UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Institut für Informatik

Implementierung einer 4-Bit MiniCPU in VHDL auf einem FPGA

Studienarbeit

im Studiengang Informatik

vorgelegt von

Christopher Israel 205110288
Marcel Jakobs 204210335

Betreuer: Dr. Merten Joost, Institut für integrierte Naturwissenschaften, Abteilung Physik,
Fachbereich 3: Naturwissenschaften

Koblenz, im Juli 2009

Erklärung

Wir versichern, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek sind wir einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimmen wir zu.	☐	☐

Ort, Datum Unterschrift

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung wurde von Christopher Israel und Marcel Jakobs im Rahmen einer Studienarbeit bei Herrn Dr. Merten Joost erarbeitet.

Ziel war es, die MiniCPU aus der Vorlesung „Technische Informatik C“ (Jetzt „Einführung in die Digitaltechnik“) von Herrn Dr. Joost für Lehrzwecke zu implementieren.

Die MiniCPU ist eine 4-Bit CISC CPU. Sie besteht aus einem Rechenwerk, Leitwerk, Adresswerk und Arbeitsspeicher.

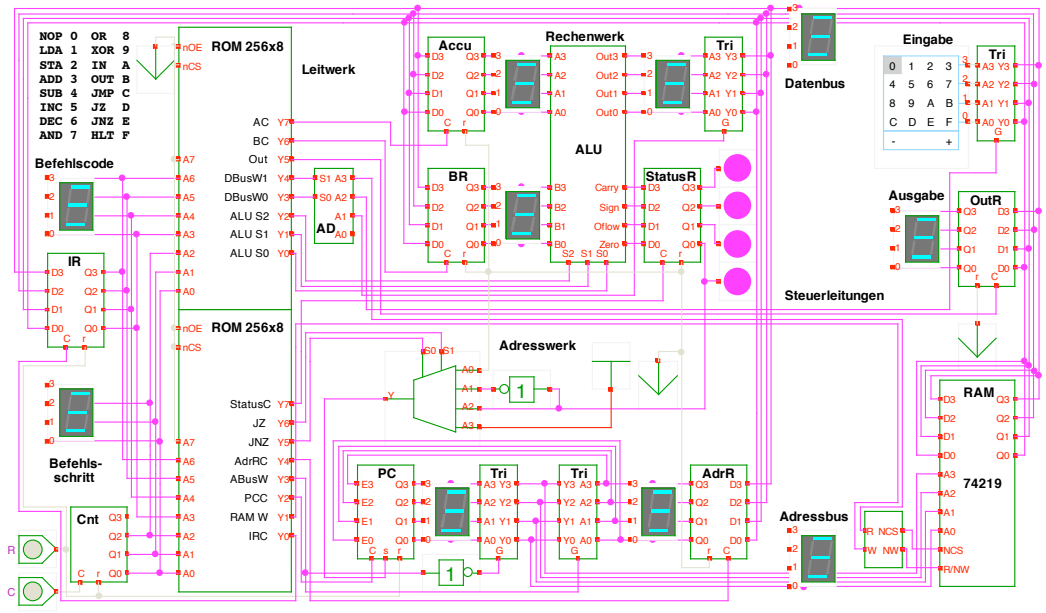
Das Rechenwerk besteht aus der Arithmetisch-Logischen Einheit (ALU), die alle Berechnungen durchführt, sowie zusätzlichen Registern. Die ALU besitzt zwei Eingangsregister. Zum Einen das Akkumulator-Register sowie ein weiteres Register (RegisterB) zum Speichern von Operanden, so dass mit jeder Operation zwei Werte verrechnet werden können. Drei Steuereingänge kodieren die auszuführende Operation.

Die ALU gibt vier Flags aus, welche in einem separaten Register (Statusregister) gespeichert werden und Informationen über die letzte Operation anzeigen. Die Flags werden in Kapitel 5.1.2 auf Seite 37 beschrieben. Das Ergebnis einer Berechnung wird auf den Datenbus geschrieben.

Die ALU unterstützt folgende Operationen:

- Identität - Der Inhalt des Akkumulators wird unverändert an den Ausgang der ALU geleitet.
- Inkrement - Der Inhalt des Akkumulators wird um 1 erhöht.
- Dekrement - Der Inhalt des Akkumulators wird um 1 verringert.
- Addition - Der Inhalt des Registers B wird zum Inhalt des Akkumulators addiert.
- Subtraktion - Der Inhalt des Registers B wird vom Inhalt des Akkumulators subtrahiert.
- AND - Der Inhalt des Registers B wird mit dem Inhalt des Akkumulators bitweise undverknüpft.
- OR - Der Inhalt des Registers B wird mit dem Inhalt des Akkumulators bitweise oderverknüpft.
- XOR - Der Inhalt des Registers B wird mit dem Inhalt des Akkumulators bitweise antivalent verknüpft.

Das Ergebnis jeder Operation wird an den Ausgang der ALU geleitet, der über einen Tristatebaustein mit dem Datenbus verbunden ist.



Der Arbeitsspeicher kann 16x4 Bit aufnehmen und ist durch den 4 Bit breiten Datenbus mit den Registern der ALU verbunden. Da die MiniCPU nach der von-Neumann-Architektur aufgebaut ist, enthält der Arbeitsspeicher sowohl Programmcode als auch Daten. Eine Ein- und Ausgabeschnittstelle ist ebenfalls an den Datenbus angeschlossen. Alle Komponenten, die auf den Bus schreiben, werden mittels Tristate-Bausteinen an den Bus gekoppelt. Ein Tristatebaustein kann neben High und Low auch den Zustand „Hochohmig“ annehmen, damit von ihm aus kein Signal auf den Datenbus gelangt, welches das gerade anliegende Signal verfälschen könnte. Da immer nur eine Komponente zur gleichen Zeit auf den Bus schreiben darf, werden auf diese Weise alle anderen Komponenten entkoppelt.

Das Adresswerk beinhaltet den 4 Bit breiten Adressbus, der in den RAM führt und dort eine Speicherstelle auswählt. Der Programmzähler, der am Ende jedes Befehls hochgezählt wird, ist durch einen Tristatebaustein mit dem Adressbus verbunden. Zusätzlich kann über den Datenbus auf das Adressregister geschrieben werden, welches ebenfalls mittels eines Tristatebaustein mit dem Adressbus verbunden ist. Dies ermöglicht den Zugriff auf beliebige Speicherstellen des RAMs. Über einen Multiplexer, der das Zero-Flag des Statusregisters auswertet, kann der Programmzähler auf den Wert des Adressregisters gesetzt werden, was (bedingte) Sprünge ermöglicht.

