Studienarbeit

Realisierung eines Echtzeit-Video-Digitalisierers unter Verwendung einer Hardware-Beschreibungssprache (AHDL, VHDL) als FPGA Prototyp

Lars H. Hahn

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich TECH

November 1995

2 Einleitung

Das Ziel der Studienarbeit war die Realisierung eines Videodigitalisieres. Als Endergebnis sollte ein funktionstüchtiger Prototyp stehen, der es ermöglicht, Videobilder in Echtzeit zu digitalisieren. Es sollte ein Bild aus dem kontinuierlichen Strom eines Videosignals digitalisiert werden. Um die Aufgabe nicht zu aufwendig werden zu lassen, beschränkt sich diese Studienarbeit auf Schwarz-Weiß-Bilder. Die Auswertung der Farbinformationen würde, wie noch ausführlicher erläutert wird, erheblich mehr Hardwareaufwand erfordern.
Um ein Videobild korrekt zu digitalisieren, ist es nötig die Synchronisationsimpulse, die den Bild- bzw. Zeilenanfang markieren, richtig auszuwerten. Ist dies gelungen, so ist es kein Problem mehr, die eigentliche Bildinformation in ein digitales Bild zu überführen.
Die Besonderheit dieses Projektes liegt darin, daß die Lösung vollkommen digital sein soll, d.h. es sollten keine analogen Komponenten verwendet werden (bis auf den Analog-Digital-Wandler). In handelsüblichen Produkten wird das Videosignal zunächst „analog aufbereitet“. Die Abtrennung der Synchronisationsimpulse (siehe 3.1.1) erfolgt dort mit Hilfe analoger Komponenten zur Signalverarbeitung.
Der Videodigitalisierer sollte in Form eines synthesefähigen Moduls einer Hardwarebeschreibungssprache[1] vorliegen, um in andere Projekte integrierbar zu sein. Die Beschreibung darf also nicht nur das Verhalten des Digitalisiers charakterisieren, sondern aus ihr muß sich auch direkt Logik synthetisieren lassen.
Im Rahmen dieser Studienarbeit beschränkt sich die Weiterverarbeitung des digitalisierten Videobildes auf die Übertragung zum PC, um es dort mit Hilfe von geeigneten Programmen betrachten und bearbeiten zu können.

[1] engl.: Hardware Description Language (HDL)

3 Grundlagen und Vorüberlegungen

3.1 Videosignal

Wichtigste Voraussetzung zum Bau eines Videodigitalisieres ist die Information über den Aufbau des Videosignals. Zur Übertragung von Videobildern gebräuchlich ist das FBAS-Signal (Farb-Bild-Austastsynchron-Signal). Das Videosignal enthält drei verschiedene Informationen: Synchronisationsimpulse (SYNCs), die Farbinformation (Chrominanz) und die Helligkeitsinformation (Luminanz). Die Bilder werden mit einer Frequenz von 50 Hz übertragen (PAL-Norm). Jedes Bild besteht aus 625 Zeilen, womit sich eine Zeilenfrequenz von 15,625 kHz ergibt. Sichtbar sind allerdings nur 575 Zeilen, die übrigen Zeilen werden während der vertikalen Synchronisation zum Rücklauf des Elektronenstahls des Monitors (Fernsehers) benötigt. Eine Zeile hat eine Länge von 64 µs (vgl. **[KRI93]**). Die Struktur des Videosignals ist in Abbildung 3-1 dargestellt.

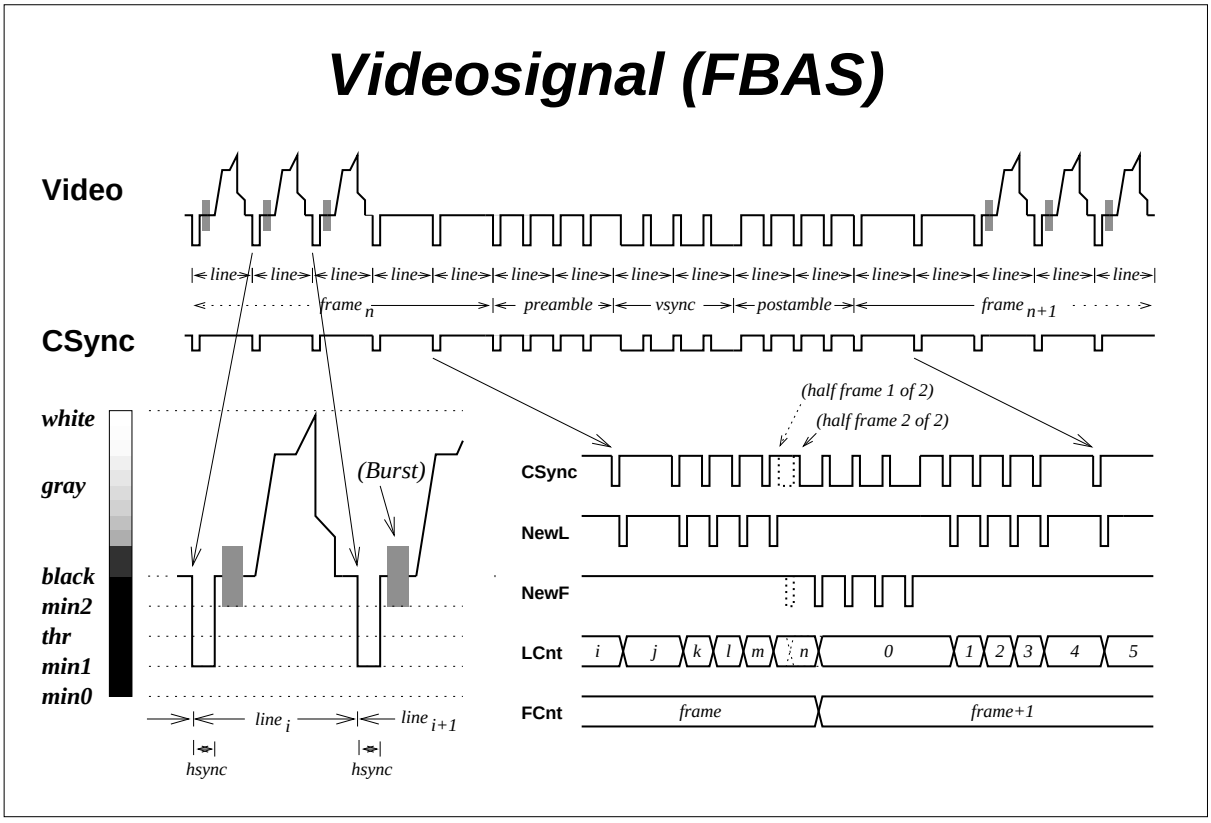

Abbildung 3-1

3.1.1 Synchronisationsimpulse (SYNCs)

Die SYNCs erkennt man daran, daß die Amplitude des Videosignals zum Zeitpunkt des SYNCs minimal im Verlauf des Signals ist. Es werden zwei Arten von SYNCs unterschieden. Dies ist zum einen der Zeilensynchronimpuls (H-Impuls) oder auch Horizontal-Sync (HSYNC) genannt, der den Zeilenbeginn markiert, und zum anderen der Vertikal-Sync (VSYNC) für den Bildanfang. Die beiden Synchronisationsimpulse unterscheiden sich durch ihre Länge und durch ihren Abstand. So dauert der HSYNC ca. 5 µs. Es gibt einen pro Zeile, also alle 64 µs einen HSYNC. Der VSYNC dauert ca. 17 µs an. Der Abstand zwischen zwei VSYNCs beträgt 32 µs. Der Zeitraum, in dem die VSYNC-Impulse andauern, hat insgesamt eine Länge von 160 µs.

3.1.2 Farbinformationen (Chrominanz)

Nach jedem HSYNC folgt der sogenannte *Burst*. Seine Frequenz beträgt im Fall der PAL-Norm etwa 4,43 MHz. Er dauert neun bis elf Perioden an. Der Burst dient zur Synchronisation der Farbinformationen. Da im Rahmen dieser Studienarbeit nur Schwarz-Weiß-Bilder digitalisiert werden, kann der Burst einfach ignoriert werden. Im endgültigen Versuchsaufbau wurde eine Schwarz-Weiß-Kamera eingesetzt. Sie liefert ein Videosignal ohne *Burst* und Chrominanz-Signal.

3.1.3 Helligkeitsinformationen (Luminanz)

Im Anschluß an den *Burst* folgen die die Bildinformation enthaltenden Videodaten. Die Signalamplitude entspricht der Helligkeit des Bildes. Je größer die Amplitude des Signals, desto heller ist der Bildpunkt. Das digitalisierte Signal in diesem Bereich repräsentiert direkt die Grauwerte für die Bildpunkte dieser Zeile. Um ein Schwarz-Weiß-Bild zu erhalten, können die Werte unmittelbar ohne weitere Bearbeitung in den Speicher geschrieben werden. Falls eine Farbkamera benutzt wird kann das aufmodulierte Farbsignal einfach ignoriert werden, da die damit verbundenen Störungen minimal sind.

3.2 Lösungsansatz

Um einen funktionstüchtigen Aufbau zu erhalten werden verschiedene Komponenten benötigt. Der im Rahmen dieser Studienarbeit fertiggestellte Prototyp besteht aus vier wesentlichen

Teilen: Analog-Digital-Wandler, Speicher, PC-Interface und einem Chip, der die Steuer-Logik aufnimmt.

Als Chip für die Aufnahme der Logik wurde ein FPGA[2] gewählt. FPGAs eignen sich besonders gut für Prototypen, da sie vor Ort (fast) beliebig oft programmierbar sind. Es entfällt somit die Wartezeit für die Chipfertigung und ein sofortiges Testen ist in der Systemumgebung möglich. Es ist somit möglich, nachdem man Änderungen im Design vorgenommen hat, die Auswirkungen sofort zu beobachten.

Um Videobilder in Echtzeit zu digitalisieren, sind etwa acht Millionen Samples pro Sekunde nötig. Dies entspricht 512 Bildpunkten pro Zeile, da eine Bildzeile 64 µs lang ist. Diese hohe Datenrate hat zur Folge, daß ein Speicher (RAM) benötigt wird, in dem die Bilder für die (langsame) Übertragung zum PC zwischengespeichert werden können. Für einen ersten Testaufbau reichen wenige KByte Speicher aus, um zunächst nur einige Bildzeilen zu digitalisieren. Für die endgültige Realisierung ist eine Speichergröße von einem MByte in jedem Fall ausreichend, da so Bilder bis zu einer Größe von 1024 mal 1024 Bildpunkten gespeichert werden können.

Als weitere Komponente wird ein Analog-Digital-Wandler benötigt, der das analoge Videosignal in einen digitalen Datenstrom umsetzt. Der AD-Wandler muß schnell arbeiten, um ausreichend Bildpunkte pro Zeile aufnehmen zu können. Die untere Grenze liegt aufgrund des Abtasttheorems bei der doppelten Videobandbreite also bei 8 bis 10 MHz. Bei 8 MHz werden 512 Bildpunkte pro Zeile aufgenommen. Um 1024 Bildpunkte, die vom RAM vorgegebene obere Grenze, zu erreichen, ist somit eine Samplingfrequenz von 16 MHz nötig. Um diese hohe Samplingrate zu erreichen, wurde als AD-Wandler ein FADC[3] eingesetzt. Dieser läßt sich problemlos mit 16 MHz betreiben. Probleme hinsichtlich einer hohen Taktrate bereiten vielmehr die FPGAs. Um Daten im DRAM abzulegen, ist ein Zugriffszyklus von sechs Takten nötig (siehe 4.4.1 DRAM Timing). Während der FADC einen Wert digitalisiert, müssen im FPGA für den DRAM-Zugriff sechs Takte ablaufen. Das entsprechende FPGA muß somit mit der sechsfachen Frequenz getaktet werden, also mit 96 MHz.

Letztendlich ist ein Interface für die Verbindung zum PC nötig. Dies muß keine besonderen Bedingungen hinsichtlich der Übertragungsgeschwindigkeit erfüllen, da das Videobild ja im RAM zwischengespeichert wird. Sinnvoll wäre es, daß ein Byte breite Daten parallel übertragen werden können, da ein serielles Übertragungsprotokoll mit deutlich mehr Aufwand

[2] Field Programmable Gate Array (FPGA)
[3] Flash Analog Digital Converter (FADC)

verbunden wäre. Ebenso wünschenswert ist eine einfache Programmierung des Interfaces auf dem PC.

4 Realisierung

4.1 Hardware-Komponenten

Im Folgenden werden die Hardware-Komponenten, die in den unterschiedlichen Aufbauten Verwendung fanden, näher beschrieben. Dies sind im einzelnen der FADC (4.1.1), die FPGAs (4.1.2), das RAM (4.1.3) und das Interface zum PC (4.1.4).

4.1.1 FADC

Zur Digitalisierung des Videosignal wird ein FADC vom Typ MC10319 der Firma Motorola verwendet. Der MC10319 ist ein parallel arbeitender 8-Bit Analog-Digital-Wandler. Er enthält 256 Komperatoren, die das Eingangssignal mit einer Referenzspannung vergleichen. Die Komperatoren sind so dimensioniert, daß die Referenzspannung von einem zum nächst größeren Komperator immer um den gleichen Betrag zunimmt. Die Ausgänge der Komperatoren werden mit Hilfe eines internen Schaltnetzes in die 8-Bit Zahlendarstellung gewandelt. Der Baustein benötigt als Versorgungsspannung +5 Volt und -5 Volt. Die maximale Samplingfrequenz, mit der der MC10319 betrieben werden kann, beträgt 25 MHz (vgl. **[MOT84]**).

