanäle zur Verfügung.

2.2 WS2812B LEDs

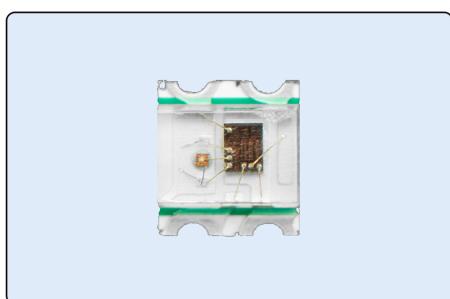
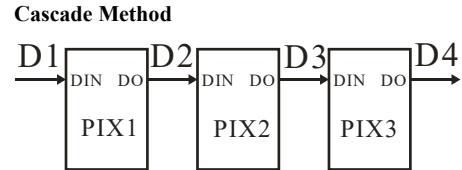
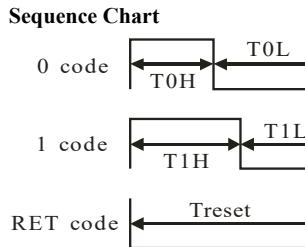


Abbildung 1: Bauform LED WS2812B
[1]

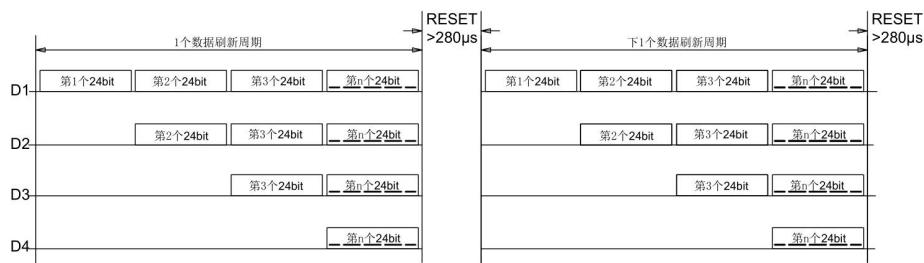
Die verwendeten LEDs wurden von World-Semi entwickelt und sind in vielen Applikationen vertreten. Es gibt sie auf Streifen als Meterware, als Matrix Module mit bis zu 32×32 LED, einzeln oder auch in den verschiedensten Formen aufgelötet.

Die SMD Komponente beinhaltet nicht nur die LEDs, sondern auch den zugehörigen Microcontroller, welcher sich im Gehäuse befindet und die LEDs direkt ansteuert. Die weite Verbreitung ist unter anderem auch auf die simple Ansteuerung zurückzuführen.

Diese LEDs besitzen vier Anschlüsse **Versorgung**, **Ground**, **Data in** und **Data out**. Das Protokoll sieht vor, dass die Anzahl der LED bekannt ist und jeweils 8 Bit pro Farbe je LED hintereinander gesendet werden, die Reihenfolge ist mit grün, rot und dann blau vorgegeben. Dies ist abweichend von der normalen Ansteuerungsreihenfolge welche RGB, also rot, grün, blau ist.



Data Transmission Method



Note: The data of D1 is send by MCU, and D2, D3, D4 through pixel internal reshaping amplification to transmit.

Composition of 24bit Data

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Note: Data transmit in order of GRB, high bit data at first.

Abbildung 2: Datenblattauszug (Mouser) [2]

Die erste LED nimmt sich die ersten 24 Bit, verarbeitet diese und gibt alle weiteren Daten von **Data in** an **Data out** weiter an die nächste LED. Sobald eine Sendepause, in Abbildung 2 als **Treset** bezeichnet, angelegt wird, ist die Übertragung beendet.

Da die Dokumentation dieser LEDs und auch die konkrete Bezeichnung der verbauten LEDs nicht eindeutig zuordenbar ist, wird auf zwei Datenblätter zurückgegriffen. Allgemein kann gesagt werden, dass diese LEDs unglaublich tolerant mit der Ansteuerung sind.

3 Entwurf und Konzept

3.1 Hierarchie

Die Hierarchie wird wie folgt implementiert. Der Microcontroller wird Daten generieren und im Arbeitsspeicher über den DMA ablegen. Diese werden, sobald sie mit dem Status gültig belegt werden, vom DMA direkt weiter an den *Physical Layer* gegeben. Dieser fungiert als Master für den IP-Core welcher in zwei Unterblöcke aufgeteilt ist, die auch nach demselben Prinzip arbeiten. Der langsamste Block, die LEDs, sind also ganz unten in der Hierarchie angeordnet und geben somit auch den Takt vor.

