Implementierung einer Opensource Softcore-CPU auf einem FPGA und Anbindung von Peripheriemodulen über deren Busschnittstelle

STUDIENARBEIT

des Studienganges Elektrotechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Johannes Steudle

21.03.2011

Bearbeitungszeitraum	10 Wochen
Matrikelnummer, Kurs	4208880, TEL08AAT
Ausbildungsfirma	Pepperl+Fuchs GmbH, Mannheim
Betreuer der Dualen Hochschule	Dipl.-Ing.(FH) Dennis Trebbels, M.Sc.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mannheim, 21.03.2011 _____

Ort, Datum

Unterschrift

Vorwort und Aufgabenstellung

Im Rahmen meiner Studienarbeit im 6. Semester ergab sich die Möglichkeit, ein Thema mit einem FPGA zu bearbeiten. Die Studienarbeit „Implementierung einer Opensource Softcore-CPU auf einem FPGA und Anbindung von Peripheriemodulen über deren Busschnittstelle“ wurde an der Dualen Hochschule Baden Württemberg Mannheim an der Fakultät Elektrotechnik durchgeführt.

Für die Studienarbeit stand ein Experimentierboard Spartan-3 von Xilinx zur Verfügung, auf dem ein Xilinx XC3S200 FPGA vorhanden ist. Im Rahmen der Studienarbeit soll eine geeignete, wenn möglich 32-bit Opensource Softcore-CPU auf diesem FPGA-Zielsystem implementiert werden. Die CPU soll über ein Bussystem, beispielsweise Wishbone, an Peripheriemodule angebunden werden. Sofern keine geeigneten Peripheriemodule oder Busschnittstellen vorhanden sind, sind diese im Rahmen der jeweiligen Studienarbeit zu erarbeiten und umzusetzen. Das Gesamtsystem soll durch zu erstellende geeignete Beispielprogramme die Funktionsfähigkeit demonstrieren.

Hierbei gliedert sich die Studienarbeit in folgende Teilpunkte:

- Auswahl einer geeigneten und interessanten Softcore-CPU (z.B. auf www.opencores.org)
- Download aller Quelldateien und Dokumentationen
- Erstellung eines Projektes in der Entwicklungsumgebung Xilinx ISE
- Implementierung der gewählten Softcore-CPU auf dem Zielsystem
- Implementierung geeigneter Beispielperipherie und Anbindung an die CPU über deren Busse
- Erstellung eines lauffähigen Demoprogramms zum Nachweis der Funktionalität des Gesamtsystems

Der vorliegende Bericht soll dem Leser als eine Art Tutorial ein Verständnis für das Projekt sowie eine selbstständige Implementierung und eigene Erweiterung ermöglichen. Verständnisgrundlage ist hierbei Kenntnis in VHDL und über die Funktionsweise einer CPU.

Ich bedanke mich an dieser Stelle bei Herrn Dennis Trebbels für seine Hilfe und stetige Diskussionsbereitschaft. Des Weiteren möchte ich Herrn Matthias Meier[1] danken, der mir mit seinem Projekt „Merkmalbasierte statische Konfigurierung von MPSoCs“ sehr helfen konnte.

[1] http://ess.cs.tu-dortmund.de/EN/Research/Projects/LavA/

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	**Bedeutung**
ACK	Acknowledge
ADR	Address
BRAM	Block RAM
CE	Chip Enable
CLK	Clock
CPU	Central Processing Unit
CYC	Cycle
DAT	Data
GCC	GNU Compiler Collection
FPGA	Field Programmable Gate Array
IR	Interrupt
IRQ	Interrupt Request
JTAG	Joint Test Action Group
LED	Light Emitting Diode
PS/2	Personal System/2
RGB	Farbschema Red Green Blue
ROM	Read Only Memory
RST	Reset
SEL	Select
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WE	Write Enable
WB	Wishbone-Bus
ZPU	Zylin CPU

Abbildungs- und Tabellenverzeichnis

1 Einleitung

1.1 Das Board und Entwicklungsumgebung

Im Rahmen der Studienarbeit stand ein Experimentierboard Spartan-3[2] von Xilinx zur Verfügung. Das Spartan-3 ist ein Einsteigerboard mit einem XC3S200 FPGA[3] (1) das sich aufgrund der geringen Anzahl an BlockRAMs (12 Stk., entspricht 24 KB) nicht unbedingt für den Einsatz eines anspruchsvollen Softcore-Prozessors mit Peripherie eignet.

Die wichtigsten Komponenten für das Projekt sind: eine PS/2- (2), eine VGA- (3), eine RS-232-Schnittstelle (4), 8 Schalter (5), 8 LEDs (6), 4 Buttons (7) und 4 Siebensegment-Anzeigen (8), des Weiteren der SRAM auf der Rückseite der Platine.

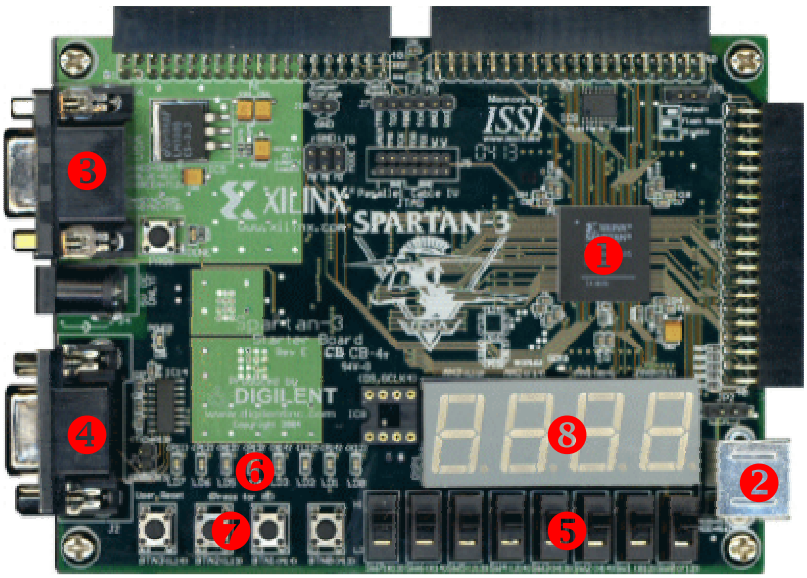


Abbildung 1: Spartan-3 Board[4]

Programmieren lässt sich das Board bzw. der FPGA mittels eines JTAG-Adapters[5]. Eine passende Software, die Xilinx ISE Entwicklungsumgebung[6] lässt sich kostenlos von Xilinx herunterladen. Für die Studienarbeit wurde die Version 10.1 verwendet. Ein Tutorial, wie das Board in Betrieb zu nehmen ist, die Entwicklungsumgebung genutzt und ein Beispielprojekt auf den FPGA geladen wird, findet sich im Anhang A.

---

[2] Vgl. [1] XILINX: Spartan-3 Starter Kit,
vgl. [2] XILINX: Spartan-3 FPGA Starter Kit Board User Guide und
vgl. [3] XILINX: Programmable Logic Design : Quick Start Handbook.

[3] Vgl. [4] XILINX: Spartan-3 FPGA Family Data Sheet.

[4] Quelle: eigene Abbildung.

[5] Vgl. [5] http://shop.trenz-electronic.de/catalog/product_info.php?cPath=30&products_id=591

[6] Vgl. [6] XILINX: Downloads.

1.2 Suche einer passenden Softcore-CPU

Begonnen hat die Studienarbeit mit der Suche nach einer frei verfügbaren und geeigneten Softcore-CPU. Ein Vorschlag war die ZPU von Øyvind Harboe. Sie ist eine kleine Stack-basierte 32 Bit CPU mit einem 8 Bit breiten Opcode. Sie hat einen einfachen und übersichtlichen Aufbau und ist deshalb optimal für die hier angestrebten Erweiterungen geeignet. Diese Opensource Softcore-CPU ist als VHDL-Projekt angelegt. Sehr nützlich ist die Möglichkeit, Programme, die in C oder C++ verfasst sind über eine spezielle GCC-Toolchain (ZPUGCC[7]) in einen Maschinencode zu wandeln, den die ZPU direkt interpretieren kann. So kann man die ZPU mittels C-Programmen steuern. Aufgrund genannter Vorteile viel die Wahl für das Projekt auf die ZPU. Verwendet wurde letztendlich deren „Small"-Variante.

Eine große Hilfe war hierbei die abgespeckte Version des Projekts von Matthias Meier[8]. Er verwendete in diesem Projekt die ZPU mit einer Wishbone-Anbindung[9] in einem Multicore-system. Für die Studienarbeit stellte er einen Auszug davon zur Verfügung. Das Projekt ist zwar eigentlich für das Board Spartan-3E von Xilinx erstellt, lässt sich aber ohne Probleme für das Spartan-3 abändern.

[7] Vgl. [7] ZYLIN CONSULTING: Zylin CPU.

[8] Vgl. [8] MEIER, Matthias: Merkmalbasierte statische Konfigurierung von MPSoCs und http://ess.cs.tu-dortmund.de/EN/Research/Projects/LavA/

[9] Vgl. [9] http://en.wikipedia.org/wiki/Wishbone_%28computer_bus%29

2 Überblick über das VHDL-Projekt

Zunächst eine Übersicht über die Struktur des Projektes:

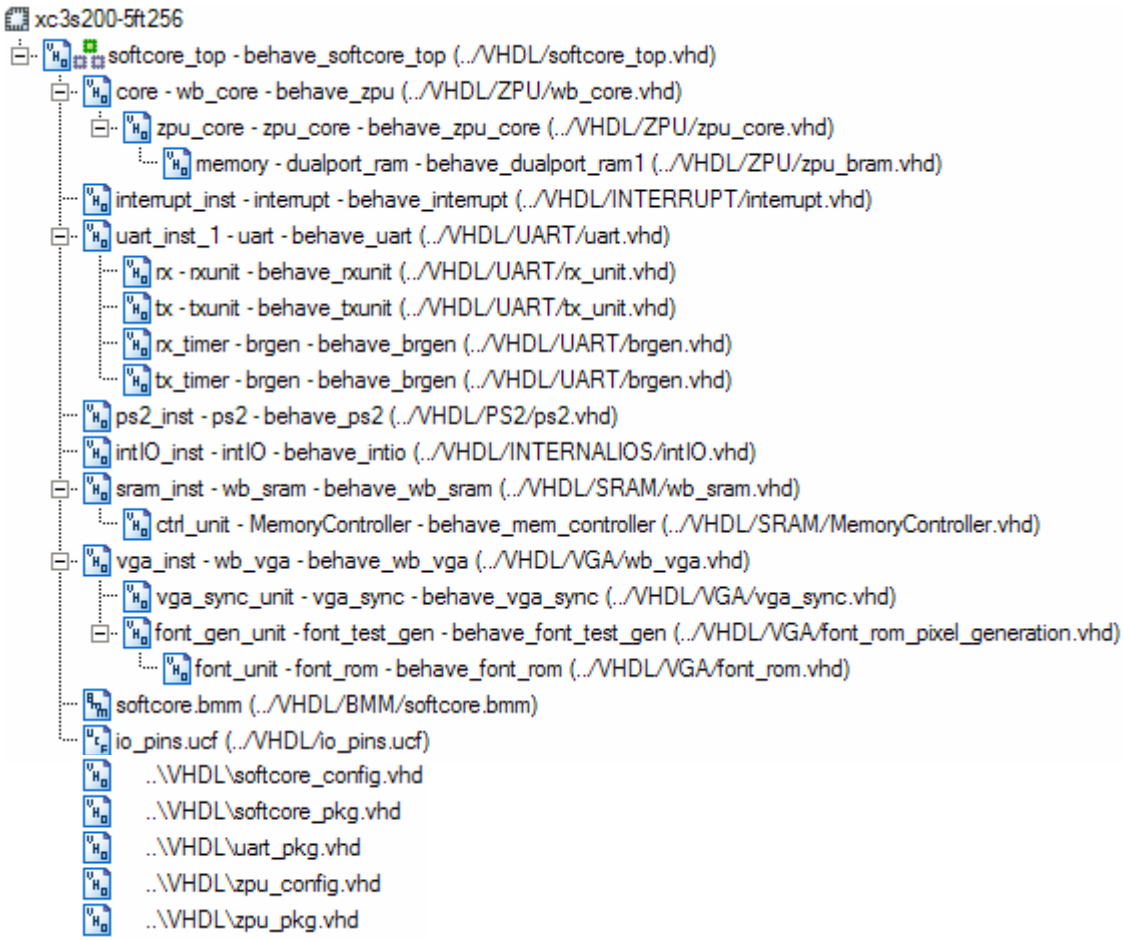


Abbildung 2: Struktur des VHDL-Projektes

Wie in Abbildung 2 zu sehen ist, befindet sich an der Spitze des Baumes, also als Top Modul, die Datei softcore_top.vhd. Sie verbindet alle benötigten externen Signale des Boards über die io_pins.ucf mit den Modulen. Sie beinhaltet den Wishbone-Bus und die Anbindung der Busteilnehmer. Hierzu gehört auch die Festlegung der Adressbereiche der Teilnehmer.