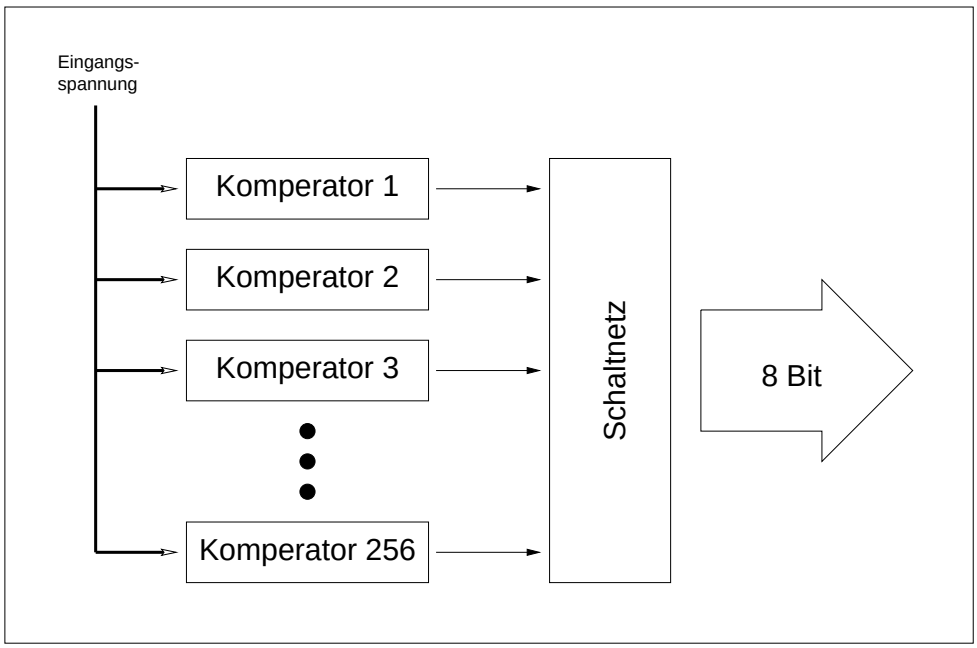

Abbildung 4-1

4.1.2 FPGA

Ein FPGA (Field Programmable Gate Array) ist ein programmierbarer digitaler Baustein. Er ist intern in die sogenannten Logikzellen unterteilt, die in einer zweidimensionalen Matrix angeordnet sind. Zwischen den Logikzellen gibt es Verbindungen, die vom Benutzer programmierbar sind. Um logische Schaltungen realisieren zu können, muß diese auf die Logikzellen aufgeteilt werden. Die erforderlichen Verbindungen müssen passend programmiert werden.

Altera-EPM7128

Um die Prototypen aufzubauen, wurden zwei FPGAs vom Typ EPM7128 der Altera MAX7000-Serie verwendet. Sie enthalten jeweils 128 Logikzellen und verfügen über 84 I/O-Pins. Jeweils 16 Logikzellen sind zu einer strukturellen Einheit zusammengefaßt, die als Logic Array Block (LAB) bezeichnet wird. Abbildung 4-2 zeigt die Struktur eines LABs. In einem EPM7128 existieren somit 8 LABs. (vgl. **[KLI93]**)

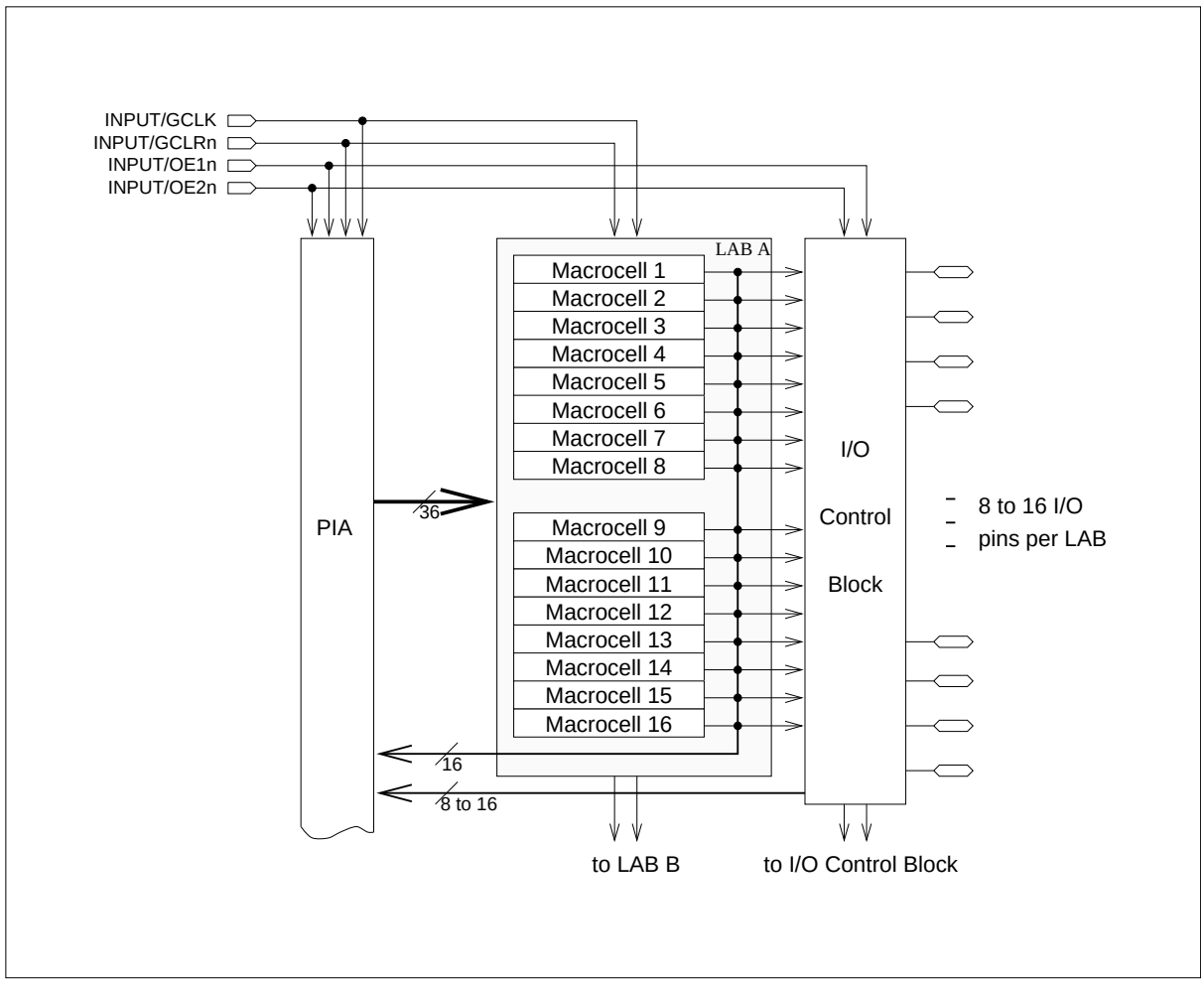

Abbildung 4-2

Die Programmierung des FPGAs erfolgte mit Hilfe einer Hardwarebeschreibungssprache. Mit ihr wird das Verhalten einer Schaltung beschrieben, keineswegs also die genaue Struktur der Logik auf Gatterebene. Die Umsetzung der Verhaltensbeschreibung in Logikfunktionen nimmt ein Design-Compiler vor. Er nimmt dem Anwender reine Fleißarbeiten, wie etwa die Minimierung von Schaltnetzen und Schaltwerken, Laufzeitoptimierungen u.ä. ab. Die Aufgabe des Entwerfers verlagert sich dadurch von der Gatter- auf die Register-Transfer-Ebene. Die Abbildung der so synthetisierten Logik auf einen vorhanden FPGA-Baustein, wird ebenfalls von einem Compiler erledigt.

4.1.3 RAM

Das Funktionsprinzip Dynamischer Speicher (DRAMs) ist einfach: Das Speicherelement besteht im Kern aus einem Kondensator, der für die beiden Zustände eines Bits die Darstellungsmöglichkeiten „aufgeladen“ und „entladen“ bietet. Dieses Prinzip birgt zugleich den größten Vorteil wie den größten Nachteil von DRAMs. Der Vorteil liegt darin, daß eine dynamische Speicherzelle aus erheblich weniger „Bauteilen“ besteht als eine statische Speicherzelle. Gegenüber statischen Bauteilen vergleichbarer Integrationsdichte bringen dynamische etwa viermal so viele Bits unter. Der Nachteil der DRAMs: Kein Dielektrikum isoliert hundertprozentig, außer vielleicht bei Temperaturen in der Nähe von Null Kelvin (absoluter Nullpunkt). Da man im Inneren eines RAM-ICs davon deutlich entfernt ist, wird die Speicherkapazität also mit der Zeit ihre Ladung und damit ihren Informationsgehalt verlieren, sofern nicht geeignete Gegenmaßnahmen getroffen werden. Jene Gegenmaßnahmen sind als REFRESH bekannt. Der REFRESH ist eine Aufgabe der externen RAM-Steuerung. Dies bedeutet, daß zusätzliche Logik im FPGA nötig wird, die bei Benutzung von SRAMs nicht notwendig ist. (vgl. **[ASS90]**)

4.1.4 PC-Interface

Als Verbindungsglied zwischen Prototyp und PC wurde eine universelle I/O-Karte gewählt. Diese verfügt über drei PIA-Bausteine PPI[4] 8255. Jeder dieser Bausteine ermöglichen die Ein- bzw. Ausgabe von 24 Bit (3 x 8 Bit).

Der 8255 ist ein programmierbarer Mehrzweck-E/A-Baustein für die Mikroprozessoren der Intel 80x86 Familie. Der Baustein hat 24 E/A-Anschlüsse, die in zwei Gruppen von je zwölf

[4] programmierbarer peripherer Interface-Baustein (PPI)

Anschlüssen getrennt programmiert und im wesentlichen in drei Betriebsarten benutzt werden können. (vgl. **[KOL90]**)

4.2 Software-Komponenten

4.2.1 VHDL vs. AHDL

Der an der Universität verfügbare Compiler des Altera MAX+plusII Systems ist in der Lage, zwei unterschiedliche Hardwarebeschreibungssprachen zu verarbeiten und in Logik umzusetzen. Die erste Alternative ist AHDL[5], eine vom Hersteller Altera speziell entwickelte Sprache. Die zweite Möglichkeit ist VHDL[6], die am weitesten verbreitete Hardwarebeschreibungssprache. Sie wurde 1983 vom amerikanischen Department of Defence initiiert und ist seit Ende 1987 als IEEE Standard 1076 genormt (vgl. **[MAE93]**).

AHDL kann zwei Vorteile auf sich vereinen. Zum einen besitzt sie einen einfachen Syntax, zum anderen ist die Sprache optimal auf die Altera-FPGAs abgestimmt. Mit ihr kann man das Verhalten der Hardware so beschreiben, daß eine Abbildung auf die vorhandenen Hardwarestrukturen direkt und besonders effizient möglich ist.

Demgegenüber ist VHDL praktisch die „Standardsprache“, um Hardware zu beschreiben. VHDL wird von vielen, sehr unterschiedlichen Systemen unterstützt. Dies hat eine hohe Portabilität der Sprache zur Folge. Ein Ergebnis, das als ASIC[7]-Komponente in anderen Projekten verwendbar sein soll, macht eine Beschreibung in VHDL fast zwingend erforderlich.

4.2.2 PC-Software

Für die Programme, die auf dem PC laufen, wurde die Sprache C / C++ gewählt. In C gestaltet sich die Programmierung des parallelen Interfaces (siehe 4.1.4 PC-Interface) besonders einfach, da dieses über Hardwareregister angesprochen wird.

4.3 Prototyp - Stufe 1

Der erste Aufbau dient der Prüfung der grundsätzlichen Machbarkeit. Er soll zunächst so einfach wie möglich sein, um grundsätzliche Probleme (insbesondere im Bereich des FADC) zu

[5] Altera Hardware Description Language (AHDL)
[6] Very High Speed Integrated Circuit HDL
[7] anwendungsspezifische intergrierte Schaltungen

eliminieren und um einen funktionsfähigen Ansatz zu erhalten. Somit bestand der erste Versuchsaufbau nur aus dem FADC, einem FPGA und 8K Static-RAM.

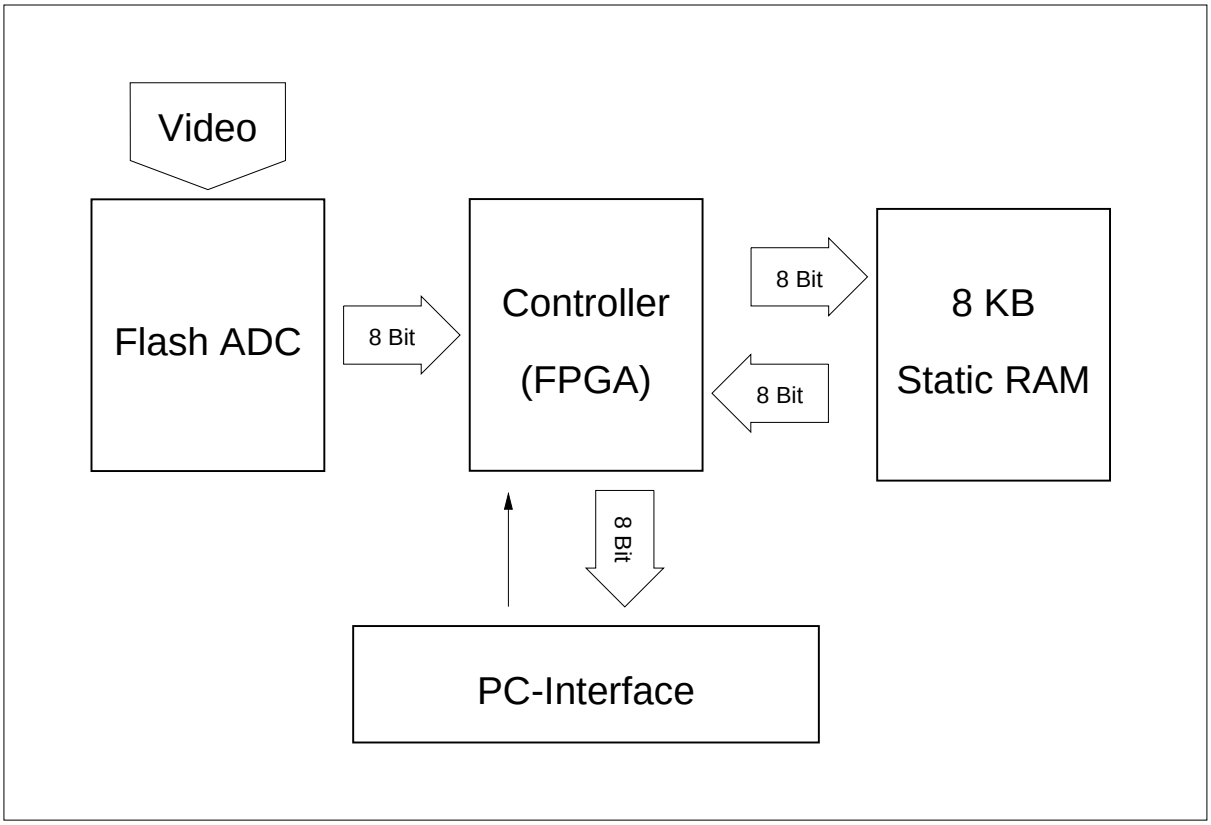

Abbildung 4-3

Für den ersten Versuch wurde bewußt ein kleiner statischer RAM-Baustein gewählt. Die Ansteuerung ist extrem einfach und die Größe des Speichers (8 KByte) reichte aus, um ein paar Bildzeilen aufzunehmen. Wie sich später herausstellte, war diese Entscheidung richtig, da die Ansteuerung der dynamischen RAMs doch erhebliche Probleme bereitete.
Das FPGA wurde für den ersten Prototypen mit Hilfe von AHDL programmiert. Die Wahl lag aufgrund der oben erwähnten Vorteile nahe, sollte doch schnell ein einfacher und funktionstüchtiger Aufbau realisiert werden.

