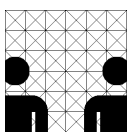
Diplomarbeit
im Studiengang Informatik

Entwurf und Implementierung einer hardwarebasierten Vorverarbeitungseinheit für ein omnidirektionales Sichtsystem

am Arbeitsbereich für
Technische Aspekte Multimodaler Systeme,
Universität Hamburg

vorgelegt von
Christian Piehl
Dezember 2008



betreut von
Prof. Dr. Jianwei Zhang
Dr. Andreas Mäder

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Einleitung

1

1.1 Motivation

Seit Menschengedenken versuchen die Menschen sich ihr Leben einfacher und bequemer zu gestalten. Sie bauten Häuser, um sich vor dem Wetter zu schützen, sie nutzten die Kraft der Tiere, um den Acker zu bewirtschaften und sie bauten Automobile, um sich einfacher und schneller fortbewegen zu können. Auch heutzutage versuchen die Menschen sich das Leben weiter zu erleichtern. Gerade durch die Informatik sind hierbei vielfältige neue Möglichkeiten entstanden dies zu tun. Navigationssysteme helfen uns bei der Orientierung an fremden Orten, das Internet ermöglicht uns innerhalb von Sekunden den Zugriff auf riesige Mengen an Informationen und mobile Geräte wie Mobiltelefone, Smartphones und PDAs geben uns die Möglichkeit überall erreichbar zu sein und überall auf Adressen und Termine zugreifen zu können.

Somit ist es leicht ersichtlich, dass die Menschen den Wunsch hegen auch bei den vielfältigen Aufgaben im Haushalt durch neue Technologien unterstützt zu werden. Eine Möglichkeit dies zu tun ist einen menschenähnlichen Roboter zu konstruieren, der die zu erledigenden Aufgaben auf die gleiche Art und Weise erledigt wie ein Mensch. Auch der Arbeitsbereich TAMS (Technische Aspekte Multimodaler Systeme) am Fachbereich Informatik der Universität Hamburg arbeitet an so einem Roboter (s. Abbildung 1.1).

1.2 Der Service-Roboter TASER

Die TASER (TAMS Service-Robot) genannte Roboterplattform basiert auf einer modifizierten Version der mobilen Plattform „MP-L655“ von Neobotix. Sie ist ausgestattet mit zwei „SICK LMS200“ Laser Entfernungsmessern, einem Industriecomputer mit einem Pentium IV mit 2,4 GHZ und zwei „PA10-6C“ Roboterarmen von Mitsubishi Heavy Industries (MHI) an denen jeweils eine „BarrettHand

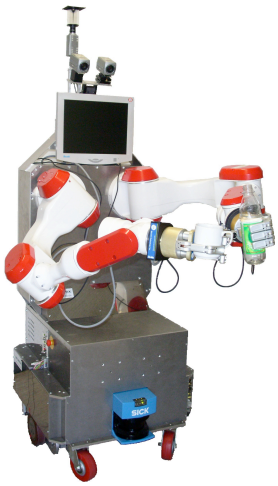


Abbildung 1.1: Der Service-Roboter TASER, Quelle: [TAMS]

BH8-262“ Roboterhand mit 3 Fingern der Firma Barrett Technology Inc. angebracht ist. Des weiteren ist sie mit einem Schwenk/Neige-Kamerakopf bestückt auf dem zwei Firewire-Kameras sitzen, welche Stereosicht ermöglichen und sie ist ausgerüstet mit einem omnidirektionalen Sichtsystem bestehend aus einer hochauflösenden Firewire-Kamera und einer Optik mit einem hyperbolischen Spiegel. Abbildung 1.2 verdeutlicht den Systemaufbau des Roboters. Die Roboterplattform ermöglicht den Studenten und den Mitarbeitern des Arbeitsbereiches das Forschen in einem breiten Themenfeld. Zu diesem gehören u.a. die Navigation und Lokalisation der Roboter-Plattform mit Laserscannern und Kameras, das Greifen oder Modifizieren von Objekten mit den Barretthänden, die dreidimensionale Rekonstruktion der näheren Umgebung anhand von Videodaten, die Entwicklung und Erprobung von Algorithmen auf Basis der Daten der omnidirektionalen Kamera oder der Entwurf eines informationszusammenführenden Systems, welches anhand multimodaler Sensordaten das Gesamtsystem steuert.

1.3 Problembeschreibung

Gleichzeitig in alle Richtungen sehen zu können stellt einen grossen Vorteil dar. Man braucht sich nicht darum zu kümmern, wann die Kamera in welche Richtung schaut und wann man sie woanders hinsehen lässt. Es gibt diverse Methoden, wie man diesen Rundumblick ermöglichen kann (siehe auch [Zha07], Seite 532):

Eine schwenkbare oder rotierende Kamera stellt eine einfache Möglichkeit dar den ganzen Raum optisch zu erfassen, bringt aber auch viele Probleme mit

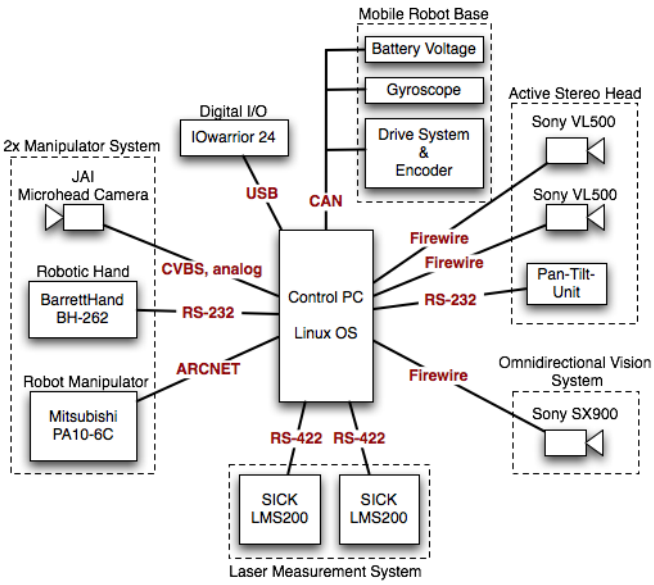


Abbildung 1.2: Systemaufbau des TASER, Quelle: [TAMS]

sich. So kann es zu Ungenauigkeiten bei der Rotation um das optische Zentrum kommen und es entsteht zusätzlicher Energieaufwand für die Bewegung der Kamera. Des weiteren ist die Verarbeitung der Bilder bei dynamischen Szenen schwierig und die Umrechnungen ins Panoramabild sind aufwendig.

Verwendung mehrerer Kameras Bei dieser Methode werden mehrere Kameras so nebeneinander montiert, dass sie gleichzeitig den ganzen Raum erfassen. Ein Beispiel hierfür stellt die RingCam von Microsoft dar (siehe [MicRC]). Sie besteht aus fünf IEEE1394-Kameras, die jeweils eine Auflösung von 640x480 Pixeln haben und erzeugt daraus einen Rundumblick mit einer Gesamtauflösung von 3000x480 Pixeln. Probleme können aber durch die unterschiedlichen Verzeichnungen, Farben und Helligkeiten der einzelnen Kameras entstehen. Ausserdem ist die gemeinsame Kalibrierung der Kameras schwierig.

Weitwinkellinsen haben eine sehr starke Verzeichnung und weisen sehr hohe Anschaffungskosten auf.

Catadioptrische Kamerasysteme bestehen aus konvexen Spiegeln und darunter angebrachten Kameras. Der vertikale Blickwinkel ist unterschiedlich und die volle Auflösung der Kamera wird nicht genutzt. Auch kann nicht jedes omnidirektionale Bild perspektivisch korrekt in ein Panoramabild umgerechnet werden.

Abbildung 1.3: Ein Bild der omnidirektionalen Kamera des TASER, Quelle: [TAMS]

Das beim TASER verwandte Sichtsystem gehört zu den catadioptrischen Kamerasystemen. Hierdurch ergeben sich einige Nachteile. Während das Bild einer normalen Kamera sofort für einen Menschen nutzbar ist, kann man mit dem Bild dieser omnidirektionalen Kamera nicht immer etwas anfangen. Die Orientierung fällt erheblich schwerer und durch die Verzerrung sind viele Details wie z.B. Schrift deutlich schwieriger zu erkennen. Abbildung 1.3 zeigt ein solches durch die omnidirektionale Kamera des TASER geliefertes Bild. Des weiteren basieren nahezu alle bekannten Algorithmen, wie z.B. die vertikale Kantendetektion, auf der normalen Sichtweise und können nicht direkt auf die omnidirektionale Sichtweise umgesetzt werden. Aus diesem Grund ist es sinnvoll das omnidirektionale Bild zuerst in ein Panoramabild umzurechnen und dann mit diesem Bild weiterzuarbeiten. Wie ein so umgerechnetes Bild aussieht zeigt Abbildung 1.4. Diese Umrechnung lässt sich auf dem TASER relativ leicht in Software bewerkstelligen, was allerdings entscheidende Nachteile mit sich bringt. Die Implementierung eines Umrechnungsalgorithmus in Software ist nicht sehr performant und verbraucht sehr viel der kostbaren Rechenleistung, die auf dem TASER nur begrenzt zur Verfügung steht. Nur für

Abbildung 1.4: Das aus Abbildung 1.3 errechnete Panoramabild, Quelle: [TAMS]

diese Aufgabe einen weiteren Rechner zum Einsatz zu bringen würde den Stromverbrauch des Systems drastisch erhöhen und auch in einem höheren Gewicht und einer erhöhten Wärmeabgabe münden. Daher liegt es nahe diese Umrechnung in einer vorgeschalteten Hardware zu erledigen, welche nicht die eben genannten Nachteile einer Softwarelösung mit sich bringt.

1.4 Zielsetzung

Ziel dieser Arbeit ist es eine funktionsfähige Vorverarbeitungseinheit für die omnidirektionale Kamera des TASER auf Basis eines FPGA-Prototypen-Boards zu entwickeln und zu implementieren. Die Einheit soll die Bilder der Kamera über den Firewire-Bus entgegennehmen, sie zu einem Panoramabild umrechnen und anschliessend wieder über den Firewire-Bus ausgeben. Dabei wird es zwingend erforderlich sein, dass die Bilder in einem Zwischenspeicher komplett abgelegt werden, da der zu verwendende Algorithmus nur auf vollständigen Bildern zum Einsatz gebracht werden kann und die Bilder zu gross für eine direkte Verarbeitung durch den FPGA sind. Ausserdem muss sich erst noch zeigen, ob der FPGA in der Lage ist die Bilder ebenso schnell umzurechnen und auszugeben wie sie von der Kamera geliefert werden. Nur dann wäre eine direkte Verarbeitung überhaupt möglich. Aus diesem Grund muss das System so entwickelt werden, dass die Bild- und Datenrate des produzierten Videosignals unabhängig von der Bild- und Datenrate des angenommen Videosignals sind. Des weiteren ist auch nicht klar, ob die Anzahl der Logikelemente des FPGA ausreicht, um die gewünschte Funktionalität komplett aufzunehmen.

1.5 Aufbau der Arbeit

In Kapitel 2 wird die verwendete Hardware vorgestellt, damit ein Überblick darüber gewonnen werden kann, welche Komponenten zum Einsatz kamen.

In Kapitel 3 werden einige Grundlagen kurz erklärt, deren Kenntnis für das Verständnis dieser Arbeit notwendig ist.

Kapitel 4 stellt die Software vor, die für die Entwicklung des Systems notwendig war und beschreibt die Vorgehensweise vom Entwurf bis zum fertigen System.

In Kapitel 5 werden die Ergebnisse dieser Arbeit beschrieben und das fertige System vorgestellt und detailliert erläutert.

Kapitel 6 fasst die gesamte Arbeit zusammen, zieht ein Fazit und erläutert die Möglichkeiten, welche sich mit diesem System nun bieten.

In Kapitel 7 wird abschliessend erörtert, an welchen Stellen das System noch verändert und verbessert werden kann.

Verwendete Hardware

2

Das Herzstück der Hardware bildet das „Nios Embedded Processor Development Board“ von Altera. Das Board ist bestückt mit 64 MiB SDRAM in Form eines SODIMM-Speicherriegels. An den PCI-Mezzanine-Connectors ist eine kleine Zusatzplatine angeschlossen, an der wiederum die Firewire-Platine angesteckt ist. Diese verbindet das System über ein sechspoliges Firewire-Kabel mit der Sony DFW-SX900 Kamera und über ein weiteres sechspoliges Firewire-Kabel mit einem Computer. Abbildung 2.1 zeigt diesen Hardware-Aufbau.

Abbildung 2.1: Der Hardware-Aufbau

2.1 Nios Embedded Processor Development Board

Das „Nios Embedded Processor Development Board“ ist ein FPGA-Prototypen Board von Altera (siehe Abbildung 2.2), welches es ermöglicht einfache eingebettete Systeme zu entwickeln und direkt auf der Hardware zu testen. Es ist vorkonfiguriert mit dem Referenz-Design eines 32 Bit Nios Prozessors, der mit Software geladen werden kann und diese auf dem Board ausführt. Abbildung 2.3 verdeutlicht den Systemaufbau dieses Boards.

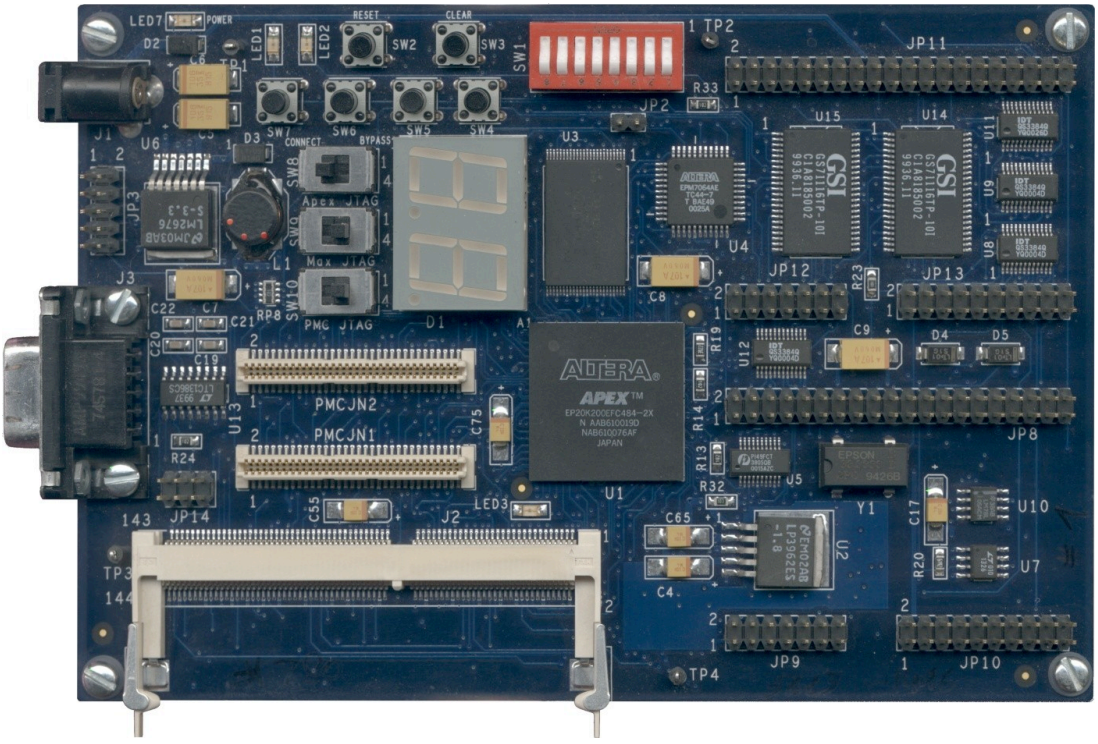


Abbildung 2.2: Das FPGA-Prototypenboard von Altera, Quelle: [TAMS]

Die Eckdaten dieses Boards (aus [Alt03]):

- ein APEX 20K200E Baustein
- 1 MiB (512 K x 16 Bit) Flash Speicher - vorkonfiguriert mit dem 32 Bit Nios Referenz Design
- 256 KiB SRAM (in zwei 64 K x 16 Bit Chips)
- On-board logic zur Konfiguration des APEX Bausteins vom Flash Speicher
- Small Outline DIMM (SODIMM) Sockel, zur Verwendung von Standard SDRAM Modulen

- Zwei IEEE-1386 peripheral component interconnect (PCI) mezzanine connectors (PMC)
- Ein serieller RS-232 Anschluss
- Ein Benutzerdefinierbarer 8 Bit DIP-Schalter
- Vier Benutzerdefinierbare Tasten
- Zwei 7-Segment LED Displays
- Zwei frei verwendbare LEDs
- Joint test action group (JTAG) Anschluss für die ByteBlaster II und MasterBlaster Verbindungskabel
- Power-on reset Schaltung

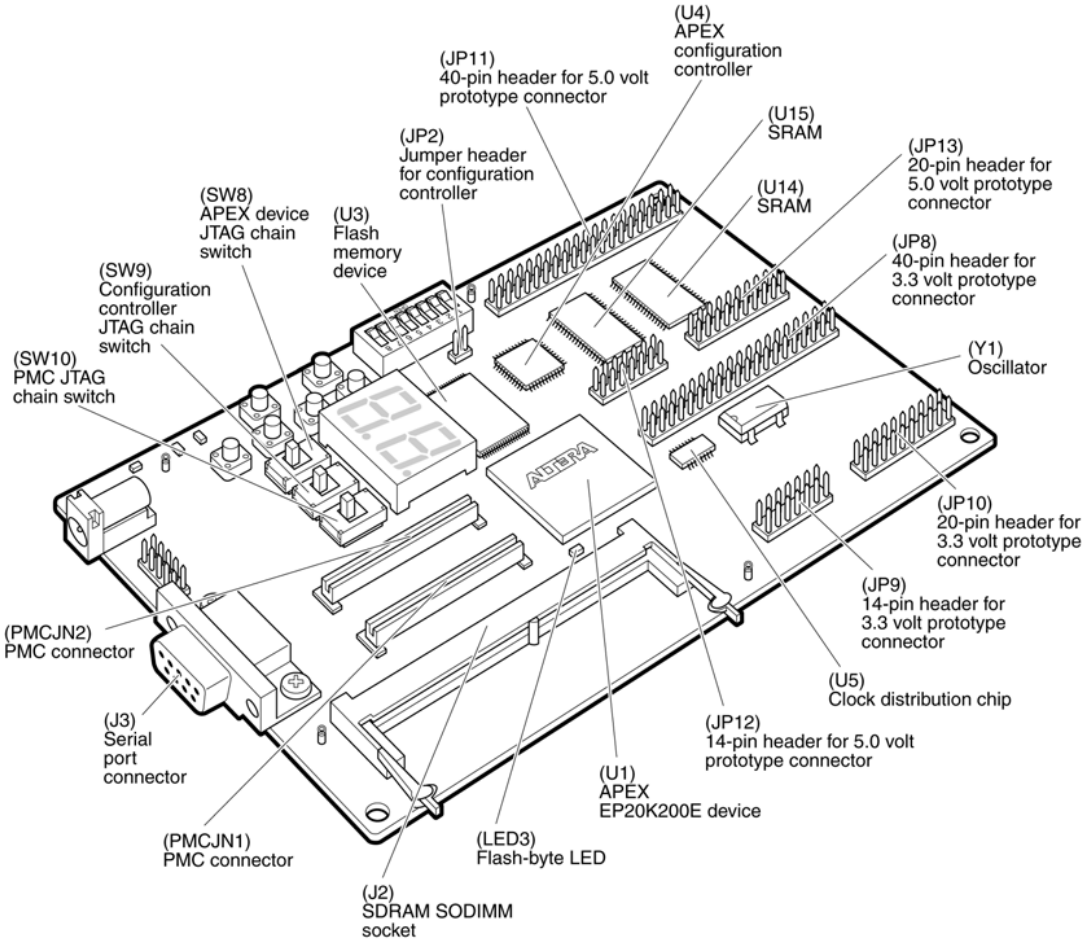


Abbildung 2.3: Der Systemaufbau des FPGA-Prototypenboards, Quelle: [Alt03]

Apex 20K200E

Der „Apex 20K200E“ ist ein FPGA der Firma Altera. Er kann bis zu 8.320 Logikelemente aufnehmen und bringt 106.496 Bit Speicherzellen mit, die von der Logik direkt verwendet werden können, um z.B. FIFOs zu realisieren. Weitere Details dieses FPGA findet man in [Alt04].

2.2 SDRAM

Der eingesetzte SDRAM-SODIMM war ein 64 MiB Modul von Micron vom Typ MT8LSDT864H-10E (siehe [Mic04]). Dieses verwendet Speicher-ICs vom Typ MT484M16A2-8E, welche in [Mic02] beschrieben werden.

2.3 Firewire-Platine

Die Firewire-Platine (siehe Abbildung 2.4) wurde am Arbeitsbereich TAMS auf Basis eines Referenz-Designs von Texas Instruments gefertigt. Sie ist bestückt mit dem Texas Instruments TSB41AB3 (siehe [TSB41]), einem Physical Layer Controller Chip und dem Texas Instruments TSB12VL01B (siehe [TSB12]), einem Link Layer Controller Chip. Sie besitzt drei Firewire 400 Anschlüsse und wird durch eine kleine Tochterplatine über die PMC-Anschlüsse an das FPGA-Board angeschlossen und verbindet somit das FPGA-Board mit dem Firewire-Bus.