Über die softcore_top.vhd werden alle benötigten Wishbone-Interfaces eingebunden, beginnend mit der als Wishbone-Master konfigurierten ZPU, sowie einem Interrupt-Controller, einem UART, einer PS/2-(Keyboard-)Schnittstelle, einem Baustein, der die LEDs, Schalter, Taster und 7-Segment-Anzeigen des Boards steuert, einem SRAM-Controller und einer VGA-Schnittstelle.

Die Datei wb_core.vhd enthält das Wishbone-Interface der ZPU. Sie setzt den Adress-, den Datenbus und die Steuersignale der ZPU in die Wishbone-Kommunikation um. Die ZPU selbst ist in der Datei zpu_core.vhd enthalten. Der Baustein ZPU enthält die Zustandsmaschine dieser, den Speicherzugriff mit Interpretation des Opcodes, sowie den Zugriff auf externe

Peripherie mittels des Wishbone-Interfaces (s. Kapitel 3). Sie bindet den als Programmspeicher und Stack genutzten BRAM mit ein (Datei zpu_bram.vhd und softcore.bmm).

Alle Dateien im Projekt mit der Endung _pgk.vhd beinhalten die Festlegung von Konstanten und Definitionen von Komponenten. Speziell die Datei zpu_pkg.vhd enthält die wertemäßige Definition der Opcodes der ZPU.

Die Dateien mit der Endung _config.vhd beinhalten die Festlegung von Konstanten, also global zu verwendende Werte. In der softcore_config.vhd kann z.B. die CPU-Taktfrequenz und die Baudrate des UART festgelegt werden. In der zpu_config.vhd die Adressbreiten, Datenbreiten, Stackgröße und Programmspeichergröße.

Die Datei interrupt.vhd enthält den Interrupt-Controller und sein Wishbone-Interface. Zusätzlich zu den Wishbone-Signalen hat er Eingänge für die Interrupts anderer Peripherie-Geräte und einen Ausgang, um den Interrupt der ZPU zu setzen. Wenn über letzteren der ZPU ein Interrupt mitgeteilt wird und diese ihn annimmt (bzw. Interrupt aktiviert hat), sendet der Controller ihr die Interruptnummer. Per Software kann der ZPU mitgeteilt werden, was bei Auftreten eines Interrupt mit diesem gemacht werden soll. Dies bedeutet auch, dass der Software-Programmierer wissen muss, welche Interruptnummer welchem Peripherie-Gerät zugeordnet ist.

Die Datei uart.vhd enthält das Wishbone-Interface des UART. Ihm untergeordnet sind die Bausteine zum Senden (Datei tx_unit.vhd) und Empfangen (Datei rx_unit.vhd). Beide haben jeweils noch einen Timer-Baustein (brgen.vhd), einen so genannten Taktteiler. Diese werden genutzt, um einerseits zyklisch den UART nach empfangenen Daten abzufragen und andererseits synchron dazu über den UART zu senden. Die Timerzeiten berechnen sich hierbei über die Baudrate (in der Datei softcore_config.vhd festgelegt). Beim Eintreffen von Daten wird ein Interrupt ausgelöst, der an den Interrupt-Controller gemeldet wird. Die Bausteine zum Senden und Empfangen sind einfache 8N1 Module von www.opencores.org.

Die Datei ps2.vhd enthält eine ganz einfache PS/2-Schnittstelle mit Wishbone-Interface. Hierzu wird das von der Tastatur gesendete Taktsignal abgefragt und dann das 10 Bit breite Telegramm eingelesen. Auch hier wird mittels Interrupt signalisiert, dass Daten zur Verfügung stehen.

Die Datei intIO.vhd enthält das Wishbone-Interface, um die Schalter, Buttons, LEDs und 7-Segment-Anzeigen auf dem Board zu steuern. Hierbei werden die Schalter und Buttons bei Ansprechen ihrer Adresse direkt eingelesen bzw. die LEDs direkt über die unteren 8 Bit des Datenworts geschaltet. Da die 7-Segment-Anzeigen nicht getrennt unterschiedliche Werte

erhalten können, gibt es eine Zustandsmaschine, die zeitlich nacheinander zwischen den 4 Anzeigen hin- und herschaltet. Hierfür wird aus dem 32 Bit breiten Datenwort je ein Byte pro Anzeige verwendet, wobei das höchstwertigste Byte der linken Anzeige usw. entspricht.

Die Datei wb_sram.vhd enthält das Wishbone-Interface des SRAM-Controllers. Dieser selbst ist in der Datei MemoryController.vhd enthalten. Da der SRAM nur eine Adressbreite von 18 Bit hat, wird das höchstwertigste Adressbit verwendet, um zwischen Lesen und Schreiben zu entscheiden.

Die Datei wb_vga.vhd enthält das Wishbone-Interface für den VGA-Controller. Hier werden im Prinzip zwei Projekte vereinigt. Zum einen eine reine Ansteuerung der Farbkanäle des VGA-Ports, hierzu ein Implementierungsbeispiel in Kapitel 5.2. Und zum anderen die Möglichkeit Text auf dem Bildschirm darzustellen (80x30 Zeichen). Hierbei stellt Bit 3 die Entscheidung zwischen diesen Modi dar, im Falle der Farbansteuerung werden Bit 0 bis 2 für die Farbwerte RGB genutzt. Andernfalls wird im Moment einfach ein statischer Text erzeugt, der am Bildschirm ausgegeben wird (Datei font_rom_pixel_generation.vhd). In der Datei font_rom.vhd ist ein ROM enthalten, dass mit den einzelnen darzustellenden Zeichen gefüllt ist. Für beide Modi wird die Auflösung 640x480 verwendet. Die Datei vga_sync.vhd beinhaltet die Ansteuerung des VGA-Ports, sie generiert letztendlich das Signal, das den Bildschirm abtastet.

3 Aufbau und Funktionsweise der ZPU

Die ZPU funktioniert im Prinzip wie eine Zustandsmaschine, die während der Abarbeitung im Wesentlichen die Phasen Fetch (Befehl holen), Decode (Befehl decodieren) und Execute (Befehl ausführen) durchläuft:

- Fetch:
  In diesem Zustand lädt die ZPU anhand des Program Counters (pc) einen kompletten 32 Bit Wert aus dem Programmspeicher. Hierbei wird der 32 Bit Wert über die oberen 30 Bit des Program Counters adressiert. Es ist zu beachten, dass der Program Counter zwar eine Breite von 32 Bit hat, die eigentliche Adressbreite des Programmspeichers jedoch abhängig von seiner Größe ist (s. u.). Dementsprechend kann eventuell nicht die volle Breite des Program Counters genutzt werden.

- Decode:
  Da die Opcodes nur 8 Bit breit sind, wird anhand der letzten zwei Bits des Program Counters entschieden, welche 8 Bit des 32 Bit breiten Speicherwertes dekodiert werden. So adressiert der Program Counter letztendlich immer 8 Bits. Dies ist ähnlich einer in CPUs genutzten Pipeline. Pipelines werden genutzt, um die Taktrate auf den Programmspeicher zu reduzieren. Hier würde durch den 32 Bit Speicherwert die CPU mit einem 8 Bit breiten Opcode 4x schneller takten können, als die maximale Taktrate des Speichers es erlaubt. Realisiert ist die ZPU jedoch so, dass sie in jedem Fetch-Zyklus auf den Speicher zugreift und einen 32 Bit Wert holt und dann nur 8 Bit nutzt. Insofern ist die Speicherzugriffstaktrate gleich der des Fetch-Zyklus.

- Execute:
  Abhängig vom dekodierten Opcode springt die ZPU in der Execution-Phase in verschiedene Zustände, um die Instruktionen abzuarbeiten. Des Weiteren wird der Program Counter um Eins erhöht.[10]

Nach dem Fetch-State existiert zusätzlich noch ein Fetch-Next-State. Dies bedeutet, dass mindestens 4 Takte benötigt werden, um einen Befehl abzuarbeiten. Bei der Taktfrequenz der ZPU von 50 MHz ergibt dies eine maximale Frequenz von 12,5 MHz.

Die ZPU besitzt direkt untergeordnet eine BlockRAM-Einheit, die sie als 32 Bit breiten Programmspeicher verwendet. Hierbei handelt es sich um einen Dualport-BlockRAM: Über bei-

[10] Vgl. [8] MEIER, Matthias: Merkmalbasierte statische Konfigurierung von MPSoCs.

de Ports kann gleichzeitig auf die gleiche Adresse schreibend oder lesend zugegriffen werden. Jedoch nicht von beiden Ports gleichzeitig auf die gleiche Adresse schreibend. Außer dem Programm ist hier auch der Stack abgelegt, also auch der Datenspeicher. Der Speicher hat im Moment eine Größe von 16 KB.

Im VHDL-Code entspricht die Struktur der ZPU einer Entity, die den ZPU Core integriert und nach außen hin eine Wishbone-Anbindung zur Verfügung stellt. Die Entity des ZPU Cores wiederum implementiert den Dualport-RAM.

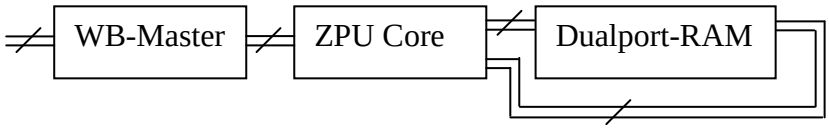


Abbildung 3: Schema ZPU-Struktur

State-Diagramm der ZPU ohne Berücksichtigung der einzelnen Bearbeitungsschritte:

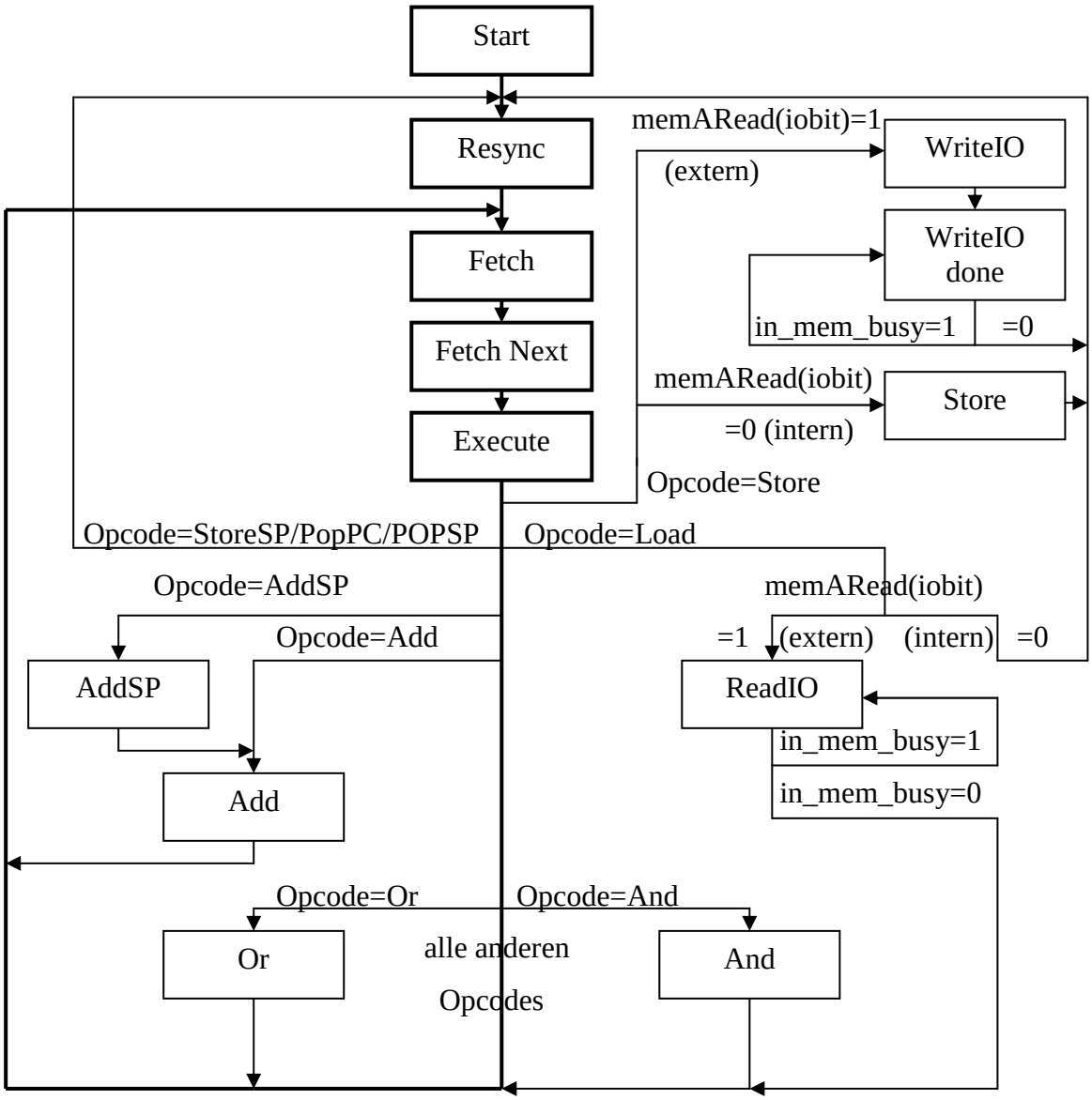


Abbildung 4: State-Diagramm der ZPU

4 Wishbone-Bus

Der Wishbone-Bus ist ein Opensource Hardware Computerbus, über den die Einheiten einer integrierten Schaltung miteinander kommunizieren können. Wishbone ist ein logischer Bus, er definiert also nicht elektrische Informationen. Stattdessen definiert die Spezifikation Signale, Taktzyklen und High- und Low-Pegel. Dies macht es so einfach, ihn in VHDL zu verwenden. Alle Signale sind dabei synchron zum Taktsignal.