4.3.1 Der erste Test

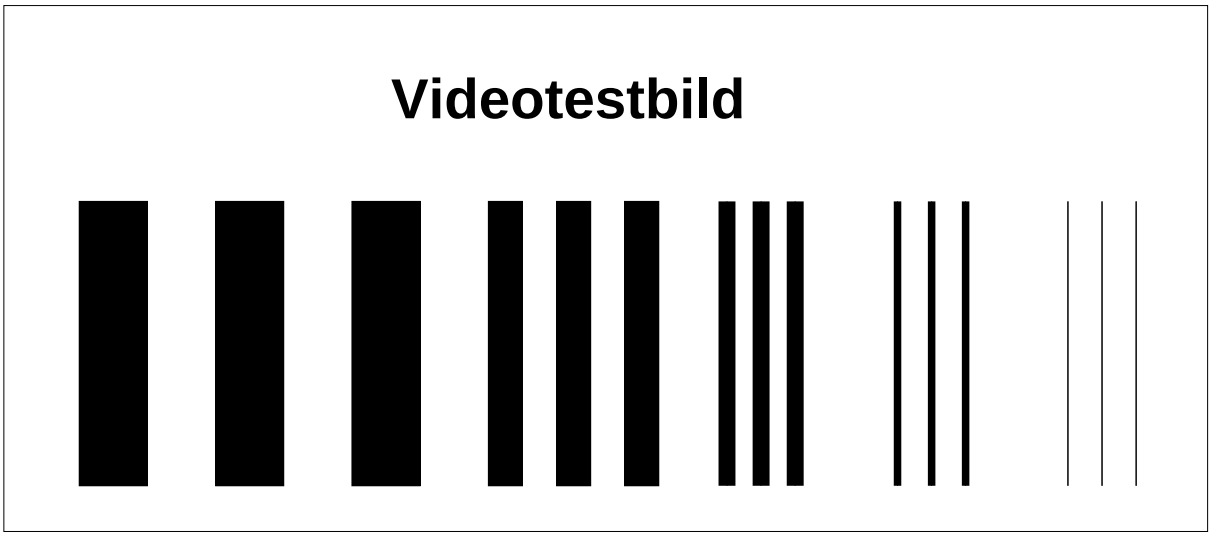

Abbildung 4-4

Um die korrekte Arbeitsweise des ersten Prototypen zu überprüfen, wurden zunächst nicht das digitalisierte Videosignal in das SRAM geschrieben, sondern die Ausgabe eines „Hilfszählers". Nachdem diese kontinuierliche Zahlenfolgen richtig beim PC angekommen war, folgte der Test mit „echten" Videodaten. Das Problem war ein Motiv zu finden, das auch in wenigen Bildzeilen zu erkennen war. Es wurde schließlich ein Bild entsprechend Abbildung 4-4 als Vorlage gewählt, indem jede Bildzeile identisch ist. Abbildung 4-5 zeigt das Ergebnis des ersten Versuchs. Es sind deutlich die unterschiedlich breiten Streifen der Vorlage zu erkennen.

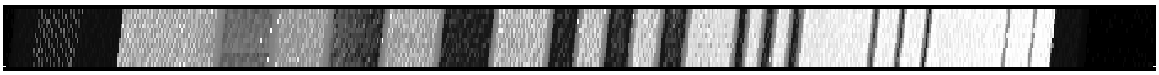

Abbildung 4-5

4.3.2 Datenübertragung zum PC

Die Verbindung zum PC stellt eine Interfacekarte auf der Basis von 8255 Bausteinen her (siehe auch 4.1.4 PC-Interface). Für die Übertragung der Bildinformation vom RAM zum PC wird eine sehr einfache Methode eingesetzt. Der PC setzt eine Ready-Leitung auf „1", wenn er bereit ist, den nächsten Wert zu empfangen. Es wird dabei vorausgesetzt, daß der Prototyp die Daten immer ausreichend schnell liefern kann. Diese Annahme kann ohne Bedenken gemacht werden, da die Daten des RAMs nach spätestens sechs Takten zur Verfügung stehen und die verwendeten Taktraten (10 - 32 MHz) ausreichend hoch sind im Vergleich zum Bustakt des PCs.

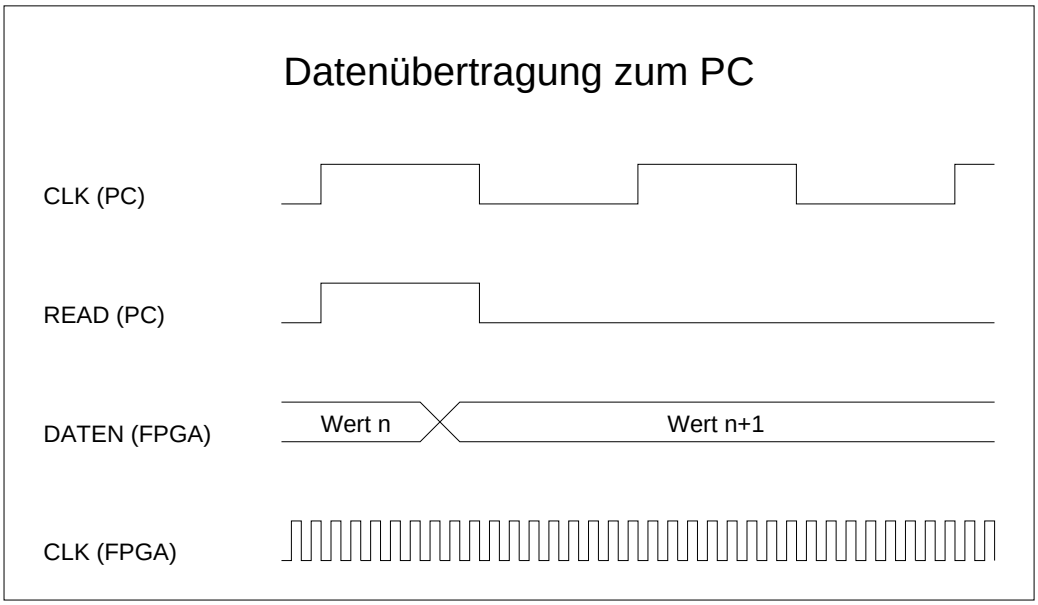

Abbildung 4-6

4.4 Prototyp - Stufe 2

Mit dem zweiten Aufbau wurde das Videosignal komplett mit allen Synchronisationsimpulsen digitalisiert. Auf diese Weise wurden „echte“ Daten in ausreichender Menge gewonnen, um mit ihrer Hilfe den endgültigen Prototypen erfolgreich simulieren zu können. Die gewonnenen Werte dienten der VHDL Simulationsumgebung der Prototyp-Stufe 3 zur Nachbildung des FADC. Die Abbildung 4-7 zeigt einen Abschnitt der so gewonnenen Daten.

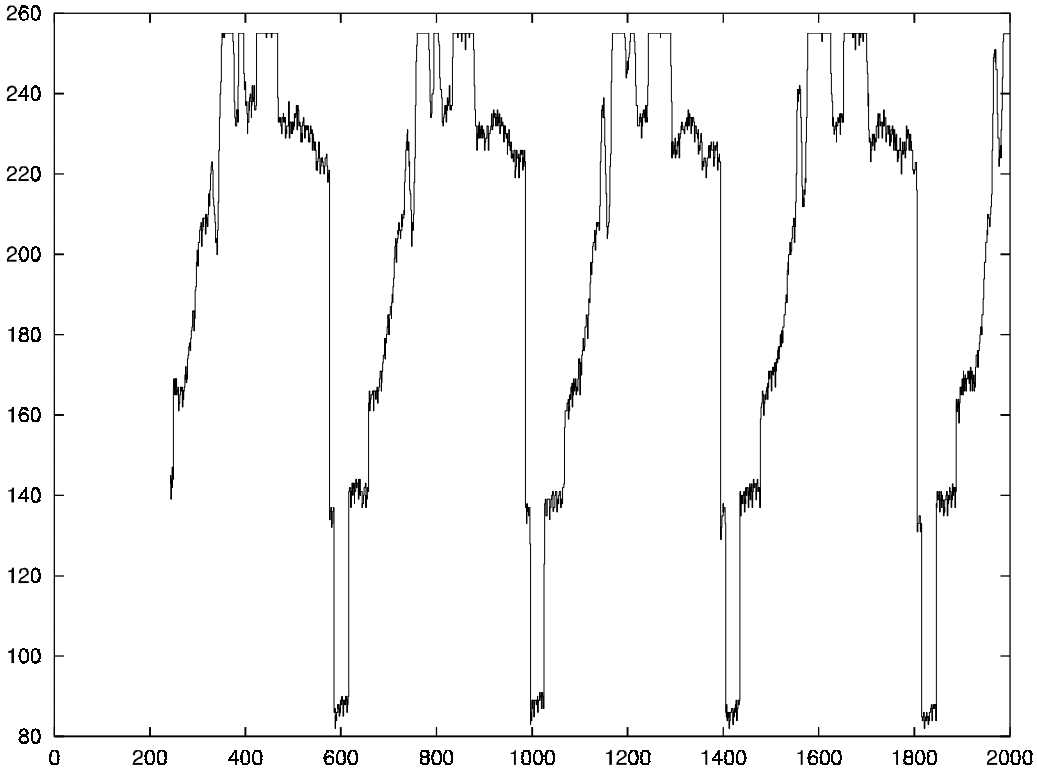

Abbildung 4-7

Weiterhin sollte das DRAM Timing in der Systemumgebung getestet werden (siehe 4.4.1 DRAM Timing). Benötigt wurden weiter der FADC, ein FPGA, dessen Funktionalität nun in VHDL beschrieben wurde, sowie jetzt ein 1 MB dynamisches RAM, um genügend Daten zwischenspeichern zu können.

Die Funktionsweise der Prototyp Stufe 2 ist ähnlich einfach wie die der Stufe 1. Wurden im Rahmen der Prototyp Stufe 1 8 KByte Daten in den Speicher geschrieben, konnten jetzt 1 MByte zwischengespeichert werden. Im Anschluß werden die Daten nach dem gleichen Verfahren wie bei Stufe 1 zum PC übertragen (siehe 4.3.2 Datenübertragung zum PC).

Die gewonnen Daten wurden zunächst mit einem C-Programm (siehe 5.2.1 PC-Syncfilter) bearbeitet, um den Algorithmus zur Isolierung der Horizontal- und Vertikal-Synchonisationsimpulse (HSYNCs und VSYNCs) zu überprüfen.

Nach erfolgreicher Programmierung in C, erfolgte im Rahmen der Prototyp Stufe 3 die Implementation in VHDL.