All Funktionsblöcke sind um vielfaches schneller wie Datenrate für die LEDs, ein Problem stellt die Ansteuergeschwindigkeit also nicht dar. Darauf wird in Kapitel 3.4 näher eingegangen.

3.2 FPGA - Physical Layer

Die Aufgabe des FPGA wird es sein, das korrekte *Timing* zu generieren und die Fähigkeit bereitzustellen, die gewünschten Farb- und Helligkeitswerte zu senden. Das Konzept sieht vor, dass die Daten direkt aus dem Random Access Memory (RAM) geholt werden, sobald diese zur Verfügung stehen.

Um die gewünschten Kanalanzahl zu erreichen, muss der IP-Core mit dem entsprechendem DMA-Kanal instanziert und implementiert werden. Die interne Kommunikation im FPGA zwischen den Modulen wird über den Advanced eXtensible Interface (AXI) Bus realisiert.

3.3 IP-Core

Der IP-Core ist nach außen hin AXI-Stream kompatibel gehalten und das Protokoll nach Spezifikation 1.0[3] implementiert. Die Eingänge sind demnach entsprechend benannt und die Datentypen und Busbreiten korrekt gewählt.

Da das Protokoll einen **nReset** vorschreibt, der IP-Core allerdings als **Reset** erstellt wurde, existiert eine Negation des Eingangs **aresetn**.

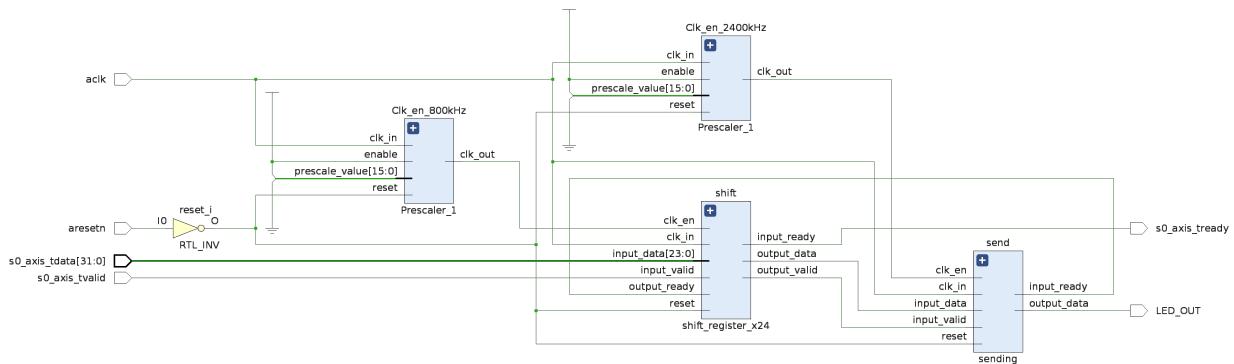


Abbildung 3: Blockschaltbild des IP Cores
eine größere Version ist im Anhang Kapitel 5.1 zu finden

Input:

Die Eingänge sind einfach und AXI-Stream kompatibel gehalten. Der Daten-Input **s0_AXIS_tdata** ist ein 24 Bit breiter Vector, welcher kompatibel mit den bestehenden Busbreiten vom DMA sein muss. Da sich dieser nur auf 32 Bit konfigurieren lässt, ist im *Top Modul* eine Konvertierung von 32 auf 24 Bit vorhanden.

Mehr zur Konnektivität zum AXI-Bus im Kapitel 3.5.