Für die Studienarbeit wurde die 32 Bit Variante des Wishbone-Busses verwendet (32 Bit Adress- und 32 Bit Datenbreite). Als Topologie wird der Shared Bus verwendet, d.h. alle Teilnehmer sitzen am selben Adress- und Datenbus und es existiert nur ein Master, nämlich die ZPU. Sollten mehrere Slaves angeschlossen sein, wird anhand der auf dem Adressbus liegenden Adresse der entsprechende Slave aktiviert. Alle Peripherie-Geräte sind als Slaves angesetzt. Ein Beispiel für eine Verbindung vom Master zu einem Slave ist in Abbildung 5 zu sehen. Hierbei enthält der Block SysCon den Taktgenerator des FPGAs und eine Verbindung zum Reset-Button. Eine Erklärung der Signale folgt in Tabelle 1.

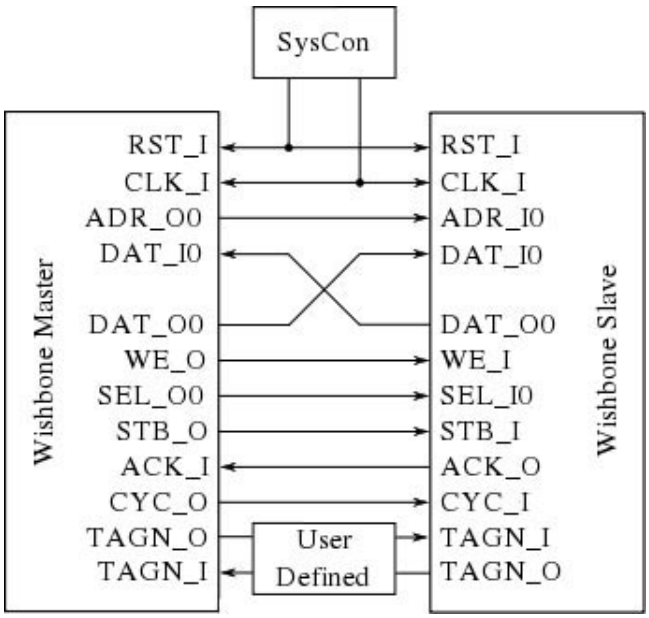


Abbildung 5: Wishbone-Bus[11]

[11]Vgl. [9] http://en.wikipedia.org/wiki/Wishbone_%28computer_bus%29

RST_I	Synchrones Resetsignal
CLK_I	Synchrones Taktsignal
ADR_O0, ADR_I0	32 Bit Adressbus
DAT_O0, DAT_I0	32 Bit Datenbus für Daten vom Master zum Slave
DAT_I0, DAT_O0	32 Bit Datenbus für Daten vom Slave zum Master
WE_O, WE_I	WE kennzeichnet, ob der aktuelle Buszyklus ein Schreib- oder Lesezyklus ist. WE ist negiert während des Lesezyklus.
SEL_O, SEL_I	4 Bit breiter Selectbus. Jedes Bit aktiviert ein Byte aus dem übermittelten 32 Bit Datenwort. Beispielsweise für SEL_O = 1111 sind alle 32 Bit aktiviert, für SEL_O = 0011 nur die unteren 16 Bit.
STB_O, STB_I	STB zeigt einen gültigen Datentransferzyklus an. Wird vom Master gesetzt, um anzuzeigen, dass eine gültige Adresse auf dem Adressbus liegt.
ACK_I, ACK_O	ACK markiert das Ende eines Buszyklus durch einen Slave, also Write complete oder Read-Daten vorhanden.
CYC_O, CYC_I	Definiert den Beginn und das Ende eines Buszyklus. Der Master setzt das Signal um einen neuen Bustransfers zu initiieren.
TAGN_O, TAGN_I	Der Tag-Bus wird in diesem Projekt nicht genutzt.

Tabelle 1: Signale des Wishbone-Busses[12]

[12] Vgl. [10] DON ; JJEZAK: Wishbone Interconnect und
vgl. [9] http://en.wikipedia.org/wiki/Wishbone_%28computer_bus%29

1. Beispiel für ein Wishbone-Output-Interface

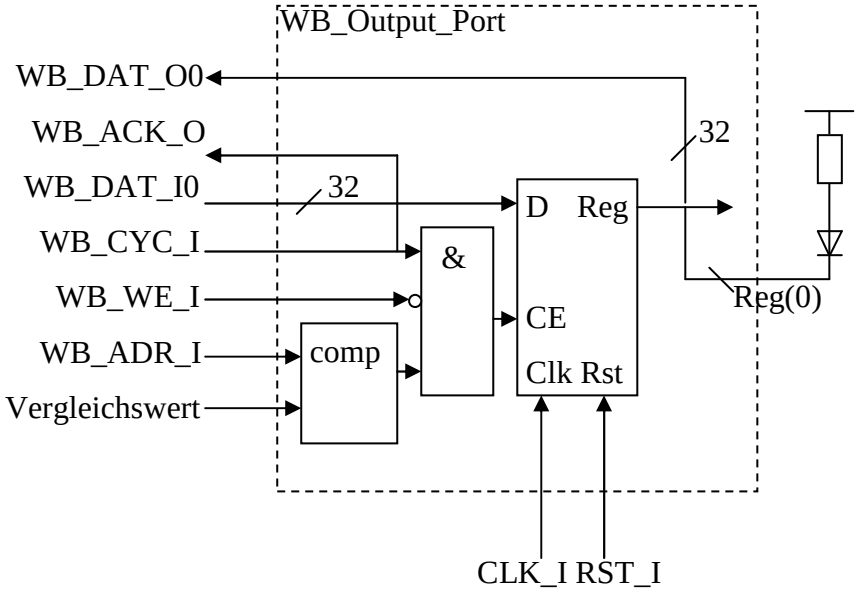


Abbildung 6: Wishbone-Output-Interface[13]

2. Beispiel für ein Wishbone-Input-Interface

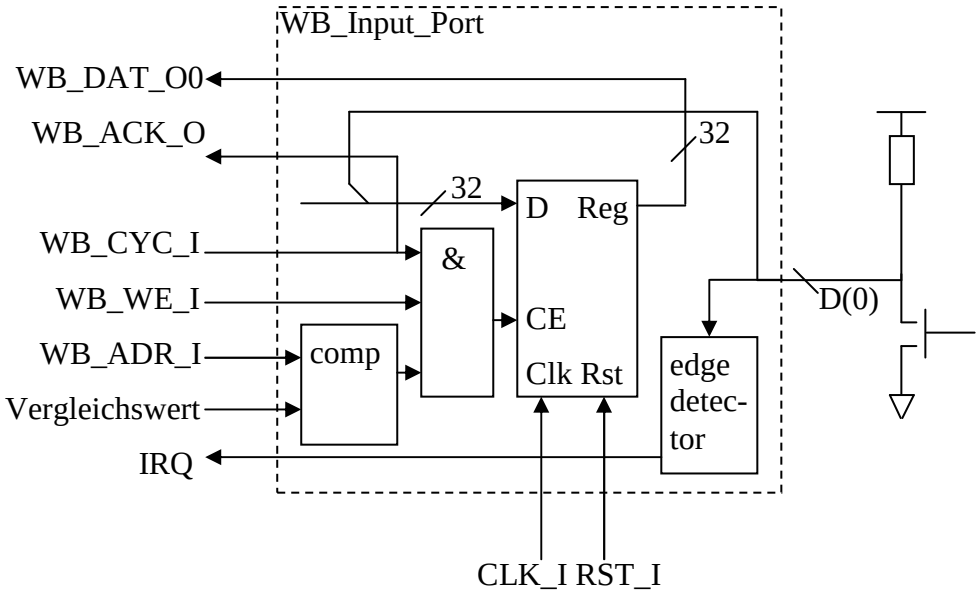


Abbildung 7: Wishbone-Input-Interface[14]

[13] Vgl. [11] http://www.armadeus.com/wiki/index.php?title=A_simple_design_with_Wishbone_bus

[14] Vgl. [11] http://www.armadeus.com/wiki/index.php?title=A_simple_design_with_Wishbone_bus

5 Peripherie

5.1 Integrierte Peripherie

Während der Studienarbeit wurden im Folgenden genannte Peripherie-Geräte an den Wishbone-Bus angeschlossen. Die Adressen für die Interfaces sind hart kodiert und lassen sich durch Anpassung im softcore_top.vhd anpassen.

1. RS-232-Schnittstelle:

   Einen UART mit 8 Datenbits, 1 Stopbit, Parity none und 115200 Baud (in der Datei softcore_config.vhd festgelegt). Bei Eintreffen eines Zeichens am UART wird ein Interrupt ausgelöst. Über Abfragen der RX-Adresse des UART kann das empfangene Zeichen dann ausgelesen werden.

   RX-Adresse: 0x80101004

   TX-Adresse: 0x80101008

2. Internal I/Os:

   Anbindung der 8 Schalter, 4 Taster, 8 LEDs und vier 7-Segment-Anzeigen auf dem Board. Eine Abfrage der Adressen der Schalter und Taster liest den Zustand der Schalter (8 Bit) bzw. Taster (4 Bit) ein. Ein Senden eines 8 Bit Wertes an die LED-Adresse schaltet die LEDs. Ein Senden eines 32 Bit Wertes an die Adresse der 7-Segment-Adresse schaltet die 4 Anzeigen. Hierbei setzt das höchstwertigste Byte die Anzeige links und das zweithöchstwertigste die zweite Anzeige von links usw. Welches Bit des jeweiligen Bytes dann welches Segment schaltet, findet sich in Abbildung 8.

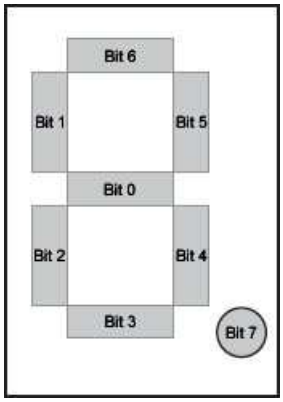


Abbildung 8: 7-Segment-Anzeige[15]

[15] Quelle: Eigene Abbildung.