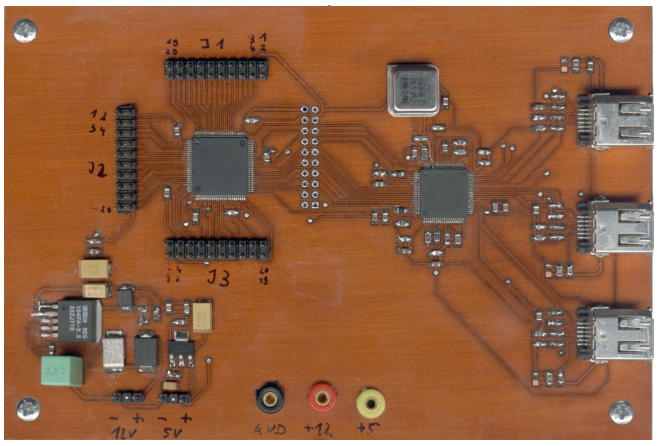


Abbildung 2.4: Die Firewire Erweiterungsplatine, Quelle: [TAMS]

2.4 Das omnidirektionale Sichtsystem

Für das Sichtsystem kam eine Sony „DFW-SX900“ zur Verwendung (siehe Abbildung 2.5). Die „DFW-SX900“ ist eine hochauflösende Firewire-Kamera. Sie arbeitet mit der SXGA Auflösung von 1280 x 960 Pixeln und überträgt die Daten mit bis zu 400 Megabit pro Sekunde über den Firewire-Bus. Weitere Informationen findet man in [Son01]. Sie unterstützt die „IEEE 1394 Digital Camera Specification“ in der Version 1.30. Diese wird in Abschnitt 3.6 näher erläutert. Die Kamera ist ausgerüstet mit der Linse „HF9HA-1B“ von Fujinon (siehe Abbildung 2.6) und einem omnidirektionalen Sichtsystem von ACCOWLE Vision vom Typ „Panorama Eye“ (siehe Abbildungs 2.7).

Abbildung 2.5: Die hochauflösende Kamera „DFW-SX900“ von Sony

Abbildung 2.6: Die Linse „HF9HA-1B“ von Fujinon, Quelle: [Fujinon]

Abbildung 2.7: Mehrere omnidirektionale Sichtsysteme vom Typ „Panorama Eye“ von ACCOWLE Vision, Quelle: [Accowle]

Grundlagen

3

3.1 Das Firewire-Bussystem

Firewire ist ein serielles Bussystem, welches ursprünglich von Apple als SCSI-Nachfolger entwickelt wurde.

Einige Eigenschaften von Firewire:

Peer-to-Peer: Firewire benötigt, im Gegensatz zu USB, keinen zentralen Host. Dadurch können alle angeschlossenen Geräte direkt untereinander Kommunizieren ohne eine Host-CPU zu benötigen.

Erweiterbarkeit: Es können bis zu 63 verschiedene Geräte an einem Bus hängen und bis zu 1024 Busse können über Brücken miteinander verbunden werden.

Selbstkonfiguration: Der gesamte Bus konfiguriert sich eigenständig selber. Es wird dazu keine Host-CPU benötigt.

Netzwerkfähig: Aufgrund der hohen Geschwindigkeit und der grossen Erweiterbarkeit, eignet sich Firewire auch für den Aufbau von Netzwerken.

Hot-Plug: Geräte können während des Betriebes ein- und ausgesteckt werden.

Plug and Play: In den Bus eingebrachte Geräte werden selbstständig vom Bus erkannt und eingebunden.

grosser Adressraum: Jedes Gerät verfügt über einen 48 Bit breiten Adressraum, über den auf 256 TiB zugegriffen werden kann.

Stromversorgung: Geräte können über den Bus mit Strom versorgt werden.

zwei verschiedene Kommunikationsarten: Das Firewire-Protokoll kennt zwei Kommunikationsarten: Bei der asynchronen Kommunikation wird den Geräten die Zustellung der Daten garantiert. Bei der isochronen Kommunikation hingegen werden die Daten in konstanten Intervallen über den Bus gesen-

det. Dabei wird dieser Kommunikation eine vorher fest zugesagte Bandbreite garantiert.

faire Arbitrierung: Ein fairer Algorithmus sorgt dafür, dass isochrone Verbindungen eine zugesicherte Bandbreite erhalten, während asynchrone Verbindungen einen fairen Zugang zum Bus erhalten.

Fehlererkennung und -behandlung: Das Erkennen und Behandeln von Fehlern ist Teil des Firewire-Protokolls.

Status- und Kontrollregister: Jedes Gerät besitzt eine fest vorgegebene Menge an „Control and Status Registers“ (CSR), die dazu benutzt werden können den Status des Gerätes auszulesen oder es zu konfigurieren. Dazu gehört auch das „Configuration ROM“, welches Informationen darüber enthält, von welchem Hersteller ein Gerät stammt und welche Fähigkeiten es besitzt.

Konfiguration

Der Bus konfiguriert sich eigenständig, indem ein Bus Reset ausgeführt wird, welcher dafür sorgt, dass sich alle angeschlossenen Geräte neu auf dem Bus anmelden. Bei diesem Bus Reset werden verschiedene Rollen zwischen den Geräten verteilt, welche diese ausführen können:

Bus Master: Der Bus Master stellt Dienste bereit, die allen Geräten am Bus zugutekommen. So erstellt er u.a. eine „Speed Map“, welche die Geschwindigkeiten auflistet, die zwischen zwei Geräten am Bus genutzt werden kann. Auch die Verteilung der Stromversorgung ist Aufgabe des Bus Master.

Cycle Master: Bei der isochronen Kommunikation ist es wichtig, dass alle Geräte zeitlich synchronisiert ihre Daten senden. Hierfür nimmt ein Gerät am Bus die Rolle des Cycle Masters ein, welcher eine interne Uhr verwaltet und in regelmässigen Abständen „Cycle Start“ Pakete versendet, die den Anfang eines Zyklus kennzeichnen.

Isochronous Ressource Manager: Der Isochronous Ressource Manager kümmert sich um die Ressourcenverteilung der isochronen Kommunikationskanäle. Er behält die Übersicht über die zugeteilten Kanalnummern und die zugeteilte Bandbreite.

Jede dieser Rollen kann von einem anderen Gerät ausgeführt werden. Es kann sich aber auch ein Gerät um alle Rollen kümmern.

Asynchrone Kommunikation

Bei der asynchronen Kommunikation werden die Daten über eine explizite 64 Bit Adresse an ein bestimmtes Gerät am Bus gesendet. Hierbei findet eine Fehlerüberprüfung statt. Anhand des „Response Code“ können im Fehlerfall Datenpakete erneut angefordert werden. Die Kommunikation basiert dabei auf drei Transaktionstypen: „Read“, „Write“ und „Lock“. Für jeden dieser Typen gibt es zwei Arten von Paketen: „Request“ und „Response“. Ein Datentransfer zwischen zwei Geräten erfolgt immer indem ein „Request“-Paket versendet wird, welches mit einem „Response“-Paket beantwortet wird. Dabei stellt „Read“ eine Leseanfrage und „Write“ eine Schreibanfrage dar. Mit „Lock“ hingegen können Speicherbereiche eines Gerätes für einen Zeitraum so blockiert werden, dass kein anderes Gerät auf sie zugreifen kann.

Durch das Bus-Management wird dafür gesorgt, dass für die asynchrone Kommunikation mindestens 20 % der gesamten Bandbreite zur Verfügung stehen. Diese wird unter allen Geräten fair verteilt, so dass jedes Gerät irgendwann seine Daten senden kann.

Isochrone Kommunikation

Bei der isochronen Kommunikation werden die Daten nicht direkt an ein Gerät, sondern als „Broadcast“-Nachricht an alle Geräte gesendet. Eine Empfangsbestätigung wird hierbei nicht versendet und es ist auch nicht möglich Daten erneut anzufordern. Bevor ein Gerät isochrone Daten versenden kann, muss es erst beim „Isochronous Ressource Manager“ einen Kanal und die benötigte Bandbreite für die Kommunikation erhalten.

Die isochrone Kommunikation besitzt eine höhere Priorität als die asynchrone Kommunikation und es können maximal 80 % der gesamten Bandbreite des Busses für diese Kommunikation benutzt werden. Die isochrone Kommunikation eignet sich somit vor allem für Audio- und Videodaten, die zwar zeitkritisch sind, aber bei denen es nicht darauf ankommt, ob alle Daten absolut einwandfrei ihr Ziel erreichen.

Weitere Informationen über das Firewire-Protokoll findet man in [WikFW] oder im sehr ausführlichen [And99].

3.2 Verwendung des Link Layer Controller Chips TSB12LV01B

Die Verwendung des TSB12LV01B erfolgt über den Zugriff auf seine Register. Hierüber werden die Einstellungen vorgenommen, die für die gewünschte Funktion notwendig sind. Ausserdem werden die Register dazu benutzt den Status des Chips und den Zustand der Eingangs- und Ausgangs-FIFOs zu erfahren (siehe Abbildung 3.1). Der Zugriff erfolgt dabei durch einem normalen Lese- oder Schreibvorgang. Damit dieser aber nicht immer ausgeführt werden muss, nur um z.B. herauszubekommen ob sich Daten im Eingangs-FIFO befinden, ist es möglich einige der Register auf die „General Purpose Output“-Ausgänge zu legen. So liegt dort immer der aktuelle Wert der ausgewählten Register an und kann direkt verwendet werden.

FIFOs

Der TSB12LV01B besitzt einen FIFO, welcher sich in drei Teile gliedert: GRF, ATF und ITF. Siehe dazu Abbildung 3.2.

GRF: Der GRF ist der „General Receive FIFO“ und für den Empfang von Daten vorgesehen.

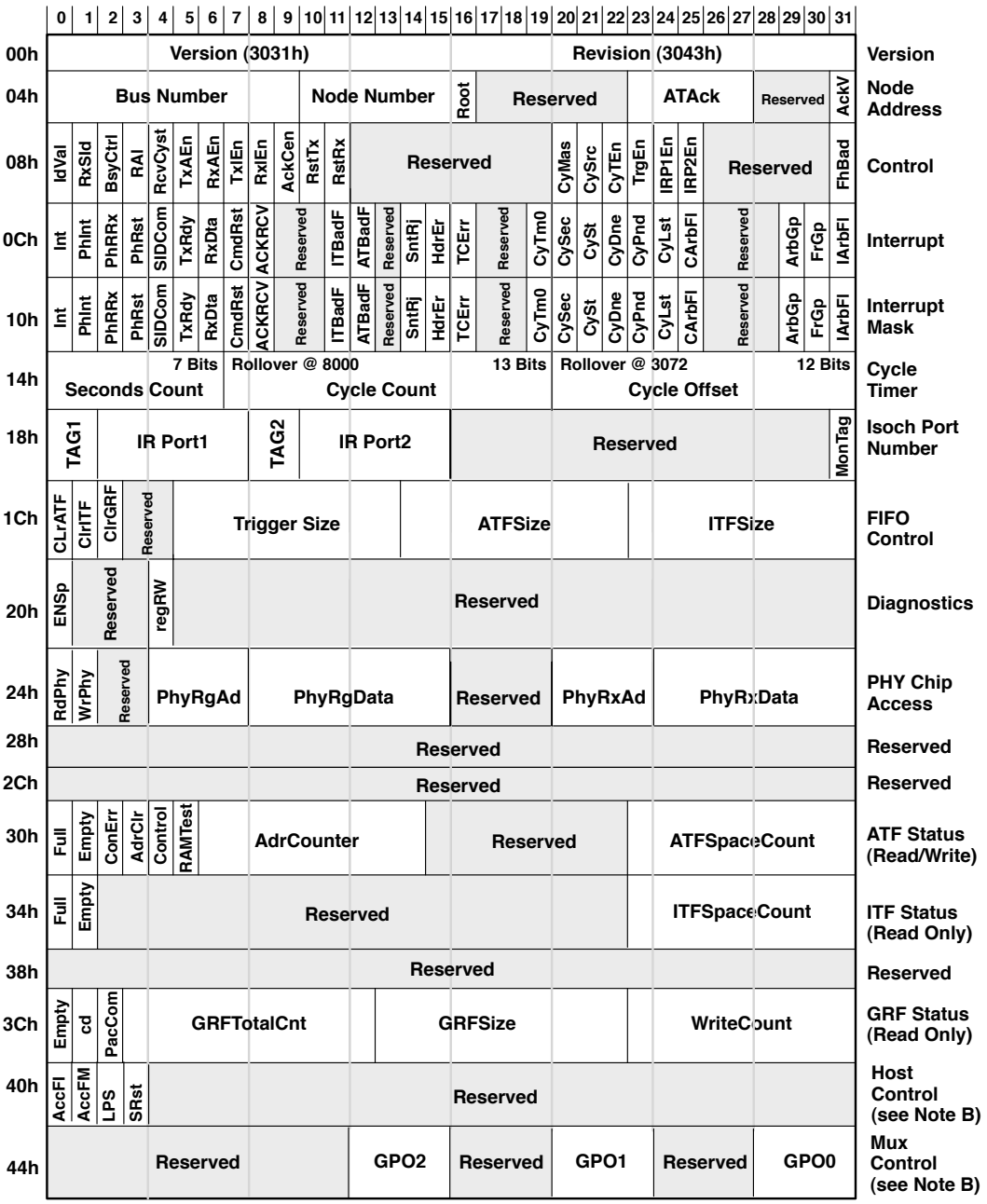
ATF: Der ATF ist der „Asynchronous Transfer FIFO“, in welchen Datenpakete geschrieben werden, die als asynchrone Datenpakete über den Firewire-Bus gesendet werden sollen.

ITF: ITF steht für „Isochronous Transfer FIFO“ und bezeichnet den für den Versand von isochronen Datenpaketen verantwortlichen FIFO.

Die Grösse jedes FIFOs ist frei wählbar, allerdings stehen für alle FIFOs gemeinsam lediglich 512 Quadlets zur Verfügung, so dass die Partitionierung dieses Speichers wohl überlegt sein sollte.

Der Zugriff auf GRF, ATF und ITF

Um die Daten aus dem GRF auszulesen genügt es einen Lesezugriff auf die passende Adresse vorzunehmen. Siehe dazu Abbildung 3.3. Dadurch erhält man jenes Quadlet der Daten, welches bereits am längsten im FIFO liegt. Das Quadlet wird anschliessend aus dem FIFO gelöscht und das nächste Quadlet ist über die gleiche Adresse verfügbar.



NOTES: A. All gray areas (bits) are reserved bits.
B. This register is new to the TSB12LV01B and does not exist in the TSB12LV01A.

Abbildung 3.1: Die Register des TSB12LV01B, Quelle: [TSB12]

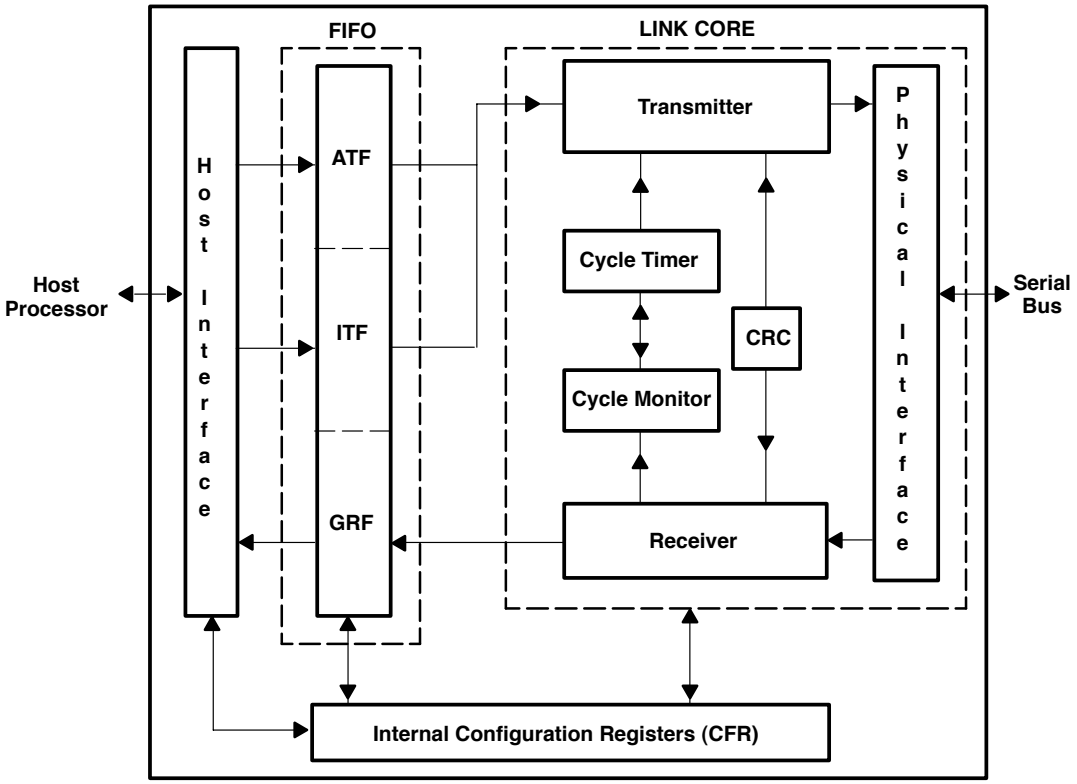


Abbildung 3.2: Der Aufbau des TSB12LV01B, Quelle: [TSB12]

Das Schreiben in die Versende-FIFOs funktioniert ein wenig anders, aber für ATF und ITF gleich: Für beide FIFOs existieren unterschiedliche Adressen, in welche die Daten geschrieben werden müssen.

Möchte man die Daten im Burst-Modus zum Chip übertragen, so schreibt man sie einfach in die „Burst“-Adresse. Mit jedem Takt wird so ein Quadlet übertragen. Hört der Burst-Zugriff auf, so signalisiert das dem Chip, dass das Paket komplett übertragen wurde.

Überträgt man die Daten nicht im Burst-Modus, so geht man anders vor. Das erste Quadlet eines Paketes wird in die „First“-Adresse geschrieben. Die darauf folgenden Daten werden in die „Continue“-Adresse geschrieben und mindestens das letzte Quadlet wird in die „Continue & Update“-Adresse geschrieben. Sobald Daten in diese Adresse geschrieben wurden beginnt der Chip damit die Daten auf dem Firewire-Bus zu übertragen. Es können nun noch weitere Daten in diese Adresse geschrieben werden, allerdings muss dafür Sorge getragen werden, dass dies rechtzeitig passiert, da sonst das Paket nicht vollständig übertragen wird. Es ist sogar möglich direkt nach der „First“-Adresse die „Continue & Update“-

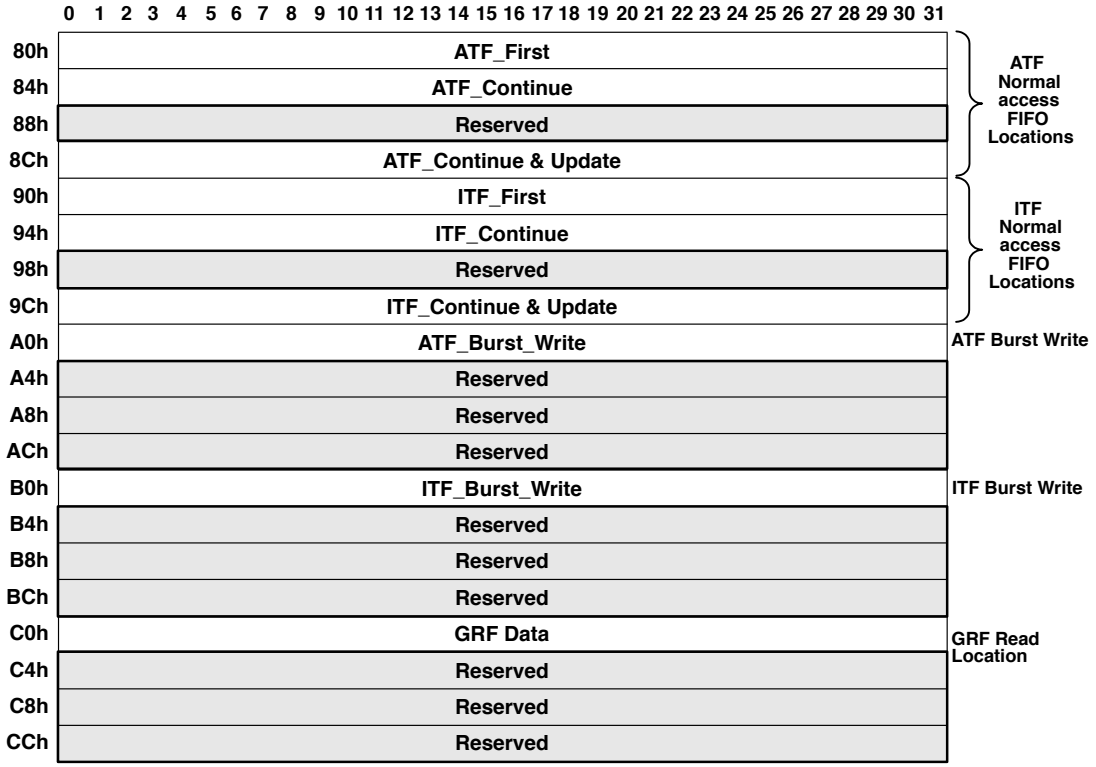


Abbildung 3.3: Der Zugriff auf die FIFOs des TSB12LV01B, Quelle: [TSB12]

Adresse zu nutzen und somit die Übertragung direkt nach dem zweiten Quadlet zu starten.

Weitere ausführliche Informationen über diesen Baustein und seine Verwendung findet man in [TSB12].

3.3 Verwendung der Einheit sdramIO

Die VHDL-Einheit „sdramIO“ wurde von Dr. Andreas Mäder vom Arbeitsbereich TAMS zur Verfügung gestellt. Sie ermöglicht es auf den SDRAM-SODIMM zuzugreifen, ohne sich mit dem Timing des Speicherriegels auseinandersetzen zu müssen. Der Zugriff erfolgt hierbei über ein sehr einfach zu handhabendes Verfahren, bei dem ein Zugriff gestartet wird, indem die Request-Leitung auf den negierten Wert der Acknowledge-Leitung gesetzt wird. Ist der Vorgang abgeschlossen, so antwortet die „sdramIO“, indem sie den Wert der „Acknowledge“-Leitung auf den Wert der „Request“-Leitung setzt. Das hat den Vorteil, dass nach einem Lese- oder Schreibzugriff keine Änderung der gesetzten Signale notwendig ist. Dies ist beson-

ders dann nützlich, wenn zwei Einheiten nicht mit dem gleichen Takt arbeiten oder der Takt nicht synchron in beiden Einheiten ankommt.