Es sind verschiedene Funktionsblöcke erstellt worden, um die gewünschten Funktionen zur Verfügung zu stellen. Diese werden folgend kurz erläutert:

shift_register_x24:

Hier wird ein Schieberegister realisiert, welches parallel die 24 Bit des **s0_AXIS_tdata** lädt. In diesen stehen die Farb- und Helligkeitswerte einer LED, welche drei Farben besitzt. Diese Informationen werden dann, nach Anforderung, seriell an den nächsten Block ausgegeben. Die eingelesenen Bits werden in diesem Dokument als Logik-Bits bezeichnet, um die Unterschiede zum nächsten Block zu vermeiden. Bevor das Schieberegister leer wird, wird ein neuer 24 Bit Vector vom DMA angefordert.

sending:

Dieser Block nimmt die Daten des Schieberegisters an und gibt diese in dem richtigen Takt für die LEDs aus. Sobald ein Logik Bit gesendet wurde, wird signalisiert, dass die Übertragung beendet wurde und das nächste Logik Bit benötigt wird. Die ausgegebenen Bits werden nachfolgend als Transfer Bits bezeichnet.

Ein Logik Bit besteht aus drei Transfer Bits.

Genauer aufgeschlüsselt wird eine logische 1 durch das *Pattern* 110 und eine 0 durch 100 dargestellt. Es ist nun zu erkennen, dass der **sending**-Block drei mal schneller arbeitet wie das Shift-Register, dies wird durch den nächsten Funktionsblock realisiert.

pre_scaler:

Der *Prescaler* Block teilt die 100 MHz in die erforderlichen *Timing* Impulse. Dieser Block ist in zwei Instanzen für 2,4 MHz und 800 kHz vorhanden.

3.4 Timing

Die folgende Tabelle visualisiert das zur Verfügung stehende zeitliche Potenzial der Programmable Logic (PL):

	Logik Bits	Transfer Bits	Formel	Dauer
		1	$T_T = \frac{1}{800 \text{ kHz}}$	1,25 µs
	1	3	$T_L = 3 \cdot T_T$	3,75 µs
1 LED	24	72	$T_{LED} = 24 \cdot T_L$	90 µs
256 LED	6144	18432	$T = 256 \cdot T_{LED}$	23,04 ms
Komplett (Adafruit)	6144	18432	$T_{best} = 256 \cdot T_{LED} + T_{reset}$	23,09 ms
Komplett (Mouser)	6144	18432	$T_{worst} = 256 \cdot T_{LED} + T_{reset}$	23,32 ms

Es ergibt sich, dass für diese Anzahl an LEDs, eine Ansteuerung mit 60 Hz, aus der Spezifikation heraus nicht möglich ist. Da die beiden Datenblätter von Adafruit[4] und Mouser[2] unterschiedliche Werte für T_{reset} zeigen, wird beides berechnet. Die Ansteuerfrequenz beträgt im schlechtesten Falle somit $f_{ges} = \frac{1}{23,32 \text{ ms}} = 42,88 \text{ Hz}$.

Um nun auch den Microcontroller zu betrachten, wird überlegt, dass dieser die vier Kanäle unabhängig voneinander mit Daten versorgt. Es wird ein Buffer aus 256×32 Bit beschrieben, welcher an den DMA übergeben wird. Es ist also nötig, dass spätestens alle 23,09 ms neue Daten vorliegen. Da der Zynq mit 125 MHz Taktet wird davon ausgegangen, dass dies immer gegeben ist.

3.5 Datentransfer

Der DMA Controller wird als Master festgelegt und gibt die Daten an den IP-Core weiter. Somit liegen die Daten auf seinem Ausgang **s0_AXIS_tdata** an und gleichzeitig wird **s0_AXIS_tvalid** auf 1 gesetzt, um die Gültigkeit zu signalisieren. Dadurch erkennt der Slave, hier der erstellte IP-Core und in diesem das Schieberegister, dass neue, gültige Daten vorhanden sind.

Das Schieberegister übernimmt diese Daten und quittiert dies mit einem einen Zyklus langem *High-Signal* auf **s0_AXIS_tready**. Dieses Verfahren wird auf dem AXI-Bus verwendet und die Spezifikation dazu ist bildlich dargestellt in der Abbildung 4. In Abbildung 5 ist der Start einer Übertragung in der Vivado Simulation dargestellt.

Dieses Übertragungsverfahren wird auch innerhalb des IP-Cores zwischen dem Schieberegister und dem Sendeblock verwendet.