Adresse der Taster: 0x80080000
Adresse der Schalter: 0x80080001
Adresse der LEDs: 0x80080002
Adresse der 7-Segment-Anzeigen: 0x80080003

3. PS/2-Schnittstelle:

   Eine einfache Tastatur-PS/2-Schnittstelle, die einen Interrupt auslöst, wenn eine Taste auf der Tastatur gedrückt wird. Hierbei wird der Scancode der Tastatur übermittelt.
   Adresse der PS/2-Schnittstelle: 0x80100000C

4. SRAM-Speicher:

   Einer der beiden board-internen SRAM-Speicher erhielt ein Wishbone-Interface. Der SRAM hat eine Datenbreite von 16 Bit und eine Adressbreite von 18 Bit. Zum Schreiben an den SRAM werden zuerst an die Data-In-Adresse die Daten geschrieben und dann an die SRAM-Address-Adresse die 18 Bit breite Adresse geschrieben. Da die Wishbone-Adressen 32 Bit breit sind, wird das höchstwertigste Bit (Bit 31) genutzt, um den Lese- oder Schreibmodus zu setzen. Geschrieben wird wenn Bit 31 = 0 und gelesen für Bit 31 = 1.
   Adresse für SRAM-Adresse: 0x800C0000
   Adresse um Daten zu schreiben: 0x800C0001
   Adresse um Daten zu lesen: 0x800C0002

5. VGA-Schnittstelle:

   Die VGA-Schnittstelle wurde mit zwei simplen Anwendungsmöglichkeiten integriert, die im Moment bereits zwei Grundfunktionalitäten des Bildschirms beherrscht. Zum einen ist es möglich, über einen 3 Bit Wert die Farben des Bildschirms zu setzen (8 Farben). Durch setzen des nächst höheren Bits wird über eine Zustandsmaschine aus einem ROM der die Zeichen enthält ein statischer Text angezeigt. Beides in üblicher Standard-Auflösung von 640x480. Mehr dazu in Kapitel 5.2.
   Adresse der VGA-Schnittstelle: 0x800E0000

Des Weiteren ist ein Interrupt-Controller als Wishbone-Slave integriert. An diesen werden die Interrupts der Wishbone-Slaves weitergereicht und dieser teilt der ZPU dann mit, von welcher Adresse der Interrupt kam. Jeder Interrupt muss eine eigene IRQ-Leitung besitzen, hierfür ist bei Hinzufügen eines weiteres Wishbone-Slaves mit Interrupt die Konstante irqLines in der Datei softcore_top.vhd anzupassen. Standardmäßig sind alle Interrupts deaktiviert. Sie können über die IR-Control-Adresse (0x8000000C) in der Software als vorhanden gekennzeichnet

und dann über die IR-Enable-Adresse (0x80000008) aktiviert werden. Wenn ein Interrupt ausgelöst wird, wird die Funktion *_zpu_interrupt ()* aufgerufen. Welcher Interrupt gesetzt ist, lässt sich über die Interrupt-Adresse 0x80000000 abfragen.

5.2 Weitere Peripherie erstellen

Auf www.opencores.org gibt es viele frei verfügbare Peripherie-Geräte mit Wishbone-Schnittstelle zum downloaden. Die Schnittstelle muss trotzdem meist etwas angepasst werden. In diesem Kapitel wird anhand eines Beispiels vorgestellt, wie ein Peripherie-Gerät von Grund auf erstellt und mittels Wishbone-Interface angeschlossen wird. Mit diesem Wissen ist es möglich sein, auch bereits vorgefertigte Bausteine, ob mit oder ohne Wishbone-Schnittstelle, an die ZPU anzuschließen. Dieses Beispiel implementiert einen VGA-Controller.[16]

Sinnvoll ist es, einen Baustein zu verwenden, der bereits funktioniert, also in einem eigenen Projekt getestet wurde. So kann man diese Fehlerquelle beim Anbinden schon einmal ausschließen.

5.2.1 Hardware

Die Wishbone-Signale sind in Record Types zusammengefasst, die in der Datei softcore_pkg.vhd definiert sind. Für die Eingangssignale zum Master gibt es die Struktur *wb_master_in*, sie setzt sich wie folgt zusammen:

```
type wb_master_in is record
      data        : std_logic_vector(31 downto 0);
      ack         : std_logic;
end record;
```

Für die Ausgangssignale vom Master gibt es folgende Struktur *wb_master_out*:

```
type wb_master_out is record
      addr        : std_logic_vector(31 downto 0);
      data        : std_logic_vector(31 downto 0);
      sel         : std_logic_vector(3 downto 0);
      we          : std_logic;
      stb         : std_logic;
      cyc         : std_logic;
end record;
```

Somit reicht es einem Wishbone-Interface als Eingänge das Takt-, das Resetsignal und ein Signal vom Typ *wb_master_out* zu verpassen (*slave_in*), sowie einen Ausgang vom Typ

[16] Autor des VGA-Controllers ohne Wishbone-Anbindung: Dennis Trebbels

wb_master_in (*slave_out*). Je nach Bedarf kann man noch ein Interruptsinal als Ausgang und in der .ucf-Datei gemappte Ein- oder Ausgangssignale definieren.

Das VGA-Interface bekommt die notwendigen Signale für die Wishbone-Kommunikation und ein Interruptsignal, das jedoch im Moment nicht genutzt wird und insofern auf 0 gelegt ist. Des Weiteren die Signale die auf den VGA-Port gemappt werden: Ein Signal *hsync* für die horizontale Synchronisation, ein Signal *vsync* für die vertikale Synchronisation und ein dreikanaliges Signal *rgb* für die drei Farbkanäle. Damit lassen sich dann 8 Farben darstellen.

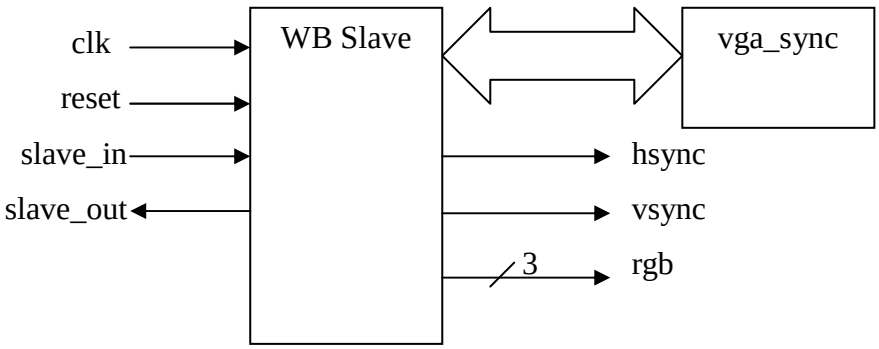


Abbildung 9: VGA-Interface Schema

Zuerst wird ein neues VHDL-Modul angelegt mit dem Namen vga_sync.vhd. Der Quellcode findet sich im Anhang B. Nun ein weiteres VHDL-Modul mit dem Namen wb_vga.vhd. Die hier zu erstellende Entity trägt den Namen *wb_vga* und wird der Wishbone-Slave. Er erhält alle Ein- und Ausgänge, wie in Abbildung 9 gezeigt. Damit die Strukturen *wb_master_in* und *wb_master_out* genutzt werden können, wird nach dem Einbinden der Standardbibliotheken noch die Bibliothek *work* eingebunden zusammen mit *use work.softcore_pkg.all*.

Bis jetzt sieht der Baustein *wb_vga* wie folgt aus:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.softcore_pkg.ALL;

entity wb_vga is
      port (
            clk, reset        : in std_logic;
            interrupt         : out std_logic;
            slave_in          : in wb_master_out;
            slave_out         : out wb_master_in;
            hsync, vsync      : out std_logic;
            rgb               : out std_logic_vector(2 downto 0)
      );
end wb_vga;

architecture Behavioral of wb_vga is
begin
end Behavioral;
```

Zunächst wird jetzt dieser leere Wishbone-Slave angeschlossen. Dies bedeutet, die 3 Signale für den VGA-Port bis zur obersten Ebene durchzuschleifen und den Slave an den Bus anzuschließen. Dazu wird in der .ucf-Datei folgendes hinzugefügt:

```
#=========================================================
# VGA outputs
#=========================================================
NET "ioVGA_rgb<2>"  LOC = "R12" | DRIVE=8 | SLEW=FAST;
NET "ioVGA_rgb<1>"  LOC = "T12" | DRIVE=8 | SLEW=FAST;
NET "ioVGA_rgb<0>"  LOC = "R11" | DRIVE=8 | SLEW=FAST;
NET "ioVGA_vsync"   LOC = "T10" | DRIVE=8 | SLEW=FAST;
NET "ioVGA_hsync"   LOC = "R9"  | DRIVE=8 | SLEW=FAST;
```

Damit sind die Signale vom VGA-Port im FPGA angeschlossen. Die oberste Ebene ist die Entity *softcore_top* in der softcore_top.vhd. Hier werden diese Signale als weitere Ports hinzugefügt:

```
ioVGA_hsync, ioVGA_vsync: out std_logic;
ioVGA_rgb: out std_logic_vector(2 downto 0);
```

Jetzt muss noch die Datei softcore_pkg.vhd angepasst werden. Hier werden die Komponenten definiert. Deswegen muss hier ebenfalls der neue Baustein *wb_vga* hinzugefügt werden, damit er verwendet werden kann:

```
component wb_vga is

      port (
            clk, reset        : in std_logic;
            interrupt         : out std_logic;
            slave_in          : in wb_master_out;
            slave_out         : out wb_master_in;
            hsync, vsync      : out std_logic;
            rgb               : out std_logic_vector(2 downto 0)
      );
end component;
```

Jetzt kann der Baustein *wb_vga* in der Entity *softcore_top* eingebunden werden. Hierfür wird wie bei den anderen Wishbone-Slaves auch ein Portmapping durchgeführt. Vorher müssen jedoch die notwendigen Signale definiert werden. In der *architecture* der *softcore_top* (in softcore_top.vhd) wird vor *begin* folgendes eingefügt:

```
signal vga_out    : wb_master_in;
signal vga_in     : wb_master_out;
signal irq_vga    : std_logic;
```

Diese Signale dienen zur Ankopplung des Wishbone-Busses an *wb_vga* und zur Ankopplung des Interrupt-Signals des *wb_vga* an den Interrupt-Controller. Um den Interrupt anschließen zu können, muss der Interrupt-Controller eine weitere Interrupt-Leitung erhalten. Hierfür die Konstante *irqLines* um 1 erhöhen. Nun kann die neue Interrupt-Leitung an den Interrupt des *wb_vga* angeschlossen werden (Beispiel für neuen Interrupt mit der Nummer 4):

```
irq(4) <= irq_vga;
```

Das Portmapping kann im Prinzip von einem anderen Slave kopiert werden und an *wb_vga* angepasst werden:

```
vga_inst: wb_vga
      port map (
            clk => clk,
            reset => reset,
            interrupt => irq_vga,
            slave_in => vga_in,
            slave_out => vga_out,
            hsync => ioVGA_hsync,
            vsync => ioVGA_vsync,
            rgb => ioVGA_rgb
      );
```

Die Signale *hsync*, *vsync* und *rgb* sind bereits bis auf die oberste Ebene durchgeschleift und dann an den VGA-Port angeschlossen, der Interrupt, das Takt- und das Resetsignal sind ebenfalls angeschlossen. Jetzt fehlt nur noch der Anschluss an den Wishbone-Bus über die Signale *vga_in* und *vga_out*. Diese sind zwar definiert und auf der einen Seite an *wb_vga* angeschlossen, jedoch noch nicht an den Wishbone-Bus.

Hier muss erst einmal eine freie Adresse ausgewählt werden, über die der VGA-Controller letztendlich von der ZPU angesteuert werden soll, beispielsweise 0x800E0000. Die Daten vom VGA-Controller zum Master gehen über das Signal *vga_out* an das Signal *master_in*. Wenn nun von der ausgewählten Adresse gelesen wird, muss *vga_out* auf *master_in* gemappt werden. Hierfür kann man einfach den Multiplexer am Ende der softcore_top.vhd wie folgt erweitern:

```
master_in <=
      irq_out    when  (master_out.addr(31  downto  11)  =  "1000"  &  '0'  &
std_logic_vector(to_unsigned(0, (26 - 11 + 1)))) else
      uart1_out  when  (master_out.addr(31  downto  11)  =  "1000"  &  '0'  &
"0000001000000010") else
      ps2_out    when  (master_out.addr(31  downto  11)  =  "1000"  &  '0'  &
"0000001000000000") else
      intIO_out  when  (master_out.addr(31  downto  11)  =  "1000"  &  '0'  &
"0000000100000000") else
      sram_out   when  (master_out.addr(31  downto  11)  =  "1000"  &  '0'  &
"0000000110000000") else
      vga_out    when  (master_out.addr(31  downto  11)  =  "1000"  &  '0'  &
"0000000111000000") else
      dummy_out;
```

Die Adresse *„1000“ & ‚0’ & „0000000111000000“* entspricht 0x800E0000.

Ebenso muss jetzt noch *vga_in* angeschlossen werden:

```
vga_in.addr <= master_out.addr;
      vga_in.data <= master_out.data;
      vga_in.sel  <= master_out.sel;
      vga_in.we   <= master_out.we;
      vga_in.stb  <= master_out.stb;
      vga_in.cyc  <=  master_out.cyc  when  (master_out.addr(31  downto  11)  =
"1000" & '0' & "0000000111000000") else '0';
```

Nun ist die *wb_vga* komplett angeschlossen. Beispielsweise kann man sie so füllen, dass man die drei RGB-Werte per ZPU steuern kann. Hierfür wird die Entity *vga_sync* eingebunden. Diese steuert das analoge VGA-Signal. Wie ein VGA-Bildschirm genau angesteuert wird, wird hier nicht erläutert.[17] Des Weiteren werden unbenötigte Signale auf 0 gesetzt, wie das Interruptsignal und *slave_out.data*, also der Datenbus zum Master. ACK wird ganz einfach auf CYC gesetzt (s. Kapitel 4). Dann wird noch ein Prozess benötigt, der die vom Master geschickten Daten auswertet und entsprechend die RGB-Farben setzt. Die *architecture* von *wb_vga* sieht dann wie folgt aus:

[17] Vgl. [12] ZIMMER, Tobias: Ansteuerung des VGA-Anschlusses über Atmega.