Das Leitwerk setzt alle Steuerleitungen für Register, Tristatebausteine, Multiplexer und die ALU und besteht aus zwei 256x8 Bit ROMs. Das Befehlsregister beinhaltet den gerade ausgeführten Befehl und wird am Anfang eines jeden Befehls aus dem RAM geladen. Jeder Befehl besteht nun aus bis zu acht Schritten, die durch den Mikrotakt gesteuert werden. Befehlscode und Befehlsschritt adressieren die ROMs, die für jede Kombination eine Belegung der Steuerleitungen gespeichert haben. Befehle, die aus weniger als acht Schritten bestehen werden mit „NOPs“ [1] auf acht Schritte aufgefüllt. Eine Auflistung aller Befehle der MiniCPU befindet sich in Kapitel 5.3

[1] Mit NOPs sind hier Mikroschritte gemeint, die keine Aktion ausführen

auf Seite 40.

Durch die Begrenzung auf 4 Bit sowohl für den Daten- als auch für den Adressbus sind nur 16 Befehle möglich und auch das RAM kann nur 16 Wörter (a 4 Bit) aufnehmen.

Eine detaillierte Beschreibung der MiniCPU befindet sich in Kapitel 5 auf Seite 37

Die MiniCPU sollte auf einem FPGA laufen und alle Ausgaben, die man in der Hades-Simulation von Herrn Dr. Joost sehen konnte, auf Siebensegmentanzeigen ausgeben. Zudem war geplant, neben der Eingabe über Taster und der Ausgabe auf einer Siebensegmentanzeige eine parallele Ein- und Ausgabe auf jeweils vier Pins zu ermöglichen sowie verschiedene Möglichkeiten zur Taktung der MiniCPU zur Verfügung zu stellen (Manuell, 1Hz und 100Hz).
Des Weiteren sollte die Möglichkeit bestehen, Werte im RAM zu bearbeiten und somit das Programm im RAM zu modifizieren.
Dabei wurde versucht, so wenig wie möglich von der Architektur und dem Aufbau der MiniCPU abzuweichen, was aufgrund der Funktionsweise eines FPGAs leider nicht zu 100 Prozent möglich ist.

2 Hardware

2.1 Was ist ein FPGA?

FPGA - Field Programmable Gate Array ist ein IC der 100.000 oder mehr Logikblöcke enthält, die sich nahezu beliebig verschalten lassen. Es gibt drei verschiedene FPGA-Architekturen:

- Antifuse (einmal beschreibbar, nichtflüchtig)
- (E)EPROM/Flash (mehrmals beschreibbar, nichtflüchtig)
- SRAM (mehrmals beschreibbar, flüchtig)

Der Großteil der zur Zeit verfügbaren FPGAs basiert auf SRAM-Zellen, die nach dem Anschalten über einen EEPROM- oder Flash-Speicher konfiguriert werden. Manche Hersteller bieten auch FPGAs an, bei denen ein Konfigurationsspeicher auf dem Chip integriert ist. Der in dieser Arbeit verwendete FPGA basiert ebenfalls auf SRAM-Zellen. Daher wird im Folgenden nur auf diese Architektur eingegangen. Der Vorteil der SRAM-Technologie ist, dass sie schneller ist als (E)EPROM oder Flash, sich aber trotzdem mehrmals beschreiben lässt. Nachteilig am SRAM ist, dass der FPGA nach jedem Einschalten neu programmiert werden muss. Der FPGA setzt sich aus folgenden Komponenten zusammen:

Übersicht: Komponenten eines FPGA

IOB (Input Output Block)	ist eine Verbindung zu einem physischen Pin des FPGA.
LUT (Lookup Table)	eine Reihe von SRAM-Zellen, mit denen sich je eine Logikfunktion nachbilden lässt.
Slice	ist eine Kombination mehrerer LUTs, FlipFlops und verbindender Logik.
CLB (Complex Logic Block)	ein Block von 4 miteinander verbundenen Slices.
Block RAM	ein Bereich zusammenhängender RAM-Zellen, die den Platz mehrerer CLBs einnehmen.
Multiplikatoren	können 18 Bit Ganzzahlen multiplizieren und belegen den Platz eines CLBs.
DCM (Digital Clock Manager)	dient zum Verwalten und Ändern des Taktes mittels PLL oder Taktteiler. Taktsignale können damit bis zu 180° phasenverschoben werden.

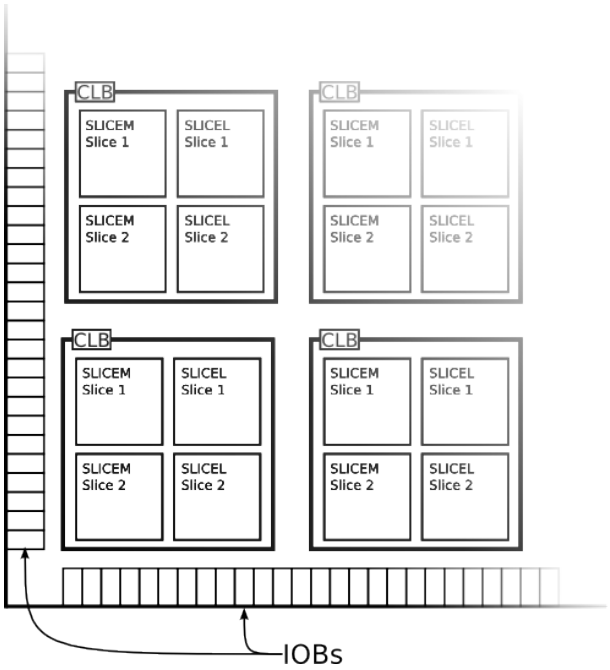


Abbildung 1: Übersicht über die Komponenten des XC3S100E

Die IOBs stellen die Schnittstelle zwischen den Pins des FPGAs und der programmierbaren Logik im Inneren des FPGA dar. In einem IOB befinden sich Treiber, die auf die verschiedenen vom FPGA unterstützen Signalpegel eingestellt werden können. Es ist auch möglich, die Ausgänge auf hohe Impedanz zu stellen (Tristate-Funktionalität), oder eingebaute Pullup- oder Pulldown-Widerstände zu aktivieren. Alle mit dem IOB in Bezug stehenden Signale können invertiert, durch interne Flip-Flops gepuffert oder mit dem Takt synchronisiert werden. Eingangssignale können optional mit einem Delay-Element um bis zu 4000 Picosekunden verzögert werden.