Auszug aus der Port-Liste der „sdramIO“:

```
port(
    ioReq   : in  std_logic;
    ioWri   : in  std_logic;
    ioAck   : out std_logic;
    ioNewI  : out std_logic;
    ioNewO  : out std_logic;
    ioAddr  : in  std_logic_vector(22 downto 0);
    ioDataI : in  std_logic_vector(63 downto 0);
    ioDataO : out std_logic_vector(63 downto 0);
    [..]
);
```

Zum Lesen aus dem Speicher geht man folgendermassen vor: In einem ersten Zustand wird die Request-Leitung negiert, das Write-Signal auf ’0’ gesetzt und eine Adresse angelegt. Hier ein Beispiel-Code aus einem Automaten, der „sdramIO“ ansteuert:

```
when read1 =>
    if ioReq_sig = ioAck then
        ioReq_sig <= not ioAck;
        ioWri <= '0';
        ioAddr <= "xxxx xxxx xxxx xxxx xxxx xxx";
        state := read2;
    end if;
```

Hierbei ist „ioReq_sig“ ein dem Ausgang „ioReq“ vorgeschaltetes Signal, das es ermöglicht den aktuellen Wert des Ausganges zu lesen, was bei einer Ausgangsleitung direkt sonst nicht möglich ist.

Im nächsten Zustand werden dann, sobald der Speicher den Lesevorgang abgeschlossen hat, nur noch die Daten von den Datenleitungen ausgelesen:

```
when read2 =>
    if (ioReq_sig = ioAck) and (ioNewO = '1') then
        data := ioDataI;
        state := idle;
    end if;
```

Die Leitung „ioNewO“ gibt dabei an, dass Daten bereit stehen, während „ioAck“ schon den Wert ändert, sobald der Speicher die Werte auf den Leitungen angenommen hat und bereit ist für weitere Daten. Diese Unterscheidung wird gebraucht, um den Burst-Modus des Speichers zu benutzen, der hier aber nicht weiter erläutert wird.

Das Schreiben funktioniert ähnlich, nur das hierbei das Write-Signal auf ’1’ gesetzt wird und zusätzlich zu der Schreibadresse auch gleich Daten auf die Datenleitungen gelegt werden:

```
when write1 =>
    if ioReq_sig = ioAck then
        ioReq_sig <= not ioAck;
        ioWri <= '1';
        ioData0 <= x"xxxx xxxx xxxx xxxx";
        ioAddr <= "xxxx xxxx xxxx xxxx xxxx xxx";
        state := write2;
    end if;
when write2 =>
    if (ioReq_sig = ioAck) and then
        state := nextstate;
    end if;
```

3.4 Algorithmen zur Umrechnung von omnidirektionalen Bildern in Panoramabilder

Für die Umrechnung eines omnidirektionalen Bildes in ein Panoramabild existieren mehrere Verfahren:

Die einfache direkte Umwandlung tastet das omnidirektionale Bild gleichmässig ab.

Die hyperboloidale Projektion bringt auch die Krümmung des verwendeten Spiegels in die Berechnung mit ein.

Die einfache direkte Umwandlung sei hier kurz anhand des folgenden Java-ähnlichen Pseudocodes erläutert, welcher [Zha07] entnommen wurde. Dem Code wurde noch Zeile 12 hinzugefügt, da der Funktion „createPanorama()“ ein Rückgabewert fehlte.

```
01 Image createPanorama(int width, int height, Image omni)
02 {
03     Image panorama = new Image(width, height);
04     for (int i=0; i<width; i++)
05     {
06         for (int j=0; j<height; j++)
07         {
08             panorama.setPixel(
09                 i, j, getPixelFromOmnidirectional(i, j, omni));
10         }
11     }
12     return panorama;
13 }
14
15 Pixel getPixelFromOmnidirectional(int i, int j, Image omni)
16 {
17     double radius = outerRing -
18         ((j / height) * (outerRing - innerRing));
19     double alpha = - (i / width) * (2 * PI);
20
21     double x = centerX - radius * sin(alpha);
22     double y = centerY + radius * cos(alpha);
23
24     return omni.getInterpolatedPixel(x, y);
25 }
```

Das Zielbild wird bei diesem Algorithmus Reihe für Reihe von oben nach unten aufgebaut. Eine Reihe im Zielbild findet sich im Ursprungsbild als Gerade zwischen dem inneren Ring, welcher das untere Ende des Spiegels darstellt und das Bild der schwarzen Nadel beinhaltet, und dem äusseren Ring, welcher das obere Ende des Spiegels darstellt, wieder. Der oberste Pixel einer Reihe liegt dabei am äusseren Ring, während der unterste Pixel am inneren Ring liegt (siehe Abbildung 3.4).

Das Vorgehen lässt sich dabei in die folgenden Phasen unterteilen:

1. **Berechnung der Polarkoordinaten:** Aus den Koordinaten eines Zielpixels wird berechnet, wo sich diese Stelle im Ursprungsbild befindet. Dazu werden die Polarkoordinaten dieser Stelle ausgehend vom Mittelpunkt des omnidirektionalen Kamerabildes berechnet.

   Für den Radius wird der Abstand zwischen den beiden Ringen mit der vertikalen Position des Zielpixels als Anteil der Gesamthöhe des Zielbildes multipliziert und anschliessend vom äusseren Ring abgezogen (Zeile 17 und 18).

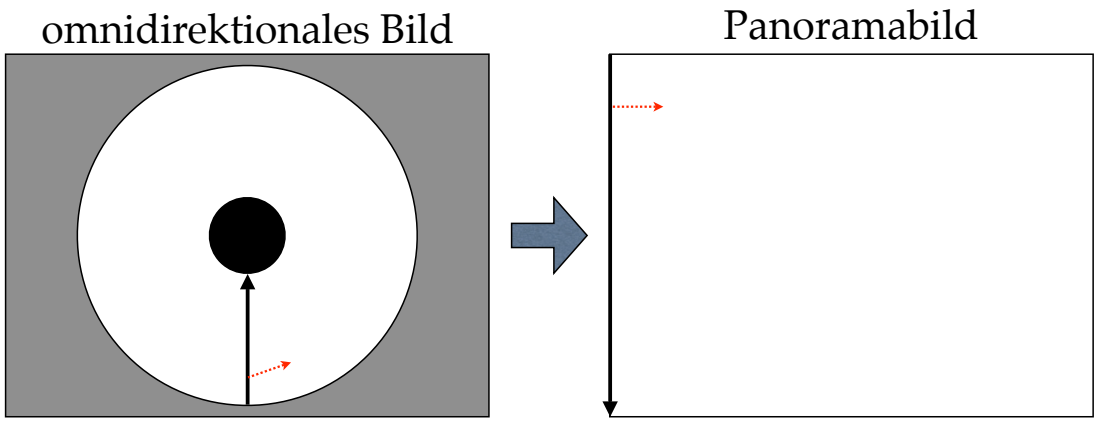


Abbildung 3.4: Vorgehensweise der direkten Umwandlung: Das omnidirektionale Bild wird mit einer rotierenden Linie abgetastet.

So werden für jede Reihe im Zielbild lediglich die Bildbereiche im Ursprungsbild zwischen den beiden Ringen von aussen nach innen abgetastet.

Für den Winkel wird die horizontale Position des Zielpixels als Anteil der Gesamtbreite des Zielbildes genommen und dieser Anteil auf einen vollen Kreis bezogen (Zeile 19). So kreist beim Umrechnen des gesamten Bildes die das Bild abtastende Linie genau einmal um das Ursprungsbild.

2. Berechnung der kartesischen Koordinaten: Mit dem Winkel und dem Radius können nun die kartesischen Koordinaten des Ursprungspixels berechnet werden. Siehe dazu [WikPol]. Diese berechnet man durch:

$$x = r \cdot \cos \alpha \tag{3.1}$$
$$y = r \cdot \sin \alpha \tag{3.2}$$

Um auf die Form in Zeile 21 und 22 zu kommen, müssen mehrere Dinge beachtet werden. Das normale Polarkoordinatensystem geht davon aus, dass der Wert von y nach oben grösser und nach unten kleiner wird. Bei diesen Bildern hingegen wird der Wert von y von oben nach unten grösser. Somit muss das Koordinatensystem an der x-Achse durch eine Negation gespiegelt werden:

$$x = r \cdot \cos \alpha \tag{3.3}$$
$$y = -r \cdot \sin \alpha \tag{3.4}$$

Des weiteren werden noch 90 Grad vom Winkel subtrahiert, damit die abtastende Linie senkrecht nach unten beginnt. Dies ist im Grunde nicht notwendig, erzeugt aber einen schöneren Eindruck, wenn man beide Bilder miteinander vergleicht: Der Bereich im omnidirektionalen Bild, der ohne Neigung

des Kopfes direkt betrachtet werden kann und somit direkt ins Auge springt ist der Bereich oberhalb des Mittelpunktes (siehe dazu Abbildung 1.3 auf Seite 12). Damit dieser sich auch im Panoramabildes in der Mitte befindet (siehe Abbildung 1.4 auf Seite 13) müssen vom Winkel noch 90 Grad abgezogen werden:

$$x = r \cdot \cos(\alpha - 90°) \tag{3.5}$$
$$= r \cdot \sin\alpha \tag{3.6}$$
$$y = -r \cdot \sin(\alpha - 90°) \tag{3.7}$$
$$= r \cdot \cos\alpha \tag{3.8}$$

Damit man die Formeln aus Zeile 21 und 22 erhält muss man sie nun noch so umstellen, dass man einen negativen Winkel erhält und anschliessend noch die Koordinaten des Mittelpunktes addieren:

$$x = --r \cdot \sin\alpha \tag{3.9}$$
$$= -r \cdot \sin(-\alpha) \tag{3.10}$$
$$y = r \cdot \cos\alpha \tag{3.11}$$
$$= r \cdot \cos(-\alpha) \tag{3.12}$$

$$x = centerX - r \cdot \sin(-\alpha) \tag{3.13}$$
$$y = centerY + r \cdot \cos(-\alpha) \tag{3.14}$$

3. Interpolation: Da die Koordinaten des Ursprungspixels rationale Zahlen sind, liegt die berechnete Stelle im Ursprungsbild zwischen mehreren Pixeln. Um diesem Umstand gerecht zu werden interpoliert man nun aus diesen Pixeln den Farbwert, welcher im Zielbild eingesetzt wird.

Für die hyperboloidale Projektion hingegen sei auf weiterführende Literatur verwiesen, wie [Zha07].

3.5 VHDL Festkommazahlarithmetik

Sowohl Fliess- als auch Festkommazahlarithmetik sind zwar Bestandteile der Hardwarebeschreibungssprache VHDL, aber sie sind nicht synthetisierbar. Dies bedeutet, dass sie zwar für die Simulation einer Schaltung benutzt werden können, aber dass aus ihnen keine Gatternetzliste generiert werden kann, die dann auf einem Chip eingesetzt werden könnte.

Aus diesem Grund ist es erforderlich selbst dafür zu sorgen, dass eine entwickelte Schaltung in der Lage ist mit Fliess- oder Festkommazahlen umzugehen, wenn

das benötigt wird. Hier wurde dafür auf einen mit VHDL93 kompatiblen Entwurf der „EDA Industry Working Groups“ zurückgegriffen, welcher unter [EDA06] zu finden ist.

Zur Verwendung kamen die beiden Dateien „math_utility_pkg.vhdl“ und „fixed_pkg_c.vhdl“. Bindet man diese als Pakete ein, so stehen anschliessend mehrere Festkommatypen und die dazu passende Arithmetik bereit. So erzeugt z.B. folgender Code ein 32 Bit breites Signal ohne Vorzeichen mit einer Genauigkeit von 16 Bit vor dem Komma und 16 Bit hinter dem Komma:

```
signal beispiel : ufixed(15 downto -16);
```

Aufpassen muss man bei diesen Datentypen vor allem bei der Multiplikation: Die Arithmetik schneidet nicht wie sonst üblich den errechneten Wert auf den Wertebereich von einem der Eingabewerte zurecht, sondern bildet einen Ausgabewert mit einem Wertebereich, der alle aus dieser Rechnung möglichen Werte abdeckt. Hier muss man dann selbst über die mitgelieferte Funktion „resize()“ einen Wertebereich für das Ergebnis festsetzen.

3.6 Firewire Digital Camera Specification

Die „Firewire Digital Camera Specification“ [DCS98] stellt einen Standard für Kameras dar, die ihre Daten über den Firewire-Bus übertragen. Dadurch ist es möglich Hard- und Software zu entwickeln, die problemlos mit unterschiedlichen Kameras umgehen kann. Die „Firewire Digital Camera Specification“ legt u.a. Register fest, die gelesen und geschrieben werden können. Dadurch können die Spezifikationen der Kameras abgefragt und Einstellungen vorgenommen werden. Auch die Paketgrössen der isochronen Datenpakete für verschiedene Videoformate legt die Spezifikation fest. Siehe dazu auch [WikDCS].

3.7 Das YUV-Farbmodell

Das YUV-Farbmodell beschreibt eine Farbe anhand zweier Komponenten. Die erste ist die Luminanz, die Lichtstärke pro Fläche, die mit Y bezeichnet wird. Die zweite ist die Chrominanz, die den Farbanteil darstellt. Diese besteht aus den Komponenten U und V. Das YUV-Farbmodell ist entstanden als das Farbfernsehen eingeführt wurde. Das Farbsignal musste zusätzlich zum Schwarz/Weiss-Signal übertragen werden, um die Abwärtskompatibilität mit den alten Schwarz/Weiss-Fernsehern zu gewährleisten.

Eine Besonderheit dieses Farbmodells ergibt sich durch den Aufbau der Netzhaut des menschlichen Auges: Helligkeitsinformationen werden in einer höheren Auflösung wahrgenommen als Farbinformationen. Dies machen sich viele YUV-Formate zunutze, indem sie die Auflösung der Chrominanz reduzieren oder für aneinander liegende Pixel zwar unterschiedliche Luminanzwerte aber die gleichen Chrominanzwerte nutzen. So z.B. das Format YUV 4:2:2. Hier werden für jeweils zwei direkt nebeneinander liegende Pixel die gleichen Chrominanzwerte verwendet. Da für Y, U und V jeweils 8 Bit verwendet werden reduziert sich so die Datenmenge pro Pixel auf 16 Bit. Weitergehende anschauliche Informationen über dieses Farbmodell findet man in [WikYUV].

Methoden und Werkzeuge

4

4.1 Entwurfsprogramme

Altera Quartus II 7.2

Das Programm „Quartus II“ (siehe Abbildung 4.1) von Altera wurde genutzt, um den mit dem Editor erzeugten VHDL-Code zu kompilieren und zu einer Gatternetzliste zu synthetisieren. Für die Synthese war Quartus mit den entsprechenden Altera Bibliotheken für den Apex-FPGA-Chip ausgestattet. Ausserdem wurde Quartus dazu genutzt, den Apex-Chip mit der Gatternetzliste zu programmieren.

Kate

Als Editor für den VHDL-Code kam „Kate“ (KDE Advanced Text Editor) zum Einsatz (siehe Abbildung 4.2). Obwohl es sich hierbei nur um einen eher einfachen Texteditor handelt, kann er mit guter VHDL-Unterstützung glänzen. Mit Syntax-Highlighting und der Funktion mehrere Zeilen auf einmal auszukommentieren, bietet er alles für einen schnellen Einstieg in die VHDL-Programmierung.

IEEE1394Diag 1.0.3

„IEEE1394Diag“ ist ein Diagnose-Programm für den Firewire-Bus. Es ermöglicht den Firewire-Bus zu überwachen und diverse Operationen auszuführen. So zeigt es z.B. an, welche Geräte sich am Firewire-Bus befinden (siehe Abbildung 4.3) und ist in der Lage deren Konfigurationsregister auszulesen und neue Werte hineinzuschreiben. Es ist ausserdem in der Lage einen Bus-Reset zu initiieren und kann einen isochronen Datenstrom erzeugen und auf dem Bus aussenden.

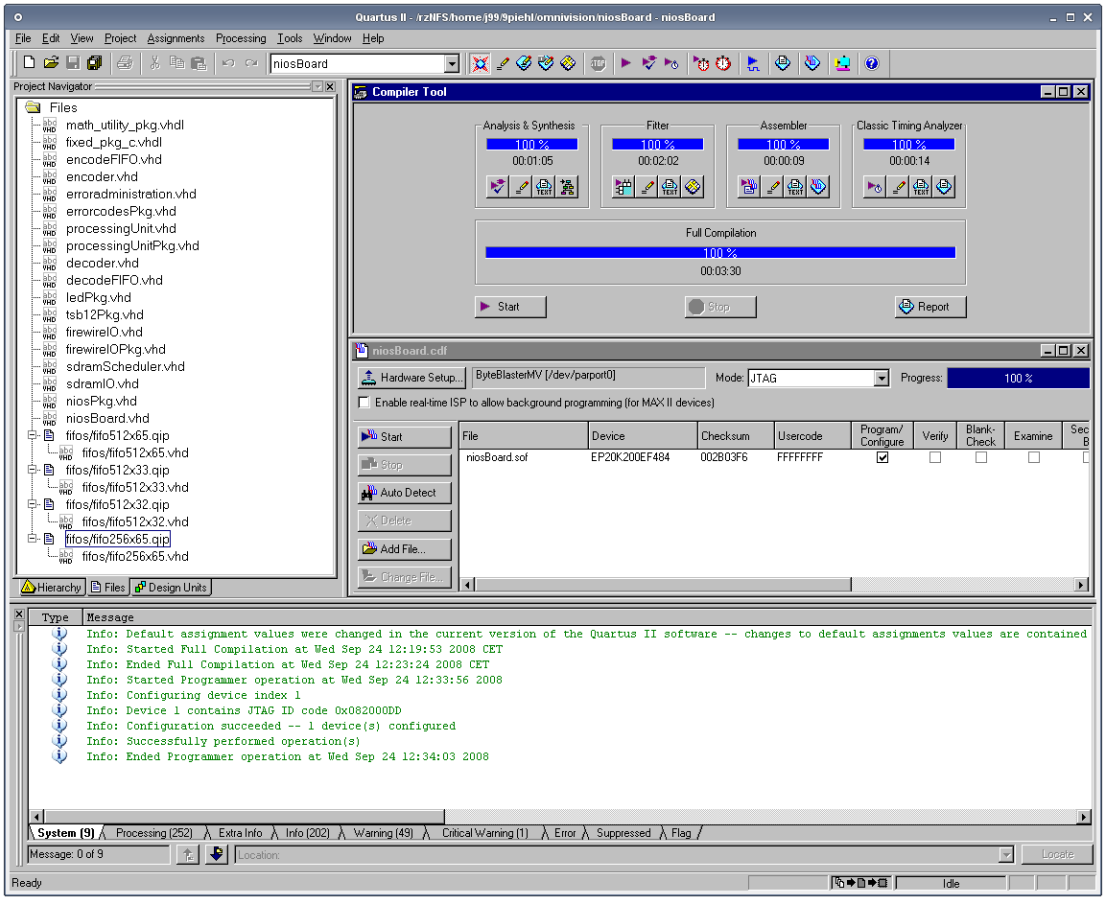


Abbildung 4.1: Altera Quartus mit geöffnetem Compiler- und Programmer-Tool

Coriander 2.0.0-rc5

„Coriander“ wurde dazu benutzt um die Kamera zu konfigurieren und zu starten. Es liest aus den Konfigurationsregistern der Kamera aus, welche Einstellungsmöglichkeiten die Kamera bietet und ermöglicht es diese zu ändern. Auf diese Weise lassen sich auch Kameraspezifische Einstellungen verändern. Die Kamera DFW-SX900 von Sony z.B. erlaubt neben dem Einstellen von Bildrate und Übertragungsgeschwindigkeit auch Einstellungen, die direkt das Bild betreffen, wie Helligkeit, Schärfe und Sättigung (siehe Abbildung 4.4).

dumpiso

Mit dem Kommandozeilenprogramm „dumpiso“ ist es möglich den Datenverkehr auf den isochronen Kanälen auf dem Firewire-Bus mitzulesen und abzuspeichern.