```
architecture Behavioral of wb_vga is

	signal rgb_reg: std_logic_vector(2 downto 0);
	signal video_on: std_logic;
begin
	-- instantiate VGA sync circuit
	vga_syn_unit: entity work.vga_sync
		port map(clk=>clk, reset=>reset, hsync=>hsync, vsync=>vsync,
			video_on=>video_on, p_tick=>open, pixel_x=>open,
			pixel_y=>open);

	interrupt <= '0';

	slave_out.ack <= slave_in.cyc;

	slave_out.data <= (others => '0');

	-- rgb buffer
	process (clk, reset)
	begin
		if (reset='1') then
			rgb_reg <= (others=>'0');
		elsif (clk'event and clk='1' and slave_in.cyc = '1' and
slave_in.we = '1') then
			rgb_reg <= slave_in.data(2 downto 0);
		end if;
	end process;
	rgb <= rgb_reg when video_on='1' else "000";
end Behavioral;
```

5.2.2 Software

Ein C-Programm für die ZPU kann jetzt die Adresse des VGA-Controllers als *volatile int* Zeiger definieren. Wenn man die Zustände der Taster am Board an den VGA-Controller schickt, kann man dann darüber die Bildschirmfarbe steuern. Mehr zur Programmierung im Kapitel 6.

```
/* IO-Adrressen */
volatile int* intIOBTN = (volatile int*)0x80080000;
volatile int* vga      = (volatile int*)0x800E0000;

int main()
{
	while(1)
	{
		*vga = *intIOBTN;
	}

	return 0;
}
```

6 Programmierung der ZPU

Die ZPU lässt sich in C bzw. C++ programmieren. Zum Erstellen der Quellcodes kann z.B. Notepad++ verwendet werden. Zum Kompilieren wird die spezielle ZPUGCC Toolchain verwendet. Diese ist jedoch für Linux gedacht. Zur Studienarbeit stand nur ein Windows-Rechner zur Verfügung. Deswegen wurde auf diesem Cygwin installiert und unter Cygwin die ZPUGCC in Betrieb genommen. Wie Cygwin installiert, welche Komponenten für die Programmierung erforderlich sind und wie die ZPUGCC auf diesem in Betrieb genommen werden kann, findet sich in Kapitel 6.3.

6.1 Ansprechen von Hardware

Die absolute Hardware-Adresse kann über Zeiger des Typs *volatile int* gesetzt werden. Als Beispiel hier die Definition der Adressen für die Schalter und die LEDs auf dem Board. Als Wert wird direkt die Adresse angegeben, auf die der Hardwaremultiplexer in der Datei softcore_top.vhd überprüft.

```
volatile int* intIOSW  = (volatile int*)0x80080001;
volatile int* intIOLED = (volatile int*)0x80080002;
```

Über Dereferenzierung des Zeigers *intIOSW* lässt sich der Zustand der Schalter auslesen. Ebenso lässt sich über das Dereferenzieren des Zeigers *intIOLED* der Zustand der LEDs setzen.

```
while(1)
{
      *intIOLED = *intIOSW; // Liest den Wert der Schalter aus und setzt
                            // dementsprechend die LEDs.
                            // die while-Schleife dient dazu, dauerhaft den
                            // Zustand der Schalter auf die LEDs zu
                            // spiegeln
}
```

6.2 Interrupts

Für die Nutzung der Interrupts müssen diese zuerst aktiviert werden. Ab dem Zeitpunkt der Aktivierung reagiert die ZPU auf diese.

```
volatile int* interrupt_enable = (volatile int*)0x80000008;
volatile int* interrupt_ctrl   = (volatile int*)0x8000000c;

void init(void)
{
      // Interrupts aktivieren
      *interrupt_ctrl = 0x07; // Interrupt 1, 2 und 3 (Bit 0, 1 und 2)
                              // setzen
      *interrupt_enable = 0x07; // Interrupt 1, 2 und 3 (Bit 0, 1 und 2)
                                // aktivieren
}

int main()
{
      init();
};
```

Um die Interrupts Abfragen zu können, muss die Funktion *_zpu_interrupt ()* verwendet werden. Ein Beispiel ist in folgendem Code zu sehen:

```
volatile int* interrupt_stat = (volatile int*)0x80000004;

void _zpu_interrupt(void)
{
      int interrupt = *interrupt_stat;
      if ((interrupt & 0x04) != 0)
      {
            // Interrupt 3 (Bit 3) ausgelöst.
      }

      if ((interrupt & 0x02) != 0)
      {
            // Interrupt 2 (Bit 2) ausgelöst.
      }
}
```

Mit 32 Bit Datenbreite sind so maximal 32 Interrupts möglich, da jedes Bit einen Interrupt präsentiert. Mit den if-Anweisungen in dem Code-Beispiel werden die einzelnen Bits isoliert und dann abgefragt.

6.3 Kompilieren

Kompiliert wird der Quellcode mit ZPUGCC. Hierbei wird eine .elf-Datei erstellt, die direkt in den BlockRAM geladen werden kann.

1. Cygwin herunterladen und installieren:

   Das Setup zu Cygwin kann unter http://www.cygwin.com/ heruntergeladen werden.[18] *Setup.exe* öffnen und auf *Weiter* klicken, *Download from Internet* markieren und auf *Weiter* klicken. Das Installationsverzeichnis festlegen und auf *Weiter* klicken. Nun das Verzeichnis festlegen, in das die für die Installation benötigten temporären Packages abgelegt werden sollen, wieder mit *Weiter* bestätigen. Dann *Direct Connection* wählen und auf *Weiter* klicken. Einen der Download-Mirrors auswählen und wieder auf *Weiter* klicken. Nun wird im Dialog eine Auswahl der zu installierenden Pakete angezeigt. Hier *Devel* anklicken, es öffnet sich die Einzelauflistung. Hier alle *GCC*-Einträge für C und C++ auswählen (einmalig Skip anklicken). *GDB*, *GIT*, *LIBGCC* und *MAKE* ebenso. Dann mit *Weiter* den Download und die Installation bestätigen. Am Ende auf *Fertigstellen* klicken.

2. ZPUGCC in Betrieb nehmen:

   Der Compiler ZPUGCC kann unter http://opensource.zylin.com/zpudownload.html unter dem Punkt *ZPU binary toolchains - latest stable* für Cygwin heruntergeladen werden.[19] Dieses Archiv im Cygwin-Programmordner (z.B. C:\Programme\cygwin)unter *home* im Benutzerordner entpacken. Das Archiv enthält einen Ordner *install*, diesen einfach in *zpugcc* umändern. Da der Ordner *zpugcc* nun direkt im Benutzerordner liegt, kann man den Compiler nach dem Starten von Cygwin direkt mit `zpugcc/bin/zpu-elf-gcc` aufrufen.

Unter Cygwin folgenden Befehl zum Kompilieren in die Konsole eingeben:

```
[Pfadname an dem ZPUGCC liegt, z.B. zpugcc]/bin/zpu-elf-gcc -Os -phi [Absoluter Pfadname]/[Dateiname].c -o [Absoluter Zielpfadname]/[Zieldateiname].elf -Wl,--relax -Wl,--gc-sections –g
```

Hierbei reicht es, die Haupt-C-Datei anzugeben, weitere Header, auch eigene etc. bindet ZPUGCC automatisch mit ein und kompiliert sie.

Man kann auch eine Textdatei mit der Endung .sh erstellen, in der oben genannter Befehl abgespeichert ist und diese dann unter Cygwin aufrufen. Dies entspricht den Batch-Dateien unter Windows, nur dass die .sh-Dateien nicht mit Doppelklick ausführbar sind. Ausführbar sind diese mit folgendem Kommandozeilenbefehl:

```
sh [Absoluter Pfadname]/[Dateiname].sh
```

[18] Direkter Download-Link: http://cygwin.com/setup.exe

[19] Direkter Download-Link: http://opensource.zylin.com/zpu/zpugcccygwin.tar.bz2

6.4 Programmspeicher laden – Data2MEM und .bmm-Files[20]

Nach dem Erstellen der .elf-Datei durch die Kompilierung muss der Inhalt in den Programmspeicher (BlockRAM) geladen werden. Dieser befindet sich in der Datei zpu_bram.vhd. Abhängig von der Größe des Prozessorsystems und des verwendeten FPGAs kann die Synthese des Systems sehr langwierig sein. Damit nicht für jede Änderung am Programmspeicher der Cores das ganze System neu synthetisiert werden muss, wird von Xilinx das Kommandozeilen-Tool Data2MEM[21] in der Entwicklungsumgebung des ISE zur Verfügung gestellt. Dieses ist standardmäßig bei der Installation von Xilinx ISE dabei. Ebenfalls spart man viel Arbeit, da man nicht jedes Byte einzeln hineinkopieren muss.

Die Daten, die zur Konfigurierung eines FPGAs benötigt werden und damit auch die Programme der Softcores, befinden sich in der so genannten .bit-Datei. In dieser .bit-Datei kann Data2MEM ein altes Programm durch ein Neues ersetzen. Um diese Transformation durchzuführen, benötigt Data2MEM Informationen darüber, wo Daten ersetzt werden sollen und wie diese aufgeteilt werden. Diese Informationen werden im BlockRAM Memory Map File, der so genannten .bmm-Datei, angegeben. Die .bmm-Datei ist eine lesbare Textdatei und hat für die ZPU folgenden Aufbau:

```
ADDRESS_MAP softcore_top PPC405 0

    ADDRESS_SPACE memory1 RAMB16 [0x00000000:0x00003FFF]
        BUS_BLOCK
           core/zpu_core/memory/RAMB16_S4_S4_inst7 [31:28];
           core/zpu_core/memory/RAMB16_S4_S4_inst6 [27:24];
           core/zpu_core/memory/RAMB16_S4_S4_inst5 [23:20];
           core/zpu_core/memory/RAMB16_S4_S4_inst4 [19:16];
           core/zpu_core/memory/RAMB16_S4_S4_inst3 [15:12];
           core/zpu_core/memory/RAMB16_S4_S4_inst2 [11:8];
           core/zpu_core/memory/RAMB16_S4_S4_inst1 [7:4];
           core/zpu_core/memory/RAMB16_S4_S4_inst0 [3:0];
        END_BUS_BLOCK;
    END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

Dem Core stehen 16 KB (4096 x 32 Bit Datenbreite) Programmspeicher zur Verfügung, die in 8 BlockRAMs aufgeteilt werden. Der Adressbereich wird unter *ADDRESS_SPACE* in Bytes angegeben. Die Aufteilung der Programmdaten in die 8 BlockRAMs ist unter *BUS_BLOCK* aufgelistet. Dabei werden nicht jeweils 512 Zeilen von 32 Bit Werten in einem BlockRAM gespeichert, sondern 4096 Zeilen mit jeweils 4 Bit des 32 Bit Wertes.

---

[20] Vgl. [13] XILINX: Data2MEM User Guide,
vgl. [14] XILINX: Using Block RAM in Spartan-3 Generation FPGAs und
vgl. [8] MEIER, Matthias: Merkmalbasierte statische Konfigurierung von MPSoCs.

[21] Vgl. [13] XILINX: Data2MEM User Guide.

Es sind 8 Instanzen vom Typ RAMB16_S4_S4 definiert. Die 16 deutet auf die Gesamtgröße von 16 KB hin. S4 auf die Datenbreite von 4 Bit. Das doppelt verwendete S4 bezeichnet das es ein Dualport-RAM ist. Hieraus folgt, dass 4 K x 4 Bit Breite = 16 KB Gesamtgröße sind. Da 8 Instanzen à 4 Bit breit definiert sind, können diese durch entsprechende Ansteuerung eine 32 Bit Breite ergeben. Es wird also ein Speicher der Größe 4 K und 32 Bit breit erstellt.[22]

Die genauen Instanznamen bzw. Pfade, wie *core/zpu_core/memory/RAMB16_S4_S4_inst7*, sind mit Hilfe des Floorplaners im ISE zu ermitteln. Es beschreibt die ineinander verschachtelten Bausteine: Die Entity *core* bindet die Entity *zpu_core* ein, die Entity *zpu_core* bindet die Entity *memory* ein usw. Und letztendlich sind in der Entity *memory* die 8 Instanzen beschrieben.