LUT
Lookup Tables (LUTs) sind Bestandteile von SRAM-basierten FPGAs, die zur Erzeugung von Logikfunktionen benutzt werden. Sie entsprechen einfachen SRAM-Zellen. Eine LUT kann jede Logikfunktion nachbilden, die nicht von mehr Variablen abhängig ist, als die LUT Adresseingänge hat, indem für jede Kombination der Eingangsvariablen der entsprechende Ausgangswert gespeichert wird. LUTs werden daher auch Funktionsgeneratoren genannt.

CLBs sind die Bauteile, die im FPGA benutzt werden, um Logikschaltkreise zu implementieren. Die CLBs sind im FPGA in einer Gitterstruktur in Zeilen und Spalten angeordnet. Innerhalb dieses Gitters befinden sich auch die anderen Komponenten des FPGAs (wie Block RAMs, Multiplikatoren und DCMs). Jedes CLB enthält vier miteinander verbundene Slices, die in zwei Paaren SLICEM und SLICEL angeordnet sind. Jedes Slice enthält zwei Lookup-Tables (LUTs), die beliebige logische Funktionen mit bis zu vier Eingängen realisieren können. LUTs sind ein-

fache SRAM-Zellen, über deren Adresse ein Ergebnis ausgewählt wird. Möchte man Funktionen mit mehr als vier Eingängen realisieren, kann man über die ebenfalls in den Slices enthaltenen „Wide Function Multiplexer“ beide LUTs eines Slices oder sogar die LUTs mehrerer Slices in einem CLB zusammenschließen und so auch Funktionen mit bis zu 79 Eingängen realisieren. [7] Die LUTs der in den SLICEMs enthaltenen Slices lassen sich auch als 1x16 Bit RAM oder 16-Bit Schieberegister verwenden. Die Größe dieses „distributed RAM“ lässt sich auch durch das Zusammenschalten mehrerer Slices oder CLBs erhöhen. Jedes Slice enthält weiterhin zwei Flip-Flops und zusätzliche Logik für arithmetische Berechnungen.

Slice
„Slice“ ist die Bezeichnung von Xilinx für die Kombination von mehreren LUTs, FlipFlops und verbindender Logik. Ein Slice ist ein Bestandteil eines CLBs. Andere Hersteller verwenden ihre eigenen Bezeichnungen für ähnliche Elemente. So nennt der Hersteller Altera die Kombination aus einer LUT, einem Register und verbindender Logik ein „Logic Element“

Neben CLBs und IOBs enthält der FPGA auch Block RAM, eine größere Menge zusammenhängender Speicherzellen, die den Platz mehrerer CLBs im Gitter belegen, sowie DCMs, die Taktsignale bearbeiten können. Die DCMs im Spartan 3E XC3S100E können ein vorhandenes Taktsignal um bis zu 180°Phasenverschieben, durch Angleichung an die Phase eines vorhandenen Taktsignals Clock Skew (Taktversatz) beheben und aus einem vorhandenen Taktsignal durch Multiplikation oder Division ein neues Taktsignal synthetisieren.

Der FPGA besitzt eine separate Infrastruktur, bestehend aus Leitungen mit niedriger Kapazität und geringen Signallaufzeiten, die ausschließlich zum Transport von Taktsignalen gedacht ist. Beim Spartan 3E können über dieses Taktnetz, das in vier voneinander unabhängige Bereiche aufgeteilt ist, bis zu acht Taktsignale übertragen werden.

Die Verbindungen der CLBs, IOBs und der Block RAMs untereinander laufen über die Routing Infrastruktur. Jedes Element kann über eine Schaltmatrix an Leitungen des Routing-Netzes angeschlossen werden. Es gibt vier Arten von Verbindungen:

- Long Lines verlaufen vertikal und horizontal über den ganzen Chip und können mit jeder sechsten Schaltmatrix verbunden werden (bidirektional)
- Hex Lines verlaufen vertikal und horizontal und können mit jeder dritten Schaltmatrix verbunden werden (unidirektional)
- Double Lines verlaufen vertikal und horizontal und können jede Schaltmatrix jeweils mit der übernächsten in Leitungsrichtung verbinden (unidirektional)
- Direct Connections existieren von jeder Schaltmatrix zu den direkt (horizontal, vertikal oder diagonal) benachbarten Schaltmatrizen

CLBs können auf „durchschalten“ gestellt werden, um Signale beispielsweise von dem Ausgang einer Long-Line Verbindung über das Routing-Netz zu einem anderen Ziel weiterzuleiten.

[6] [4]

2.2 Unterschiede zu CPLDs

Ein CPLD (Complex Programmable Logic Device) ist genau wie ein FPGA ein programmierbares logisches Bauteil. Es gibt allerdings mehrere Unterschiede zwischen FPGAs und CPLDs:

- CPLDs sind üblicherweise EEPROM-basiert, während viele FPGAs SRAM-basiert sind. Aufgrund dessen müssen FPGAs meist nach dem Anschalten erst von einem Flash-Speicher oder über JTAG konfiguriert werden, während CPLDs nach dem Einschalten sofort Einsatzbereit sind.
- CPLDs beinhalten wesentlich weniger Logik als FPGAs. Dadurch lassen sich größere Schaltungen nur in FPGAs realisieren. Beispiel: größter CPLD: 12.000 Logikgatter, kleinster FPGA: 100.000 Logikgatter.
- CPLDs haben durch ihren einfachen Aufbau geringere Durchlaufzeiten. Bei FPGAs können Signale an beliebige Module auf dem Chip geleitet werden, während bei CPLDs die Verdrahtung der Module weitestgehend fest ist.
- CPLDs haben meistens nur ein FlipFlop für jeden Ausgabepin, während FPGAs FlipFlops in jedem Logikmodul haben. CPLDs sind daher ungünstig für Bauteile, die viele Flipflops benötigen.