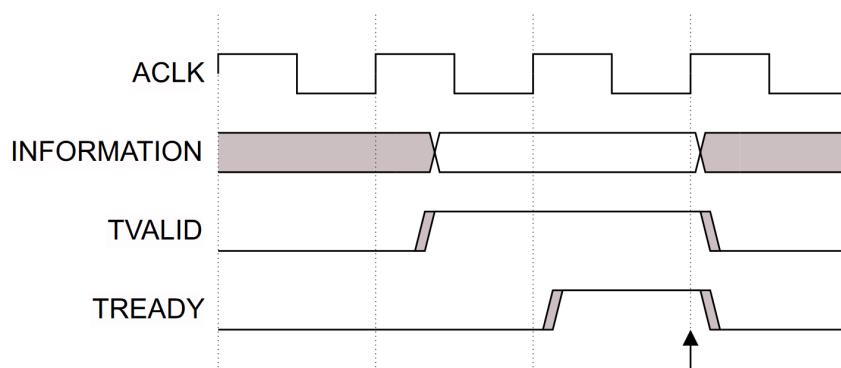


Figure 2-1 TVALID before TREADY handshake

Abbildung 4: Transfer Protocol nach Spezifikation [3]

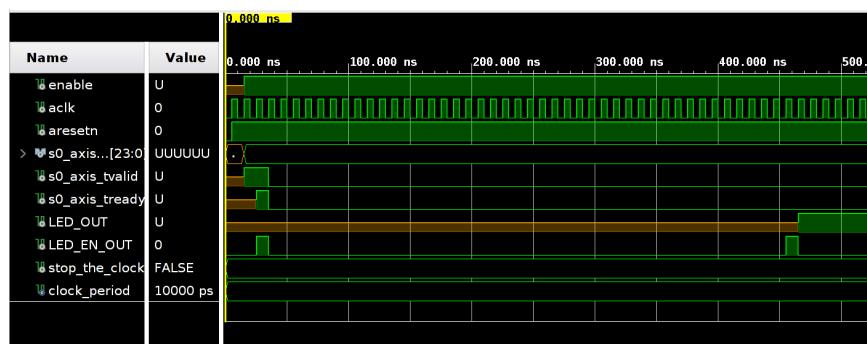


Abbildung 5: Simulation des IP-Cores

3.6 Blockdesign

Dieser IP-Core wird in das Blockdesign eingefügt und die ganze Schaltung wie folgt dargestellt.