Nur mit diesen Angaben lässt sich die .bmm-Datei jedoch noch nicht für die Transformation mittels Data2MEM verwenden, da bisher die Angabe des exakten BlockRAMs fehlt. Hier empfiehlt es sich die Arbeit direkt durch das ISE, genauer gesagt durch Bitgen, erledigen zu lassen. Dazu wird die .bmm-Datei in das ISE-Projekt eingebunden und während des Generate Programming File-Prozesses wird die .bmm-Datei automatisch um die fehlenden Angaben ergänzt. Nach diesem Schritt findet sich im Projekt-Ordner eine neue .bmm-Datei, die im Dateinamen um „_bd“ erweitert ist und nun die Ortsangaben für alle Instanzen in folgender Form enthält:

```
ADDRESS_MAP softcore_top PPC405 0
    ADDRESS_SPACE memory1 RAMB16 [0x00000000:0x00003FFF]
        BUS_BLOCK
           core/zpu_core/memory/RAMB16_S4_S4_inst7 [31:28] PLACED = X0Y1;
           core/zpu_core/memory/RAMB16_S4_S4_inst6 [27:24] PLACED = X0Y5;
           core/zpu_core/memory/RAMB16_S4_S4_inst5 [23:20] PLACED = X1Y3;
           core/zpu_core/memory/RAMB16_S4_S4_inst4 [19:16] PLACED = X0Y3;
           core/zpu_core/memory/RAMB16_S4_S4_inst3 [15:12] PLACED = X0Y0;
           core/zpu_core/memory/RAMB16_S4_S4_inst2 [11:8] PLACED = X0Y2;
           core/zpu_core/memory/RAMB16_S4_S4_inst1 [7:4] PLACED = X0Y4;
           core/zpu_core/memory/RAMB16_S4_S4_inst0 [3:0] PLACED = X1Y2;
        END_BUS_BLOCK;
    END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

Neben der .bmm-Datei muss Data2MEM noch die ursprüngliche .bit-Datei und das Programm als .elf-File übergeben werden. Die .bit-Datei liegt normalerweise auf oberster Ebene des Projektverzeichnisses ab und trägt auch den Namen des Projektes. Mit folgendem Befehl wird die Transformation der .bit-Datei ausgeführt:

[22] Vgl. [14] XILINX: Using Block RAM in Spartan-3 Generation FPGAs.

```
C:\Programme\Xilinx\[Programmversion, z.B. 10.1]\ISE\bin\nt\data2mem -bm
[Absoluter Pfadname]\[Dateiname].bmm -bd [Absoluter Pfadna-
me]\[Dateiname].elf -bt [Absoluter Pfadname]\[Dateiname].bit -o b [Absolu-
ter Pfadname]\[Zieldateiname].bit
```

Speziell für das vorhandene Projekt lauten die Dateinamen:

softcore.bmm

main.elf

softcore_top.bit

softcore_top_fw.bit

Ausgeführt werden kann das konsolenbasierte Data2MEM über die Windows-Konsole. Alternativ kann man eine Batch-Datei mit dem entsprechenden Befehl erstellen. Hierfür den Befehl in eine Textdatei kopieren und die Dateiendung in .bat ändern. Bei Verwendung einer Batch-Datei kann man auch zum Pfad der Batch-Datei relative Pfadangaben verwenden.

Das Ergebnis dieser Transformation ist die neue .bit-Datei softcore_top_fw.bit mit geändertem Programmspeicher. Dieses wird dann statt der softcore_top.bit auf den FPGA geladen. Hierfür kann wie gewohnt die Software iMPACT genutzt werden, um über den JTAG-Adapter die neue .bit-Datei auf den FPGA zu laden.[23]

Kommentare können, wie aus der Sprache C bekannt, mit // oder /* und */ gesetzt werden. Allerdings ist zu beachten, dass Kommentare nur vor dem eigentlichen Code als Kommentare gewertet werden. Ein Kommentar zwischen den Zeilen des Codes erzeugt einen Fehler.

Passend zu der .bmm-Datei muss natürlich auch der zu füllende Speicher in VHDL beschrieben sein. In der Datei zpu_bram.vhd werden die entsprechenden Instanzen definiert. Jede Instanz erhält einen Namen und einen Typ, z.B. *RAMB16_S4_S4*. Dem folgt ein *generic map*, bei dem, egal um welchen Typ es sich handelt, die Konstanten von *INIT_00* bis *INIT_3F* komplett mit 0 gefüllt werden (128 Bit bzw. 32 Zeichen in Hex). Anschließend erfolgt das Portmapping. Hier als Beispiel das Portmapping für einen Dualport-RAM des Typs *RAMB16_S4_S4*. Die auskommentierten Zeilen sind Ports, die nur in der S9er, S18er und S36er Konfiguration vorhanden sind (s. Tabelle 2).

[23] s. Anhang A