[6] [2]

2.3 Auswahl der Hardware

Es standen mehrere FPGA-Boards in der engeren Auswahl:

- das Xilinx Spartan-3E Starter Kit von Digilent für 122,96 Euro
- das Zefant LC3E-100 für 69 Euro
- das Zefant LC3E-250 für 79 Euro

Das Spartan-3E Starter Kit ist ein vollständiges Kit, das alles enthält, was man zum Programmieren des FPGAs benötigt (Netzteil, Programmierkabel, Software und das FPGA-Board). Das im Kit enthaltene Board ist auch mit einiger Hardware bestückt, wie einer LCD-Anzeige, einer Ethernet-Schnittstelle, RS232- und PS2-Ports, sowie mehreren Tastern, Schaltern und LEDs. Es sind allerdings nur insgesamt 18 frei belegbare Ein- und Ausgänge vorhanden. Auf dem Board befindet sich auch ein Hirose-Stecker mit weiteren 100 Ein- und Ausgängen, jedoch war es nicht möglich, eine Bezugsquelle für ein passendes Gegenstück zu diesem Stecker zu lokalisieren. Dadurch ist der Zugang auf diese Pins effektiv versperrt.

Bei den Zefant-Boards wird nichts mitgeliefert außer dem Board selbst; es ist auch außer einem SRAM- und einem Flashspeicher keine nennenswerte zusätzliche Hardware auf dem Board bestückt. Die Boards weisen allerdings zwei 40-polige Stiftleisten auf, die direkt an die Pins des FPGAs angeschlossen sind. Zum Programmieren ist zusätzlich ein JTAG-Kabel notwendig, welches ca. 20 Euro kostet.

Die Entscheidung ist auf das Zefant LC3E-100 Board gefallen, da das Spartan-3E Starter Kit aufgrund der begrenzten Anzahl von benutzbaren Ein/Ausgabepins schlecht geeignet ist, wenn externe Anzeigen oder Eingabemöglichkeiten angeschlossen werden sollen. Beim Zefant-Board ist dies jedoch kein Problem, da viele Ein/Ausgabepins verfügbar sind. Weitere Vorteile der Zefant-Boards sind:

- keine unnötige Hardware auf dem Board
- günstiger Preis
- kompakte Form

2.4 Beschreibung des gewählten FPGA und Entwicklungsboards

Das Zefant LC3E-100 Board enthält den Xilinx Spartan 3E FPGA mit 100.000 Gates (XC3S100E), einen 512K * 8-Bit SRAM, einen Intel Strataflash mit 32MBit, einen 50 MHz Oszillator zur Takterzeugung sowie Spannungswandler, um aus 5V Eingangsspannung die verschiedenen Versorgungsspannungen für den FPGA zu erzeugen. Es sind zwei 40-polige Stiftleisten vorhanden, die neben den der Spannungsversorgung auch die Ein- und Ausgänge des FPGA enthalten. Eine weitere 40-polige Stiftleiste kann eingelötet werden, um von außen Zugang zu den Pins des Flash-Speichers zu erhalten.
Des Weiteren sind ein DIP-Schalter mit 4 Bit sowie 2 Taster und einige LEDs vorhanden.

3 Einführung in VHDL

In diesem Abschnitt wird eine kurze Einführung in die wesentlichen Sprachelemente von VHDL gegeben.

3.1 Was ist VHDL

Um FPGAs oder andere PLDs zu programmieren, muss man die einzuspeichernde Digitalschaltung in einer Form notieren, die ein Computer lesen und in eine Konfigurationsdatei für den FPGA umwandeln kann. Dazu verwendet man eine sogenannte Hardware Description Language (HDL). Es gibt mehrere HDLs, am häufigsten werden jedoch VHDL und Verilog verwendet. VHDL ist eine Abkürzung für VHSIC HDL (Very High Speed Integrated Circuit Hardware Description Language). Hardware Description Languages sind jedoch keine Programmiersprachen in dem Sinne, dass mit ihnen ausführbarer Code generiert werden kann. Vielmehr wird aus dem VHDL-Code in einem Synthese-Vorgang eine Netzliste generiert. Diese beschreibt die einzelnen Komponenten und wie sie miteinander verbunden sind. In einem Place-And-Route-Vorgang werden die Komponenten auf dem „Floorplan“[1] des FPGA verteilt und miteinander verbunden. Nach diesem Schritt kann eine Konfigurationsdatei für den FPGA erstellt werden.

3.2 Notation

Anweisungen enden im Allgemeinen mit einem Semikolon. Komponentendeklarationen enden mit „end component;“, während die zugehörigen Entitätsdeklarationen mit „end“ und dem Namen der Entität enden. Deklarationen von Ports einer Komponente werden mit Semikola getrennt. Literale für einzelne Bits müssen mit einfachen Anführungszeichen geschrieben werden, während Bitvektoren mit doppelten Anführungszeichen geschrieben werden.
Kommentare werden mit doppeltem Bindestrich `--` eingeleitet und gelten für den Rest der Zeile. Die Groß- und Kleinschreibung sowie Einrückungen werden in VHDL ignoriert und dienen lediglich der besseren Lesbarkeit des Codes.

[1] Siehe auch die entsprechenden Erklärungen im Glossar auf Seite 79

Übersicht: Sprachelemente von VHDL

Modul	ist eine logische Einheit. Dies kann ein Schaltnetz oder ein Schaltwerk sein.
Signal	ist eine Verbindung zwischen zwei Modulen. Es kann auch fest auf einen Wert gesetzt werden. Man kann sich Signale wie Leitungen vorstellen.
Variable	ist eine Speichereinheit für einen Wert in einem Prozess.
Entity	ist das Schlüsselwort, mit dem ein Modul definiert wird.
Architecture	ist das Schlüsselwort mit dem die Implementation eines Moduls eingeleitet wird.
Component	ist eine Signatur eines Moduls, welche benötigt wird um ein Modul zu exportieren.
Package	ist eine Sammlung von Components.