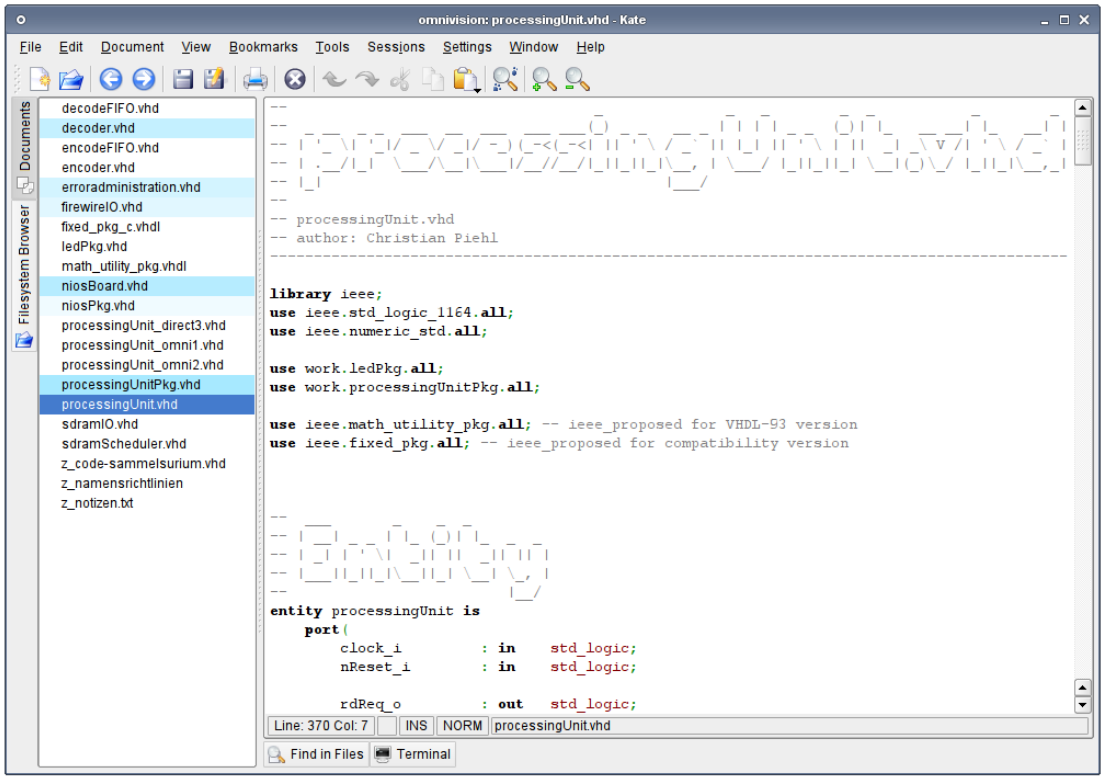


Abbildung 4.2: Der Editor Kate mit mehreren geöffneten Dokumenten

Das Abbild des Datenverkehrs kann dann anschliessend mit einem Hexeditor, wie z.B. „KHexEdit“ analysiert werden. Dies ermöglicht es zu überprüfen, welche Datenpakete über den Bus gehen und ob die eigenen Pakete richtig formatiert sind.

```
> dumpiso -c0 dumpfile
```

speichert die Daten aus Kanal 0 in der Datei „dumpfile“ ab. Anschliessend kann man mit

```
> khexedit dumpfile
```

die Daten im Hexeditor KHexEdit betrachten (siehe Abbildung 4.5).

isoViewS

Das Programm „isoViewS“ wurde von Dr. Andreas Mäder vom Arbeitsbereich TAMS entwickelt und zeigt die in isochronen Datenströmen übertragenen Video-

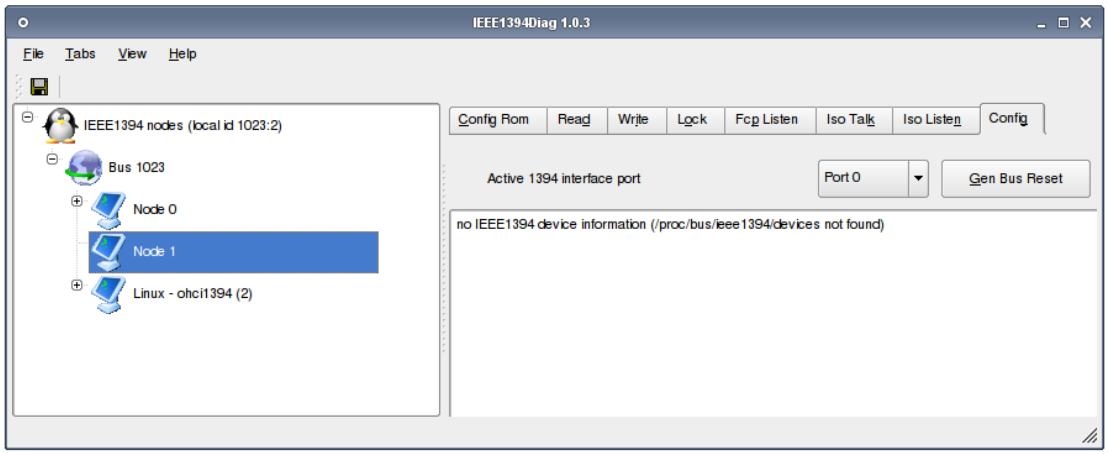


Abbildung 4.3: IEEE1394Diag zeigt Geräte am Firewire-Bus hierarchisch an

daten auf dem Computerbildschirm an. Da andere Video-Programme, die diese Daten vom Firewire-Bus lesen, am anderen Ende ein Gerät erwarten, welches sich vollständig an die „Firewire Digital Camera Specification“ hält, ermöglicht dieses Programm so den Entwurf von weniger komplexen Systemen und den Test von Systemen, die noch nicht vollständig implementiert sind.

```
> isoViewS -c0 --xSize 1280 --ySize 960
```

zeigt den Datenstrom aus Kanal 0 und mit einer Auflösung von 1280 x 960 Pixeln grafisch auf dem Bildschirm an.

4.2 Vorgehensweise

4.2.1 Entwurf

Zu Anfang galt es einen Entwurf für das Gesamtsystem zu erstellen. Aus einem vorangegangenen Projekt am Arbeitsbereich TAMS, dessen Fortführung diese Diplomarbeit im Grunde ist, waren bereits die ersten Grundzüge bekannt. So sollte es eine zentrale Funktionseinheit mit dem Namen „sdramScheduler“ geben, die über die „sdramIO“ mit dem Speicher kommuniziert und den Zugriff auf diesen verwaltet. Daneben sollte es die „processingUnit“ geben, die den auszuführenden Algorithmus enthält und über die „sdramScheduler“ lesend und schreibend auf den Speicher zugreift. Des weiteren sollte sich die Einheit „firewireIO“ um die Kommunikation mit der Firewire-Platine und dem Firewire-Bus kümmern und

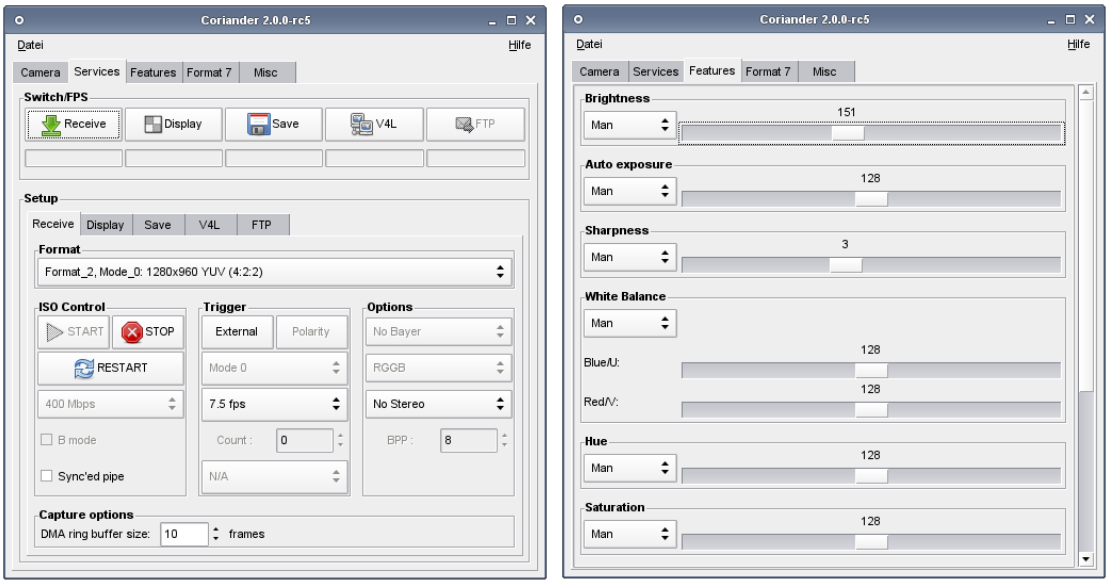


Abbildung 4.4: Coriander ermöglicht es Kameraspezifische Einstellungen vorzunehmen

die Videodaten an die „sdramScheduler“ übertragen und von ihr annehmen. Abbildung 4.6 verdeutlicht diesen Entwurf.

4.2.2 Implementierung

Als Takt für das gesamte System wurden 25 MHz gewählt. Die Firewire-Platine und der Speicher wurden direkt mit diesem Takt angesteuert, während der FPGA den Takt um 8000 Pikosekunden verzögert bekam. Diese Einstellung war notwendig, da der Takt aufgrund höherer Laufzeit die Firewire-Platine nur verzögert erreicht. Mit dieser Einstellung hatte auch der Arbeitsbereich TAMS bereits gute Erfahrungen gemacht.

Die Einheiten wurden dem Datenstrom folgend implementiert. Begonnen wurde bei der „firewireIO“. Hier galt es zunächst die Kommunikation mit der Firewire-Platine aufzunehmen, indem die Zugriffsprotokolle für den Lese- und Schreibvorgang implementiert wurden. Dies gestaltete sich schwieriger als erwartet, da die Anbindung der Firewire-Platine an das FPGA-Board aufgrund der Laufzeitverzögerung nicht gänzlich zuverlässig funktionierte. Der ursprüngliche Plan die Burst-Modi der Firewire-Platine zu nutzen, um die Daten schneller übertragen zu können, musste bereits an dieser Stelle der Entwicklung verworfen werden. Die einfachen Lese- und Schreibmodi funktionierten jedoch und so konnte damit be-

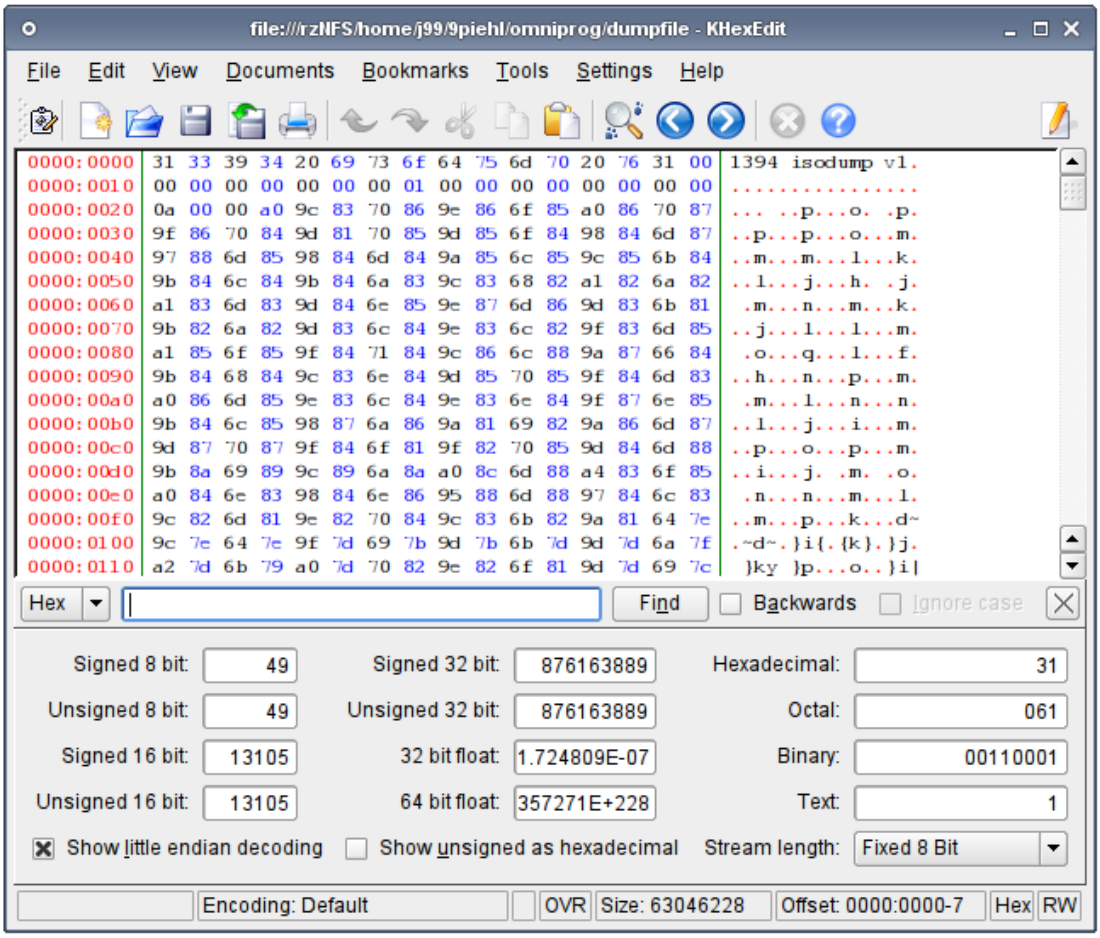


Abbildung 4.5: Ein gespeicherter isochroner Datenstrom im Hexeditor KHexEdit

gonnen werden die Konfigurationsregister der Firewire-Platine auszulesen und neu zu schreiben, wodurch Zugriff auf den Firewire-Bus möglich wurde.

Um trotzdem noch eine schnelle Datenübertragung zu realisieren wurde versucht den Takt zu erhöhen. Da die Lese- und Schreibfunktionen auch bei 50 MHz noch einwandfrei funktionierten und dies der höchste für die Chips auf der Firewire-Platine zulässige Takt ist wurde das gesamte System nun mit diesem Takt betrieben. Durch den Wegfall der Burst-Modi war es nun auch nicht mehr erforderlich, dass FPGA und Firewire-Platine absolut synchron laufen, weshalb auch auf die Taktverschiebung verzichtet werden konnte.

Als nächstes wurde der Zugriff auf den asynchronen Datenpuffer der Firewire-Platine implementiert. Die Daten mussten Paketweise aus dem entsprechenden Register der Firewire-Platine ausgelesen werden und anschliessend anhand ihres

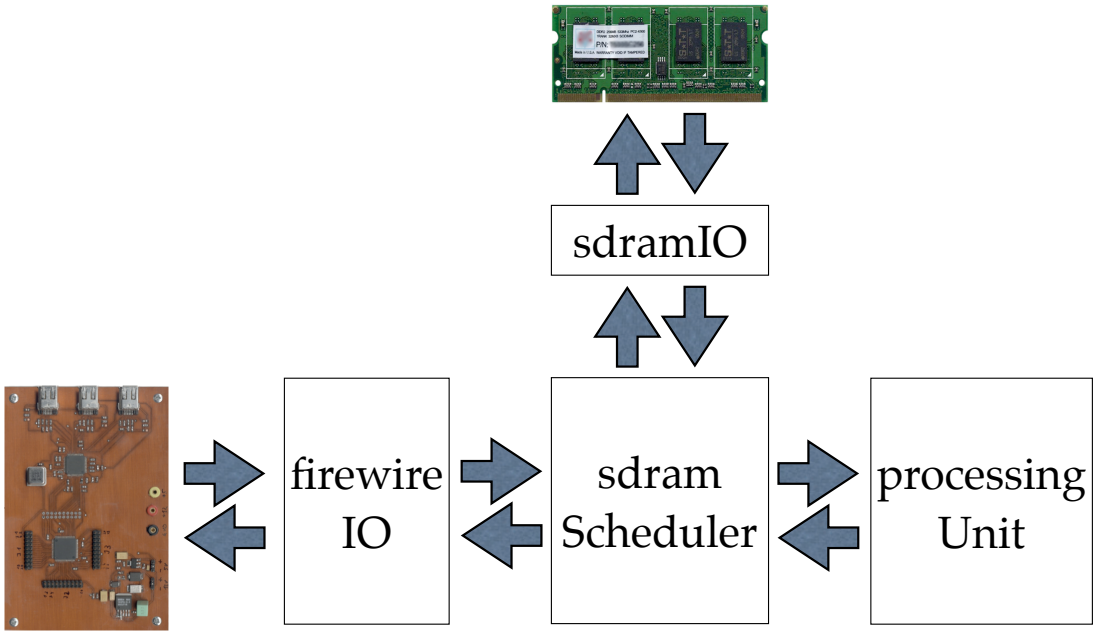


Abbildung 4.6: Der erste Entwurf des Gesamtsystems

Headers identifiziert werden. Hierbei fiel auf, dass die Übertragung der Daten doch nicht einwandfrei funktionierte, da die ausgelesenen Datenpakete nicht das Format hatten, dass sie eigentlich hätten haben sollen. Es stellte sich schlussendlich heraus, dass die Lesefunktion einen Fehler enthielt und fälschlicherweise den Burst-Modus aktivierte, wodurch in mehreren aufeinanderfolgenden Takten mehrere Datenpakete übertragen wurden, aber die „firewireIO“ nur eines davon speicherte.

Nach dem Beheben dieses Fehlers wurde das Versenden von asynchronen Daten implementiert, damit sich das System auf dem Firewire-Bus identifizieren kann. Mit dem Programm „IEEE1394Diag“ wurde der korrekte Versand und das korrekte Format dieser Nachrichten überprüft und getestet, ob das System auf entsprechende Anfragen auf dem Firewire-Bus richtig antwortet. Mit Hilfe des gleichen Programms wurde dann dafür gesorgt, dass das System bei einem Bus-Reset auf dem Firewire-Bus komplett neu gestartet wird.

Es folgte das Dekodieren des isochronen Videodatenstroms. Wie bei den asynchronen Daten, musste dieser erst Paketweise aus den entsprechenden Registern der Firewire-Platine ausgelesen und dann von den Firewire-spezifischen Headern befreit werden, bevor er an die nächste Einheit weitergereicht werden konnte.

Des weiteren wurde ein einfacher Ausgabedatenstrom erzeugt, um die Ausgabe von Videodaten über einen isochronen Kanal des Firewire-Busses zu testen. Dieses Videosignal wurde mit einem Computer empfangen und mit dem Programm

„dumpiso“ darauf untersucht, ob der Versand des Datenstromes einwandfrei funktioniert und die Informationen im Header und die Frame-Grösse stimmten. Als dies der Fall war, konnte das Signal mit dem Programm „isoViewS“ auf dem Bildschirm angezeigt werden.

Als nächste Einheit wurde ein FIFO generiert, der die Daten, die im Speicher abgelegt werden sollen, erst einmal zwischenspeichert. Leider funktionierte der Read-Ahead-Modus der von Quartus generierten FIFOs nicht und so musste auf den normalen Modus zurückgegriffen werden, bei dem die Daten angefordert werden müssen, bevor sie einen Takt später gelesen werden können. Es stellte sich heraus, dass es sinnvoll ist an dieser Stelle auch die Umkodierung des Datenstroms von 32 auf 64 Bit vorzunehmen. Dadurch wurde aus dem einfachen FIFO der „decodeFIFO“, welcher eine Kombination aus mehreren kleineren Einheiten darstellt: Einem 33 Bit breiten FIFO (32 Bit Daten und 1 Steuerbit), einer Dekodiereinheit und einem 65 Bit breiten FIFO (64 Bit Daten und 1 Steuerbit).

Nun konnte die zentrale Einheit, die „sdramScheduler“ umgesetzt werden. Die Anbindung des SDRAM klappte dank der „sdramIO“ schnell und reibungslos und auch das Scheduling war schnell implementiert. In die „sdramScheduler“ wurde anfangs erstmal eine direkte Umleitung der Eingangsdaten auf die Ausgangsdaten eingebaut, um damit zu testen, ob die Daten bis hierher soweit korrekt übertragen wurden und ob das Speichern und Lesen einwandfrei lief.

Anschliessend wurde die „encodeFIFO“ geschrieben, die sich lediglich dadurch von „decodeFIFO“ unterscheidet, dass die Datenbreite von 64 Bit auf 32 Bit heruntergesetzt wird. Jetzt war es erstmals möglich Bilder vom Firewire-Bus auszulesen, zu speichern und wieder auszugeben. Mit der Software „isoViewS“ konnten diese auch sogleich auf dem Monitor eines an den Firewire-Bus angeschlossenen Computers angezeigt werden. Die Übertragung funktionierte soweit, aber das Bild war noch deutlich verrauscht. Nach langer Fehlersuche stellte sich heraus, dass der Speicher mit der verwendeten Taktfrequenz nicht zurecht kam und sich dadurch Fehler in das Bild einschlichen. Dies wurde nur zufällig durch einen Fehlerfall offenbar, bei dem keine Videodaten mehr beim Speicher ankamen und so immer das gleiche Bild ausgegeben wurde. Einige Bildfehler veränderten sich bei jedem einzelnen übertragenen Bild und entstanden so offensichtlich durch fehlerhaftes Auslesen aus dem Speicher. Andere Bildfehler wiederum, die sich farblich von den fluktuierenden Fehlern deutlich unterschieden, waren immer an den gleich Stellen zu finden und ihre Anzahl blieb auf den ersten Blick konstant. Nach einer kleinen Arbeitspause, in der das System in diesem Zustand belassen wurde, stellte sich heraus, dass dem nicht so war und diese Fehler an Anzahl zunahmen, so dass auf einen Fehler im Refresh des Speichers und damit auf eine zu hohe Taktfrequenz geschlossen werden konnte. Der Takt wurde deshalb erneut angepasst und auf 33 MHz reduziert.

Zu guter letzt wurde die Einheit „processingUnit“ erstellt. Um die allgemeine Funktionalität der Schnittstelle zur „sdramScheduler“ zu testen wurde als erster Algorithmus wieder die direkte Abbildung des Ursprungsbildes auf das Zielbild umgesetzt. Erst als dies funktionierte folgte der Algorithmus, welcher das omnidirektionale Ursprungsbild in ein Panoramabild umrechnet. Dies erforderte einen deutlich höheren Aufwand als die direkte Umleitung, da nun nicht mehr mit ganzen Zahlen gerechnet werden konnte, sondern rationale Zahlen zum Einsatz kommen mussten. Hierfür wurde auf die mit VHDL93 kompatiblen Pakete eines Entwurfs für Festkommazahlarithmetik von der „EDA Industry Working Groups“ zurückgegriffen.

Da bereits dieser Algorithmus nur vereinfacht zum Einsatz kam, wurde auf eine Implementierung von komplexeren Algorithmen zur Umrechnung der omnidirektionalen Bilder verzichtet.

4.2.3 Überprüfung der Funktionalität

Die Verifikation wurde immer direkt auf dem FPGA-Board vorgenommen, da sich die Kommunikation mit dem Firewire-Bus und der Firewire-Platine nur schlecht hätte simulieren lassen und die Simulation keinen Rückschluss darauf zugelassen hätte, ob ein Entwurf auch auf dem FPGA-Board wirklich funktioniert. Ausserdem war die Anbindung der Firewire-Platine an das FPGA-Board nicht ganz zuverlässig und produzierte des öfteren unvorhersehbare und schwer nachvollziehbare Fehler. Eine Simulation wäre hier der Situation nicht gerecht geworden.

Stattdessen wurde auf die LEDs und die 7-Segment-Anzeigen zurückgegriffen, um Meldungen bezüglich des Status einer Einheit anzeigen zu lassen. Da aber immer nur eine Einheit zur Zeit auf diese Ports zugreifen kann und der Zugriff darauf stets fest in den Code eingebettet war kam es schnell zu der Situation, dass man zu einem bestimmten Zeitpunkt besser die Ausgabe einer anderen Einheit gebraucht hätte. Dies führte zur Entwicklung der Einheit „errorAdministration“, die sich um die Verwaltung der Statusausgaben kümmern und somit jederzeit Zugriff auf den Status aller Einheiten erlauben sollte.