```
port map (DOA => memARead(3 downto 0),

          DOB => memBRead(3 downto 0),
          --DOPA => open,
          --DOPB => open,
          ADDRA => memAAddr(addrBitBRAM downto minAddrBit),
          ADDRB => memBAddr(addrBitBRAM downto minAddrBit),
          CLKA => clk,
          CLKB => clk,
          DIA => memAWrite(3 downto 0),
          DIB => memBWrite(3 downto 0),
          --DIPA => "00",
          --DIPB => "00",
          ENA => '1',
          ENB => '1',
          SSRA => '0',
          SSRB => '0',
          WEA => memAWriteEnable,
          WEB => memBWriteEnable);
```

Organization	Memory Depth	Data Width	Parity Width	DI/DO	DIP/DOP	ADDR	Single-Port Primitive	Total RAM Kbits
512x36	512	32	4	(31:0)	(3:0)	(8:0)	RAMB16_S36	18K
1Kx18	1024	16	2	(15:0)	(1:0)	(9:0)	RAMB16_S18	18K
2Kx9	2048	8	1	(7:0)	(0:0)	(10:0)	RAMB16_S9	18K
4Kx4	4096	4	-	(3:0)	-	(11:0)	RAMB16_S4	16K
8Kx2	8192	2	-	(1:0)	-	(12:0)	RAMB16_S2	16K
16Kx1	16384	1	-	(0:0)	-	(13:0)	RAMB16_S1	16K

Tabelle 2: Einteilungsmöglichkeiten des BlockRAMs[24]

		Port A					
		16Kx1	8Kx2	4Kx4	2Kx9	1Kx18	512x36
Port B	16Kx1	_S1_S1					
	8Kx2	_S1_S2	_S2_S2				
	4Kx4	_S1_S4	_S2_S4	_S4_S4			
	2Kx9	_S1_S9	_S2_S9	_S4_S9	_S9_S9		
	1Kx18	_S1_S18	_S2_S18	_S4_S18	_S9_S18	_S18_S18	
	512x36	_S1_S36	_S2_S36	_S4_S36	_S9_S36	_S18_S36	_S36_S36

Tabelle 3: Zuordnung bei Dualport-RAM[25]

[24] Vgl. [14] XILINX: Using Block RAM in Spartan-3 Generation FPGAs, Seite 10

[25] Vgl. [14] XILINX: Using Block RAM in Spartan-3 Generation FPGAs., Seite 10

7 Zusammenfassung und Ausblick

Insgesamt bot das Projekt einen sehr interessanten Einstieg in die Welt der FPGAs und Softcores. Für dieses umfassende Thema wäre ein größerer Zeitraum wünschenswert gewesen. Das erreichte Ergebnis bietet eine sehr gute Grundlage für weitere ausbauende Projekte in diesem Gebiet. Durch die Darstellung des vorliegenden Berichts als eine Art Tutorial bietet es auch die Möglichkeit, mit den erreichten Ergebnissen ohne großen Zeitaufwand den aktuellen Stand nachzuvollziehen und direkt darauf aufzubauen. Für weiterführende Projekte empfiehlt es sich auf das Nachfolger-Board Spartan-3E umzusteigen, um mehr Ressourcen (v.a. Block-RAM) verwenden zu können. Eine Anpassung des vorliegenden Projekts an dieses Board ist sehr leicht möglich.

Erreicht wurde in diesem Projekt die erfolgreiche Implementierung einer 32 Bit Softcore-CPU und deren Anbindung an ein Bussystem, was das Anschließen von Peripheriegeräten ermöglichte. Auch dem letzten Punkt der anfangs vorgenommenen Aufgaben, der Ansteuerung mittels Software in C konnte Rechnung getragen werden. Alle Komponenten lassen sich über die ZPU ansteuern und nutzen.

Über die Implementierung der Beispielperipheriemodule wie UART, PS/2, SRAM und I/Os hinaus wurde sogar eine VGA Schnittstelle von Grund auf entwickelt und angebunden. Sie bietet für weitere Projekte die Möglichkeit, dass die bereits implementierte Textgenerierung (80x30 Zeichen auf 640x480 Bildschirmauflösung) über die Software angesprochen werden kann. Des Weiteren wäre eine variable Bildschirmauflösung möglich.

Ein weiterer Punkt ist die bereits funktionierende und im Ordner „future_use“ abliegende PS/2-Schnittstelle. Dies sind zwei Projekte, einmal für Tastatur und einmal für die Maus. Diese können anstatt der bereits verwendeten PS/2-Schnittstelle eingesetzt werden. Z.B. könnte sie so genutzt werden, dass man per Software entscheiden kann, ob Maus oder Tastatur angeschlossen ist und dementsprechend dann den richtigen Baustein aktivieren.

Eine weitere Implementierungsmöglichkeit bietet der zweite SRAM-Baustein auf der Rückseite des Boards. Da beide eine Datenbreite von 16 Bit haben, könnte man hier der ZPU, die ja mit 32 Bit arbeitet, einen 32 Bit breiten Datenspeicher zur Verfügung stellen.

Mit den eben genannten drei Erweiterungen ist es z.B. möglich, einen MiniPC auf dem FPGA-Board zum laufen zu bringen. Per Software könnte eine Art Minibetriebssystem geschrieben werden, so dass man durch die Benutzerschnittstellen Tastatur/Maus und Bildschirm auch richtig mit dem Benutzer agieren kann.

Literaturverzeichnis

[1] XILINX: *Spartan-3 Starter Kit*. URL http://www.xilinx.com/products/devkits/HW-SPAR3-SK-UNI-G.htm. Einsichtnahme: 11.01.2011. Xilinx.

[2] XILINX: *Spartan-3 FPGA Starter Kit Board User Guide*. URL http://www.xilinx.com/support/documentation/boards_and_kits/ug130.pdf. Aktualisierungsdatum: 20.06.2008. Xilinx.

[3] XILINX: *Programmable Logic Design : Quick Start Handbook*. URL http://www.xilinx.com/publications/products/cpld/logic_handbook.pdf. Aktualisierungsdatum: 12.06.2006. Xilinx.

[4] XILINX: *Spartan-3 FPGA Family Data Sheet*. URL http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf. Aktualisierungsdatum: 04.12.2009. Xilinx.

[5] o. V.: *Digilent XUP USB-JTAG Programming Cable Academic 23272*. URL http://shop.trenz-electronic.de/catalog/product_info.php?cPath=30&products_id=591. Aktualisierungsdatum: 24.07.2009. trenz electronic Online Shop.

[6] XILINX: *Downloads*. URL http://www.xilinx.com/support/download/index.htm. Einsichtnahme: 11.01.2011. Xilinx.

[7] ZYLIN CONSULTING: *Zylin CPU*. URL http://opensource.zylin.com/zpudownload.html. Einsichtnahme: 11.01.2011. Zylin Consulting.

[8] MEIER, Matthias: *Merkmalbasierte statische Konfigurierung von MPSoCs*. URL http://ess.cs.tu-dortmund.de/Teaching/Theses/2009/DA_Meier_2009.pdf. Aktualisierungsdatum: 12.05.2009. Technische Universität Dortmund. – Diplomarbeit

[9] o. V.: *Wishbone (computer bus)*. URL http://en.wikipedia.org/wiki/Wishbone_%28computer_bus%29. Aktualisierungsdatum: 28.01.2011. Wikimedia Foundation Inc.

[10] DON ; JJEZAK: *Wishbone Interconnect*.URL http://kujo.cs.pitt.edu/index.php/Wishbone_Interconnect. Aktualisierungsdatum: 23.11.2009. University of Pittsburgh.

[11] o. V.: *A simple design with Wishbone bus*. URL http://www.armadeus.com/wiki/index.php?title=A_simple_design_with_Wishbone_bus. Aktualisierungsdatum: 29.07.2010. Armadeus Project.

[12] ZIMMER, Tobias: *Ansteuerung des VGA-Anschlusses über Atmega*. URL http://www.uni-koblenz.de/~physik/informatik/ECC/vga.pdf. Aktualisierungsdatum: 12.08.2007. Universität Koblenz.

[13] XILINX: *Data2MEM User Guide*. URL http://www.xilinx.com/itp/xilinx10/books/docs/d2m/d2m.pdf. Einsichtnahme: 11.01.2011. Xilinx. – UG437 (v. 2.0)

[14] XILINX: *Using Block RAM in Spartan-3 Generation FPGAs*. URL http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf. Aktualisierungsdatum: 01.05.2005. Xilinx.

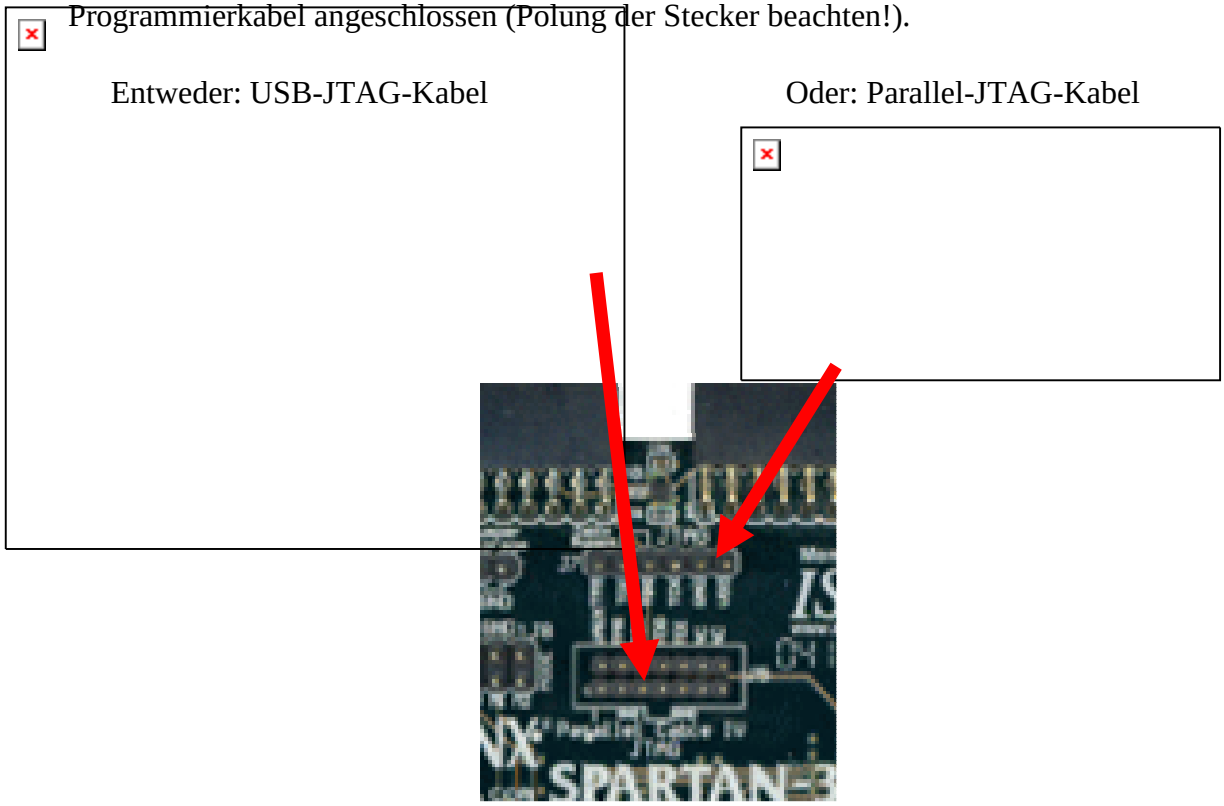
Anhang A - Inbetriebnahme von Board und Entwicklungsumgebung

1. Board anschließen:

   Zur Benutzung des Boards wird die mitgelieferte Spannungsversorgung und das JTAG-Programmierkabel angeschlossen (Polung der Stecker beachten!).

   Entweder: USB-JTAG-Kabel

   Oder: Parallel-JTAG-Kabel



2. Entwicklungsumgebung installieren:

   Download von *Xilinx ISE WebPack 10.1* unter http://www.xilinx.com/support/download/index.htm[26] (s. Abbildung 10).

[26] Vgl. [6] XILINX: Downloads

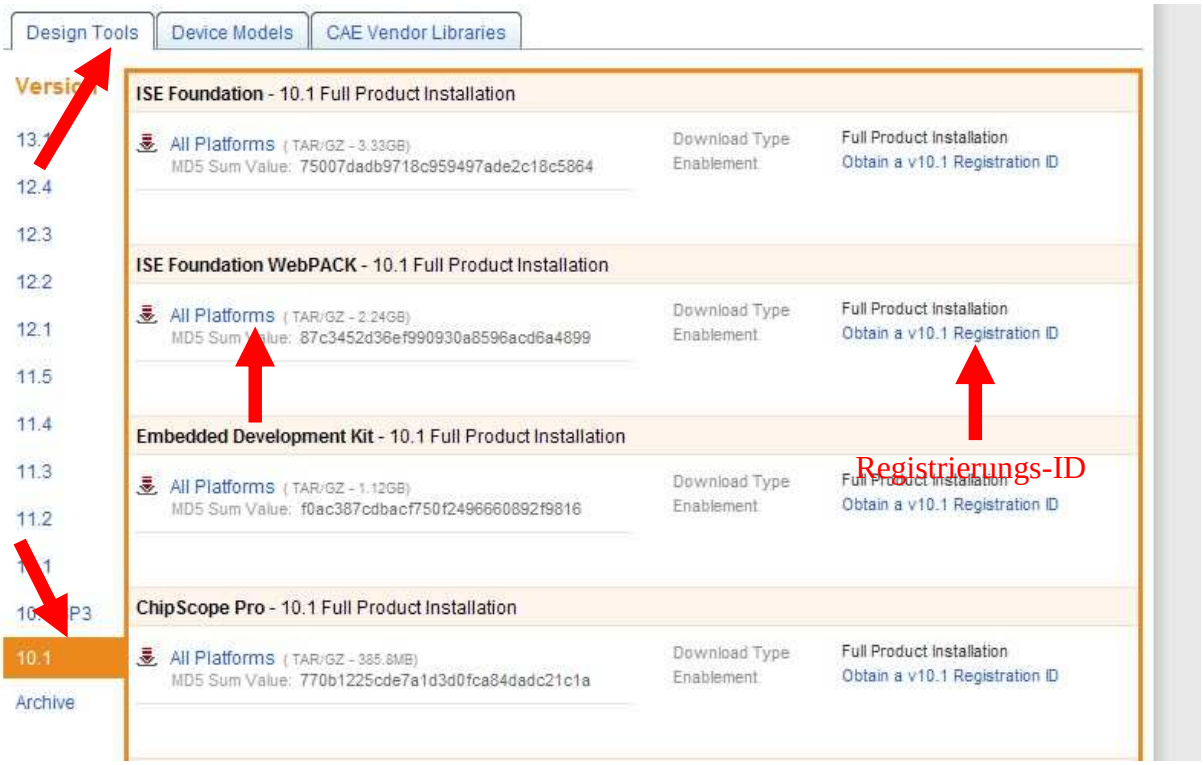


Abbildung 10: Download und Registrierung von Xilinx ISE 10.1

Hierfür ist eine kostenlose Anmeldung bei Xilinx erforderlich. Nach dem Download das Archiv entpacken und installieren. Nach der Installation das Produkt registrieren, um es sinnvoll nutzen zu können (s. Abbildung 10).

3. Erstes Beispielprojekt für das Board in VHDL erstellen:

   - Den *Project Navigator* unter *Xilinx ISE Design Suite 10.1* im Startmenü starten
   - Ein neues Projekt anlegen (Menü *File > New Project*). Im folgenden Dialog einen Projektname und -verzeichnis und als *Top-level source type* HDL auswählen. Achtung: Keine Dateinamen und -Pfade mit Sonderzeichen (wie z.B. Leerzeichen) wählen! Auf *Next* klicken. Nun muss der FPGA korrekt eingetragen werden, in Abbildung 11 sind die Daten für das Spartan-3 Board gezeigt.

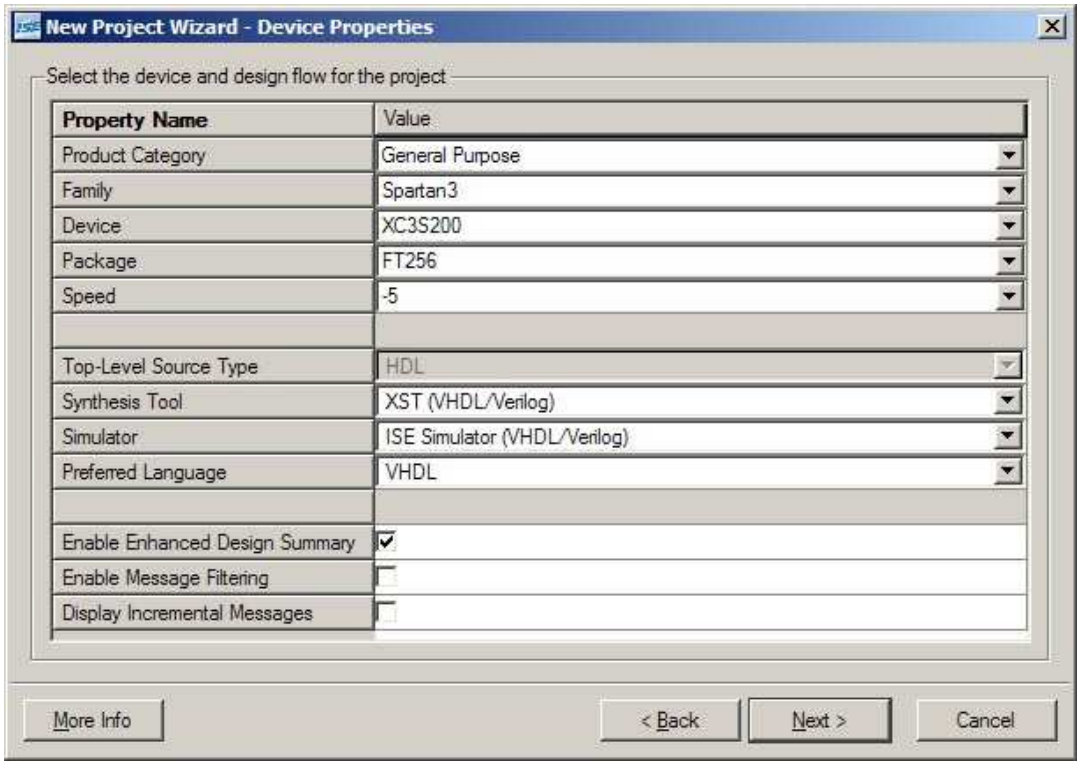


Abbildung 11: Projekteigenschaften für Spartan-3 Board einstellen

- Auf *Next* bzw. *Finish* klicken, bis sich der Dialog schließt. Links im Fenster *Sources* wird jetzt das Projekt angezeigt. Im Moment ist nur der FPGA dem Projekt hinzugefügt. Mit Rechtsklick auf das Projekt und Klick auf *New Source* lässt sich ein neues VHDL-Modul hinzufügen. Hierfür im folgenden Dialog *VHDL Module* auswählen und einen Dateinamen festlegen. Außerdem den Haken vor *Add to project* setzen. Dann auf *Next* klicken.

- Im folgenden Fenster lassen sich die Ein- und Ausgangsignale des Bausteins einstellen, den das VHDL-Modul enthält. Diese für das Beispiel hier wie in Abbildung 12 gezeigt anpassen.

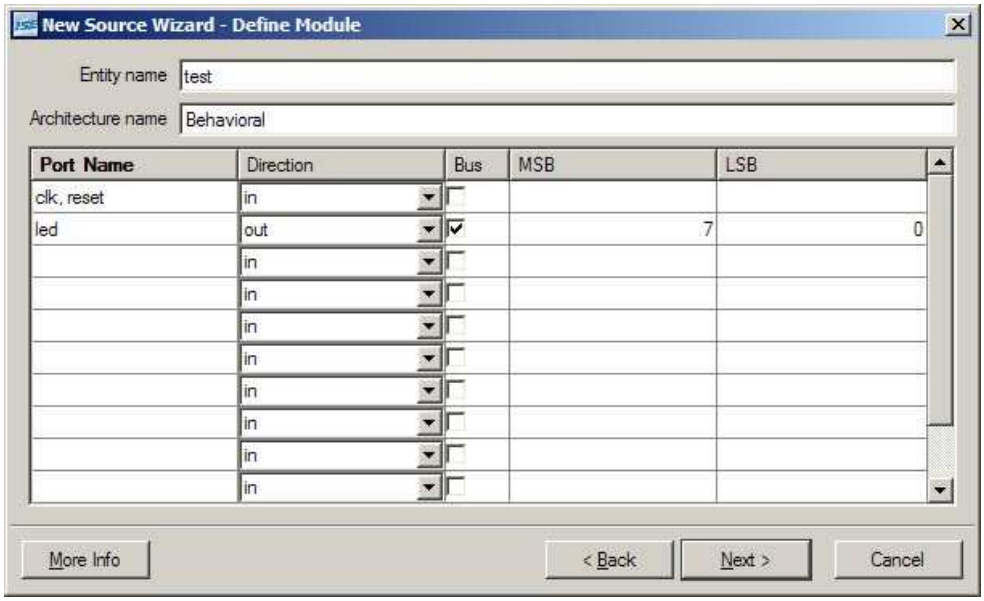


Abbildung 12: Ports einstellen

- Dann auf *Next* bzw. *Finish* klicken, bis sich der Dialog schließt. Nun ist im Fenster *Sources* das eben neu erstellte VHDL-Modul zu sehen. Mit Doppelklick öffnet es sich im Hauptfenster. Der leere Baustein oder Entity sollte nun wie folgt aussehen (bis auf den individuell vergebenen Namen, der hier *test* lautet):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity test is
    Port ( clk, reset : in  STD_LOGIC;
           led : out  STD_LOGIC_VECTOR (7 downto 0));
end test;

architecture Behavioral of test is

begin


end Behavioral;
```