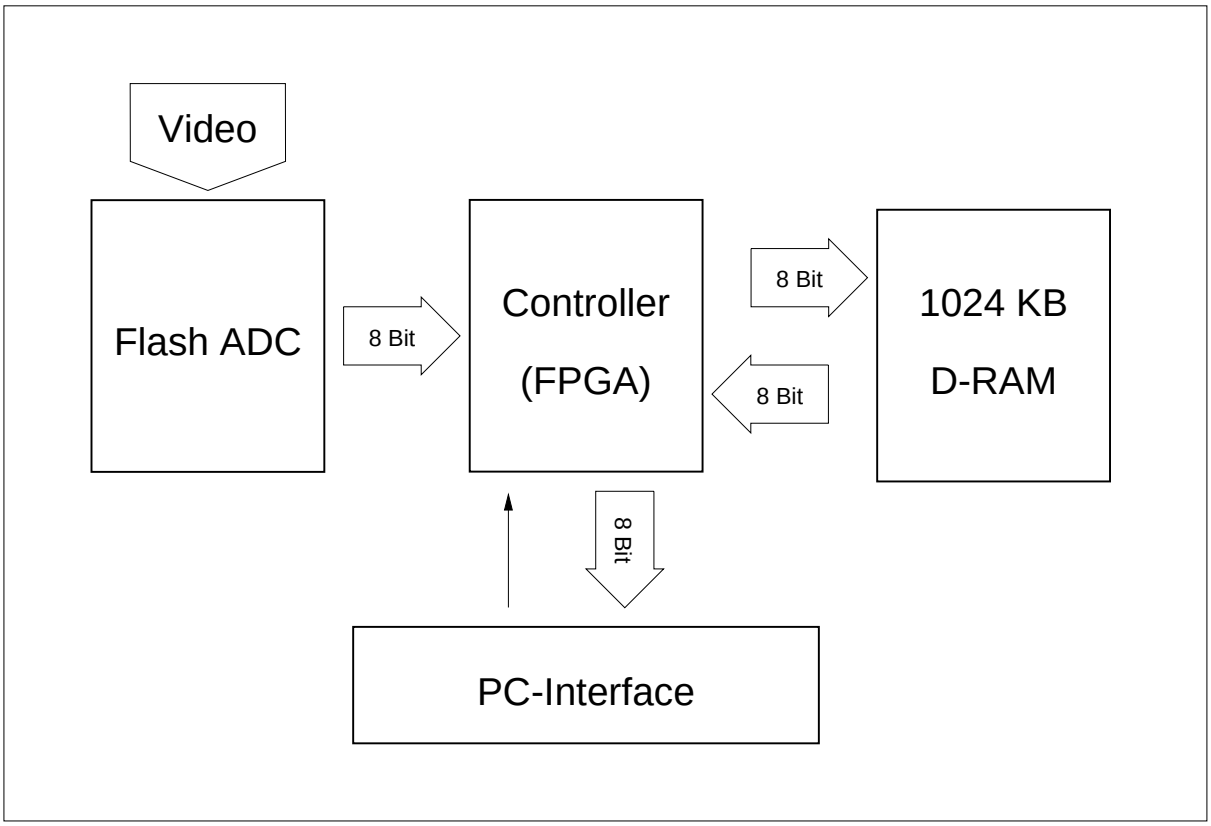

Abbildung 4-8

4.4.1 DRAM Timing

Die Ansteuerung eines SRAMs ist denkbar einfach: Es wird die Adresse angelegt und das Datum wird geschrieben oder gelesen. Die Ansteuerung des DRAMs hingegen muß nach genauen zeitlichen Spezifikationen erfolgen. Die Benutzung eines DRAMs ist somit ungleich komplizierter.

Die Adresse ist in eine Zeilen- (ROW) und Spalten-Adresse (COLUMN) aufgeteilt. Beide Adressenteile werden dem DRAM auf einem Bus multiplexartig zugeführt. Es ist somit nötig, zunächst die ROW- und dann die COLUMN-Adresse am Adreßbus anzulegen. Sobald die RAS[8]- und CAS[9]-Impulse anliegen, werden die Werte des Adreßbusses als gültig übernommen (siehe Abbildung 4-9). Weiterhin muß beim DRAM im Gegensatz zum SRAM für ein REFRESH gesorgt werden (siehe 4.1.3 RAM).

[8] Row Address Strobe
[9] Column Address Strobe

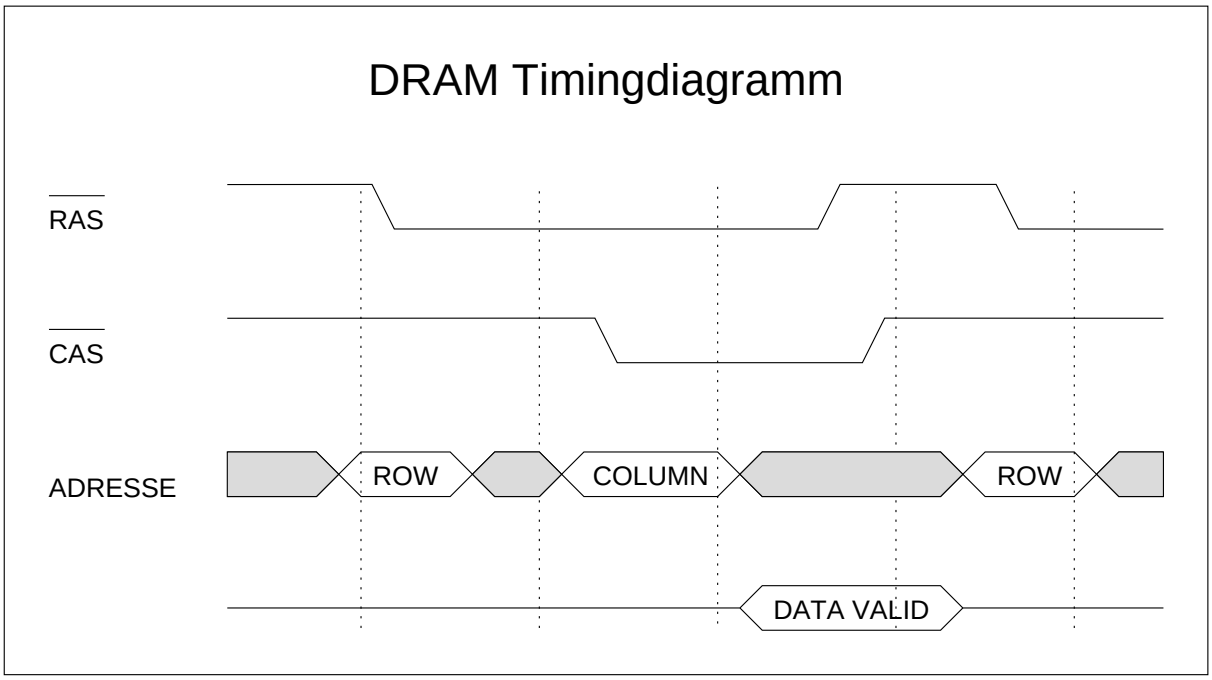

Abbildung 4-9

In der Tat bereitete die Ansteuerung des DRAMs erhebliche Probleme. Die Anzahl der möglichen Fehlerquellen hatte, wie oben beschrieben, stark zugenommen. Gab es bei der Prototyp Stufe 1 keine Probleme mit dem Beschreiben und Auslesen des statischen RAMs, so war beim dynamischen RAM zunächst keinerlei sinnvoller und nachvollziehbarer Zugriff möglich. Eine Fehlersuche mit Hilfe der Simulation war nur bedingt möglich, da die Simulationsumgebung keinesfalls perfekt war. Sie gab das Verhalten der DRAMs nur sehr oberflächlich wieder.

4.5 Prototyp - Stufe 3

Ziel ist ein funktionstüchtiges, eigenständiges System. Neu gegenüber der bisherigen Realisierungen ist die zuverlässige Erkennung von H- und VSYNCs. Dieses wurde bisher von einem C-Programm zu Testzwecken geleistet. Die direkte Umsetzung des C-Programms in VHDL war leider nicht möglich. Die Problematik wird in Abschnitt 5.2.1 (PC-Syncfilter) behandelt.

Aufgrund des gestiegenen Funktionsumfanges wurde ein zweites FPGA nötig, da die 128 Logikzellen eines Bausteins nicht mehr ausreichten, die gestellte Aufgabe zu lösen. Ebenfalls nötig wurde eine umfangreiche Simulationsphase, um die Korrektheit der SYNC Erkennung zu überprüfen.

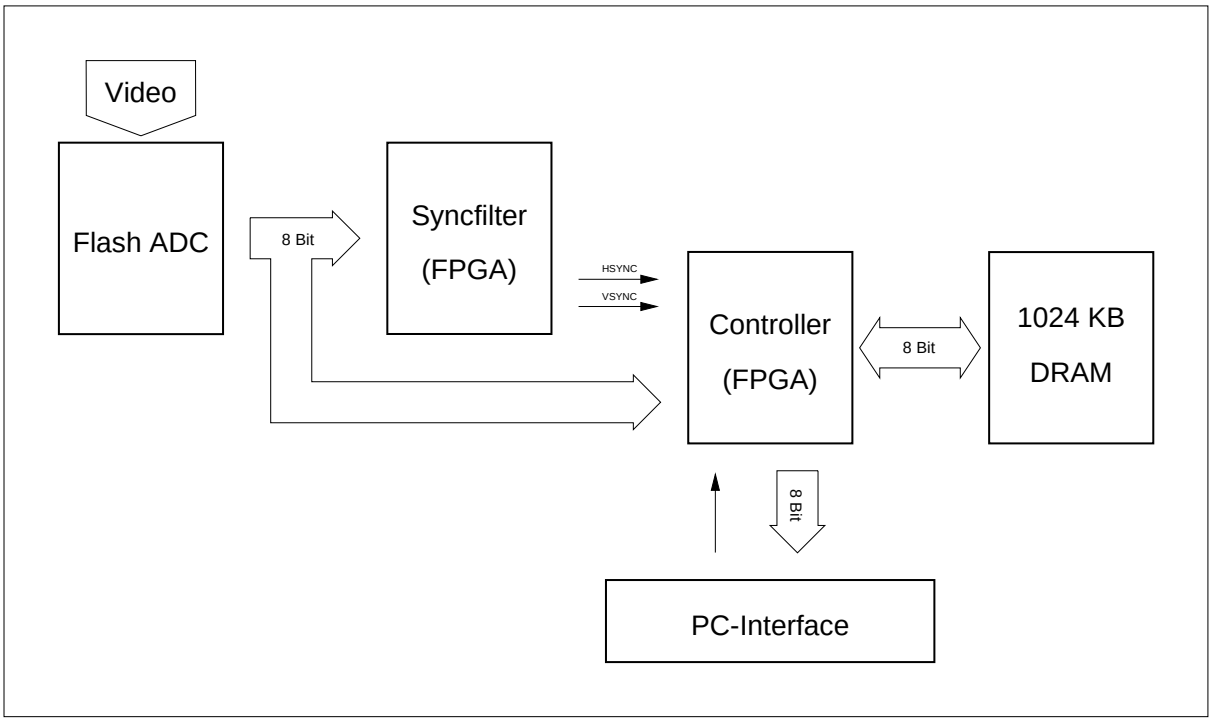

Abbildung 4-10

4.5.1 Erkennung der Synchronisationsimpulse

Der zentrale Teil des Digitalisieres ist der sogenannte „Syncfilter“. Er erkennt aus dem kontinuierlichen Datenstrom des digitalisierten Videosignals die H- und VSYNCs.

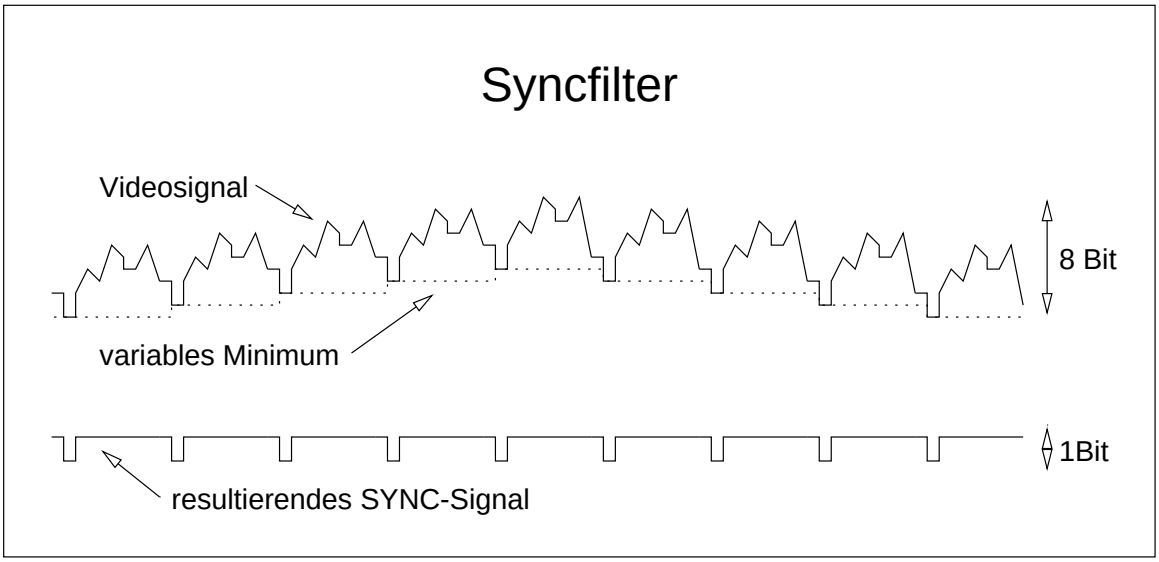

Abbildung 4-11

Ein wichtiger Teil des „Syncfilters“ ist ein Prozeß[10] der permanent den kleinsten Wert (Minimum) der digitalisierten Daten ermittelt. Das Minimum liegt nicht fest auf einem Wert,

[10] Ein Prozeß in VHDL ist ein Modul (Funktionseinheit). Alle Prozesse laufen parallel ab.

sondern ist aufgrund von Schwankungen im Videosignal Änderungen unterworfen. Es ist somit nötig, das Minimum immer wieder neu zu ermitteln und es dem Verlauf des Videosignals anzupassen. Das Minimum bezieht sich somit immer auf einen bestimmten Abschnitt des Videosignal und kann sich im Verlauf des Signals verändern.
Erreicht wird diese dynamische Ermittlung des Minimums dadurch, daß immer nach Ende eines HSYNCs das aktuelle Minimum um eins erhöht wird. Das korrigierte Minimum wird permanent mit dem aktuellem Wert des Datenstroms verglichen. Ist der Wert kleiner, wird das Minimum wieder erniedrigt. Ist der Wert größer oder gleich, dann war die leichte Erhöhung angebracht. Da sich das Minimum des Datenstroms nicht schlagartig ändert, führt dieses Verfahren zu sehr guten Ergebnissen.
Das so ermittelte aktuelle Minimum wird permanent vom derzeitigem Wert subtrahiert. Falls das Ergebnis der Subtraktion Null oder sehr klein ist, wird entschieden, daß ein SYNC vorliegt. Zwischen H- und VSYNC wird mit Hilfe der Länge des SYNCs unterschieden. Mit Eintreten der Vorderflanke des SYNCs startet ein Zähler, bei seiner Rückflanke wird aufgrund des Zählerstandes auf H- oder VSYNC geschlossen.