Ergebnisse

5

5.1 Übersicht über das Gesamtsystem

Wie man Abschnitt 4.2 entnehmen kann, hat sich der Aufbau des Systems im Laufe der Implementierung ein wenig verändert. Den endgültigen Aufbau kann man Abbildung 5.1 entnehmen. Das System trägt nun den Titel „omniVision“

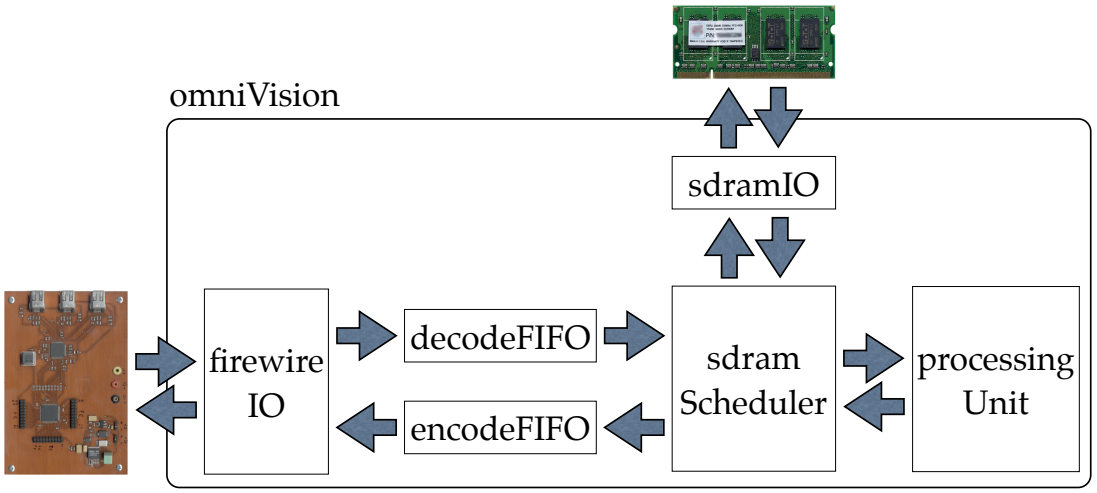


Abbildung 5.1: Der endgültige Systemaufbau

und gliedert sich in die folgenden Einheiten:

firewireIO Diese Einheit kümmert sich um die Kommunikation mit der Firewire-Platine und dem Firewire-Bus. Sie reicht den vom Firewire-Bus gelesenen isochronen Datenstrom an die Einheit „decodeFIFO“ weiter und nimmt die fertig verarbeiteten Videodaten von „encodeFIFO“ entgegen und gibt sie über den Firewire-Bus als isochronen Datenstrom aus.

decodeFIFO Die Einheit „decodeFIFO“ kümmert sich um die Weiterleitung der Daten von der „firewireIO“ zur „sdramScheduler“. Dabei wird der Datenstrom dekodiert.

encodeFIFO Die Einheit „encodeFIFO“ kümmert sich um die Weiterleitung der Daten von der „sdramScheduler“ zur „firewireIO“. Dabei wird der Datenstrom neu kodiert.

sdramScheduler Dieser Speicher-Scheduler verwaltet den Speicher und beantwortet die Lese- und Schreibanfragen von der „decodeFIFO“, der „encodeFIFO“ und der „processingUnit“.

sdramIO Die Einheit „sdramIO“ kümmert sich um den Zugriff auf den SDRAM. Weitere Informationen über diese Einheit findet man in Abschnitt 3.3. An dieser Einheit wurden für den Einsatz in „omniVision“ keine Änderungen vorgenommen.

processingUnit Die Verarbeitungseinheit „processingUnit“ beherbergt den verwendeten Algorithmus und errechnet aus dem Quelldatenstrom den Zieldatenstrom.

Die sich durch diesen Aufbau ergebenden Datenströme werden in Abbildung 5.2 dargestellt.

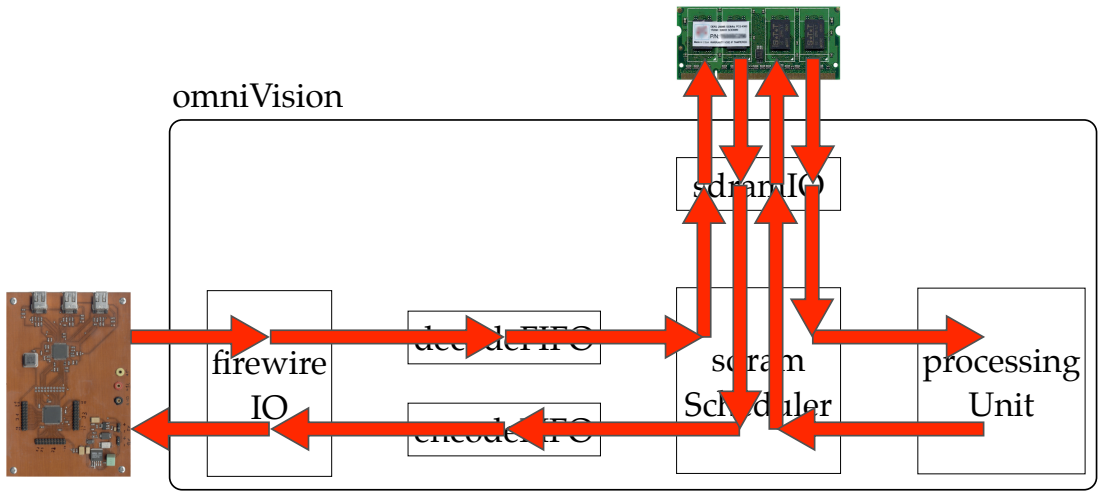


Abbildung 5.2: Die Datenströme in „omniVision“

5.2 Strukturbaum

Als gute Entwurfsmethode für VHDL gilt die Trennung von Struktur und Verhalten. Eine Einheit soll, soweit dies möglich ist, entweder eine Strukturbeschreibung oder eine Verhaltensbeschreibung enthalten. Diese Methode wird u.a. in [LWS94] empfohlen.

Unter einer Strukturbeschreibung versteht man die Verbindung von mehreren Einheiten zu einer grösseren Einheit. Die Funktion bzw. das Verhalten dieser Einheit ergibt sich somit aus dem Zusammenspiel der verbundenen Einheiten. Gleichzeitig entsteht so auch eine Hierarchie unter den Einheiten. Eine Verhaltensbeschreibung indes beschreibt direkt das Verhalten der Einheit, ohne auf andere Einheiten zurückzugreifen.

Da Einheiten mit Strukturbeschreibung schlussendlich immer auf Einheiten mit Verhaltensbeschreibung verweisen, ergibt sich durch diese Entwurfsmethode für das Gesamtsystem ein Baum, dessen Knoten Einheiten mit Strukturbeschreibung und dessen Blätter Einheiten mit Verhaltensbeschreibung sind.

Diese Entwurfsmethode wurde für „omniVision“ durchgehend eingehalten. Die Darstellung des Strukturbaumes findet man in Abbildung 5.3.

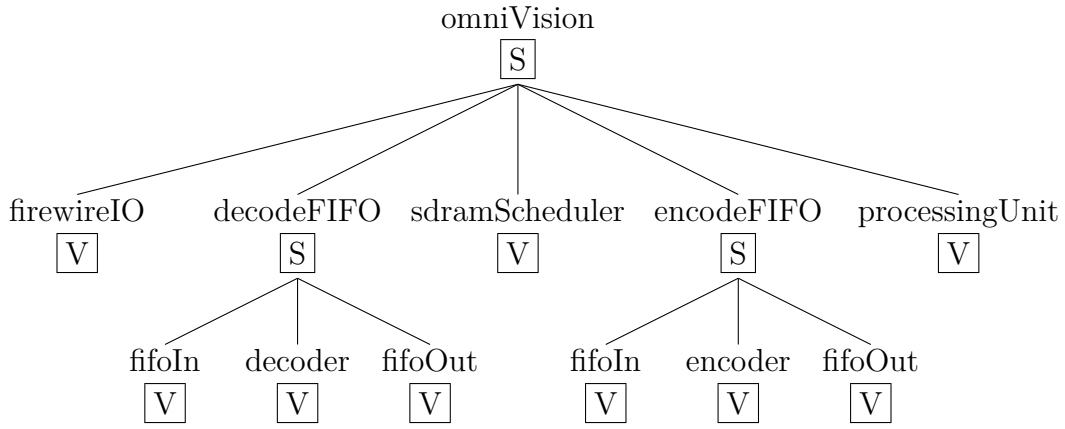


Abbildung 5.3: Der Strukturbaum von „omniVision“

5.3 Die Einheit omniVision

Die Einheit „omniVision“ verbindet alle anderen Einheiten miteinander. Hier erfolgt die Verteilung des Takt- und des Reset-Signals. Aus dem ursprünglichen Taktsignal wird durch Negation ein komplementäres Taktsignal erzeugt. Beide

Taktsignale werden dann nach Bedarf an die Einheiten verteilt. Des weiteren wird das „fwBusReset“-Signal, welches die „firewireIO“ bei einem Firewire-Bus-Reset ausgibt, verarbeitet. Dieses Signal sorgt dafür, dass alle Einheiten bis auf die „firewireIO“ neu gestartet werden. Ein Reset über den Reset-Knopf der FPGA-Platine hingegen sorgt dafür, dass alle Einheiten einen Reset vollziehen.

5.4 Die Einheit firewireIO

5.4.1 Ein- und Ausgänge

Eine Darstellung der Ein- und Ausgänge der Einheit „firewireIO“ findet man in Abbildung 5.4. Ihre Funktion im einzelnen:

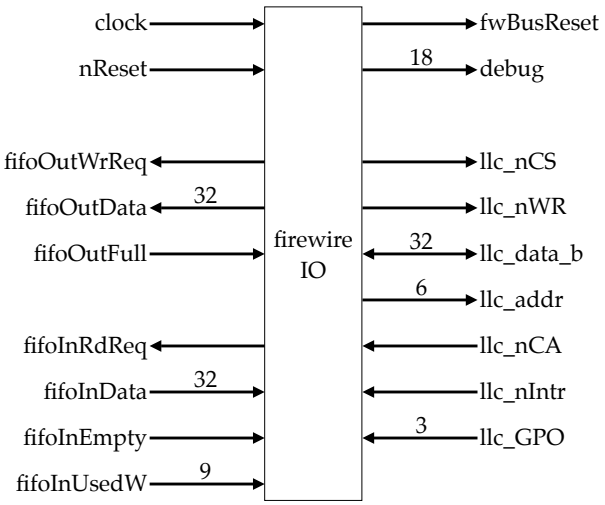


Abbildung 5.4: Die Ein- und Ausgänge der Einheit „firewireIO“

clock liefert das Taktsignal.

nReset liefert den Reset. Diese Leitung ist „active-low“.

fifoOutWrReq fordert den Ausgangs-FIFO auf die Daten in den FIFO zu schreiben.

fifoOutData befördert die Daten an den Ausgangs-FIFO.

fifoOutFull gibt an, ob der Ausgangs-FIFO voll ist.

fifoInRdReq fordert Daten vom Eingangs-FIFO an.

fifoInData liefert die Daten vom Eingangs-FIFO.

fifoInEmpty gibt an, ob der Eingangs-FIFO leer ist.

fifoInUsedW gibt an, wieviele Elemente sich im Eingangs-FIFO befinden.

fwBusReset wird benutzt, wenn ein Bus-Reset auf dem Firewire-Bus auftritt. Ein über diese Leitung ausgelöster Reset setzt alle anderen Einheiten bis auf „firewireIO“ selber zurück. Dies ist notwendig, da „fwBusReset“ direkt an „llc_nIntr“ angeschlossen ist und der Interrupt erst zurückgesetzt werden muss, bevor diese Leitung wieder auf '1' zurückgeht. „fwBusReset“ ist „active-low“.

debug befördert die Statusinformationen von „firewireIO“ zur „errorAdministration“.

llc_nCS teilt der Firewire-Platine mit, dass man einen Lese- oder Schreibvorgang vornehmen möchte. Der Name steht für „cycle start“. Diese Leitung ist „active-low“.

llc_nWR wählt zwischen Lese- und Schreibzugriff aus. Der Name steht für „write request“. Diese Leitung ist „active-low“. Eine '1' bedeutet somit einen Lesezugriff, während eine '0' einen Schreibzugriff selektiert.

llc_data_b befördert die Daten von und zu der Firewire-Platine.

llc_addr überträgt die zu lesende oder zu schreibende Adresse an die Firewire-Platine.

llc_nCA gibt an, dass die Firewire-Platine einen Zugriff ausgeführt hat. Der Name steht für „cycle acknowledge“.

llc_nIntr gibt an, ob es zu einem Interrupt gekommen ist. Diese Leitung ist „active-low“.

llc_GPO liefert kontinuierlich den Status von drei ausgewählten Registern. GPO steht dabei für „general-purpose output“.

5.4.2 Zustände

Den Zustandsautomaten der Einheit „firewireIO“ findet man in Abbildung 5.5. Der Übersichtlichkeit halber wurden für die Funktionalität unwichtige Zustände weggelassen. Die Funktion der Zustände im einzelnen:

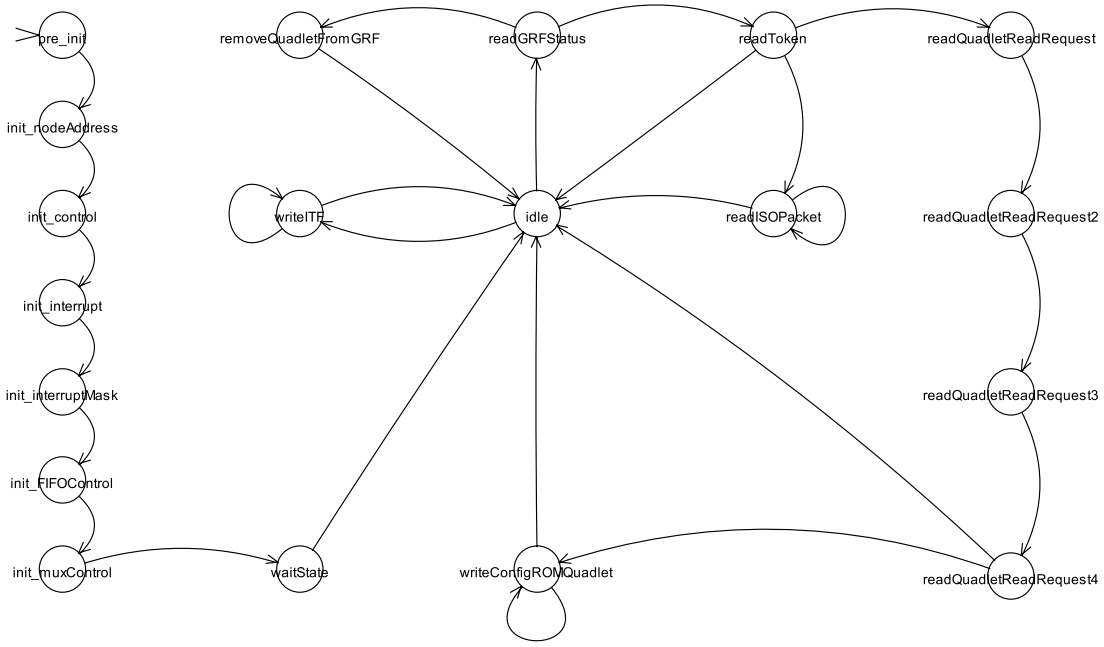


Abbildung 5.5: Der Zustandsautomat der Einheit „firewireIO“

pre_init

Dies ist der Startzustand. Er ist dafür da einige Takte zu warten, bis mit der Initialisierung begonnen wird. Dies ist notwendig, damit die Komponenten der Firewire-Platine sich initialisieren können. Anschliessend geht es im Zustand „init“ weiter.

init_nodeAddress

Dieser Zustand dient dazu die „Node Address“-Register zu setzen. Dies wird in „omniVision“ nicht benutzt ist aber der Vollständigkeit halber vorhanden. Es folgt der Zustand „init_control“.

init_control - todo

Hier wird das „Control“-Register gesetzt. Es werden die folgenden Werte auf ’1’ gesetzt:

- „IDVal“ sorgt dafür, dass Pakete, die an die „Node ID“ gesendet werden, angenommen werden.
- „TxAEn“ aktiviert den Versand von asynchronen Datenpaketen.
- „RxAEn“ aktiviert den Empfang von asynchronen Datenpaketen.
- „AckCEn“ aktiviert den Versand von Bestätigungspaketen für den Empfang von „Write Request“-Paketen, wenn der GRF nicht voll war und diese aufnehmen konnte.
- „TxIEn“ aktiviert den Versand von isonchronen Datenpaketen.
- „RxIEn“ aktiviert den Empfang von isonchronen Datenpaketen.
- „RstTx“ sorgt für einen Reset des Transmitters.
- „RstRx“ sorgt für einen Reset des Receivers.
- „TrgEn“ aktiviert die „Trigger“-Funktion. Diese sorgt dafür, dass empfangene Pakete in kleinere Pakete aufgeteilt werden. Die Grösse dieser Pakete wird mit der „Trigger Size“ festgelegt. Dies ermöglicht es zum einen Pakete zu empfangen, die grösser sind als der „GRF“ und zum anderen bereits Daten auszulesen, während der Rest des Paketes noch empfangen wird.
- „IRP1En“ aktiviert den Empfang von isochronen Daten auf dem Kanal, der im Register „IR Port 1“ eingestellt ist. Dort ist standardmässig Kanal 0 eingestellt. Da dies auch der Kanal ist, auf dem die Kamera sendet, wird dieser Wert von „omniVision“ nicht verändert.

Anschliessend geht es im Zustand „init_interrupt“ weiter.

init_interrupt

Hier werden alle evtl. bisher aufgetretenen Interrupts gelöscht. Dies geschieht durch das Überschreiben aller Werte mit ’1’. Der nächste Zustand ist „init_interruptMask“.

init_interruptMask

Die Interrupt-Maske bestimmt, bei welchen Interrupts die Firewire-Platine den „llc_nIntr“-Ausgang aktiviert. Hier wird nur der Wert von „PhRst“ auf ’1’ gesetzt. Somit gibt die Firewire-Platine nur dann einen Interrupt über die „llc_nIntr“-Leitung aus, wenn es einen Bus Reset auf dem Firewire-Bus gibt. Weiter geht es im Zustand „init_FIFOControl“.

init_FIFOControl

In diesem Zustand werden die Einstellungen für die FIFOs der Firewire-Platine vorgenommen. Es werden die folgenden Werte gesetzt:

- „ClrATF“ sorgt dafür, dass der ATF gelöscht wird.
- „ClrITF“ sorgt dafür, dass der ITF gelöscht wird.
- „ClrGRF“ sorgt dafür, dass der GRF gelöscht wird.
- „TriggerSize“ bestimmt die Grösse in die grössere Pakete geteilt werden. Hier wurde 50 Quadlets als Grösse ausgewählt.
- „ATFSize“ bestimmt die Grösse des ATF. Dem ATF werden 8 Quadlets zugeordnet.
- „ITFSize“ bestimmt die Grösse des ITF. Dem ITF werden 161 Quadlets zugeordnet.

Es folgt der Zustand „init_muxControl“.

init_muxControl

Hier wird bestimmt, welche Funktion bzw. welche Register über den Ausgang „llc_GPO“ ausgegeben werden:

- GPO0 trägt den Wert von „GRF empty“. Hiermit lässt sich sofort erkennen, ob Daten im GRF liegen, die gelesen werden können.
- GPO1 trägt den Wert von „ITF full“. Damit lässt sich erkennen, ob der ITF bereits voll ist.
- GPO2 trägt den Wert von „ITF empty“. Dies erlaubt es jederzeit auszulesen, ob der ITF leer ist und Daten hineingeschrieben werden können.

Der nächste Zustand ist „waitState“.

waitState

Dieser Zustand ist dazu da der Firewire-Platine genug Zeit zu lassen, damit sie auf die Änderungen der Register reagieren kann. Es folgt der Zustand „idle“.

idle

Der Zustand „idle“ stellt den zentralen Zustand dar, in den immer wieder zurückgekehrt wird. Hier wird entschieden, welche Aktion als nächstes vorgenommen wird. Befinden sich Daten im GRF, so wird in den Zustand „readGRFStatus“ gewechselt, um diese auszulesen. Ist der GRF hingegen leer, so wird nachgeschaut, ob Daten im Eingangs-FIFO liegen und ob der ITF frei ist. Ist beides der Fall, so wird in den Zustand „writeITF“ gewechselt, um die Daten vom FIFO an die Firewire-Platine durchzureichen.

readGRFStatus

Dieser Zustand liest das Register „GRF Status“ der Firewire-Platine aus. Der Wert „cd“ in diesem Register gibt an, ob das oberste Quadlet im GRF ein Token ist oder nicht. Ein Token ist ein von der Firewire-Platine generierter Header, welcher u.a. Informationen darüber enthält, welche Daten im folgenden Paket enthalten sind. Ist das nächste Quadlet im GRF ein Token, so wird in den Zustand „readToken“ übergegangen, ist es keines, so folgt der Zustand „removeQuadletFromGRF“.

removeQuadletFromGRF

Der Zustand „removeQuadletFromGRF“ dienst dazu das oberste Quadlet vom GRF zu entfernen. Dies gescheit, indem einfach ein Lesezugriff gestartet wird, ohne das die von der Firewire-Platine ausgegebenen Daten von den Datenleitungen abgegriffen werden. Anschliessend geht es zurück in den Zustand „idle“.

readToken

In diesem Zustand wird das Token aus dem GRF gelesen. Der „transaction code“ gibt dabei Auskunft darüber, was für ein Paket empfangen wurde. Hat dieser den hexadezimalen Wert ’4’ so wurde ein „Quadlet Read Request“ empfangen und es wird in den Zustand „readQuadletReadRequest“ verzweigt. Hat dieser den Wert ’A’, so wurden isochrone Daten empfangen und es wird in den Zustand „readISOPacket“ gewechselt. Zusätzlich dazu wird der Wert „Write Count“ in einer Variable gesichert. Dieser Wert sagt aus, wie viele Daten-Quadlets sich in diesem Paket befinden. In jedem anderen Fall geht die Einheit in den Zustand „idle“ zurück. Dies bedeutet, dass jede andere Art von Paketen im GRF schlicht ignoriert wird, da sie für die einwandfreie Funktion des Systems irrelevant sind.

readISOPacket

Hier werden die empfangenen isochronen Daten aus dem GRF ausgelesen und an den Ausgangs-FIFO übergeben. Der Wert aus „Write Count“ wird hierbei dazu verwendet die richtige Anzahl an Quadlets aus dem GRF auszulesen. Erst wenn alle Daten-Quadlets vom GRF gelesen wurden geht die Einheit in den Zustand „idle“ zurück.

readQuadletReadRequest

Dieser Zustand und seine Nachfolgezustände lesen ein „Quadlet Read Request“-Paket aus dem GRF aus. Dabei werden die gelesenen Daten in die entsprechenden Variablen gesichert, insofern sie für eine Antwort auf dieses Paket notwendig sind. In „readQuadletReadRequest“ wird das „Transaction Label“ gesichert. Dies ist eine eindeutige Nummer für jede Transaktion zwischen zwei Firewire-Nodes.