3.3 Datentypen

In VHDL gibt es eine Reihe von Datentypen. Dazu zählen:

bit	Kann die Werte '1' und '0' annehmen. bit und bit_vector sollten jedoch nicht verwendet werden, da std_logic dasselbe leistet und noch zusätzliche Zustände unterstützt
std_ulogic	Beinhaltet neben '0' und '1' auch einen Zustand mit hoher Impedanz 'Z', einen undefinierten Zustand 'U', einen unbekannten Zustand 'X', schwache Signalstärken 'L' 'H' und 'W', sowie einen don't-care-Zustand '-'. Es ist jedoch nicht möglich, verschiedene Werte gleichzeitig auf ein Signal zu legen, wodurch Bussysteme mit diesem Datentyp nicht möglich sind
std_logic	Entspricht std_ulogic. std_logic hat allerdings im Gegensatz zu std_ulogic eine Auflösungsfunktion, die bei verschiedenen Signalquellen bestimmt, welcher Wert sich durchsetzt. Dadurch ist es auch möglich, verschiedene Werte gleichzeitig auf ein Signal zu legen womit auch Bussysteme implementierbar sind. [1]
boolean	Kann die Werte 'true' und 'false' annehmen
integer	Ermöglicht die Repräsentation von numerischen Werten. Dies ist jedoch auch mit std_logic_vector möglich.
unsigned	Ein Untertyp von integer, der nicht vorzeichenbehaftete Werte repräsentiert
signed	Ein Untertyp von integer, der vorzeichenbehaftete Werte repräsentiert
real	Ermöglicht die Repräsentation von Fließ- bzw. Festkommazahlen.

3.4 Signale und Variablen

In VHDL unterscheidet man zwischen Signalen und Variablen. Beide Strukturelemente können zum Speichern von Daten von beliebigen Datentypen verwendet werden. Variablen können allerdings nur innerhalb eines Prozesses verwendet werden. Der entscheidende Unterschied liegt darin, dass Variablen „direkt“ verändert werden, während Signale einen innerhalb eines Prozesses in sie gespeicherten Wert erst nach Beendung des Prozesses übernehmen. Signale repräsentieren Verbindungen zwischen Modulen, während Variablen Werte speichern. Man könnte Signale auch als Leitungen betrachten, die verschiedene Module verbinden bzw. auf die bei einer Zuweisung entsprechende Signalpegel gelegt werden. Aus diesem Grund ist es nicht möglich, Signalen mehrere Werte nacheinander „zuzuweisen“. Variablen hingegen speichern einen Wert, der beliebig geändert werden kann. Dies ermöglicht es, innerhalb eines Prozesses mit Variablen so umzugehen, wie man auch in einer Programmiersprache mit Variablen umgehen würde. So kann man eine Variable in einem Prozess auf einen Wert setzen, später diesen Wert lesen und der Variable einen neuen Wert zuweisen.

Signale

Signale werden grundsätzlich innerhalb einer Architecture-Beschreibung einer Entity deklariert (siehe Kapitel 3.6 auf Seite 18). Nach dem Schlüsselwort „signal“ kommt der Signalname beziehungsweise eine Liste von Signalnamen gefolgt von dem Variablentypen für das Signal.

Beispiel:

```
architecture xyz_arch of xyz is
signal sig1, sig2 :  std_logic_vector(3 downto 0);
signal sig3       :  std_logic;
signal bit1       :  bit_vector(3 downto 0);
signal bit2       :  bit;
begin
...
end xyz_arch;
```

Signale können mit dem „<=“ Operator auf einen Wert gesetzt oder mit einem anderen Signal verbunden werden.

Beispiel:

```
sig1 <= "1111";
sig2 <= sig1;
sig3 <= '1';
```

Variablen

Variablen werden innerhalb eines Prozesses deklariert und sind ausschließlich innerhalb dieses Prozesses gültig. Variablen werden genauso deklariert wie Signale, der einzige Unterschied ist,

dass das Schlüsselwort „variable“ verwendet wird.

Beispiel:

```
process(bla)
variable sig1, sig2 : std_logic_vector(3 downto 0);
variable sig3       : std_logic;
variable bit1       : bit_vector(3 downto 0);
variable bit2       : bit;
begin
...
end process;
```

Variablen können mit dem „:=“ Operator auf einen Wert gesetzt werden.

Beispiel:

```
sig1 := "1111";
sig2 := sig1;
sig3 := '1';
```

Vektoren

Signale können neben Datentypen wie std_logic auch Vektoren beinhalten. Dabei werden mehrere Werte in einem Signal oder einer Variablen zusammengefasst. So ist z.B.

```
signal test : std_logic_vector(3 downto 0)
```

ein 4-Bit Vektor vom Typ std_logic. Über den &-Operator können mehrere Signale zu einem Vektor aneinander gehängt werden.

Beispiel:

```
signal abc, bca :  std_logic_vector(2 downto 0);
signal a        :  std_logic := '0';
signal b        :  std_logic := '1';
signal c        :  std_logic := '0';

begin

abc <= a & b & c;   -- := "010"
bca <= b & c & a;   -- := "100"
```

Durch die Angabe eines Indexes in Klammern kann auch auf einzelne Bits oder Teilvektoren des Vektors zugegriffen werden.

Beispiel:

```
signal abc :  std_logic_vector(2 downto 0) := "010";
signal ab  :  std_logic_vector(1 downto 0);
signal a   :  std_logic;
signal b   :  std_logic;
signal c   :  std_logic;

begin
a  <= abc(0);
b  <= abc(1);
c  <= abc(2);
ab <= abc(1 downto 0);
```

Um alle oder einen Teil der Bits eines Vektors auf einen Wert zu setzen gibt es das Schlüsselwort `others`, mit dem sich solche Ausdrücke verkürzen lassen. Das folgende Beispiel definiert einen 20-Bit Vektor, der mit 0 initialisiert wird und setzt dann alle Bits außer dem höchstwertigsten auf 1.

Beispiel:

```
signal bigvector :  std_logic_vector(19 downto 0) := (others=>'0');

begin
bigvector(18 downto 0) <= (others => '1')
```

3.5 Bedingte Zuweisung

Ein Multiplexer lässt sich in VHDL einfach erstellen, indem man bedingte Zuweisung mit „when“ verwendet. Dies ist eine Alternative zu der Verwendung von Prozessen mit if-then-else oder case-when Konstrukten.

Beispiel:

```
ausgang <= eingang1 when steuersignal = '1' else eingang2;
```

Im Beispiel wird dem Signal `ausgang` der Wert des Signals `eingang1` zugewiesen, wenn das Steuersignal den Wert 1 besitzt. Ansonsten wird der Wert des Signals `eingang2` zugewiesen.

3.5.1 Typendeklarationen

In VHDL kann man eigene Typen deklarieren. Dazu benutzt man das Schlüsselwort „type“ gefolgt von dem neuen Typnamen und der eigentlichen Deklaration des Typs.