4.5.2 Aufteilung des Videosignals

Mit Hilfe der richtig erkannten H- und VSYNCs ist das Videobild einfach in seine unterschiedlichen Abschnitte zu unterteilen. Nach dem Initialisieren der Schaltung wird auf einen VSYNC (Bildanfang) gewartet. Sobald das Ende des VSYNCs erreicht ist, wird die erste Bildzeile digitalisiert. Mit jedem HSYNC (Zeilenbeginn) wird der Zeilenzähler um Eins erhöht. Der Zeilenzähler wird direkt zur Generierung der ROW-Adresse des DRAMs verwendet. Somit ist eine maximale Zeilenbreite von 1024 Punkten möglich.
Diese Methode ermöglicht die Speicherung der Bilddaten in einer einheitlichen Weise, unabhängig von der angelegten Taktfrequenz. Bei einer höheren Frequenz werden einfach entsprechend mehr Werte gespeichert. Die praktische Grenze bei diesem Prototypen liegt wie erwähnt bei 1024 Einträgen pro Zeile. Der vorliegende Aufbau läßt sich, aufgrund der relativ langsamen verfügbaren FPGAs, aber nicht einmal annähernd so hoch takten, daß die Grenze von 1024 Bildpunkten erreicht würde.
Das Digitalisieren wird mit dem Erkennen des nächsten VSYNCs beendet. Es folgt die schon beschriebene Übertragung der Daten zum PC (siehe 4.3.2 Datenübertragung zum PC). Während der Übertragung darf der kontinuierliche REFRESH des DRAMs nicht vernachlässigt werden, um keine Werte zu verlieren. Ist die Übertragung zum PC abgeschlossen, wird mit dem Digitalisieren des nächsten Bildes begonnen.

4.6 Simulation und Systemtest

Bei Schaltungssimulation wurden zwei Methoden angewandt, die es ermöglichen, Fehler, trotz der großen Datenmengen, schnell zu erkennen. Zum einen wird die Simulation mit den zuvor gewonnenen „realen“ Daten durchgeführt, zum anderen erfolgt die Ausgabe der Simulation (des Bildes) in einem X-Windows-Fenster.

Normalerweise wird die Simulation mit generierten, der Realität nachempfunden Werten durchgeführt. Zum Testen des Designs des Digitalisierers wurde ein anderer Weg gewählt. Es wurden mit Hilfe der Prototyp Stufe 2 echte Daten gewonnen und gespeichert, um sie dann als „reale“ Werte für die Simulation verwenden zu können.

Die zweite Besonderheit betrifft die Ausgabe der Simulation. In der Regel erfolgt diese mit Hilfe von „Waves“. Hierbei werden die Ausgangswerte durch Zahlenfolgen dargestellt. Bei den großen Datenmengen, die beim Videodigitalisieren anfallen, ist dies sehr unübersichtlich. Die Fehlersuche ist so nur unter großem Zeitaufwand möglich. Im Rahmen dieser Studienarbeit kam eine andere Methode zur Fehlersuche zu Einsatz. Mit Hilfe der Simulationsausgabe (H- und VSYNCs) wurde das digitalisierte Videobild direkt in ein X-Windows-Fenster ausgegeben. So war es möglich, einen Fehler quasi mit einem Blick zu erkennen.

Der Systemtest erfolgte unter Echtzeitbedingungen in einem eigenen Hardwareaufbau. Da die FPGAs vor Ort praktisch beliebig oft programmiert werden konnten, war ein Test in dem jeweiligen Aufbau direkt möglich. Somit konnte ein Fehler, der erst in der Systemumgebung, nicht aber in der Simulation, auftritt, sofort erkannt und schnell behoben werden.

5 Aufbau (Technische Beschreibung)

In den folgenden Abschnitten wird die Implementation des Digitalisierers genau beschrieben. Der VHDL- und C-Sourcecode ist in voller Länge abgedruckt und ausführlich kommentiert. Einen Überblick über das Zusammenspiel der Unterschiedlichen Komponenten der Hardware gibt Abbildung 5-1.

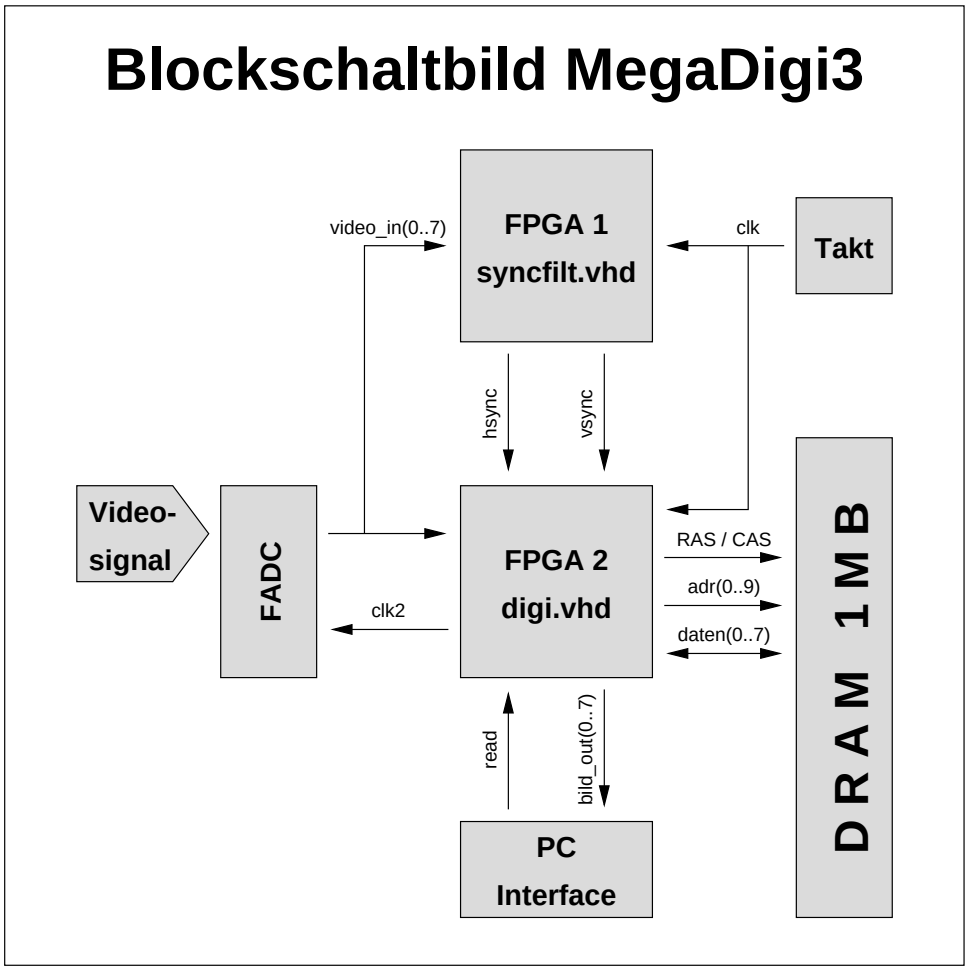

Abbildung 5-1

5.1 FPGA Logik

Das Verhalten der FPGA Logik der Prototyp Stufe 3 wurde mit Hilfe von VHDL beschrieben. VHDL unterstützt die Aufteilung der Beschreibung in einzelne Module (Komponenten). Im folgenden sind die Teilkomponenten des Systems aufgeführt und beschrieben.

5.1.1 Eingabepuffer

Der „Eingabepuffer“ hat die Aufgabe, das READ-Signal[11] mit dem Aufbau zu synchronisieren. Dies ist nötig, da der PC mit einem eigenen Takt arbeitet und Signale vom PC somit asynchron zum Takt der FPGAs auftreten.

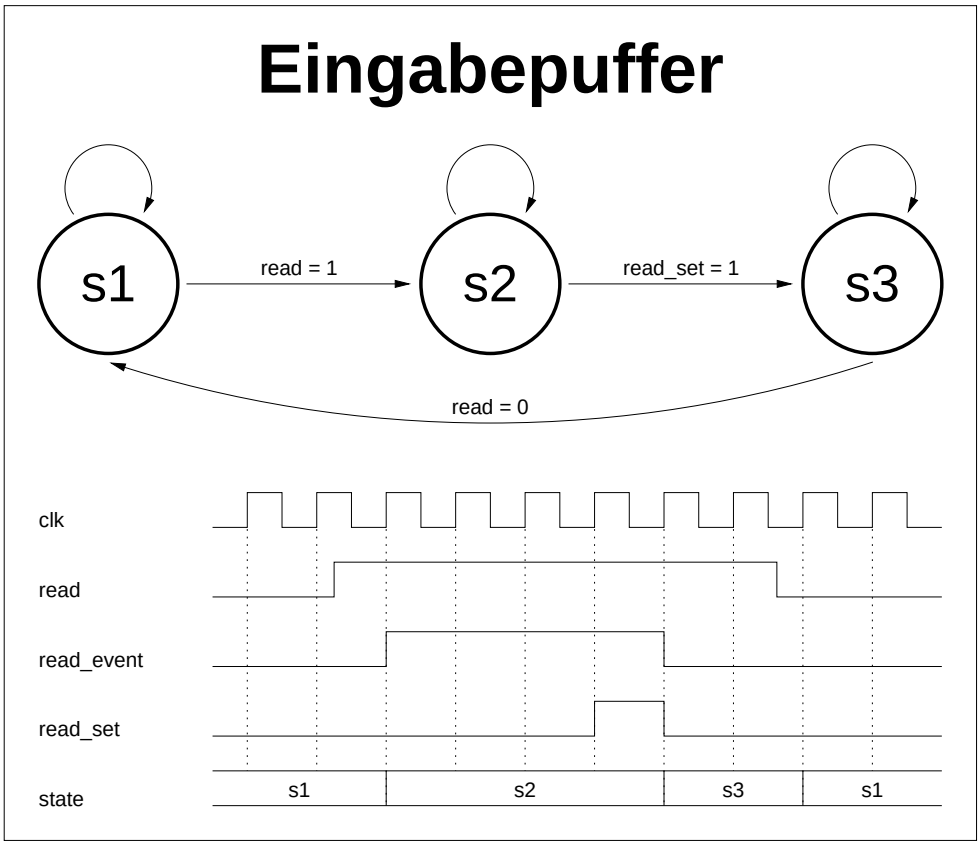

Abbildung 5-2

Das READ-Signal wird vom PC gesendet, wenn er bereit ist, ein Byte zu empfangen. Das READ-Signal kann dabei zu jedem Zeitpunkt des RAM-Zugriffs bzw. des REFRESH-Zykluses erfolgen. Weiterhin wird das Signal mit Sicherheit länger als einen (FPGA-)Takt andauern, da der Bustakt des PC deutlich langsamer ist als der Takt des FPGAs. Dies berücksichtigt der „Eingabepuffer“ und liefert ein Signal namens „READ_EVENT“, das nur solange auf „1“ liegt, bis ein Zugriffszyklus des DRAMs durchlaufen wurde und somit ein Wert ausgelesen wurde. Dieser Wert liegt nun permanent auf dem Bus zur PC-Interface-Karte an. Es muß jetzt noch gewartet werden bis das READ-Signal vom PC wieder auf „0“ geht, um dann ein neues Signal akzeptieren zu können. Grundsätzlich wird (zu Recht) angenommen, daß die Prototyphardware deutlich schneller als der PC arbeitet. Das FPGA ist im Vergleich zum PC

[11] „Ready“ Signal vom PC

schnell genug, um zwischen zwei READ-Signalen das DRAM auszulesen (siehe 4.3.2 Datenübertragung zum PC). Im Prinzip synchronisiert der „Eingabepuffer“ den Lesetakt des PCs mit dem READ-REFRESH-Zykluses des Prototyps.

```
--
-- Digital Scope III * (C) 1995 L.H.Hahn
--               Rev. 5.0 /23.08.95
--
-- inpbuf.vhd (input buffer)

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity inpbuf is
  port(
    reset      : in std_logic;
    clk        : in std_logic;
    read       : in std_logic;
    read_set   : in std_logic;
    read_event : out std_logic);
end inpbuf;

architecture behaviour of inpbuf is
  type st is (s1, s2, s3);
  signal cs, ns: st;

begin
  read_event <= '1' when cs = s2 else '0';

  READ_BUF: process(clk, cs, read, read_set)
  begin
    case cs is
      when s1 => if read='1' then ns <= s2; else ns <= s1; end if;
      when s2 => if read_set='1' then ns <=s3; else ns <= s2; end if;
      when s3 => if read='0' then ns <= s1; else ns <= s3; end if;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then cs <= s1;
    elsif (clk'event and clk = '1') then cs <= ns; end if;
  end process;
end behaviour;
```

5.1.2 Basic-Counter

Der „Counter“ stellt einen einfachen Zähler dar. Er ist zehn Bit breit, besitzt ein synchrones (`syn_reset`) und ein asychrones Reset (`asy_reset`), sowie ein Enable-Signal (`enable`).

```
--
-- Digital Scope III   Lars H. Hahn 1995
--               Rev. 5.0 /23.08.95
--
-- count.vhd (10-Bit-Zaehler)

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
```

```
entity count is
  port(
    asy_reset  : in   std_logic;
    syn_reset  : in   std_logic;
    enable     : in   std_logic;
    clk        : in   std_logic;
    raus       : out  std_logic_vector(9 downto 0)
           );
end count;

architecture behaviour of count is
  signal counter : std_logic_vector(9 downto 0);

begin

  raus <= counter;

  COUNT: process(clk,asy_reset,syn_reset)
  begin
    if (asy_reset='0') then
      counter <= "0000000000";
    elsif (clk'event and clk='1') then
      if (syn_reset='0') then
        counter <= "0000000000";
      elsif (enable = '1') then
        counter <= counter + '1';
      end if;
    end if;
  end process;
end behaviour;
```