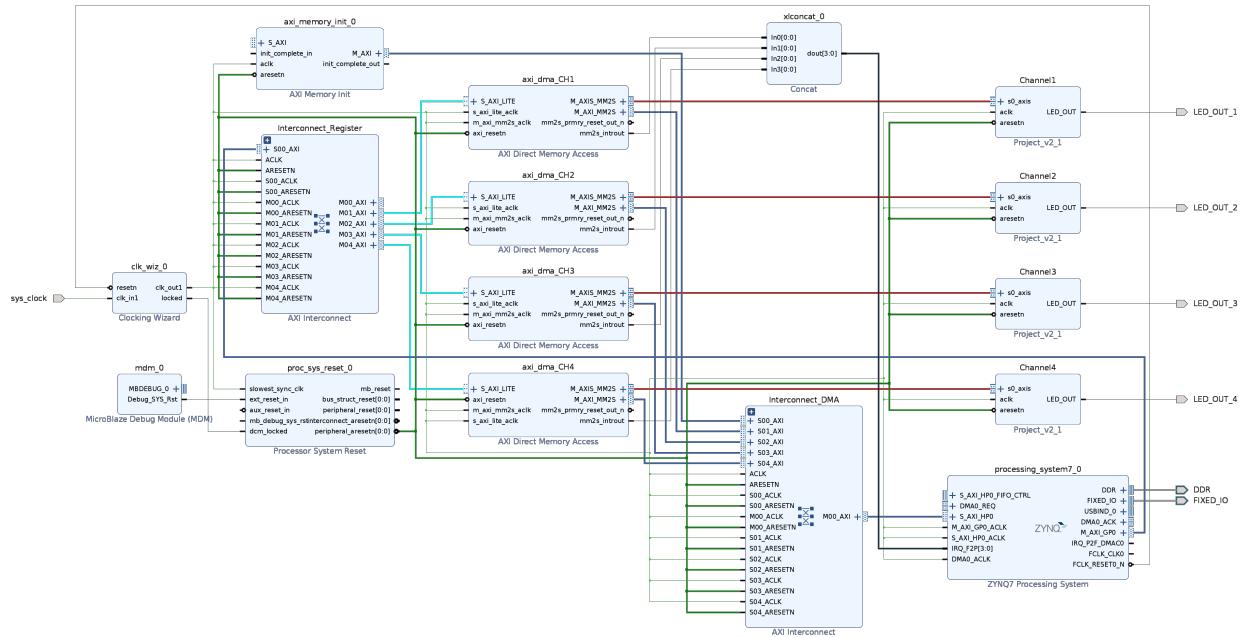


Abbildung 6: Blockschaltbild des Block Designs

eine größere Version ist im Anhang Kapitel 5.2 zu finden

In Abbildung 6 wird sichtbar wie der IP-Core mit dem zugehörigen DMA Kanal viermal instanziert wird, um alle geforderten Kanäle abzubilden.

3.7 DMA - Direct Memory Access

Der DMA fungiert als Master gegenüber dem IP-Core und stellt die Daten direkt bereit. Über zwei *Interconnects* werden die vier DMA Kanäle mit dem ZYNQ verbunden. Der Register *Interconnect* gibt als Master den DMA Kanälen die Adresse vor. Über den *Interconnect DMA* wird die Verbindung vom zum ZYNQ und dessen *High Performance AXI* ermöglicht und dadurch der DD3 Speicherchip verbunden.

Um die Interruptfunktionen der DMA Module zu verwenden, wird ein *Concat* Modul vorgeschaltet und dieses direkt auf den Interrupt ReQuest (IRQ) des ZYNQ gefuhrt. Konfiguriert werden diese dann so, dass ein Interrupt auslosen, wenn das Modul Daten in Richtung PL gesendet hat. Dies ist dann der Tx-Pfad des IRQ.

3.8 Microcontroller

An oberster Stelle der Hierarchie steht der ZYNQ. Die Aufgabe des ZYNQ ist es den DMA Kanal mit Daten zu befüllen, diese generiert er und schreibt diese in den DMA.

Es wurden keine Transmit Buffer verwendet, da der Code nicht auf Performance ausgelegt ist und keine Zeitkritische Funktion erstellt wird.

Da der Funktionsumfang des APSoC sehr groß ist wurden die benötigten Informationen aus der technischen Dokumentation [5] und den diversen Beispielprogrammen von Xilinx auf Github [6] als Hilfe genommen.

DMA Initialisierung

Um die DMA Funktion des APSoC verwenden zu können, müssen die einzelnen Kanäle erstellt und initialisiert werden. Dazu werden die Strukturen erstellt, an die von Xilinx bereitgestellten Funktionen übergeben, welche dann mit den entsprechenden Parametern die DMA Kanäle initialisieren. Besonders ist hier, dass nur eine Richtung aktiviert wird. Der DMA hat nur Zugriff auf den Physical Layer und kann keine Daten zum Microcontroller senden.

Interrupt Initialisierung

Die Initialisierung des *Interrupthandlers* ist komplex und vom Ablauf her sehr starr, er erforderte einiges an Nachforschung. Um die vier Kanäle unabhängig voneinander zu befüllen, wenn diese bereit sind, ist die Funktion über *Interrupthandler* ausgelegt. Die andere und einfachere mögliche Variante über Polling ist nicht sinnvoll und wurde nur zu Testzwecken für einen Kanal implementiert. Da die Anforderung allerdings mehrere Kanäle vorschreibt, wurde auf ein *Interrupt* Konzept umgebaut. Jeder DMA Kanal besitzt eine eigene Verbindung zum ZYNQ, welche in der Hardware umgesetzt und in der Software richtig eingestellt werden muss.

Main

Die *Main* enthält einen Block, um alle notwendigen Initialisierungen aufzurufen und läuft danach in einer Dauerschleife in der auf die *Flags* der *Interrupts* gewartet wird. Ist die Bedingung erfüllt, wird für den korrespondierenden Kanal die Funktion zum Update der Daten aufgerufen. Die Farbe der LED ist hier eine Konstante. Soll diese sich auch ändern, muss eine weitere Funktion eingebaut werden und hier der Farbwert angepasst werden. Aus der *Main* wird auch die Funktion **SimpleTransfer** Funktion des DMA aufgerufen, welche die Daten aus dem Buffer übergibt.