Beispiel:

```
type Tmemory is array (0 to 15) of std_logic_vector(3 downto 0);
```

Der Typ Tmemory beschreibt ein Signal, welches aus 16 4-Bit Vektoren besteht. Dieser Typ wurde in der VHDL-Implementation der MiniCPU zum Speichern der Werte des RAMs benutzt.

3.6 Grundlegende Strukturelemente

In VHDL werden Hardware-Designs in Module unterteilt, die beliebig oft instanziiert werden können. Diese Module bestehen aus einer Entity und einer Architecture. [1]

Port

In einer Entity- oder Component-Deklaration werden die Eingangs- und Ausgangssignale eines Moduls innerhalb einer Port-Deklaration festgelegt. Dort wird für jedes Ein/Ausgabesignal die Bezeichnung, die Richtung des Datenflusses (in, out, inout oder buffer) sowie der Datentyp festgelegt.

Beispiel: Port-Deklaration eines 2:1 Multiplexers mit 4-Bit Signalen:

```
Port(eingang1     : in std_logic_vektor(3 downto 0); -- Erster Eingang mit 4 Bit
     eingang2     : in std_logic_vektor(3 downto 0); -- Zweiter Eingang mit 4 Bit
     steuersignal : in std_logic;                    -- 1 Bit zur Steuerung
     ausgang      : out std_logic_vektor(3 downto 0) -- Ausgangssignal mit 4 Bit
);
```

Entity

Die Entity entspricht der „Signatur“ des Moduls, in dem die Eingangs- und Ausgangspins des Moduls festgelegt werden.

Beispiel: Entity-Deklaration eines 2:1 Multiplexers mit 4-Bit:

```
entity multiplexer is
    Port(eingang1     : in std_logic_vektor(3 downto 0); -- Erster Eingang mit 4 Bit
         eingang2     : in std_logic_vektor(3 downto 0); -- Zweiter Eingang mit 4 Bit
         steuersignal : in std_logic;                    -- 1 Bit zur Steuerung
         ausgang      : out std_logic_vektor(3 downto 0) -- Ausgangssignal mit 4 Bit
    );
end multiplexer;
```

[1] Siehe auch Glossar auf Seite 79

Architecture

Die Architecture beschreibt den „inneren“ Aufbau des Moduls.

Beispiel: Architecture eines 2:1 Multiplexers mit 4-Bit:

```
architecture multiplexer_logic of multiplexer is
  ausgang <= eingang1 when steuersignal = '0'
        else eingang2 when steuersignal = '1'
        else "XXXX";
end multiplexer_logic;
```

Der Ausgang wird in diesem Beispiel auf `eingang1` gesetzt, wenn das Steuersignal den Wert 0 hat. Er wird auf `eingang2` gesetzt, wenn das Steuersignal den Wert 1 hat. Ist der Wert des Steuersignals weder 0 noch 1 (also undefiniert, unbekannt oder hochohmig), wird der Ausgang auf undefiniert gesetzt.

3.7 Packages

Wie in Programmiersprachen macht es auch in VHDL Sinn, Code in eigenständige Abschnitte zu unterteilen, die bei Bedarf eingebunden werden können. Dies wird in VHDL durch Packages realisiert. Ein Package ist im Grunde eine Sammlung von Modulen und Typendeklarationen.

Um mehrere Module zu einem Package zusammenzufügen, erstellt man eine .vhd-Datei, in die man eine Umgebung mit dem Schlüsselwort „package“ einfügt. Innerhalb der Package-Umgebung deklariert man die Module als Components um einen Export zu ermöglichen. Nach der Package-Umgebung fügt man die Entity- und Architecture-Deklarationen der Module ein, die zum Package gehören sollen. Dabei ist darauf zu achten, dass vor jeder Entity-Deklaration die für das jeweilige Modul benötigten Libraries eingebunden werden.

Component

Components werden benötigt, wenn mehrere Entities zum Export in einem Package zusammengefasst werden sollen. Die Packages können in andere Module eingebunden werden um deren Entities zu instanziieren. Dies entspricht grob den Funktionsdeklarationen in C Header-Dateien. In der Component werden analog zu der Entity die Eingangs- und Ausgangspins des Moduls festgelegt. Diese müssen mit denen der zugehörigen Entity übereinstimmen, da sonst die Synthese fehlschlägt. In einem Package müssen zu Beginn die Components der enthaltenen Entities deklariert werden, da ansonsten in den Teilen eines übergeordneten Moduls, welches das Package verwendet, die im Package enthaltenen Module nicht gefunden werden.

Beispiel: Component eines 2:1 Multiplexers mit 4-Bit:

```
component multiplexer
    Port(eingang1     : in std_logic_vektor(3 downto 0); -- Erster Eingang mit 4 Bit
         eingang2     : in std_logic_vektor(3 downto 0); -- Zweiter Eingang mit 4 Bit
         steuersignal : in std_logic;                    -- 1 Bit zur Steuerung
         ausgang      : out std_logic_vektor(3 downto 0) -- Ausgangssignal mit 4 Bit
    );
end component;
```

Der Unterschied zwischen Entity und Component ist, dass eine Entity zwingend notwendig ist um ein Modul zu definieren, während die Component-Deklaration für den Export des Moduls benötigt wird. Ein Modul, welches nur eine Entity, aber keine Component-Deklaration besitzt, kann nur als Topmodul[1] genutzt werden. Wie man in folgendem Beispiel sehen kann gibt es zu jedem Modul neben der Entity-Deklaration auch eine Component-Deklaration, damit das Modul exportiert werden kann (Hier wurde beispielhaft nur für or2 die Entity- und Architecture-Deklaration eingefügt)

Beispiel: Package mit Logikgattern

```
package logikgatter is

component or2
    Port(a,b : in std_logic;
         out : out std_logic);
end component;

component and2
    Port(a,b : in std_logic;
         out : out std_logic);
end component;

[...]

end logikgatter;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or2 is
    Port(a,b : in std_logic;
         out : out std_logic);
end or2;

architecture or2_implementation of or2 is
begin
    out <= a or b;
end or2_implementation;

[...]
```

[1] Siehe auch die entsprechenden Erklärungen im Glossar auf Seite 79

3.8 Port Maps

In einer Architecture-Definition können zur Vereinfachung auch andere Module verwendet werden. Die Instanziierung eines Moduls geschieht mit dem Befehl `port map`. Dazu wählt man mehrere Signale aus und „mappt“ sie an die Ein- und Ausgangs-Ports des einzubindenden Moduls. Dadurch wird eine neue Instanz des Moduls erzeugt, die mit den ausgewählten Signalen verbunden wird.