5.1.3 Zähler

Das Modul `zaehler.vhd` instanziert drei 10-Bit-Zähler, dieser wurde zuvor im Modul `counter.vhd` beschrieben. Die Zähler dienen zur Generierung der Adressen für den RAM-Zugriff. Um einen Adressraum von 1 MByte (1048576 Byte) ansprechen zu können, werden 20 Bit breite Adressen benötigt. Beim Zugriff auf das DRAM wird die Adresse in zwei Teile zu je zehn Bit aufgeteilt, die sogenannte Zeilen- bzw. Spaltenadresse. Es sind somit nur zehn Adreßleitungen zum DRAM nötig. Sie werden sowohl zum Übermitteln der Zeilen- als auch der Spaltenadresse genutzt. Um dies zu erreichen, ist ein Multiplexer nötig. Es wird somit je ein Zähler für die Zeilen- und die Spaltenadresse, sowie ein weiterer Zähler für den REFRESH-Zugriff benötigt.

```
--
-- Digital Scope III * (C) 1995 L.H.Hahn
--               Rev. 5.0 /23.08.95
--
-- zaehler.vhd

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity zaehler is
  port(
```

```
    reset      : in std_logic;
    clk        : in std_logic;
    enaLOW     : in std_logic;
    enaHIGH    : in std_logic;
    enaREFR    : in std_logic;
    rst        : in std_logic;
    refr_mode  : in std_logic;
    amx        : in std_logic;
    cLOW1      : out std_logic;
    cLOW5      : out std_logic;
    cHIGH1     : out std_logic;
    adr        : out std_logic_vector(9 downto 0)
    );
end zaehler;

architecture behaviour of zaehler is

component count
  port(
    asy_reset  : in   std_logic;
    syn_reset  : in   std_logic;
    enable     : in   std_logic;
    clk        : in   std_logic;
    raus       : out  std_logic_vector(9 downto 0)
    );
end component;

signal counterHIGH, counterLOW, counterREFR, adr_help:
                                              std_logic_vector(9 downto 0);
signal rstLOW : std_logic;

begin
  count1: count port map(reset, rst,    enaHIGH, clk, counterHIGH);
  count2: count port map(reset, rstLOW, enaLOW,  clk, counterLOW);
  count3: count port map(reset, rst,    enaREFR, clk, counterREFR);

  cLOW1    <= '1' when counterLOW="1111111111" else '0';
  cHIGH1   <= '1' when counterHIGH="1111111111" else '0';
  cLOW5    <= '1' when (counterLOW >= 512) else '0';
  adr_help <= counterREFR when refr_mode='1' else counterLOW;
  adr      <= counterHIGH when amx='0' else adr_help;
  rstLOW   <= '0' when (rst='0' or enaHIGH='1') else '1';
end;
```

5.1.4 DRAM Ansteuerung

Die Signale zur Ansteuerung des DRAMs werden mit Hilfe eines einfachen Automaten erzeugt. Dieser durchläuft zyklisch sechs Zustände (dram1, dram2, ..., dram6; siehe Abbildung 5-3). Für jeden Zugriff auf das DRAM werden somit sechs Taktzyklen benötigt. Die Signale RAS, AMX[12] und CAS werden, wie man ebenfalls aus Abbildung 5-3 ersehen kann, direkt in Abhängigkeit vom jeweiligen Zustand erzeugt. Da sowohl beim Lese- als auch beim Schreibzugriff auf das RAM die gleichen Signale (RAS, CAS) zur Ansteuerung benötigt werden, können sie kontinuierlich erzeugt werden. Um asynchrone, externe Signale mit dem

[12] Das Signal AMX dient intern zur Auswahl der „richtigen“ Adresse (ROW oder COLUMN).

Prototypen zu synchronisieren, wird das Signal „RAM_CYCLE“ erzeugt. Es gibt das Ende eines Zugriffszyklus an.

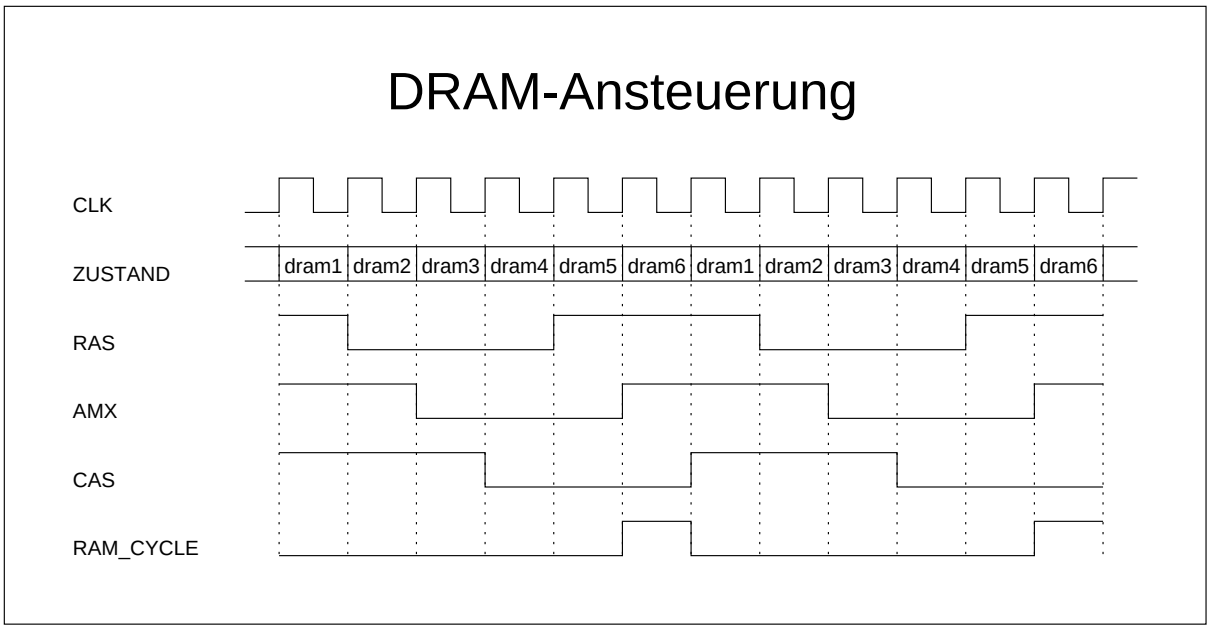

Abbildung 5-3

```
--
-- Digital Scope III   Lars H. Hahn 1995
--               Rev. 5.0 /23.08.95
--
-- dram.vhd (DRAM Timing)

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity dram is
  port(
    reset:     in std_logic;
    clk:       in std_logic;
    ras:       out std_logic;
    amx:       out std_logic;
    cas:       out std_logic;
    ram_cycle: out std_logic
);
end dram;

architecture behaviour of dram is
  type state_type is (dram1, dram2, dram3, dram4, dram5, dram6);
  signal current_state, next_state: state_type;

begin
  ras       <= '1' when (current_state=dram1 or
                         current_state=dram5 or
                         current_state=dram6) else '0';
  amx       <= '1' when (current_state=dram1 or
                         current_state=dram2 or
                         current_state=dram6) else '0';
  cas       <= '1' when (current_state=dram1 or
                         current_state=dram2 or
                         current_state=dram3) else '0';
```

```
  ram_cycle <= '1' when (current_state=dram6) else '0';

  D_RAM: process(current_state)
  begin
    case current_state is
      when dram1     => next_state <= dram2;
      when dram2     => next_state <= dram3;
      when dram3     => next_state <= dram4;
      when dram4     => next_state <= dram5;
      when dram5     => next_state <= dram6;
      when dram6     => next_state <= dram1;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= dram1;
    elsif (clk'event and clk = '1') then
      current_state <= next_state;
      end if;
  end process;
end behaviour;
```

5.1.5 Der zentrale Steuerungsautomat

Im Modul `masterfsm.vhd` wird der zentrale Steuerungsautomat (siehe Abbildung 5-4) beschrieben. Er wartet nach der Initialisierung auf den ersten VSYNC, um dann in den Zustand `sample` zu wechseln. In diesem Zustand werden die Bilddaten Zeile für Zeile im DRAM gespeichert. Der Zustand wird mit dem nächsten VSYNC verlassen. Es folgt der Zustand `REFRESH` für den REFRESH-Zyklus, der immer wieder zum Auslesen einzelner Daten aus dem DRAM unterbrochen wird (`READ`). Wurde das komplette Bild zum PC übertragen, wird sofort nach dem nächsten VSYNC das nächste Bild aufgenommen.

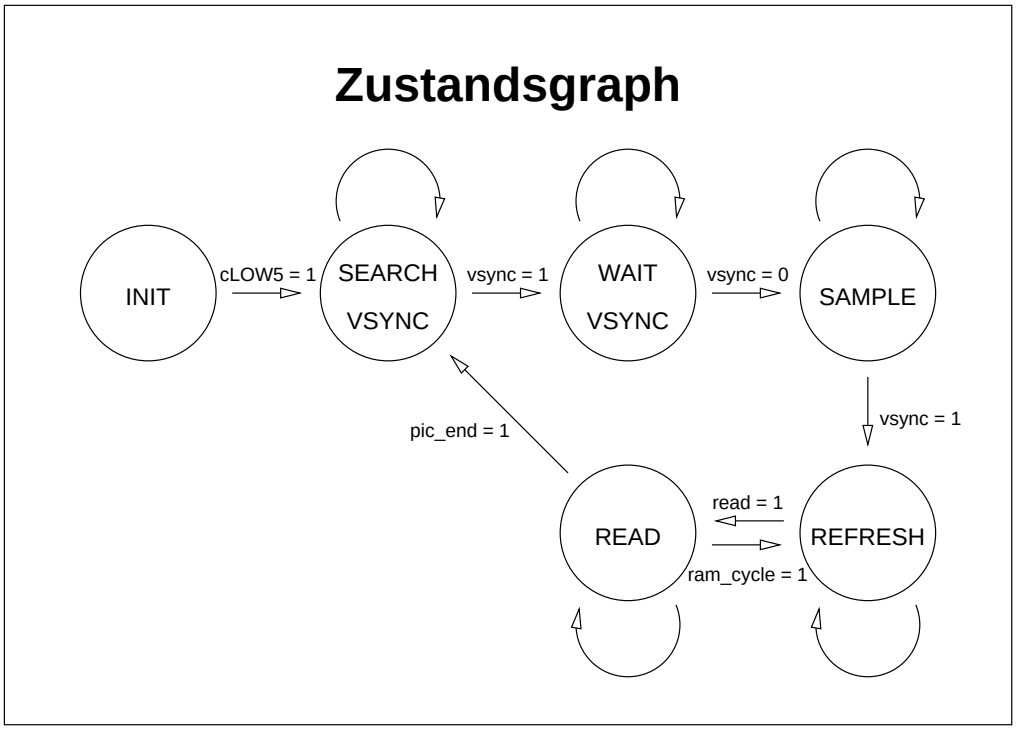

Abbildung 5-4

```
--
-- Digital Scope III   Lars H. Hahn 1995
--                Rev. 5.0 /23.08.95
--
-- masterfsm.vhd (master statemachine)

library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity masterfsm is
  port(
    reset:      in std_logic;
    clk:        in std_logic;
    vsync:      in std_logic;
    hsync:      in std_logic;
    read_event: in std_logic;
    cLOW1:      in std_logic;
    cLOW5:      in std_logic;
    cHIGH1:     in std_logic;
    ram_cycle:  in std_logic;
    we:         buffer std_logic;
    read_set:   buffer std_logic;
    enaLOW:     out std_logic;
    enaHIGH:    out std_logic;
    enaREFR:    out std_logic;
    rst:        out std_logic;
    refr_mode:  buffer std_logic
);
end masterfsm;

architecture behaviour of masterfsm is
  type state_type is
   (init, search_vsync, wait_vsync, sample, refresh, read);
  signal current_state, next_state: state_type;
  signal pic_end, read_mode, samp_mode: std_logic;
```

```
begin
  pic_end   <= '1' when (cLOW1='1' and cHIGH1='1') else '0';
  we        <= '0' when (current_state=sample) else '1';
  samp_mode <= '1' when (current_state=sample) else '0';
  read_mode <= '1' when (current_state=read) else '0';
  refr_mode <= '1' when (current_state=refresh) else '0';
  read_set  <= '1' when (ram_cycle='1' and read_mode='1') else '0';
  enaLOW    <= '1' when (((samp_mode='1' or read_mode='1') and
                       ram_cycle='1') or current_state=init) else '0';
  enaHIGH   <= '1' when ((read_set='1' and cLOW1='1') or
                       (samp_mode='1' and hsync='1')) else '0';
  enaREFR   <= '1' when (refr_mode='1' and ram_cycle='1') else '0';

  FSM: process(current_state, vsync, read_event, pic_end, ram_cycle, cLOW5)
  begin
    case current_state is
     when init         => rst <= '1';
                          if (cLOW5 = '1') then next_state <= search_vsync;
                          else next_state <= init; end if;
     when search_vsync => rst <= '1';
                          if vsync='1' then next_state <= wait_vsync;
                          else next_state <= search_vsync; end if;
     when wait_vsync   => if vsync='0' then rst <= '0';
                                            next_state <= sample;
                          else rst <= '1'; next_state <= wait_vsync;end if;
     when sample       => if vsync='1' then rst <= '0';
                                            next_state <= refresh;
                          else rst <= '1'; next_state <= sample; end if;
      when refresh     => rst <= '1';
                          if (read_event='1' and ram_cycle='1') then
                                                  next_state <= read;
                          else next_state <= refresh; end if;
      when read        => rst <= '1';
                          if pic_end='1' then next_state <= search_vsync;
                          elsif (ram_cycle='1') then next_state <= refresh;
                          else next_state <= read; end if;
    end case;
  end process;

  SYNCH: process(clk, reset)
  begin
    if (reset = '0') then
      current_state <= init;
     elsif (clk'event and clk = '1') then
      current_state <= next_state;
      end if;
  end process;
end behaviour;
```

5.1.6 MegaDigi3 (Toplevel Design)

Im Modul `digi.vhd` werden alle Komponenten des Digitalisieres instanziert und zusammengefügt, mit Ausnahme des „Syncfilters“. Dieser ist in einem eigenen Baustein untergebracht.

Weiterhin werden im Modul `digi.vhd`, in Abhängigkeit vom Write-Enable-Signal (we) die Tristatetreiber des bidirektionalen Datenbus zum RAM gesetzt. Dies ist nötig, da ja auf das RAM sowohl lesend als auch schreibend zugegriffen wird.