In „readQuadletReadRequest2“ werden die „Source ID“ und „Destination Offset High“ gesichert. Die „Source ID“ bezeichnet dabei die eindeutige „Node ID“ des Senders der Nachricht. Die „Destination Offset High“ wird mit der „Destination Offset Low“ konkateniert, welche in „readQuadletReadRequest3“ ausgelesen wird und bezeichnet die Adresse, welche der Sender der Nachricht gelesen haben möchte.

In „readQuadletReadRequest4“ wird dann überprüft, ob sich diese Adresse im „Config ROM“ befindet. Ist dies der Fall, so wird in den Zustand „writeConfigROMQuadlet“ gewechselt. Ansonsten wird die Nachricht einfach ignoriert und es geht wieder zurück in den Zustand „idle“.

writeConfigROMQuadlet

Hier wird auf das „Quadlet Read Request“-Paket mit einem „Quadlet Read Response“-Paket geantwortet. Die vorher ermittelte Adresse wird aus dem „Config ROM“ ausgelesen und die Daten an den Sender übertragen. Anschliessend geht es zurück in den Zustand „idle“.

writeITF

Dieser Zustand überträgt die Videodaten aus dem Eingangs-FIFO an die Firewire-Platine. Hierbei wird immer ein komplettes Package an die Firewire-Platine übertragen. Die Anzahl der Quadlets in einem Package ist dabei abhängig von der gewählten Übertragungsrate des Ausgabevideos und ist in der „Firewire Digital

Camera Specification“ festgelegt. Nachdem das ganze Package an die Firewire-Platine übertragen wurde geht es zurück in den Zustand „idle“.

5.4.3 Datenraten

Da es Ziel dieser Arbeit ist sowohl das Eingangsvideo als auch das Ausgangsvideo mit einer möglichst hohen Bildrate zu empfangen und zu senden, ist die Datenrate bei der Übertragung von und zur Firewire-Platine natürlich von zentraler Bedeutung.

Die Übertragung erfolgt dabei mit einer Datenbreite von 32 Bit und einem Takt von 33 MHz. Da sowohl Lese- als auch Schreiboperationen jeweils 3 Takte benötigen, was empirisch ermittelt wurde, liegt die maximale Datenrate bei:

$$\frac{32\,\text{Bit} \cdot 33\,\text{MHz}}{3\,\text{Takte}} = 352\,\text{MBit/sec} \tag{5.1}$$

Dies ist natürlich nur ein theoretischer Wert, der in der Praxis nicht ganz erreicht werden wird. Dies liegt u.a. daran, dass die Einheit zwischen der Übertragung von Videodaten von und zur Firewire-Platine auch die Register der Firewire-Platine auslesen muss. Ausserdem enthalten sowohl der empfangene als auch der gesendete Datenstrom zusätzlich zu den Videodaten auch die Header, die für den Versand über den Firewire-Bus notwendig sind.

Mit dieser Datenrate müssen sowohl der Eingabe- als auch der Ausgabedatenstrom übertragen werden. Daher muss auch betrachtet werden, welche Datenraten für die verschiedenen Übertragungsgeschwindigkeiten für das Eingangs- und das Ausgangsvideo anfallen. Beide Videos werden mit einer Auflösung von 1280 x 960 Pixeln und einer Farbtiefe von 16 Bit pro Pixel übertragen. Zusammen mit den dabei möglichen Bildraten von 7,5, 3,75 und 1,875 Bildern pro Sekunde (fps) ergeben sich folgende Datenraten:

$$7{,}5\,\text{fps} \cdot 1280 \cdot 960 \cdot 16\,\text{Bit} \approx 147\,\text{MBit/sec} \tag{5.2}$$

$$3{,}75\,\text{fps} \cdot 1280 \cdot 960 \cdot 16\,\text{Bit} \approx 74\,\text{MBit/sec} \tag{5.3}$$

$$1{,}875\,\text{fps} \cdot 1280 \cdot 960 \cdot 16\,\text{Bit} \approx 37\,\text{MBit/sec} \tag{5.4}$$

Vergleicht man nun die Datenraten der Videos mit der maximal möglichen Datenrate, so stellt man fest, dass es möglich sein sollte beide Videodatenströme mit 7,5 Bildern pro Sekunde zu übertragen. Dies ist allerdings aus einem anderen Grund nicht möglich: Mit dem Programm „Coriander“ (siehe Abschnitt 4.1) wurde ermittelt, dass der Datenstrom der Kamera bei einer Bildrate von 7,5 Bildern pro Sekunde bereits über 50% der verfügbaren Datenrate des Firewire-Busses verbraucht. Alle anderen Kombinationen von Bildraten hingegen sind technisch

möglich. Dies bedeutet aber nicht, dass sämtliche Kombinationen auch sinnvoll sind. So erscheint es wenig vorteilhaft, wenn das Ausgangsvideo mit einer höheren Bildrate ausgegeben wird, als das Eingangsvideo empfangen wird. Somit reduziert sich die Anzahl der sinnvollen und technisch möglichen Kombinationen auf fünf (siehe Tabelle 5.1). Diese Kombinationen von Bildraten werden aber nicht alle

Eingangsbildrate	**Ausgangsbildrate**
7,5 fps	3,75 fps
7,5 fps	1,875 fps
3,75 fps	3,75 fps
3,75 fps	1,875 fps
1,875 fps	1,875 fps

Tabelle 5.1: Sinnvolle und technisch mögliche Kombinationen von Bildraten für die Einheit „firewireIO“

von der „firewireIO“ unterstützt. Der Grund dafür liegt in der Ansteuerung der Firewire-Platine. Für die Ausgabe des Videos auf dem Firewire-Bus werden die Daten in den „ITF“ (isochronous transfer FIFO) genannten Zwischenspeicher der Firewire-Platine übertragen (siehe Abschnitt 3.2).

Aufgrund der vielen Probleme bei der Ansteuerung der Firewire-Platine während der Implementierung des Systems wurde hierfür eine möglichst einfache und stabile Methode ausgewählt. Die Grösse des ITF wird dabei so gross gewählt, dass exakt ein Paket plus dazugehörenden Header in den ITF passen. So kann dieses Paket komplett in den Speicher übertragen werden, bevor die Firewire-Platine damit beginnt Daten auf dem Firewire-Bus auszugeben. Da es aber auch möglich sein sollte Eingangsdatenströme mit allen möglichen Bildraten zu empfangen, war es notwendig, dass der „GRF“ (general receive FIFO) grösser als der ITF ausfällt. Da aber alle drei Speicher gemeinsam nur über 512 Quadlets verfügen und da die Paketgrösse schon bei einem Ausgabevideo mit 3,75 Bildern pro Sekunde 320 Quadlets beträgt (siehe Tabelle 5.2) und der ITF damit mehr als die Hälfte der Gesamtkapazität verbrauchen würde, wurde die Ausgangsbildrate auf 1,875 Bilder pro Sekunde festgelegt. Die somit momentan mit der „firewireIO“ möglichen Kombinationen an Bildraten findet man in Tabelle 5.3.

Bildrate	7,5 fps	3,75 fps	1,875 fps
Paketgrösse	640 Quadlets	320 Quadlets	160 Quadlets

Tabelle 5.2: Die Paketgrössen bei der Übertragung des isochronen Datenstroms

Eine Steigerung der Bildrate für das Ausgabevideo ist zwar technisch noch möglich, würde aber eine Menge Arbeit mit sich bringen. Der gesamte Ausgabealgorithmus müsste neu entwickelt werden und die Grössen von ITF und GRF müssten neu festgelegt und auf einwandfreie Funktion getestet werden. Ausserdem wäre ein weitaus dynamischeres Scheduling im Zustand „idle“ erforderlich. Dieser Aufwand ist aber erst dann sinnvoll, wenn diese Ausgangsbildrate wirklich benötigt wird, was im Zuge dieser Diplomarbeit nicht der Fall ist.

Eingangsbildrate	**Ausgangsbildrate**
7,5 fps	1,875 fps
3,75 fps	1,875 fps
1,875 fps	1,875 fps

Tabelle 5.3: Die mit der Einheit „firewireIO“ momentan möglichen Kombinationen von Bildraten

5.5 Die Einheit decodeFIFO

Die Einheit „decodeFIFO“ stellt eine Kombination zweier FIFOs mit einer dazwischen gelagerten Dekodiereinheit dar. Abbildung 5.6 verdeutlicht diesen Entwurf. Hierbei ist „decodeFIFO“ eine rein strukturelle Beschreibung, während sich sämtliche Funktionalität in den Untereinheiten verbirgt. Auf diese Art und Weise ist es problemlos möglich die verschiedenen Einheiten auszutauschen und das Gesamtsystem somit an veränderte Bedingungen anzupassen.

Abbildung 5.6: Die Eingabe-FIFOs mit der dazwischen gelagerten Dekodiereinheit

5.5.1 Ein- und Ausgänge

Eine Darstellung der Ein- und Ausgänge der Einheit „decodeFIFO“ findet man in Abbildung 5.7. Ihre Funktion im einzelnen:

clockFalling_i liefert das negierte Taktsignal.

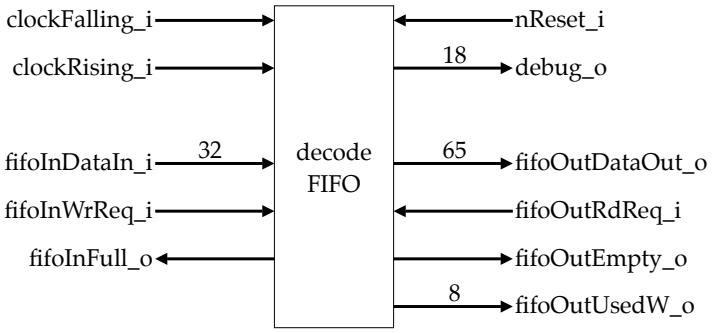


Abbildung 5.7: Die Ein- und Ausgänge der Einheit „decodeFIFO“

clockRising_i liefert das Taktsignal.

nReset_i liefert den Reset. Diese Leitung ist „active-low“.

debug_i befördert die Statusinformationen der Dekodiereinheit zur „errorAdministration“.

fifoInData_i liefert Daten für den Eingangs-FIFO.

fifoInWrReq_i gibt an, dass Daten für den Eingangs-FIFO auf den Datenleitungen bereitstehen, die geschrieben werden sollen.

fifoInFull_o überträgt die Information, ob der Eingangs-FIFO voll ist, an die „firewireIO“.

fifoOutDataOut_o befördert die Daten vom Ausgangs-FIFO an die „sdramScheduler“.

fifoOutRdReq_i liefert die Anfrage für einen Lesezugriff auf den Ausgangs-FIFO von der „sdramScheduler“.

fifoOutEmpty_o befördert die Information, ob sich Daten im Ausgangs-FIFO befinden, an die „sdramScheduler“.

fifoOutUsedW_o überträgt die Anzahl der Elemente, die sich im Ausgangs-FIFO befinden, an die „sdramScheduler“.

5.5.2 Eingesetzte FIFOs

Die FIFOs wurden durch das Entwurfsprogramm „Quartus“ generiert. Der interne Aufbau wird hier deshalb nicht weiter beschrieben. Es handelt sich beim Eingangs-FIFO „fifoIn“ um einen 32 Bit breiten FIFO mit einer Kapazität von

512 Elementen, während der Ausgangs-FIFO „fifoOut“ eine Breite von 65 Bit und eine Kapazität von 256 Elementen besitzt. Die Breite des Eingangs-FIFOs ergibt sich direkt durch die Datenbreite der Firewire-Platine, da die Daten aus dem GRF direkt an den „fifoIn“ weitergereicht werden. Die Breite des Ausgangs-FIFOs hingegen ergibt sich durch die Datenbreite des eingesetzten Speichers. Hinzu kommt noch ein weiteres Bit, welches, wenn gesetzt, den Anfang eines neuen Bildes signalisiert.

5.5.3 Die Dekodiereinheit

Die Dekodiereinheit „decoder“ transferiert die Daten aus dem Eingangs-FIFO in den Ausgangs-FIFO. Dabei werden auch die Header für den isochronen Datentransfer ausgelesen und entfernt, die sich noch im Datenstrom befinden. Den Zustandsautomaten der Dekodiereinheit findet man in Abbildung 5.8. Diese Einheit arbeitet für die Zustandsübergänge mit einer Methode, bei der beim Sprung in den Zustand „idle“ immer der darauf folgende Zustand festgelegt wird. Der Übersichtlichkeit halber wurden für die Funktionalität unwichtige Zustände weggelassen. Die Funktion der Zustände im einzelnen:

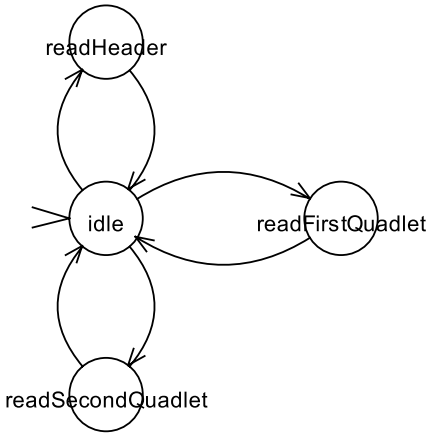


Abbildung 5.8: Der Zustandsautomat der Einheit „decoder“

idle

Dies ist der Startzustand und der zentrale Angelpunkt der Dekodiereinheit. Von hier wird in alle anderen Zustände verzweigt. Direkt beim Start bzw. nach einem Reset geht es zuerst in den Zustand „readHeader“.

readHeader

Dieser Zustand liest ein Quadlet vom Eingangs-FIFO. Da dieser Zustand nur direkt beim Start und nach einem vollständig ausgelesenen Package an die Reihe kommt enthält dieses Quadlet zwangsläufig einen Header vom Transport über den Firewire-Bus. Aus diesem wird ausgelesen, wieviele Bytes dieses Package enthält und ob es das erste Package des Bildes ist. Anschliessend wird in den Zustand „idle“ übergegangen, wobei der darauf folgende Zustand „readFirstQuadlet“ sein wird.

readFirstQuadlet

In diesem Zustand wird das Quadlet, welches vom Eingangs-FIFO gelesen wird, in einer Variable zwischengespeichert. Des weiteren wird noch das 65te Bit anhand der Informationen, die aus dem Header ausgelesen wurden, generiert und der Variable hinzugefügt. Dann geht es zurück in den Zustand „idle“, von wo es dann in den Zustand „readSecondQuadlet“ geht.

readSecondQuadlet

Hier wird das zweite Quadlet vom Eingangs-FIFO gelesen und den zwischengespeicherten Daten hinzugefügt. Diese 65 Bit werden dann an den Ausgangs-FIFO übertragen. Es folgt der Zustand „idle“. Der darauf folgende Zustand ist abhängig davon, ob das Package bereits komplett aus dem Eingangs-FIFO ausgelesen wurde. Ist dies der Fall, so folgt der Zustand „readHeader“, ansonsten geht es anschliessend im Zustand „readFirstQuadlet“ weiter.

5.6 Die Einheit encodeFIFO

Die Einheit „encodeFIFO“ stellt eine Kombination zweier FIFOs mit einer dazwischen gelagerten Kodiereinheit dar. Wie schon „decodeFIFO“ ist „encodeFIFO“ eine rein strukturelle Beschreibung mit den bekannten Vorteilen. Abbildung 5.9 verdeutlicht diesen Entwurf.

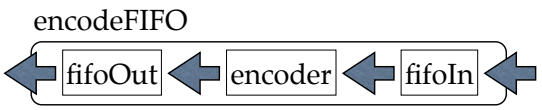


Abbildung 5.9: Die Ausgabe-FIFOs mit der dazwischen gelagerten Kodiereinheit

5.6.1 Ein- und Ausgänge

Eine Darstellung der Ein- und Ausgänge der Einheit „encodeFIFO“ findet man in Abbildung 5.10. Ihre Funktion im einzelnen:

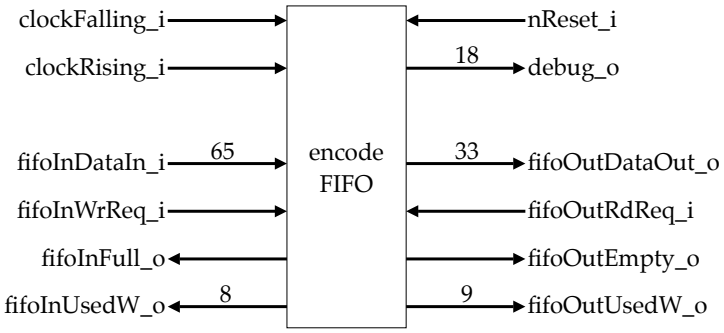


Abbildung 5.10: Die Ein- und Ausgänge der Einheit „encodeFIFO“

clockFalling_i trägt das negierte Taktsignal.

clockRising_i trägt das Taktsignal.

nReset_i ist die Reset-Leitung. Diese Leitung ist „active-low“.

debug_i befördert die Statusinformationen von der Kodiereinheit an die „error-Administration“ weiter.

fifoInData_i liefert die Daten von der „sdramScheduler“.

fifoInWrReq_i liefert die Anfrage für Schreibzugriff auf den Eingangs-FIFO von der „sdramScheduler“.

fifoInFull_o befördert die Information, ob der Eingangs-FIFO voll ist, an die „sdramScheduler“.

fifoInUsedW_o überträgt die Anzahl der Elemente, die sich im Eingangs-FIFO befinden, an die „sdramScheduler“.

fifoOutDataOut_o befördert die Daten vom Ausgangs-FIFO an die „firewireIO“.

fifoOutRdReq_i liefert die Anfrage für einen Lesezugriff auf den Ausgangs-FIFO von der „firewireIO“.

fifoOutEmpty_o befördert die Information, ob sich Daten im Ausgangs-FIFO befinden, an die „firewireIO“.

fifoOutUsedW_o überträgt die Anzahl der Elemente, die sich im Ausgangs-FIFO befinden, an die „firewireIO“.

5.6.2 Eingesetzte FIFOs

Auch die FIFOs in „encodeFIFO“ wurden von „Quartus“ generiert. Der Eingangs-FIFO „fifoIn“ besitzt eine Breite von 65 Bit und kann 256 Elemente aufnehmen. Der Ausgangs-FIFO „fifoOut“ hat eine Breite von 33 Bit und eine Kapazität von 512 Elementen. Die Breite der FIFOs ergeben sich wieder durch die Datenbreiten der angeschlossenen Einheiten. Lediglich der an die „firewireIO“ angeschlossene FIFO hat hier ein Bit mehr Breite, da er zusätzlich noch die Information transportieren muss, ob sich ein Quadlet um das erste im Bild handelt. Diese Information steckte bei der Dekodiereinheit „decoder“ noch im ankommenden Datenstrom.

5.6.3 Die Kodiereinheit

Die Kodiereinheit „encoder“ befördert die Daten vom Eingangs-FIFO in den Ausgangs-FIFO. Dabei teilt sie die im Eingangs-FIFO ankommenden Elemente auf zwei Elemente im Ausgangs-FIFO auf und passt dabei das Bit, welches das erste Quadlet in einem Bild kennzeichnet, entsprechend an. Den Zustandsautomaten der Kodiereinheit findet man in Abbildung 5.11. Der Übersichtlichkeit halber wurden für die Funktionalität unwichtige Zustände weggelassen. Die Funktion der Zustände im einzelnen:

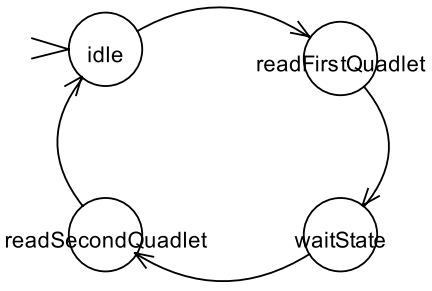


Abbildung 5.11: Der Zustandsautomat der Einheit „encoder“

idle

Der Zustand „idle“ ist der Startzustand. Hier wird darauf gewartet, dass der Eingangs-FIFO Daten enthält und der Ausgangs-FIFO nicht voll ist. Dann folgt der Zustand „readFirstQuadlet“.

readFirstQuadlet

Hier wird die erste Hälfte der Daten vom Eingangs-FIFO in den Ausgangs-FIFO übertragen. Sind die Daten im Eingangs-FIFO als erste im Bild gekennzeichnet, so wird dieses Bit dem in den Ausgangs-FIFO übertragenen Quadlet mitgegeben. Anschliessend geht es im Zustand „waitState“ weiter.

waitState

In diesem Zustand wird überprüft, ob der Ausgangs-FIFO weiterhin Daten aufnehmen kann. Ist dies der Fall so geht die Einheit über in den Zustand „readSecondQuadlet“.

readSecondQuadlet

Dieser Zustand überträgt die zweite Hälfte der Daten aus dem Eingangs-FIFO an den Ausgangs-FIFO. Danach geht die Einheit wieder in den Zustand „idle“ über.

5.7 Die Einheit sdramScheduler

5.7.1 Ein- und Ausgänge

Eine Darstellung der Ein- und Ausgänge der Einheit „sdramScheduler“ findet man in Abbildung 5.12. Ihre Funktion im einzelnen:

clock_i trägt das Taktsignal.

nReset_i ist die Reset-Leitung. Diese Leitung ist „active-low“.

debug_o befördert die Statusinformationen von der „sdramScheduler“ zur „errorAdministration“.

ioReq_o befördert die Anfrage für einen Zugriff auf den Speicher zur „sdramIO“.