Durch die Verwendung von Port Maps erspart man es sich, Elemente wie Register oder Counter bei jeder Verwendung neu definieren zu müssen.
Außerdem ist es so möglich, komplexe Hardware-Designs zusammenzusetzen ohne den Überblick zu verlieren.

Ein Port Map wird definiert durch ein frei wählbares Label, gefolgt vom Namen des zu instanziierenden Moduls. Danach kommt das Schlüsselwort „port map“ und eine Liste der zu verbindenden Signale in Klammern. Die Signale können entweder in der Reihenfolge angegeben werden, wie sie in der Port-Deklaration des Moduls stehen oder man schreibt alternativ jeweils den Namen des Signals innerhalb der Port-Deklaration zusammen mit dem zu verbindenden Signal.

BNF Syntax:

```
<label> ":" <modulname> "port map" "(" <signal> { "," <signal> } ");" .

<label> ":" <modulname> "port map" "(" <modulsignal> "=>" <localsignal>
                                { "," <modulsignal> "=>" <localsignal> } ");" .
```

Im folgenden Beispiel werden die im vorherigen Kapitel definierten Komponenten or2, and2 und xor2 instanziiert wobei jedes Mal die Signale a und b als Eingangssignale übergeben werden und die Ausgabe auf die Signale out, out2 und out3 gelegt wird.

Beispiel:

```
entity portmaptest is
    Port(out, out2, out3 : out std_logic);
end portmaptest;

architecture portmaptest_implementation of portmaptest is
signal a : std_logic := 1;
signal b : std_logic := 0;
begin
--  Label  Modulname            Signale
    ODER  : or2      port map (a, b, out);
    UND   : and2     port map (a, b, out2);
    XODER : xor2     port map (a, b, out3);

end portmaptest_implementation;
```

oder alternativ:

```
--  Label  Modulname        Signale
    ODER  : or2  port map (a   => a,
                           b   => b,
                           out => out);
    UND   : and2 port map (a   => a,
                           b   => b,
                           out => out2);
    XODER : xor2 port map (a   => a,
                           b   => b,
                           out => out3);
```

3.9 Prozesse

Neben den Möglichkeiten, die Architektur eines Moduls kombinatorisch oder durch das Instanzieren anderer Module zu beschreiben, kann man sie auch als einen Prozess modellieren.

Innerhalb eines Prozesses werden Anweisungen nacheinander ausgeführt, außerdem sind Konstrukte wie if-then-else, switch-case, und Schleifen möglich. Allerdings können nicht alle möglichen Konstrukte zur Implementation genutzt werden, da sie nicht synthetisierbar sein können. Nicht synthetisierbare Konstrukte finden Anwendung bei der Simulation von VHDL-Code. Nicht synthetisierbar sind beispielsweise taktgesteuerte Prozesse, die von mehreren Stimuli ausgelöst werden oder die „wait for“-Anweisung.

Sensitivity List

Taktgesteuerter Prozess
Ein taktgesteuerter Prozess ist ein Prozess, der bei jeder steigenden (bzw. fallenden) Flanke angestoßen wird. In der Sensitivity List des Prozesses darf dabei nur das Taktsignal stehen. Siehe auch „Flankenerkennung“ in diesem Kapitel auf Seite 23

Prozesse haben eine Sensitivity List, in der die Signale aufgezählt werden, nach deren Änderung der Prozess angestoßen wird. Bei taktgesteuerten Prozessen darf nur das Taktsignal in der Sensitivity List aufgeführt sein. Bei nicht taktgesteuerten Prozessen sollten alle Signale in die Sensitivity List aufgenommen werden, deren Änderung die Werte der Signale beeinflusst, die vom Prozess beschrieben werden. Es sollte darauf geachtet werden, dass keine Signale in der Sensitivity List stehen, die vom Prozess selbst verändert werden, da dies zu einer „Combinatorial Loop“ führt, weil der Prozess sich ununterbrochen selbst aufruft.

if-else

In Prozessen ist die Verwendung von if-then-else Konstrukten möglich.

Im folgenden Beispiel wird der Ausgang entsprechende des Steuersignals auf den Eingang oder

den invertierten Eingang gesetzt. Ist das Steuersignal weder 1 noch 0, so wird der Ausgang unabhängig vom Eingang auf 0 gelegt.

Beispiel

```
process(steuersignal, eingang)
begin
    if(steuersignal = '1') then
        ausgang <= eingang;
    elsif(steuersignal = '0') then
        ausgang <= not eingang;
    else
        ausgang <= '0';
    end if;
end process;
```

Bei der Verwendung von if-Abfragen sollte darauf geachtet werden, dass in jedem Fall ein else-Zweig existiert. Ansonsten werden in der Synthese möglicherweise unnötige Flipflops eingesetzt. Dies liegt daran, dass beim Erreichen eines nicht abgedeckten Falls die von den anderen else-Zweigen veränderten Signale auf ihrem derzeitigen Wert gehalten werden. Wenn jedoch jeder Zweig behandelt wird können in der Synthese einfache Multiplexer eingesetzt werden.

Flankenerkennung

Um Taktflanken zu erkennen benötigt man das Keyword „event“, welches in einer Bedingung bei einem Wechsel des Signalpegels true zurück gibt. Normalerweise wird es zusammen mit der Abfrage des Signals auf 1 oder 0 und-verknüpft, damit immer nur ein Flankenwechsel (steigender oder fallender) erkannt wird.
Um eine steigende Flanke des Signals `osc1` zu erkennen, kann folgende Bedingung in einem if-Statement verwendet werden:

```
process(osc1)
begin
    if(osc1'event and osc1='1') then
        -- do something
    end if;
end process;
```

case-when

Alternativ zu if-then-else Konstrukten kann man in Prozessen auch case-when Konstrukte verwenden. Diese bieten sich besonders dann an, wenn man Multiplexer mit vielen Eingangssignalen erstellen möchte oder einfach ein if-then-else mit vielen elseif-Zweigen vermeiden will.

Das folgende Beispiel setzt den Ausgang entsprechend dem Steuersignal auf einen der Eingänge oder invertierten Eingänge oder auf „1111“ bzw. „0000“.