Im Prozeß NEWCLK wird außerdem ein Takt (clk2) mit halbierter Frequenz erzeugt. Hiermit wird der FADC getaktet.

```
--
-- Digital Scope III * (C) 1995 L.H.Hahn
--               Rev. 5.0 /23.08.95
--
-- digi.vhd (toplevel design)

Library IEEE;
use ieee.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity digi is
  port(
    reset    : in std_logic;
    clk      : in std_logic;
    video_in : in std_logic_vector(7 downto 0);
    read     : in std_logic;
    hsync    : in std_logic;
    vsync    : in std_logic;
    ras      : out std_logic;
    cas      : out std_logic;
    we       : buffer std_logic;
    adr      : out std_logic_vector(9 downto 0);
    daten    : inout std_logic_vector(7 downto 0);
    bild_out : out std_logic_vector(7 downto 0);
    clk2     : buffer std_logic
    );
end digi;

architecture behaviour of digi is
component masterfsm
  port(
    reset:      in std_logic;
    clk:        in std_logic;
    vsync:      in std_logic;
    hsync:      in std_logic;
    read_event: in std_logic;
    cLOW1:      in std_logic;
    cLOW5:      in std_logic;
    cHIGH1:     in std_logic;
    ram_cycle:  in std_logic;
    we:         buffer std_logic;
    read_set:   buffer std_logic;
    enaLOW:     out std_logic;
    enaHIGH:    out std_logic;
    enaREFR:    out std_logic;
    rst:        out std_logic;
    refr_mode:  buffer std_logic
    );
end component;

component dram
  port(
    reset:     in std_logic;
    clk:       in std_logic;
    ras:       out std_logic;
    amx:       out std_logic;
    cas:       out std_logic;
    ram_cycle: out std_logic
    );
end component;
```

```
component inpbuf
  port(
    reset   : in std_logic;
    clk     : in std_logic;
    read    : in std_logic;
    read_set : in std_logic;
    read_event   : out std_logic
    );
end component;

component zaehler
  port(
    reset      : in std_logic;
    clk        : in std_logic;
    enaLOW     : in std_logic;
    enaHIGH    : in std_logic;
    enaREFR    : in std_logic;
    rst        : in std_logic;
    refr_mode  : in std_logic;
    amx        : in std_logic;
    cLOW1      : out std_logic;
    cLOW5      : out std_logic;
    cHIGH1     : out std_logic;
    adr        : out std_logic_vector(9 downto 0)
           );
end component;

signal read_event, read_set, ram_cycle, amx, refr_mode : std_logic;
signal cLOW1, cLOW5, cHIGH1, enaLOW, enaHIGH, enaREFR, rst : std_logic;
signal bild_store : std_logic_vector(7 downto 0);
signal help_store : std_logic_vector(7 downto 0);

begin
  my_masterfsm: masterfsm port map
    (reset, clk, vsync, hsync, read_event, cLOW1, cLOW5, CHIGH1, ram_cycle,
    we, read_set, enaLOW, enaHIGH, enaREFR, rst, refr_mode);
  my_dram: dram port map
    (reset, clk, ras, amx, cas, ram_cycle);
  my_inpbuf: inpbuf port map
    (reset, clk, read, read_set, read_event);
  my_zaehler: zaehler port map
    (reset, clk, enaLOW, enaHIGH, enaREFR, rst, refr_mode, amx, cLOW1,
    cLOW5, cHIGH1, adr);

  bild_out <= bild_store;
    daten <= video_in when we='0' else "ZZZZZZZZ";

  process(clk, reset, read_set)
  begin
    if (reset = '0') then bild_store <= "00000000";
    elsif (clk'event and clk='1') then
      if (read_set='1') then bild_store <= daten; end if;
      end if;
  end process;

  NEWCLK: process(clk, reset)
  begin
    if (reset = '0') then clk2 <= '0';
    elsif (clk'event and clk='1') then
      if (clk2='1') then
        clk2 <= '0';
      else
        clk2 <= '1';
      end if;
        end if;
  end process;
```

```
end;
```

5.1.7 Syncfilter (VHDL Beschreibung)

Im folgenden ist der Syncfilter beschrieben. Er ist in vier Prozesse unterteilt:

VERT	Mit jeder Rückflanke eines Synchronisationsimpulses wird das Signal `VSYNC` neu bestimmt. Dies geschieht mit Hilfe des Signals `WELCHER`, das in Abhängigkeit vom Stand des Zählers `COUNTER` gesetzt wird.
COUNT	Mit jeder Vorderflanke eines Synchronisationsimpulses wird der Zähler `COUNTER` neu gestartet. In Abhängigkeit von diesem Zähler wird bei der Rückflanke entschieden, ob ein HSYNC oder ein VSYNC vorliegt.
LATCH	Dieser Prozeß sorgt für die Zwischenspeicherung des aktuellen Synchronisationsimpulses und des Videosignals.
MINIMUM	Hier wird das „aktuelle“ Minimum im Datenstrom bestimmt. Das Verfahren, mit dem dies geschieht, wurde im Kapitel 4.5.1 - Erkennung der Synchronisationsimpulse - beschrieben.

```
--
-- Digital Scope III   Lars H. Hahn 1995
--               Rev. 5.1 /13.09.95
--
-- new_sf.vhd (Sync-Erkennung)

library ieee;
use IEEE.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity new_sf is
  port(
    reset    : in  std_logic;
    clk      : in  std_logic;
    video_in : in  std_logic_vector(3 downto 0);
    vsync    : out std_logic;
    hsync    : out std_logic
   );
end new_sf;

architecture behaviour of new_sf is

  signal mem,min,diff : std_logic_vector(3 downto 0);
  signal counter : std_logic_vector(9 downto 0);
  signal sync_vf,sync_rf,old_sync,sync,welcher : std_logic;

begin
  diff    <= "0000" when (min > mem) else (mem - min);
  sync    <= '0' when (diff="0000" or diff="0001" or diff="0010") else '1';
  sync_vf <= '1' when (old_sync = '1' and sync = '0') else '0';
  sync_rf <= '1' when (old_sync = '0' and sync = '1') else '0';
  welcher <= '1' when (counter > 100) else '0';
```

```
  hsync   <= sync_rf when (welcher = '0') else '0';

  VERT: process(clk,reset)
  begin
   if (reset='0') then
     vsync <= '0';
   elsif (clk'event and clk='1') then
     if (sync_rf='1') then
       vsync <= welcher;
     end if;
    end if;
  end process;

  COUNT: process(clk,reset,sync_vf)
  begin
    if (reset='0') then
      counter <= "0000000000";
    elsif (clk'event and clk='1') then
      if (sync_vf='1') then
        counter <= "0000000000";
      else
        counter <= counter + 1;
      end if;
    end if;
  end process;

  LATCH: process(clk,reset)
  begin
   if (reset='0') then
     mem <= "0000";
     old_sync <= '0';
   elsif (clk'event and clk='1') then
      mem <= video_in;
      old_sync <= sync;
    end if;
  end process;

  MINIMUM: process(clk,reset,sync_vf)
  begin
    if (reset='0') then
      min <= "1111";
    elsif (clk'event and clk='1') then
      if ((sync_vf = '1') or (counter = "1111111111")) then
        min <= min + 1;
      elsif (mem < min) then
        min <= min - 1;
      end if;
    end if;
  end process;

end behaviour;
```

5.2 PC Software

Für den PC entstanden im Laufe der Studienarbeit zwei Programme. Zum einen ein Programm, um den Algorithmus zum Erkennen der Synchronisationsimpulse zu testen, und zum anderen die Übertragungssoftware, um die im DRAM zwischengespeicherten Bilddaten zum PC zu übertragen.

5.2.1 PC-Syncfilter

Der „PC-Syncfilter“ entstand im Rahmen der Prototyp Stufe 2. Er dient zum Testen des Algorithmus, der die HSYNCs und VSYNCs erkennt. Als C-Programm funktionierte dieser Algorithmus hervorragend. Leider brachte die direkte Umsetzung in VHDL keine zufriedenstellenden Ergebnisse. Das C-Programm erkennt die Synchronisationsimpulse auf Grund der großen Differenz zwischen zwei Werten im Videodatenstrom. In der endgültigen VHDL Implementation blieb nur das Verfahren zur Erkennung des Minimums erhalten. Auf ein Synchronisationsimpuls wird (in der VHDL Version) immer dann entschieden, wenn der digitale Wert des Videosignals in der Nähe des Minimums liegt (siehe 5.1.7 Syncfilter).

```
#include "stdio.h"
#define TRUE 1
#define FALSE 0
typedef unsigned char BOOL;
char filename[] = "megabild.dat";

void main()
{
  FILE *datei;
  int i=0;
  unsigned char c=0,c1=0,c2=0,c3=0,min=255;
  int dif1=0,dif2=0,dif3=0,count=0;
  BOOL hsync = FALSE, vsync = FALSE;
  BOOL start = FALSE, ready = FALSE;
  datei = fopen(&filename[0],"r");

  if(datei)
  {
    while(ready==FALSE)
    {
      i++;
      count++;
      c = fgetc(datei);
      c1 = c >> 4;
      if(c1 < min) min=c1;
      dif1 = c3 - c1;
      dif2 = c1 - min;
      dif3 = c3 - min;
      if((dif1 > 2) && (dif2 < 3))
      {
        if(hsync == FALSE)
        {
          if(count < 210)
          {
            if((start==TRUE) && (vsync==FALSE)) ready = TRUE;
            else start = TRUE;
            vsync = TRUE;
            hsync = FALSE;
          }
          else
          {
            vsync = FALSE;
            hsync = TRUE;
            count = 1;
          }
        }
      }
      else if((dif1 < -2) && (dif3 < 3))
      {
```

```
        if(hsync == TRUE) hsync = FALSE;
      }
      else if((hsync==FALSE) && (vsync==FALSE) && (start==TRUE))
      {
        printf("%c",c);
      }
      c3 = c2;
      c2 = c1;
    }
    fclose(datei);
  }
  else
  {
    printf("'%s' nicht gefunden!\n",&filename[0]);
  }
}
```

5.2.2 Übertragungssoftware

Die Übertragungssoftware funktioniert nach einem sehr einfachen Handshake-Protokoll. Immer wenn der PC bereit ist einen neuen Wert zu empfangen, setzt er die READ-Leitung auf „1“. Die FPGA-Logik ist ausreichend schnell, so daß der neue Wert aus Sicht des PCs sofort anliegt. Die Daten werden in Blöcken von 4 KB auf die Festplatte des PCs geschrieben. Es werden die gesamten 1 MB Daten ausgelesen, d.h. ein Bild mit einer Größe von 1024 mal 1024 Bildpunkten. Hiervon wird aber nur ein Teil für das tatsächliche Bild benötigt, die Größe des Bildes hängt direkt von der Taktfrequenz ab, mit der der Digitalisierer betrieben wird.

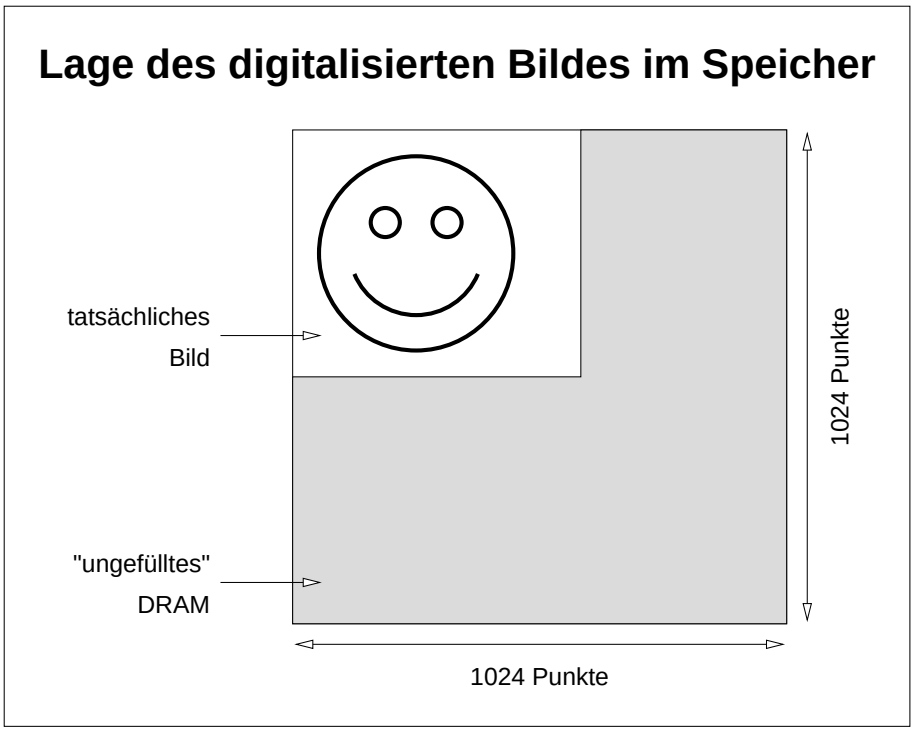

Abbildung 5-5

```
/*************************************************************
 *                                                           *
 *   Videoscope - Steuerprogramm   (C) Juli 1994 L.H.Hahn    *
 *                                                           *
 *   Version 2.0 (9.12.94)                                   *
 *   Version 95  (13.9.95)                                   *
 *                                                           *
 *************************************************************/

#include "stdio.h"
#include "fcntl.h"
#include "io.h"
#include "dos.h"

int porta = 0x0de4, portb  = 0x0de5,
    portc = 0x0de6, status = 0x0de7;
unsigned char speicher[0xfff];

unsigned char datain() /* Daten von Port B einlesen */
{
   unsigned char byte;
   byte = inportb(portb);
   return(byte);
}

void dataout(byte) /* Daten nach Port A schreiben */
unsigned char byte;
{
   outportb(porta, byte);
}

void warten()
{
   int i;
   for(i=0;i<100;i++);
}

void main()
{
  long i,j;
  int datei;
  printf("MEGADIGI III *** (C) 1995 L.H.Hahn\n");
  outportb(status, 0x82); /* 1000 0010 */

  /* 1MByte abholen */
  datei = open("megabild.dat",O_CREAT|O_BINARY);
  for(i=0;i<0x100000;i+=0x1000)
  {
       for(j=0;j<0x1000;j++)
       {
            dataout(1);
            warten();
            speicher[j] = datain();
            dataout(0);
            warten();
       }
       /* ... nun auf'e Harddisk! */
       write(datei,speicher,0x1000);
       printf("#");
  }
  close(datei);
}
```