Update_Channel

Hier wird eine einfache Animation erstellt, welche den globalen Buffer eines übergebenen Kanals beschreibt. Um einen angenehmen Farbverlauf dazustellen wird eine Sinusfunktion verwendet, die Kanalnummer verändert die Phase, damit ein Unterschied zwischen den vier Kanälen erkennbar ist.

Damit die Animation einfach in Kartesischen Koordinaten erstellt werden kann, wird eine Funktion aufgerufen, die diese in die Anordnung der LEDs umwandelt.

XYtoSerpentine

Diese Funktion überprüft, ob die derzeitige Koordinate in einer ungeraden Zeile ist und korrigiert dann nach.

3.9 Fehlersuche

FPGA / VHDL

Bei der Erstellung der PL für den IP-Core entstanden keine unüberwindbaren Schwierigkeiten. Die Implementierung der DMA Kanäle erforderte einiges an Recherche konnte aber erfolgreich abgeschlossen werden.

ZYNQ / C-Programm

Das eigentliche Programm welches am Microcontroller läuft, besteht mehr aus Initialisierung und Konfiguration der einzelnen Elemente wie IRQ und DMA, wie aus tatsächlicher Funktionen zur Animation und Ansteuerung der LEDs. Schwierigkeiten hier brachte die korrekte Verknüpfung von PL und IRQ der DMA Kanäle, sowie die korrekte Reihenfolge der Initialisierung..

Cache-Kohärenz

Im ersten Test an der Hardware gab das Cora allerdings nur Nullen aus. Dies zeigte, dass das System so weit funktionierte, aber noch ein Problem bei der Datenübergabe existiert.

Problem:

Bei Systemen, die aus einer Kombination von Microcontroller und PL bestehen, kann es zu Problemen mit der Cache-Kohärenz kommen. Der CPU-Cache speichert Kopien von Daten aus dem Hauptspeicher, um den Zugriff zu beschleunigen. Wenn der Microcontroller die Daten modifiziert, werden diese Änderungen zuerst im Cache statt im Hauptspeicher abgelegt werden. Vor allem in diesem Anwendungsfall wo die Datenmenge relativ gesehen, sehr gering ist, überträgt der Memory-Handler die Daten nicht sofort. Der DMA schickte somit immer die Initialisierungswerte (Null) an die PL.

Lösung: Cache-Flush

Vor der DMA-Übertragung, also nachdem der Buffer erstellt wurde, wird die Funktion `XIL_Cache_Flush()` aufgerufen. Diese Operation zwingt den Memory-Handler, alle modifizierten Daten im Cache sofort in den Hauptspeicher zu schreiben.

HPO Konfiguration

Bei der finalen Fehlersuche stellte sich raus, dass LEDs nur in Paaren angesteuert werden können. Wenn diese direkt in mittels Memset in den Buffer geschrieben werden, dann ist ersichtlich, dass gerade Adressen nicht übernommen werden. Kurz gesagt, der Fehler lag in einer Inkonsistenz der Busbreiten des HPO. In Vivado wurde diese auf 32 Bit gestellt, allerdings in Vitis nie aktualisiert. Durch richtige Konfiguration wurde dieses Problem behoben.

Vitis

Die Bereitgestellte Integrated Development Environment (IDE) Namens Vitis hat zudem einige Besonderheiten. Eine davon ist, dass der *Compiler* zum Beispiel die korrekt eingebundene *math.h* ignoriert. Um diese verwenden zu können, muss dies erzwungen werden, indem die *Linker Settings* angepasst werden.

4 Fazit

In diesem Projekt wurden viele neue Systeme und Abläufe entdeckt. Die Kombination von zwei großen Systemen, dem ZYNQ und dem Artix-7, bot einige Herausforderungen die in viel Recherche und in Gesprächen mit Kommilitonen geklärt wurden. Das Einarbeiten in dieses neue, umfangreiche Konzept mit dieser Hardware erforderte einiges an Umdenken, verglichen mit früheren Projekten mit einem Microcontroller.

Weiterhin bietet AMD mit dem Support Forum eine große Wissensbasis. Es gibt dort sehr viele Einblicke in die verschiedensten Anwendungsmöglichkeiten von FPGA und APSoC, wie auch deren Problem bei der Entwicklung von Systemen.