- Als Beispiel soll hier ein Lauflicht implementiert werden, dass die 8 LEDs auf dem Board verwendet. Hierfür wird die *architecture* wie folgt angepasst:

```
architecture Behavioral of test is
signal r_reg, r_next: unsigned (31 downto 0);
signal led_reg, led_next: unsigned (7 downto 0);
signal timer_reg, timer_next: unsigned (7 downto 0);
begin

process (clk,reset)
begin
      if (reset = '1') then
            r_reg <= (others => '0');
            led_reg <= "00000001";
            timer_reg <= (others => '0');
      elsif (clk'event and clk = '1') then
            r_reg <= r_next;
            led_reg <= led_next;
            timer_reg <= timer_next;
      end if;
end process;

process (r_reg, led_reg, timer_reg)
begin
r_next <= r_reg;
led_next <= led_reg;
timer_next <= timer_reg;

if (r_reg >= 25000000) then
      if (timer_reg < 7) then
            led_next <= led_reg(6 downto 0) & led_reg(7);
            timer_next <= timer_reg + 1;
      else
            led_next <= led_reg(0) & led_reg(7 downto 1);
            if (timer_reg = 13) then
                  timer_next <= (others => '0');
            else
                  timer_next <= timer_reg + 1;
            end if;
      end if;

      r_next <= (others => '0');
else
      r_next <= r_reg + 1;
      timer_next <= timer_reg;
end if;

end process;

led <= std_logic_vector(led_reg);

end Behavioral;
```

- Nach dem Speichern des VHDL-Moduls dieses im Fenster *Sources* markieren und dann im Fenster *Processes* unterhalb unter *Synthesis - XST* mit Doppelklick den *Check Syntax* durchführen. Er sollte erfolgreich sein (grüner Haken). Wenn nicht, noch mal den Inhalt des VHDL-Moduls überprüfen.

- Die am Baustein verwendeten Ports müssen nun noch an die Gehäusepins des FPGAs angeschlossen werden. Hierfür wird ein Constraints File benötigt, das man über *New Source* hinzufügt. Diesmal wird *Implementation Constraints File* aus-

gewählt, wieder ein Dateiname eingegeben und der Haken vor *Add to project* gesetzt. Wieder auf *Next* bzw. *Finish* klicken bis sich der Dialog schließt. Jetzt ist vor dem VHDL-Modul im Fenster *Sources* ein Plus erschienen. Wenn man diese anklickt, öffnet sich die untergeordnete Ebene, in diesem Fall ist dass das Constraints File. Dieses markieren und im Fenster *Processes* auf das Plus vor *User Constraints* klicken. Es öffnet sich die untergeordnete Ebene mit *Edit Constraints (Text)*. Hierauf ein Doppelklick öffnet das Constraints File im Hauptfenster zur Bearbeitung. Hier folgenden Code eintragen:

```
#============================================================
#    Pin assignment for Xilinx
#    Spartan-3 Starter board
#============================================================

#============================================================
# clock and reset signals
#============================================================
NET "clk"    LOC = "T9";
NET "reset"  LOC = "L14";

#============================================================
# 8 LEDs on the board
#============================================================
NET "led<0>"  LOC = "K12";
NET "led<1>"  LOC = "P14";
NET "led<2>"  LOC = "L12";
NET "led<3>"  LOC = "N14";
NET "led<4>"  LOC = "P13";
NET "led<5>"  LOC = "N12";
NET "led<6>"  LOC = "P12";
NET "led<7>"  LOC = "P11";

#============================================================
# Timing constraint of S3 50-MHz onboard oscillator
# name of the clock signal is clk
#============================================================
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50 %;
```

- Nun wieder im Fenster *Sources* das VHDL-Modul markieren und dann im Fenster *Processes* Doppelklick auf *Generate Programming File*. Hiermit synthetisiert Xilinx das gesamte Projekt und führt auch vorher die Schritte *Synthesize - XST* und *Implement Design* durch. Es sollten vor allen drei Punkten hinterher ein grüner Haken stehen.

- Doppelklick auf *Configure Target Device* baut die Verbindung mittels JTAG-Adapter zum Board auf. Die Warnungsmeldung vom Programm iMPACT einfach mit *OK* bestätigen. Im folgenden Dialog *Configure devices using Boundary-Scan (JTAG)* auswählen und mit *Finish* bestätigen. Nun sollte das FPGA-Board gefunden werden. Jetzt öffnet sich im Hauptfenster der Boundary Scan mit zwei Ele-

menten. Das linke ist der FPGA, das rechte der ConfigurationFLASH. Letzterer wird nicht benötigt. Es sollte sich ein Dialog öffnen, der die Configuration Files (.bit-Datei) für die zwei Bausteine verlangt. Ansonsten kann der Dialog auch mit Doppelklick auf die Bausteine geöffnet werden. Der FPGA (*xc3s200*) erhält die .bit-Datei im Hauptverzeichnis des Projektes. Der rechte Baustein wird als *Bypass* gewählt. Im nachfolgenden Dialog darauf achten, dass der Haken vor *Verify* entfernt ist und dann mit *OK* bestätigen. Nun kann mit Rechtklick auf den FPGA und *Program* das Projekt auf den FPGA geladen werden. Die LEDs sollten nun ein Lauflicht zeigen, das von rechts nach links und wieder zurück wandert.

Anhang B - vga_sync.vhd[27]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH. ALL;
use IEEE.STD_LOGIC_UNSIGNED. ALL;

entity vga_sync is
    Port ( clk,reset : in  STD_LOGIC;
           hsync,vsync : out  STD_LOGIC;
           video_on,p_tick : out  STD_LOGIC;
           pixel_x,pixel_y : out  STD_LOGIC_VECTOR (9 downto 0));
end vga_sync;

architecture Behavioral of vga_sync is
      -- VGA 640-by-480 sync parameters
      constant HD: integer := 640; -- horizontal display area
      constant HF: integer:= 16;  -- h, front porch
      constant HB: integer:= 48;  -- h, back porch
      constant HR: integer:= 96;  -- h, retrace
      constant VD: integer:= 480; -- vertical display area
      constant VF: integer:= 10;  -- v, front porch
      constant VB: integer:= 33;  -- v, back porch
      constant VR: integer:= 2;   -- v, retrace

      -- mod-2 counter
      signal mod2_reg, mod2_next: STD_LOGIC;

      -- sync counters
      signal h_count_reg, h_count_next: unsigned (9 downto 0);
      signal v_count_reg, v_count_next: unsigned (9 downto 0);

      -- output buffer
      signal h_sync_reg, v_sync_reg: STD_LOGIC;
      signal h_sync_next, v_sync_next: STD_LOGIC;

      -- status signal
      signal h_end, v_end, pixel_tick: STD_LOGIC;

begin
      -- registers
      process (clk, reset)
      begin
            if (reset = '1') then
                  mod2_reg <= '0';
                  h_count_reg <= (others => '0');
                  v_count_reg <= (others => '0');
                  h_sync_reg <= '0';
                  v_sync_reg <= '0';
            elsif (clk'event and clk = '1') then
                  mod2_reg <= mod2_next;
                  h_count_reg <= h_count_next;
                  v_count_reg <= v_count_next;
                  h_sync_reg <= h_sync_next;
                  v_sync_reg <= v_sync_next;
            end if;
      end process;

      -- mod-2 circuit to generate 25 MHz enable tick
```

[27] Quelle: Dennis Trebbels

```
      mod2_next <= not mod2_reg;

      -- 25 MHz pixel tick
      pixel_tick <= '1' when mod2_reg = '1' else '0';

      -- status
      h_end <= -- end of horizontal counter
            '1' when h_count_reg = (HD + HF + HB + HR - 1) else -- 799
            '0';
      v_end <= -- end of horizontal counter
            '1' when v_count_reg = (VD + VF + VB + VR - 1) else -- 524
            '0';

      -- mod-800 horizontal sync counter
      process (h_count_reg, h_end, pixel_tick)
      begin
            if (pixel_tick = '1') then -- 25 MHz tick
                  if (h_end = '1') then
                        h_count_next <= (others => '0');
                  else
                        h_count_next <= h_count_reg + 1;
                  end if;
            else
                  h_count_next <= h_count_reg;
            end if;
      end process;

      -- mod-525 vertical sync counter
      process (v_count_reg, v_end, h_end, pixel_tick)
      begin
            if (pixel_tick = '1' and h_end = '1') then -- 25 MHz tick
                  if (v_end = '1') then
                        v_count_next <= (others => '0');
                  else
                        v_count_next <= v_count_reg + 1;
                  end if;
            else
                  v_count_next <= v_count_reg;
            end if;
      end process;

      -- horizontal and vertical sync, buffered to avoid glitch
      h_sync_next <=
            '1' when (h_count_reg>=(HD+HF))               -- 656
                  and (h_count_reg<=(HD+HF+HR-1)) else    -- 751
            '0';
      v_sync_next <=
            '1' when (v_count_reg>=(VD+VF))               -- 490
                  and (v_count_reg<=(VD+VF+VR-1)) else    -- 491
            '0';

      -- video on/off
      video_on <=
            '1' when (h_count_reg < HD) and (v_count_reg < VD) else
            '0';

      --output signal
      hsync <= h_sync_reg;
      vsync <= v_sync_reg;
      pixel_x <= std_logic_vector (h_count_reg);
      pixel_y <= std_logic_vector (v_count_reg);
      p_tick <= pixel_tick;

end Behavioral;
```

Anhang C - Verzeichnisstruktur des Projekts

- Bericht
  - Bilder
  - Softcore Implementierung.doc
  - Softcore Implementierung.pdf
- future_use
  - PS2_Keyboard
    (...)
  - PS2_Mouse
    (...)
- ZPU
  - C
    - build.sh
    - main.c
    - main.elf
    - main.h
  - Data2MEM
    - generate_modified_bitfile.bat
  - ISE_PROJECT
    (...)
  - VHDL
    - BMM
      - softcore.bmm
      - softcore_bd.bmm
    - INTERNALIOS
      - intIO.vhd
    - INTERRUPT
      - interrupt.vhd
    - PS2
      - ps2.vhd
    - SRAM
      - MemoryController.vhd
      - wb_sram.vhd
    - UART
      - brgen.vhd
      - rx_unit.vhd
      - tx_unit.vhd
      - uart.vhd
      - uart_pkg.vhd
    - VGA
      - font_rom.vhd
      - font_rom_pixel_generation.vhd
      - vga_sync.vhd
      - wb_vga.vhd
    - ZPU
      - wb_core.vhd
      - zpu_bram.vhd
      - zpu_config.vhd
      - zpu_core.vhd
      - zpu_pkg.vhd
    - io_pins.ucf
    - softcore_config.vhd
    - softcore_pkg.vhd
    - softcore_top.vhd
  - ISE_PROJECT.ise
- Aufgabenstellung.doc
- ZPU_Matthias_Meier.zip