5.3 Die Simulationsumgebung

Das besondere der Simulationsumgebung ist, daß „reale“ Daten zur Verfügung gestellt werden. Mit Hilfe des Prototyps „MegaDigi2“ wurden Videodaten kontinuierlich gesamplet und in 1 MB großen Dateien abgelegt. Damit diese Daten in VHDL eingelesen werden können, müssen sie als ASCII Datei in folgender Form vorliegen:

```
123 127 126 120 119 150 164 187 200 201
220 229 230 245 236 218 209 187 180 167
120  90  77  76  78  75 103 109 150 155
176 180 189 ...
...
```

Eine komplette Zeile kann dann mit folgender Programmzeile in die Variable `in_line` eingelesen werden:

```
readline(datei,in_line);
```

Auf die einzelnen Zahlen kann sequentiell wie folgt zugegriffen werden:

```
read(in_line, video);
```

Die Variable `video` (Integer) muß zur weiteren Benutzung noch in ein „std_logic_vector“ konvertiert werden. Die weiteren Prozesse dienen der Erzeugung des Taktes und des RESET-Signals.

```
--
-- Testumgebung: Syncfilter fuer Digital Scope III * (C) 1995 L.H.Hahn
--                                                  Rev 5.1 /13.09.95
library synopsys;
use synopsys.distributions.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.testpkg.all;
use std.textio.all;

entity hugo_new_sf is
end;

architecture tb of hugo_new_sf is

signal clk, reset, vsync, hsync: std_logic;
signal video_in : std_logic_vector(3 downto 0);
signal video_help : std_logic_vector(7 downto 0);
file datei: text is in "megabild.asc";

component new_sf
  port(
    reset    : in     std_logic;
    clk      : in     std_logic;
    video_in : in     std_logic_vector(3 downto 0);
    vsync    : out    std_logic;
    hsync    : out    std_logic
    );
end component;

begin
```

```
  my_syncfilter: new_sf port map(reset, clk, video_in, vsync, hsync);

  video_in(0) <= video_help(3);
  video_in(1) <= video_help(2);
  video_in(2) <= video_help(1);
  video_in(3) <= video_help(0);

  getfile: process
  variable in_line: line;
  variable video: integer range 0 to 255;
  begin
    while not endfile(datei) loop
      readline(datei,in_line);
      while not endline(in_line) loop
        read(in_line, video);
        video_help <= xconv_slv(video,8);
        wait for 24 ns;
      end loop;
    end loop;
  end process;

  clk_pro: process
  begin
    clk <= '0';
    wait for 12 ns;
    clk <= '1';
    wait for 12 ns;
  end process;

  reset_pro: process
  begin
    wait for 100 ns;
    reset <= '1';
    wait on reset;
  end process;

end tb;


configuration cfg_hugo_new_sf of hugo_new_sf is
  for tb
  end for;
end;
```

6 Ergebnisse und Ausblick

Am Ende steht eine synthesefähige VHDL-Beschreibung eines Video-Digitalisieres. Weiterhin gibt es einen funktionstüchtigen Hardwareaufbau zur Aufnahme von Videobildern. Als Nebenprodukt existiert eine erprobte VHDL-Beschreibung zur Ansteuerung von DRAMs.

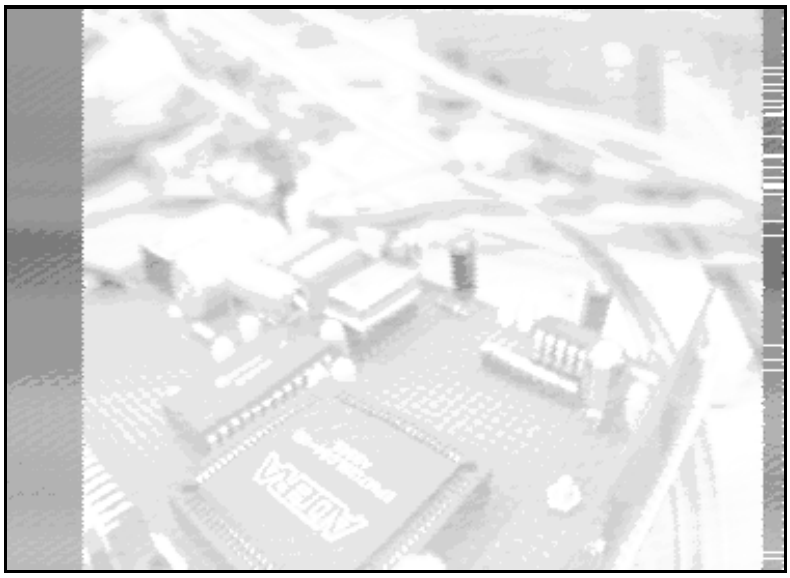

Abbildung 6-1

Die Abbildung 6-1 zeigt den Aufbau der Prototypen Stufe 2. Das Bild stellt quasi eine „Selbstaufnahme“ dar. Der linke, dunkle Streifen ist der Bereich in dem sich bei einem Farbvideosignal der *Burst* befindet. Rechts befindet sich die sogenannte Schwarzschulter.

6.1 Hardware-Aufwand

Die Entwicklung des Hardwareaufwandes im Vergleich der einzelnen Prototyp Stufen verdeutlicht die folgende Tabelle:

	Logikzellen
Prototyp Stufe 1	46
Prototyp Stufe 2	93
Prototyp Stufe 3	162

Es ist die deutliche Steigerung des Logikaufwandes zwischen den verschiedenen Stufen der Realisierung erkennbar. Die ersten beiden Stufen passen ohne Probleme in die an der Universität vorhandenen Bausteine vom Typ EPM7128 (siehe 4.1.2 FPGA). Diese erlauben die Verwendung von 128 Logikzellen pro FPGA. Die Stufe 3 paßt, entgegen der weitaus niedriger ausgefallenen Schätzung, nicht in diesen Baustein. Daher wurde das Design auf zwei FPGAs aufgeteilt. Die Aufteilung konnte sich zum Glück an funktionalen Grenzen orientieren. So beansprucht der „Syncfilter“ einen eigenen Baustein. Der gesamte übrige Logik (Ansteuerung von DRAM, PC und FADC) befindet sich im zweiten Baustein.

6.2 Erweiterungen und Verwendung

Der Prototyp der Stufe 3 läßt sich in der jetzigen Form als funktionstüchtiger Digitalisierer für Videoeinzelbilder einsetzten. Eine mögliche Erweiterung liegt darin, auch Farbbilder zu verarbeiten. Hierzu ist aber eine höhere Samplingfrequenz und somit eine höhere Taktrate nötig.
Die vorhandene VHDL Beschreibung des Syncfilters oder der DRAM Ansteuerung läßt sich in anderen Projekten der Bildverarbeitung verwenden.

7 Literaturverzeichnis

[MOT84] Motorola: Analog/Digital and Digital/Analog Conversion Manual, 1984

[KRI93] Krisch, Lothar: Fernsehtechnik: Grundlagen, Verfahren, Systeme - Vieweg, 1993

[KAN90] Kane, Gerry: CRT Controller Handbook, OSBORNE/McGraw-Hill 1990

[HEN] Hensinger: PLDs und FPGAs in der Praxis

[ASS90] Assenbaum, Johannes: DRAMs in der Übersicht, Teil 1;
aus c't 10/1990, Seite 375

[ASS90] Assenbaum, Johannes: DRAMs in der Übersicht, Teil 2;
aus c't 11/1990, Seite 417

[MAE93] Maeder, Andreas: VHDL Kurzbeschreibung zum Projekt „VLSI-Entwurf", 1993

[KLI93] Klindworth, André: Unterlagen zum FPGA-Praktikum, 1993

[KOL90] Kolter, Heinrich: Einführung in die Programmiertechnik des PPI 8255, 1990

8 Anhang

8.1 Beschaltung FADC

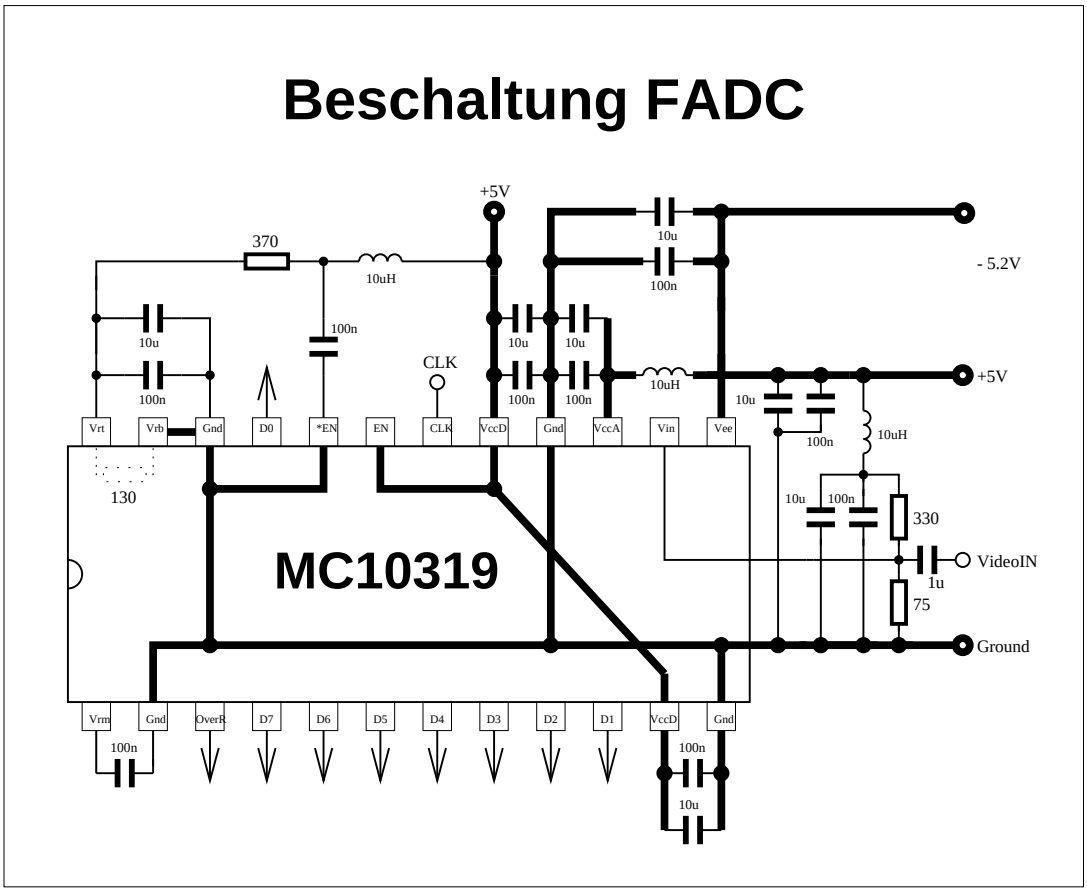

Abbildung 8-1

8.2 Pinbelegung FPGAs

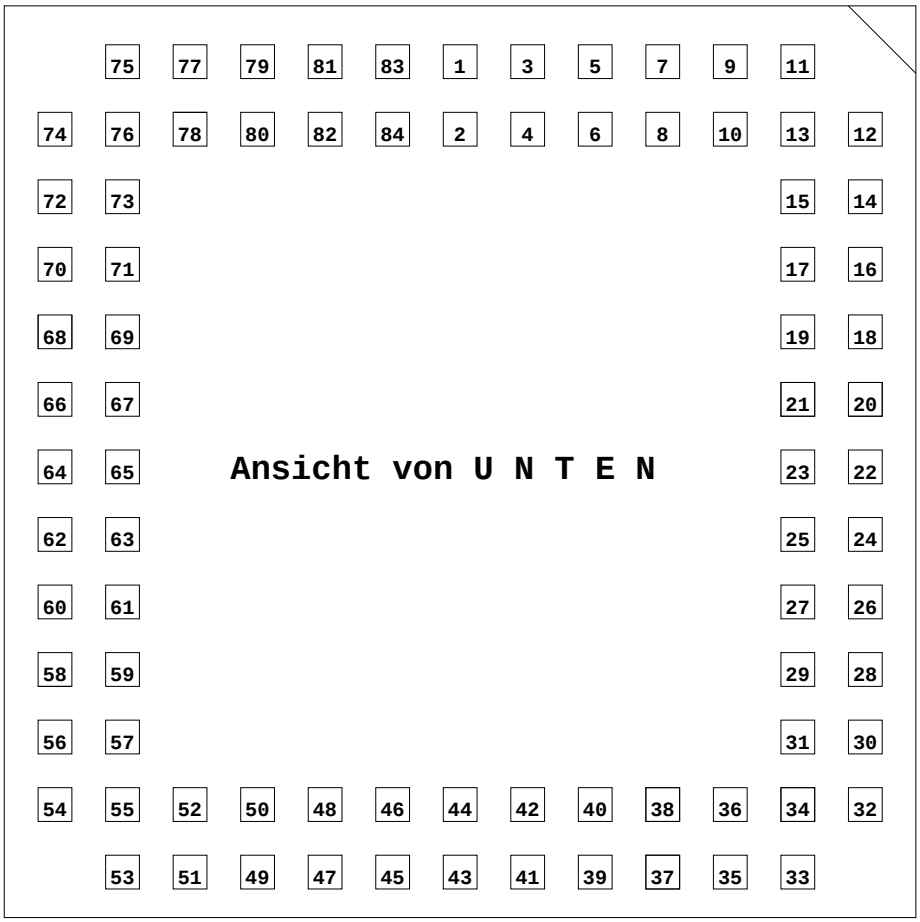

Abbildung 8-2