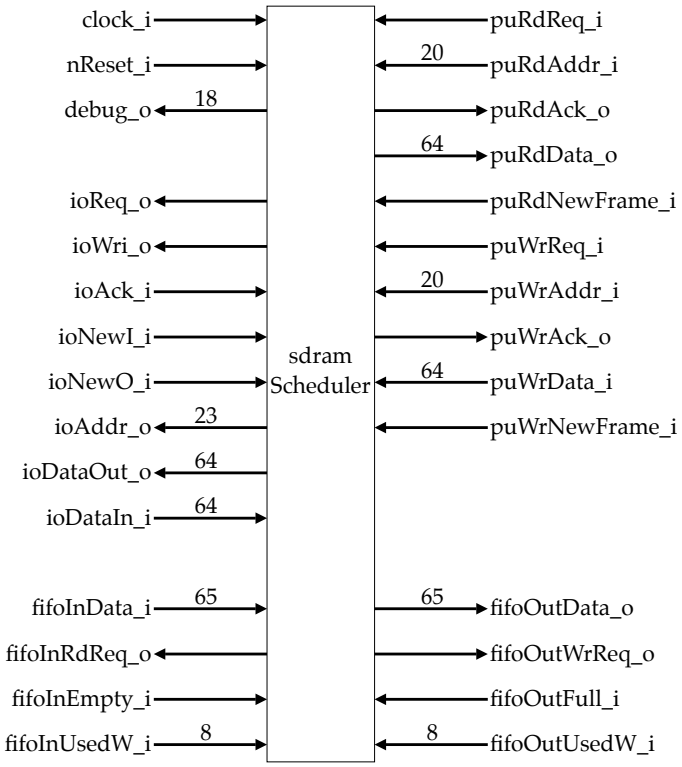


Abbildung 5.12: Die Ein- und Ausgänge der Einheit „sdramScheduler“

ioWri_o bestimmt, ob ein mit „ioReq_o“ beantragter Zugriff ein Lese- oder ein Schreibzugriff ist.

ioAck_i liefert die Information, ob ein Zugriff auf den Speicher gestartet werden kann.

ioNewI_i liefert die Information von der „sdramIO“, dass gelesene Daten an den Ausgängen bereit stehen.

ioNewO_i liefert während eines Burst-Schreibzugriffes die Information, dass neue Daten an die „sdramIO“ übertragen werden können.

ioAddr_o liefert der „sdramIO“ die Adresse, die gelesen oder geschrieben werden soll.

ioDataOut_o befördert die zu schreibenden Daten an die „sdramIO“.

ioDataIn_i liefert die gelesenen Daten von der „sdramIO“.

fifoInData_i liefert die Daten vom Eingangs-FIFO von der „decodeFIFO“.

fifoInRdReq_o befördert die Anfrage für einen Lesezugriff an die „decodeFIFO“.

fifoInEmpty_i liefert die Information, ob sich Daten im Eingangs-FIFO befinden von der „decodeFIFO“.

fifoInUsedW_i liefert die Anzahl der Elemente im Eingangs-FIFO von der „decodeFIFO“.

puRdReq_i liefert die Information von der „processingUnit“, dass ein Lesezugriff gestartet werden soll.

puRdAddr_i liefert die zu lesende Adresse von der „processingUnit“.

puRdAck_o informiert die „processingUnit“, ob ein Lesezugriff gestartet werden kann.

puRdData_o liefert der „processingUnit“ die gelesenen Daten.

puRdNewFrame_i liefert die Information, dass sich die zu lesenden Daten in einem neuen Bild befinden.

puWrReq_i liefert die Information von der „processingUnit“, dass ein Schreibzugriff gestartet werden soll.

puWrAddr_i liefert die zu schreibende Adresse von der „processingUnit“.

puWrAck_o informiert die „processingUnit“, ob ein Schreibzugriff gestartet werden kann.

puWrData_i liefert die zu schreibenden Daten von der „processingUnit“.

puWrNewFrame_i liefert die Information, dass sich die zu schreibenden Daten in einem neuen Bild befinden.

fifoOutData_o befördert die Daten zum Ausgangs-FIFO an die „encodeFIFO“.

fifoOutWrReq_o beantragt einen Schreibzugriff auf den Ausgangs-FIFO bei der „encodeFIFO“.

fifoOutFull_i liefert die Information von der „encodeFIFO“, ob der Ausgangs-FIFO voll ist.

fifoOutUsedW_i liefert die Anzahl der Elemente im Ausgangs-FIFO von der „encodeFIFO“.

5.7.2 Zustände

Den Zustandsautomaten der Einheit „sdramScheduler“ findet man in Abbildung 5.13. Der Übersichtlichkeit halber wurden für die Funktionalität unwichtige Zustände weggelassen. Die Funktion der Zustände im einzelnen:

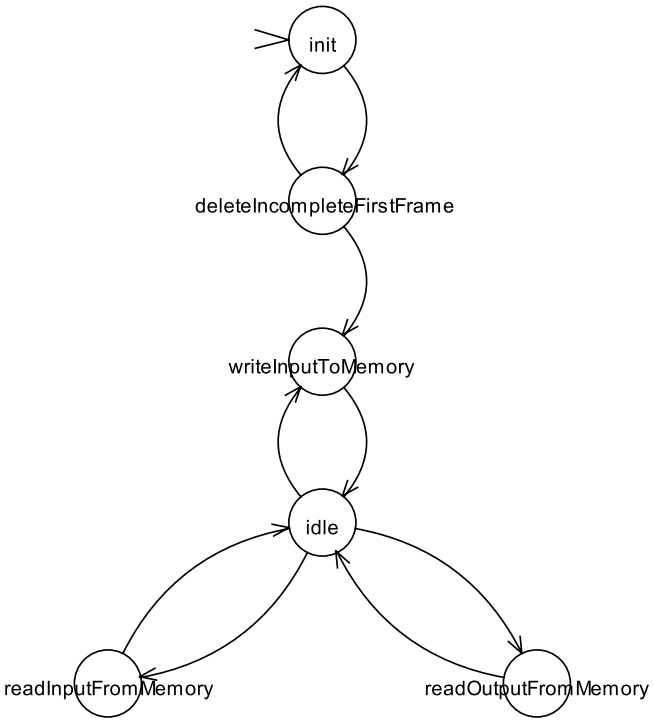


Abbildung 5.13: Der Zustandsautomat der Einheit „sdramScheduler“

init

Dies ist der Startzustand der Einheit „sdramScheduler“. Hier wird gewartet, bis sich Daten im Eingangs-FIFO befinden und dann nach „deleteIncompleteFirstFrame“ gewechselt.

deleIncompleteFirstFrame

Hier wird geschaut, ob die Daten im FIFO den Anfang eines Bildes enthalten. Ist dies nicht so, dann werden diese Daten gelöscht und es wird wieder in den Zustand „init“ übergegangen. Sind die Daten der Anfang eines Bildes, dann wird in den Zustand „writeInputToMemory“ gewechselt.

writeInputToMemory

In diesem Zustand werden die Daten aus dem Eingangs-FIFO ausgelesen und an den Speicher übertragen. Anschliessend geht es zurück in den Zustand „idle“.

idle

Dieses ist der zentrale Zustand, in den immer wieder zurück gekehrt wird. Hier findet das Scheduling zwischen den konkurrierenden Datenströmen statt, die alle vom oder zum Speicher fliessen. Von hier wird in die entsprechenden Zustände verzweigt, die jeweils für einen dieser Datenströme zuständig sind. Lediglich Schreibzugriffe von der „processingUnit“ werden hier direkt verarbeitet. Ein weiterer Zustand für diesen Datenstrom entfällt somit.

readInputFromMemory

Dieser Zustand überträgt die angeforderten Daten vom Speicher zur „processingUnit“. Danach geht es zurück in den Zustand „idle“.

readOutputFromMemory

Hier werden die Daten vom Speicher an den Ausgangs-FIFO übertragen. Anschliessend geht es zurück in den Zustand „idle“.

5.7.3 Speicher-Scheduling

Das Speicher-Scheduling wurde als reine Prioritätsliste implementiert. Die Prioritäten sind wie folgt festgelegt:

1. Schreiben des Eingangsdatenstroms in den Speicher
2. Lesen des Ausgangsdatenstroms aus dem Speicher
3. Lesen des Eingangsdatenstroms aus dem Speicher
4. Schreiben des Ausgangsdatenstroms in den Speicher

Das Schreiben des Eingangsdatenstroms steht dabei an oberster Stelle, da es ohne ein korrekt eingelesenes Eingangsvideo nicht möglich ist die fehlerfreie Funktion des Gesamtsystems zu gewährleisten. Das Lesen des Ausgangsdatenstroms folgt an zweiter Stelle, da für einwandfreies Ausgangsvideo eine unterbrechungsfreie

Übertragung der Daten innerhalb des Systems notwendig ist. An dritter Stelle und vierter Stelle folgen das Lesen des Eingangsdatenstroms und das Schreiben des Ausgangsdatenstroms. Dies sind die Funktionen, welche von der „processingUnit“ ausgelöst werden. Die Reihenfolge ist für den hier eingesetzten Algorithmus im Grunde egal, da dort entweder Daten gelesen oder Daten geschrieben werden, aber im allgemeinen ist es wichtiger, dass die Recheneinheit mit Daten versorgt wird und arbeiten kann, als dass ihr die fertigen Daten abgenommen werden.

Dieses Scheduling sorgt dafür, dass das System zuverlässig seinen Dienst verrichtet. Falls es wirklich zu Engpässen beim Speicherzugriff kommen sollte, so gehen diese alleine zu Lasten der Verarbeitungseinheit. Es würde also lediglich die Umrechnungsgeschwindigkeit sinken. Der Rest des Systems hingegen, also die Annahme des Datenstroms und die Ausgabe des Datenstroms, sollte auch dann weiter stabil funktionieren.

5.7.4 Aufteilung des Speichers

Um eine effektive Aufteilung des Speichers vorzunehmen muss man als erstes die Bildgrösse betrachten. Diese beläuft sich bei einer Auflösung von 1280 x 960 Pixeln und einer Farbtiefe von 16 Bit pro Pixel auf 2.457.600 Byte. Davon müssen nun, damit der effiziente Ablauf aller Funktionen des Gesamtsystems gewährleistet werden kann, sechs Bilder im Speicher vorgehalten werden:

1. ein Bild, in welches gerade der Eingangsdatenstrom geschrieben wird
2. ein Bild, welches ein fertig eingelesenes Eingangsbild enthält
3. ein Bild, von dem gerade die Verarbeitungseinheit liest
4. ein Bild, in welches gerade die Verarbeitungseinheit schreibt
5. ein Bild, welches ein fertig berechnetes Ausgabebild enthält
6. ein Bild, welches gerade ausgegeben wird

In Anbetracht der verwendeten Speichergrösse von 64 MiB stellt es kein Problem dar diese Anzahl an Bildern im Speicher zu sichern. Das System wäre auch mit nur 16 MiB Speicher ausgekommen. Dann aber wäre die Partitionierung des Speichers nicht so elegant zu lösen gewesen. So hingegen war es möglich den Speicher einfach in acht Stücke mit einer Grösse von 8 MiB aufzuteilen und diese mit den obersten drei Bit der Speicheradresse zu adressieren. Von diesen acht Teilen werden allerdings lediglich sechs verwendet, während die übrigen zwei Brach liegen. Tabelle 5.4 verdeutlicht dies. Die Ein- und die Ausgabe funktionieren dabei auf die gleiche Art und Weise: Es gibt jeweils ein Bild, in welches gerade geschrieben wird. Ist dieses vollständig geschrieben, so wird der Basiswert der Speicheradresse

000 Eingabebild wird geschrieben	**001** Eingabebild im Wartezustand	**010** Eingabebild wird gelesen	**011** unbenutzt
100 Ausgabebild wird geschrieben	**101** Ausgabebild im Wartezustand	**110** Ausgabebild wird gelesen	**111** unbenutzt

Tabelle 5.4: Die Partitionierung des Speichers

(die oberen drei Bit) des abgeschlossenen Bildes mit dem des wartenden Bildes vertauscht. Dadurch liegt nun das eben geschriebene Bild im Wartezustand, während das eben noch wartende überschrieben wird. Zusätzlich wird noch ein Bit gesetzt, welches der entsprechenden Lesefunktion anzeigt, dass sich ein neues Bild im Wartezustand befindet.

Das Lesen funktioniert ähnlich. Ist ein Bild vollständig ausgelesen und befindet sich im Wartezustand ein neueres Bild, dann wird die Basisadresse mit der des wartenden Bildes getauscht. Andernfalls wird auf einen Tausch verzichtet und das immer noch aktuelle Bild noch mal verwendet. Dies ist vor allem für die Videoausgabe unverzichtbar.

Diese Konstruktion macht es möglich, dass alle Funktionen reibungslos und effizient ihren Dienst verrichten können, ohne auf andere Funktionen Rücksicht nehmen oder warten zu müssen. Es wird auch dafür gesorgt, dass die verarbeiteten Bilder stets so aktuell wie möglich sind und dass Bilder, die nicht verarbeitet werden können oder veraltet sind, verworfen werfen.

5.8 Die Einheit processingUnit

5.8.1 Ein- und Ausgänge

Eine Darstellung der Ein- und Ausgänge der Einheit „processingUnit“ findet man in Abbildung 5.14. Ihre Funktion im einzelnen:

clock_i trägt das Taktsignal.

nReset_i liefert den Reset. Diese Leitung ist „active-low“.

debug_o befördert die Statusinformationen von „processingUnit“ zur „errorAdministration“.

rdReq_o liefert die Anfrage für einen Lesezugriff an die „sdramScheduler“.

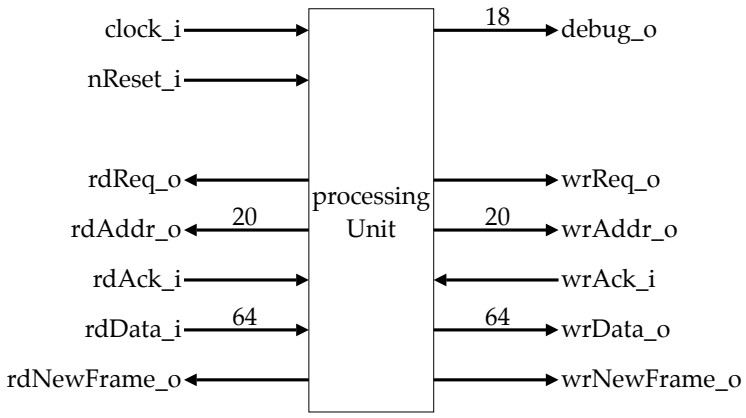


Abbildung 5.14: Die Ein- und Ausgänge der Einheit „processingUnit“

rdAddr_o liefert der „sdramScheduler“ die zu lesende Adresse.

rdAck_i liefert die Information, ob ein Lesezugriff gestartet werden kann.

rdData_i liefert die gelesenen Daten von der „sdramScheduler“.

rdNewFrame_o informiert die „sdramScheduler“, dass sich die Daten, welche gelesen werden sollen in einem neuen Bild befinden.

wrReq_o liefert die Anfrage für einen Schreibzugriff an die „sdramScheduler“.

wrAddr_o liefert der „sdramScheduler“ die zu schreibende Adresse.

wrAck_i liefert die Information, ob ein Schreibzugriff gestartet werden kann.

wrData_o liefert der „sdramScheduler“ die zu schreibenden Daten.

wrNewFrame_o informiert die „sdramScheduler“, dass sich die Daten, welche geschrieben werden sollen in einem neuen Bild befinden.

5.8.2 Der verwendete Algorithmus

Als Algorithmus zur Umrechnung der omnidirektionalen Bilder in Panoramabilder kam der in Abschnitt 3.4 erklärte Algorithmus für die einfache direkte Umwandlung zum Einsatz, welcher an einigen Stellen ein wenig an die Begebenheiten angepasst werden musste. Die folgende Auflistung der Anpassungen ist in die drei Phasen des Algorithmus unterteilt:

1. **Berechnung der Polarkoordinaten:** Bei der Berechnung des Radius und des Winkels kommen zwar Festkommazahlen zum Einsatz, die Ergebnisse aber

werden als ganze Zahlen gespeichert und somit gerundet. Dadurch wird die Genauigkeit ein wenig reduziert. Dies ist aber gerade beim Winkel unabdingbar, damit Sinus und Kosinus berechnet werden können. Diese Funktionen sind in der vorhandenen Hardware nicht enthalten und wurden daher als numerische Funktionen manuell implementiert, mit dem Winkel als Eingabewert in Grad. Mit einer Festkommazahl als Eingabewert hätten diese Listen deutlich grösser ausfallen müssen.

2. **Berechnung der kartesischen Koordinaten:** Für die Berechnung von x und y wird der Radius mit den Ausgabewerten von Sinus bzw. Kosinus multipliziert. Die dadurch entstehenden Festkommazahlen werden anschliessend als ganze Zahlen gespeichert. Dadurch wird auch hier die Genauigkeit ein wenig reduziert. Die Nachkommastellen wären aber lediglich für die Interpolation nötig und werden hier deshalb nicht weiter benötigt.

3. **Interpolation:** Auf eine Interpolation zwischen mehreren Pixeln wurde aus Geschwindigkeitsgründen komplett verzichtet. Zum einen entfällt so ein weiterer Berechnungsschritt und zum anderen würde eine Interpolation den Speicher deutlich stärker beanspruchen.

   Würde man z.B. mit Festkommazahlen für x und y die vier diese Stelle umgebenden Pixel nehmen und daraus den Zielpixel interpolieren, so hiesse das, dass statt einem vier Pixel geladen werden müssten. Da sich immer vier nebeneinanderliegende Pixel in einem Speicherbereich befinden, würde dies bedeuten, dass mindestens zwei mal, im schlechtesten Fall sogar vier mal aus dem Speicher gelesen werden müsste (siehe Abbildung 5.15). Geht man von einer Normalverteilung dieser Ereignisse aus, würde die Speicherbelastung im Schnitt auf das 2,5-fache ansteigen:

$$3 \cdot 25\% \cdot 2\,\text{Zugriffe} + 25\% \cdot 4\,\text{Zugriffe} = 2{,}5\,\text{Zugriffe} \tag{5.5}$$

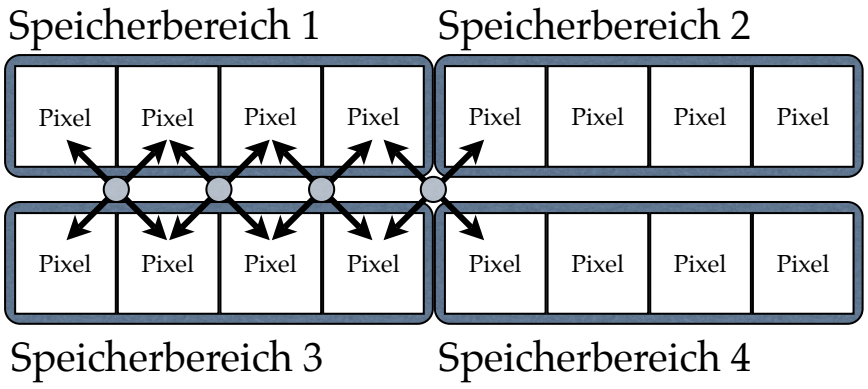


Abbildung 5.15: Interpolation würde die Anzahl der Speicherzugriffe vervielfachen

Da der hier eingesetzte Algorithmus jeden Ursprungspixel direkt aus dem Speicher lädt und das Zielbild die gleichen Dimensionen besitzt wie das Ursprungsbild, bedeutet dies, dass für jeden Zielpixel, ein kompletter Speicherbereich mit vier Pixeln geladen wird. Das Ursprungsbild wird also bereits vier mal komplett aus dem Speicher geladen. Eine Interpolation, wie eben beschrieben, würde dafür sorgen, dass jedes Bild zehn mal geladen wird. Dies ist ohne komplexere Verbesserungen des Speicherzugriffes nicht sinnvoll. Ein paar Verbesserungsmöglichkeiten sind in Kapitel 7 beschrieben.

Das Ergebnis des verwendeten Algorithmus verdeutlichen Abbildung 5.16 und 5.17. Dabei ist das Panoramabild nicht direkt das umgerechnete omnidirektionale Bild, sondern wurde wenige Sekunden nach diesem aufgenommen. Die Umrechnungsgeschwindigkeit liegt bei ca. 2 Bildern pro Sekunde. Diese Angabe wird bisher nirgends im System berechnet und wurde anhand eines über die 7-Segment-Anzeigen ausgegebenen Zählers geschätzt.

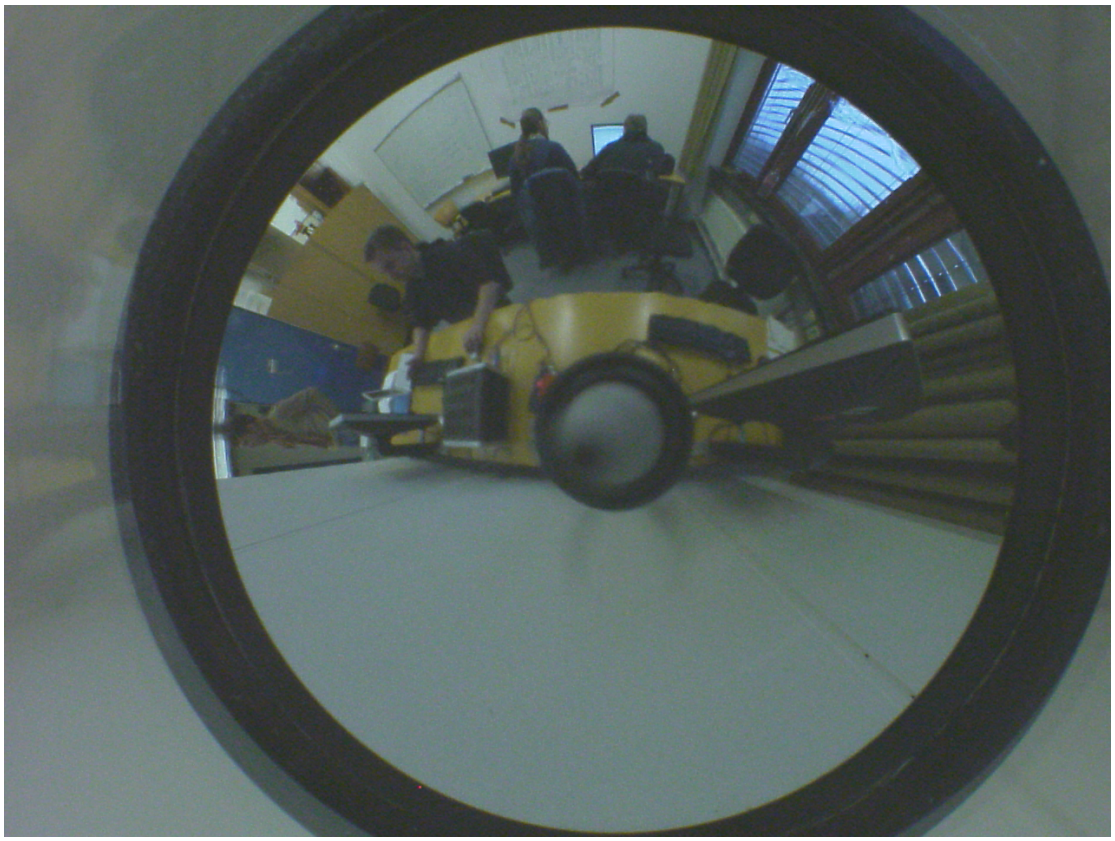


Abbildung 5.16: Ein Bild der omnidirektionalen Kamera

Abbildung 5.17: Ein Ausgabebild von „omniVision“

5.8.3 Zustände

Den Zustandsautomaten der Einheit „processingUnit“ findet man in Abbildung 5.18. Der Übersichtlichkeit halber wurden für die Funktionalität unwichtige Zustände weggelassen. Die Funktion der Zustände im einzelnen:

compute

Dies ist der Startzustand. Hier werden der Radius und der Winkel des Ursprungspixels berechnet. Danach geht es in den Zustand „compute2“.