5 Anhang

5.1 Blockschaltbild IP-Core

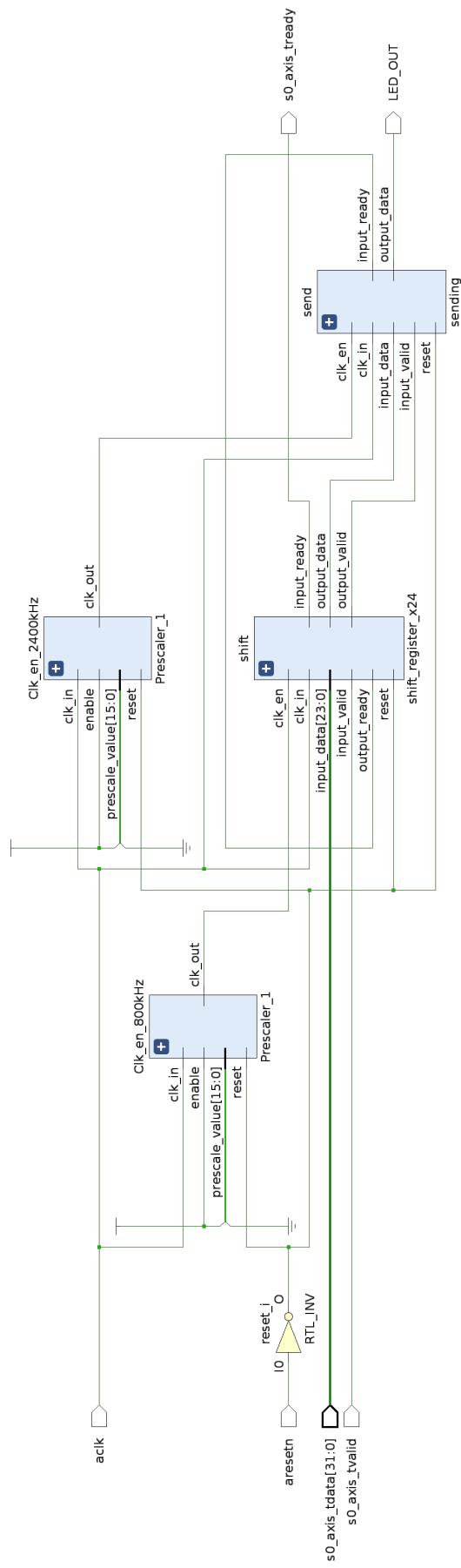


Abbildung 7: Großes Blockschaltbild des IP Cores

5.2 Blockschaltbild Blockdesign

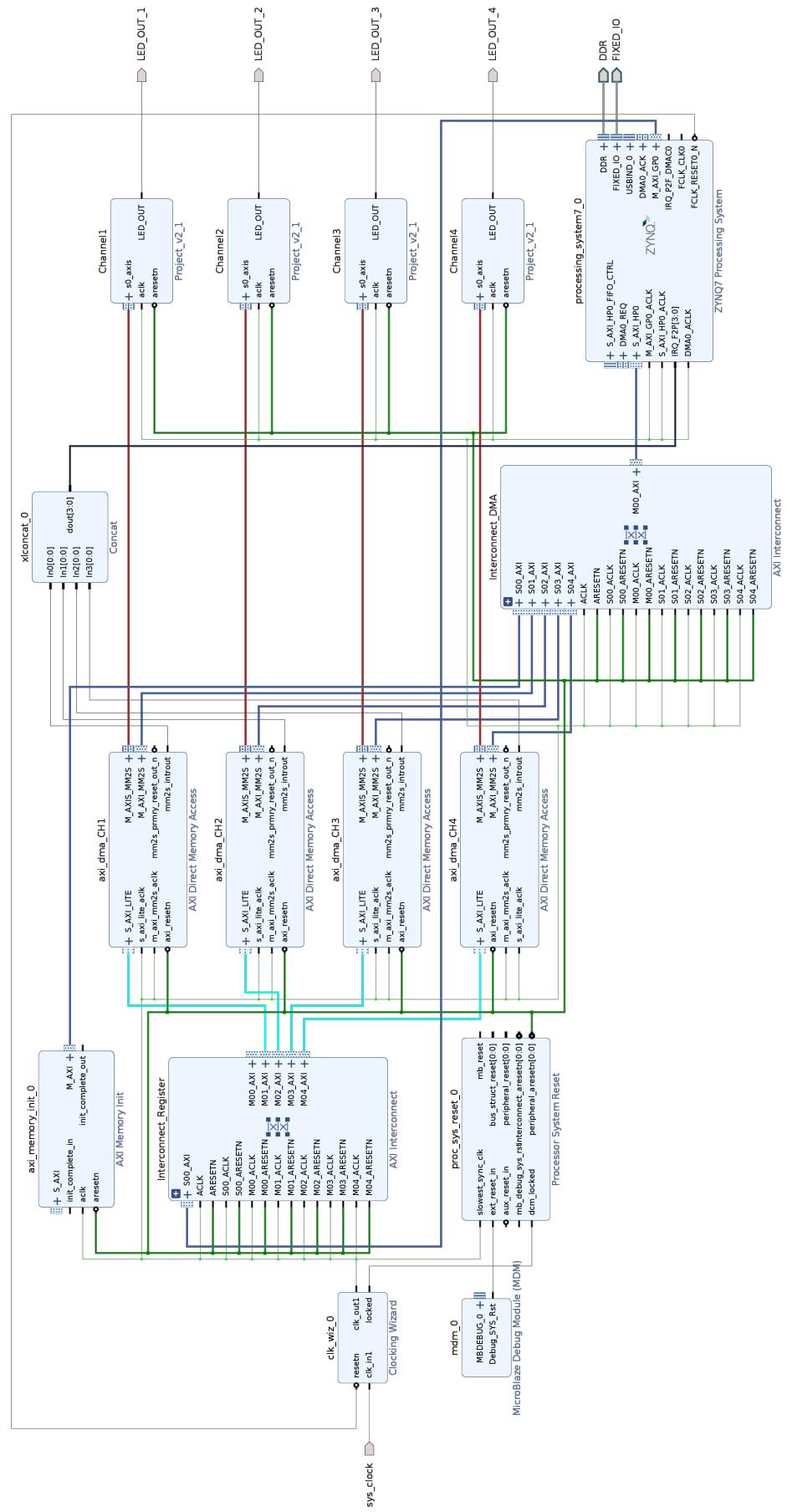


Abbildung 8: Großes Blockschaltbild des Blockdesigns

6 Quellen und Verzeichnisse

Literatur

- [1] *NeoPixel Nano* 2020. URL: <https://www.adafruit.com/product/4684> (besucht am 21.01.2024).
- [2] *NeoPixel WS2812B Datenblatt (Mouser)*. URL: https://www.mouser.com/pdfDocs/WS2812B-2020_V10_EN_181106150240761.pdf (besucht am 21.01.2024).
- [3] *AMBA 4 AXI4-Stream Protocol*. Version 1.0. 3. März 2010. URL: <https://www.arm.com/>.
- [4] *NeoPixel WS2812B Datenblatt (Adafruit)*. URL: <https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf> (besucht am 21.01.2024).
- [5] *Zynq 7000 SoC Technical Reference Manual (UG585)*. URL: <https://docs.xilinx.com/r/en-US/ug585-zynq-7000-SoC-TRM> (besucht am 27.01.2024).
- [6] *embeddedsw.git - repo for standalone software*. URL: <https://github.com/Xilinx/embeddedsw/tree/master> (besucht am 27.01.2024).

Abbildungsverzeichnis

1	Bauform LED WS2812B [1]	1
2	Datenblattauszug (Mouser) [2]	2
3	Blockschaltbild des IP Cores <small>eine größere Version ist im Anhang Kapitel 5.1 zu finden</small>	4
4	Transfer Protocol nach Spezifikation [3]	6
5	Simulation des IP-Cores	6
6	Blockschaltbild des Block Designs <small>eine größere Version ist im Anhang Kapitel 5.2 zu finden</small>	7
7	Großes Blockschaltbild des IP Cores	I
8	Großes Blockschaltbild des Blockdesigns	II

Abkürzungsverzeichnis

APSoC	All-Programmable System-on-Chip
AXI	Advanced eXtensible Interface
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
IDE	Integrated Development Environment
IP	Intellectual Property
IRQ	Interrupt ReQuest
LED	Light Emitting Diode
PL	Programmable Logic
RAM	Random Access Memory