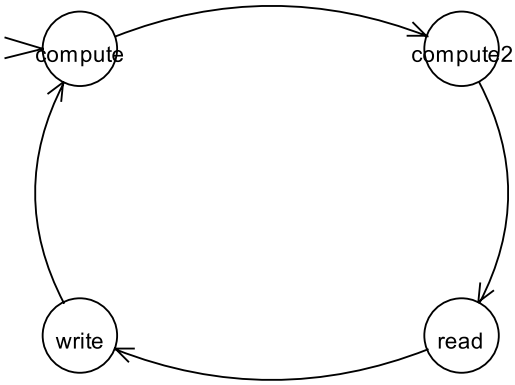


Abbildung 5.18: Der Zustandsautomat der Einheit „processingUnit“

compute2

In diesem Zustand werden mit dem Radius und mit dem Winkel der horizontale und der vertikale Abstand vom Mittelpunkt berechnet. Anschliessend geht es im Zustand „read“ weiter.

read

In „read“ wird aus dem Mittelpunkt und den Abständen vom Mittelpunkt der Pixel errechnet, welcher in das Zielbild eingesetzt wird. Aus diesem wird dann die Adresse errechnet, an welcher dieser gespeichert ist. Die Adresse wird an den Speicher-Scheduler übertragen, damit dieser die Daten aus dem Speicher laden kann. Es folgt der Zustand „write“.

write

Hier wird, sobald die aus dem Speicher gelesenen Daten bereitstehen, der Ursprungspixel aus diesen extrahiert. Dazu wird errechnet, an welcher Stelle der gelesenen Daten er gespeichert ist. Anschliessend wird aus dem Zielpixel berechnet, an welche Stelle des zu schreibenden Datenworts der Ursprungspixel gespeichert werden muss. Ist dies der letzte Pixel in diesem Datenwort, so wird der Speicher-Scheduler angewiesen dies an der entsprechenden Adresse zu speichern. Anschliessend geht es zurück in den Zustand „compute“.

5.9 Die Einheit errorAdministration

Die Einheit „errorAdministration“ entstand während der Implementierung des Systems aus dem Problem heraus, dass das FPGA-Board nur sehr wenige Möglichkeiten bietet Rückmeldungen über den aktuellen Zustand anzuzeigen. Da sie nicht Teil der Funktionalität des Gesamtsystems ist wird sie hier nur kurz erläutert.

Diese Einheit ist im Grunde ein Multiplexer. Bei ihr laufen die Statusausgaben aller anderen Einheiten zusammen, welche jeweils aus 18 Bit bestehen: Zwei mal acht Bit für die zwei 7-Segment-Anzeigen und zwei Bit für die zwei LEDs. Anhand der Einstellung auf dem DIP-Schalter wird nun die Statusausgabe einer Einheit an die 7-Segment-Anzeigen und die LEDs durchgereicht. Dadurch kann jederzeit die Ausgabe jeder Einheit betrachtet werden.

5.10 Ausnutzung des FPGA

Da die Klärung der Frage, ob die gewünschte Funktionalität überhaupt auf den vorhandenen FPGA passt, ein Ziel dieser Arbeit war, ist die Frage, wie viel Kapazität des FPGA verbraucht wurde natürlich von zentraler Bedeutung. Auch für einen weiteren Ausbau des Systems oder die Verwendung von komplexeren Algorithmen ist es wichtig zu wissen, ob noch Platz auf dem Chip vorhanden ist oder ein grösserer FPGA zum Einsatz kommen muss. In Tabelle 5.5 kann man die Ausnutzung des FPGA am Ende der Entwicklung sehen. Wie man leicht sieht wurden

	vorhanden	**verwendet**	**anteilig**
Logikelemente	8.320	3.390	ca. 40 %
Speicherbits	106.496	76.288	ca. 70 %

Tabelle 5.5: Die Ausnutzung des FPGA

die Kapazitäten des FPGA nicht vollständig verbraucht. Von den verbrauchten 76.288 Speicherbits entfallen im Übrigen 66.048 Bit auf die verwendeten FIFOs. Da diese die einzigen Einheiten waren, bei denen klar war, dass sie Gebrauch von den vorhandenen Speicherbits machen, wurden sie so entworfen, dass sie die vorhandenen Kapazitäten weitestgehend ausnutzen. Das bedeutet natürlich, dass dort auch wieder Kapazitäten eingespart werden können, indem man die FIFOs ein wenig verkleinert. Im Allgemeinen besteht also noch Raum für Erweiterungen. Welche Erweiterungen sinnvoll sind, wird in Kapitel 7 weiter ausgeführt.

Zusammenfassung und Fazit

6

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf und der Implementierung einer hardwarebasierten Vorverarbeitungseinheit für ein omnidirektionales Sichtsystem. Der grösste Teil dieser Arbeit bestand in der Einarbeitung in verschiedene technische Dokumentationen und Protokolle, deren anschliessende Implementierung und die dazu gehörenden Tests. Das Ergebnis waren meist nur wenige Zeilen Code, deren Entwicklung aber viel Zeit in Anspruch genommen hat. Auch die immer währenden Probleme mit der Stabilität der Hardware haben sich zeitlich deutlich bemerkbar gemacht.

Das primäre Ziel war es eine funktionsfähige Hardware zu implementieren, welche die omnidirektionalen Bilder der verwendeten hochauflösenden Kamera in Panoramabilder umrechnet und wieder ausgibt. Dies ist soweit gelungen. Die dafür notwendige Zwischenspeicherung sowohl des eingehenden als auch des ausgehenden Datenstroms wurde dabei so umgesetzt, dass das System in der Lage ist unabhängig von den Eingangsbildrate, der Ausgabebildrate und der Umrechnungsgeschwindigkeit seinen Dienst zu verrichten.

Die Geschwindigkeit bei der Umrechnung ist zwar nicht hoch genug um alle Bilder der Kamera zu verarbeiten, wenn diese im schnellsten Modus läuft, entspricht aber doch den Erwartungen. Hinzu kommt, dass hier noch eine Menge Raum für Optimierungen und Verbesserungen ist, welche die Geschwindigkeit durchaus soweit erhöhen könnten, dass sie den Firewire-Bus an seine Grenzen stossen lassen würde.

Auch der Frage, ob der FPGA genug Platz für die gewünschte Funktionalität bietet, wurde nachgegangen. So kann hier eindeutig gesagt werden, dass der Platz nicht nur für die jetzige Implementierung ausreicht, sondern auch für zukünftige Erweiterungen noch genügend Raum vorhanden ist.

Das System kann, so wie es ist, als Vorverarbeitungseinheit eingesetzt werden. Dabei sollte aber beachtet werden, dass es noch einige Verbesserungspunkte gibt. So ist z.B. die Qualität des Ausgabevideos noch nicht ganz optimal. Davon abgesehen erledigt das System stabil und zuverlässig seinen Dienst.

Eine andere Möglichkeit ist es das System als Framework zu nutzen. Durch die modulare Architektur ist es problemlos möglich den kompletten Algorithmus auszutauschen. So können andere Algorithmen für diese Hardware entwickelt und eingesetzt werden, ohne sich um den Rest der Hardware Gedanken machen zu müssen.

Ausblick

7

In den folgenden Abschnitten werden die möglichen Erweiterungen und Verbesserungen vorgestellt, die am bestehenden System noch vorgenommen werden können.

7.1 Optimierung des verwendeten Algorithmus

Da bei der Implementierung des Algorithmus sehr viele Abstriche gemacht wurden, um eine Grundfunktionalität herzustellen, stehen für die Optimierung des Algorithmus eine Menge an Optionen bereit.

So könnte man statt die volle Auflösung des Ausgabevideoformats zu nutzen die Berechnung des Ausgabebildes auf einen Teil der Gesamtauflösung beschränken und somit ein Breitbild erzeugen, wie dies z.B. bei der Berechnung des Panoramabildes in Abbildung 1.4 gemacht wurde. Würde man die Auflösung halbieren, dann könnte man damit die Geschwindigkeit bei der Umrechnung verdoppeln. Allerdings müsste dann dafür gesorgt werden, dass der Rest des Ausgabebildes zumindest einmalig gelöscht bzw. schwarz überschrieben wird. Dies könnte z.B. direkt beim Start des Systems vom Speicherscheduler erledigt werden.

Ein weiterer Verbesserungspunkt ist die Umrechnung an sich. Hier wurden sehr viele Berechnungsschritte mit ganzen Zahlen vorgenommen, wodurch die Genauigkeit reduziert wurde. Würde man alle Berechnungsschritte komplett mit rationalen Zahlen erledigen, so könnte das Ergebnis noch verfeinert werden. Auch der Einsatz einer Interpolation wäre denkbar. Dafür aber müsste der Zugriff auf den Speicher verbessert werden werden. Darauf wird weiter unten in diesem Kapitel noch näher eingegangen.

Eine sehr komplexe Möglichkeit die Umrechnung zu beschleunigen wäre der Einsatz von „Threading“, also mehreren nebenläufigen Prozessen, welche an der Umrechnung beteiligt sind. Dies ist bei diesem Algorithmus sehr gut möglich, da die Berechnung eines Zielpixels komplett unabhängig von der Berechnung der anderen Pixel ist. Geht man aber von der geschätzten Umrechnungsgeschwindigkeit von

2 Bildern pro Sekunde aus und bedenkt, dass man die Ausgabegeschwindigkeit mit einer Halbierung der Auflösung leicht auf das Doppelte erhöhen kann und dass die maximale Ausgabegeschwindigkeit bei lediglich 3,75 Bildern pro Sekunde liegt (siehe Abschnitt 5.4.3), dann wird diese Möglichkeit erst dann interessant, wenn man wirklich die volle Auflösung zur Ausgabe nutzen möchte.

7.2 Einsatz von anderen Algorithmen

Die Verarbeitungseinheit wurde so entworfen dass auch andere Algorithmen problemlos eingesetzt werden können. So könnte z.B. ein anderer Algorithmus zur Umrechnung des omnidirektionalen Bildes ins Panoramabild eingesetzt werden, wie z.B. die Hyperboloidale Projektion (siehe [Zha07], Seite 559). Auch wäre es denkbar zusätzlich zu dem bereits verwendeten Algorithmus noch weitere Algorithmen zum Einsatz zu bringen. So könnte direkt auf dem errechneten Panoramabild ein Algorithmus zur Kantendetektion eingesetzt werden. Dazu wäre es evtl. notwendig den Speicher-Scheduler ein wenig zu verändern, so dass die Verarbeitungseinheit auch aus dem bereits geschriebenen Bild lesen kann. Auch wäre es möglich die beiden bis jetzt brach liegenden Speicherbereiche (siehe Tabelle 5.4 in Abschnitt 5.7.4) als weitere Zwischenspeicher für die Verarbeitungseinheit nutzbar zu machen, so dass Algorithmen auf mehr als nur zwei Bildern arbeiten könnten.

7.3 Verbesserungen am Speicher-Scheduler

Die jetzige Speicheranbindung nutzt lediglich die einfachen Lese- und Schreibzugriffe auf den Speicher. Hier könnte man durch die Verwendung der Burst-Modi eine Menge an Zeit gewinnen, die dann vor allem der Verarbeitungseinheit zugutekommen würde und für komplexere Algorithmen verwendet werden könnte. Damit die Burst-Modi eingesetzt werden können ist es erforderlich, dass die FIFOs die Anzahl der in ihnen gespeicherten Elemente ausgeben. Diese Funktion der mit Quartus generierten FIFOs hat aber leider nicht funktioniert. Somit wäre dann auch dort noch Nachbesserung nötig.

Ein weiterer Verbesserungspunkt am Speicher-Scheduler ist die Aufteilung des Speichers und der Zugriff darauf durch die Verarbeitungseinheit. Die momentane Aufteilung des Speichers geht sehr verschwenderisch mit der verfügbaren Speicherkapazität um. Effektiv werden momentan lediglich 15 MiB von den verfügbaren 64 MiB genutzt. Würde man den Speicher anders aufteilen liessen sich deutlich mehr Bilder im Speicher unterbringen.

Diese könnte man dafür verwenden mehrere Eingangsbilder zu speichern, oder als weitere Zwischenspeicher für die Verarbeitungseinheit. So liessen sich auch Algorithmen einsetzen, die Zugriff auf mehrere aufeinander folgende Bilder benötigen, um ihren Dienst zu verrichten.

7.4 Verbesserungen an der Verarbeitungseinheit

Die momentane Umsetzung der Verarbeitungseinheit arbeitet beim Lesen und beim Schreiben immer direkt auf dem Speicher. Dies produziert zum einen Wartezeiten und zum anderen unnötigen Datenverkehr.

Während die Verarbeitungseinheit versucht Daten in den Speicher zu schreiben findet keine weitere Berechnung statt. Ausserdem hat das Speichern der Daten im Speicher-Scheduler die niedrigste Priorität. Somit kann es durchaus viele Takte dauern, bis die Verarbeitungseinheit mit dem Abarbeiten des Algorithmus fortfährt. Hier könnte der Einsatz eines FIFOs für das Schreiben die Verarbeitung deutlich beschleunigen.

Beim Lesen aus dem Speicher hingegen werden oft die gleichen Speicherbereiche mehrfach geladen. Auch beim momentan eingesetzten Algorithmus passiert das sehr oft. Zum einen, weil er nicht linear aus dem Ursprungsbild liest und zum anderen, weil einzelne Pixel verarbeitet werden, für die aber jeweils ein ganzer Speicherbereich geladen werden muss. Hier würde ein Cache viele Speicherzugriffe überflüssig machen.

7.5 Verbesserungen an der firewireIO

Wie man Abschnitt 5.4.3 entnehmen kann, wurde die Ausgangsbildrate aufgrund diverser Umstände fest auf 1,875 Bilder pro Sekunde eingestellt. Technisch möglich wäre aber auch die Ausgabe mit einer Bildrate von 3,75 Bildern pro Sekunde. Um dies umzusetzen müssten allerdings diverse Änderungen an der „firewireIO“ vorgenommen werden. Ausserdem müsste durch Tests sichergestellt werden, dass dies dann auch stabil und zuverlässig funktioniert.

Eine weiterer Verbesserungspunkt an der „firewireIO“ betrifft die asynchrone Kommunikation mit dem Rest des Firewire-Busses. Es wäre von Vorteil, wenn das System sich korrekt als Firewire-Kamera auf dem Bus anmelden würde. So könnte man für den Empfang, die Aufzeichnung und die Weiterverarbeitung des ausgegeben Videos auf Standard-Software zurückgreifen und könnte somit aus einem grossen Pool an Anwendungen schöpfen.

7.6 Statusanzeige und Interaktion

Die bisherige Statusausgabe der meisten Einheiten gibt lediglich anhand einfacher Zähler an, ob eine Einheit läuft oder ob es zu einem Fehler gekommen ist. Hier könnte man deutlich weiter gehen und die Ausgabe der verschiedenen Einheiten soweit verbessern, dass sie wesentlich mehr Informationen preisgeben. So ist es z.B. denkbar, dass die Verarbeitungseinheit angibt, wie viele Bilder pro Sekunde sie verarbeitet hat anstatt diese lediglich aufsummiert anzugeben. Die gleiche Angabe könnte auch der Eingangs-FIFO anzeigen, so dass man sehen kann welche Bildrate das Eingangsvideo hat.

Der Ausbau der Ausgabemöglichkeiten würde aber vor allem in Verbund mit Interaktionsmöglichkeiten Sinn ergeben. So ist es denkbar, dass man die „errorAdministration“ soweit ausbaut, dass sie Eingaben des Benutzers über die Knöpfe der FPGA-Platine an die verschiedenen Einheiten weiterreicht und dadurch diverse Einstellmöglichkeiten zur Laufzeit ermöglicht werden. So könnte man in der „firewireIO“ die Ein- und Ausgabe-Kanäle umstellen oder die Ausgabebildrate einstellen, insoweit die verschiedenen Ausgabebildraten implementiert sind. Auch die Verarbeitungseinheit könnte auf Benutzereingaben reagieren und so z.B. zwischen verschiedenen Algorithmen umschalten, die Berechnung pausieren oder Einstellungen für den Algorithmus vornehmen. So wäre es denkbar, dass man den Startwinkel bei der Berechnung des Panoramabildes verändert und so das Ausgabebild seitlich verschiebt.

Literaturverzeichnis

[Accowle] Accowle
http://www.accowle.com/english/index.html
Link überprüft am 17.12.2008

[Alt03] Altera Corporation
Nios Embedded Processor Development Board Data Sheet
Ver. 2.2, Juli 2003

[Alt04] Altera Corporation
APEX 20K Programmable Logic Device Family Data Sheet
Ver. 5.1, März 2004

[And99] Don Anderson
FireWire System Architecture
Second Edition, Addison-Wesley 1999

[DCS98] 1394 Trade Association
1394-based Digital Camera Specification
Version 1.20, Juli 1998

[EDA06] EDA Industry Working Groups
http://www.eda-stds.org/vhdl-200x/vhdl-200x-ft/packages_old/files.html
Link überprüft am 17.12.2008

[Fujinon] Fujinon
http://www.fujinon.com/
Link überprüft am 17.12.2008

[LWS94] Gunther Lehmann, Bernhard Wunder, Manfred Selz
Schaltungsdesign mit VHDL
März 1994
http://www.itiv.uni-karlsruhe.de/opencms/opencms/de/study/vhdl/book/download.html
Link überprüft am 17.12.2008

[Mic02] Micron Technology, Inc.
64Mb: x4, x8, x16 SDRAM
2002

[Mic04] Micron Technology, Inc.
64MB, 128MB, 256MB (x64, DR) 144-PIN SDRAM SO-DIMM
2004

[MicRC] Microsoft Research
RingCam
http://research.microsoft.com/~rcutler/ringcam/ringcam.htm
Link überprüft am 10.09.2008

[Sch01] Prof. Dr. F. Schubert
VHDL-Syntax
Mai 2001
http://users.etech.haw-hamburg.de/users/schubert/VHDLsynt.pdf
Link überprüft am 17.12.2008

[Son01] Sony
DFW-SX900/DFW-X700 Technical Manual
Ver. 1.0, 2001

[TAMS] Universität Hamburg, Department Informatik, Arbeitsbereichs TAMS
http://tams-www.informatik.uni-hamburg.de/index.php
Link überprüft am 17.12.2008

[TSB12] Texas Instruments
TSB12LV01B IEEE 1394-1995 High-Speed Serial-Bus Link-Layer Controller Data Manual
Mai 2002

[TSB41] Texas Instruments
TSB41AB3 IEEE 1394a-2000 THREE-PORT CABLE TRANSCEIVER/ARBITER
Oktober 2003

[WikDCS] Wikipedia
DCAM
http://de.wikipedia.org/wiki/DCAM
Link überprüft am 17.12.2008

[WikFW] Wikipedia
FireWire
http://de.wikipedia.org/wiki/FireWire
Link überprüft am 17.12.2008

[WikPol] Wikipedia
Polarkoordinaten
http://de.wikipedia.org/wiki/Polarkoordinaten
Link überprüft am 17.12.2008

[WikYUV] Wikipedia
YUV-Farbmodell
http://de.wikipedia.org/wiki/YUV-Farbmodell
Link überprüft am 17.12.2008

[Zha07] Jianwei Zhang
Intelligente Roboter
Vorlesung im Wintersemester 2007/2008
http://tams-www.informatik.uni-hamburg.de/lectures/2007ws/vorlesung/ir/doc/irWS07-2Eb.pdf
Link überprüft am 17.12.2008

Danksagungen

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mir die Erstellung der vorliegenden Arbeit ermöglichten.

Herrn Prof. Dr. J. Zhang danke ich für das, durch die Vergabe dieser Arbeit, entgegengebrachte Vertrauen und zur Verfügung gestellte Arbeitsmittel.

Ein grosser Dank geht an meinen Betreuer Dr. Andreas Mäder. Danke dafür, dass ich jederzeit an deine Tür klopfen konnte und Hilfe bekam. Ich weiss, dass das nicht selbstverständlich ist.

Mein Dank gilt auch dem gesamten TAMS-Arbeitsbereich. Danke für das nette Klima, das bei euch auf dem Flur herrscht.

Carsten Esslinger und Daniel Dorer möchte ich dafür danken, dass sie sich damals mit mir durch das Projekt gekämpft haben, welches die Grundlage meiner Diplomarbeit wurde.

Volker Tell danke ich für das Korrekturlesen, gerade wo Hardware normalerweise nicht sein Bereich ist.

Denis Klimentjew danke ich für die Bereitstellung der schönen Titelseite, für die vielen netten und hilfreichen Gespräche und das ständige Motivieren.

Meiner Freundin Katharina Mösche danke ich für die viele Geduld, während dieses stressigen Abschnitts meines Lebens.

Schlussendlich möchte ich mich bei meinen Eltern Ursula und Günter Piehl bedanken, die mich während des gesamten Studiums finanziell unterstützt haben.

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

Hamburg, den 23. Dezember 2008

(Christian Piehl)