Beispiel:

```
process(steuersignal, eingang1, eingang2)
    begin
    case steuersignal is
        when "000"  => ausgang <= eingang1;
        when "001"  => ausgang <= not eingang1;
        when "010"  => ausgang <= eingang2;
        when "011"  => ausgang <= not eingang2;
        when "100"  => ausgang <= "1111";
        when others => ausgang <= "0000";
    end case;
end process;
```

Schleifen

In Prozessen können auch for-Schleifen und while-Schleifen verwendet werden. Besonders nützlich sind Schleifen vor allem in Testbenches, da so viele Testfälle mit wenig Schreibaufwand erstellt werden können.

3.10 Testbenches und Debugging

Nach dem Erstellen eines Moduls ist es sinnvoll, die Funktion des Moduls zu testen. Dazu schreibt man eine sogenannte Testbench, in der man nacheinander den Eingangssignalen des Moduls Werte zuweist und dann in der Simulation überprüft, ob an den Ausgangssignalen des Moduls die richtigen Werte anliegen. Die Testbenches sind die VHDL-Entsprechung zu Unit-Tests in Programmiersprachen.
Eine Testbench ist eine VHDL-Datei, die einen Prozess beinhaltet, welcher der Ausführung des Tests dient. In diesem Prozess können die Eingangs-Signale des Topmoduls beliebig gesetzt werden. Das folgende Beispiel zeigt die Testbench für die ALU. Der erste Teil, in dem die ALU mittels Port Map instanziiert und die Input- und Output-Signale definiert werden, wird von ISE automatisch erstellt.

Beispiel: Testbench für die ALU

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY alu_tb_vhd IS
END alu_tb_vhd;

ARCHITECTURE behavior OF alu_tb_vhd IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT alu
    PORT(
        in1     : IN std_logic_vector(3 downto 0);
        in2     : IN std_logic_vector(3 downto 0);
        control : IN std_logic_vector(2 downto 0);
        out1    : OUT std_logic_vector(3 downto 0);
        flags   : OUT std_logic_vector(3 downto 0)
        );
    END COMPONENT;

    --Inputs
    SIGNAL in1     :  std_logic_vector(3 downto 0) := (others=>'0');
    SIGNAL in2     :  std_logic_vector(3 downto 0) := (others=>'0');
    SIGNAL control :  std_logic_vector(2 downto 0) := (others=>'0');

    --Outputs
    SIGNAL out1  :  std_logic_vector(3 downto 0);
    SIGNAL flags :  std_logic_vector(3 downto 0);

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: alu PORT MAP(
        in1     => in1,
        in2     => in2,
        out1    => out1,
        control => control,
        flags   => flags
    );
```

Im nächsten Teil des Beispiels kann man nun den eigentlichen Test ausführen, indem man die Input-Signale auf zu testende Werte legt. Die 100 ns für das global reset sollten stehen gelassen werden damit die Register und FlipFlops einen in einen definierten Zustand gelangen und die Tests unter gleichen Bedingungen statt finden können. Die eigenen Anweisungen sollten erst hinter „Place stimulus here“ eingefügt werden. Dabei sollte man darauf achten mit „wait for x ns“ genügend Zeit zwischen 2 Anweisungen zu lassen, wenn diese nicht parallel ausgeführt werden sollen. Zusätzliche Variablen können natürlich wie üblich zwischen Prozess und begin deklariert werden.
In diesem Beispiel werden die beiden Eingangsvariablen der ALU fest auf „1011“ und „0011“ gesetzt und dann alle 20 ns die Steuerleitungen der ALU geändert, so dass verschiedene Operationen mit den beiden Werten ausgeführt werden.

```
    tb : PROCESS
    BEGIN

        -- Wait 100 ns for global reset to finish
        wait for 100 ns;

        -- Place stimulus here

        in1 <= "1011";
        in2 <= "0011";
        wait for 20 ns;

        control <= "000";
        wait for 20 ns;
        control <= "001";
        wait for 20 ns;
        control <= "010";
        -- und so weiter

        wait; -- will wait forever
    END PROCESS;

END;
```

3.11 UCF zur Pinbelegung

Um Code in den FPGA zu laden, muss man zuerst festlegen, welche Ein- und Ausgaben des Toplevel-Moduls auf welchen Pins des FPGA verfügbar sein sollen. Um dieses Mapping festzulegen benutzt man eine UCF-Datei (User Constraint File). In dieser Datei wird der benutzte FPGA eingetragen und zu jeder Ein- oder Ausgabe des Toplevel-Moduls eine Zeile angelegt, die den Namen des Ports im Toplevel-Modul, die Pinnummer auf dem FPGA und die verwendeten Spannungspegel enthält. Dort kann man für den gewählten Pin auch gegebenenfalls Pullups aktivieren.

Beispiel:

```
CONFIG PART = XC3S100E-TQ144-4;

##############################################################################
# Clock Constraints
##############################################################################

# -- Clock ---------------------------------------------------------------
NET "osc1"                    LOC = "P122"    | IOSTANDARD  = LVCMOS33;

##############################################################################
# Hardware on FPGA module
##############################################################################
# -- Buttons -------------------------------------------------------------
NET "impulse"                 LOC = "P78"     | IOSTANDARD = LVCMOS33 | PULLUP;
NET "reset"                   LOC = "P84"     | IOSTANDARD = LVCMOS33 | PULLUP;

# -- LEDs ----------------------------------------------------------------
NET "debugled"                LOC = "P44"     | IOSTANDARD = LVCMOS33;
```

Zuordnung

Eine Zuordnung von einem Port im Toplevel-Modul zu einem Pin des FPGA kann man mithilfe des Schlüsselworts `NET` herstellen. Dabei notiert man nach `NET` den Namen des Ports und gibt mit `LOC =` *PINNUMMMER* den Pin des FPGA an, der verwendet werden soll. Nach einem `|` können weitere Einstellungen festgelegt werden.

Pullup Widerstände

Durch die Angabe von `| PULLUP` in einer Zuordnung wird auf dem jeweiligen Pin der Pullup aktiviert. Bei Eingabeelementen wie Tastern oder Schaltern ist es wichtig, die Pullups zu aktivieren, da so sichergestellt ist, dass zu jedem Zeitpunkt ein definierter Wert anliegt.

Pegeldefinitionen

Mit `| IOSTANDARD =` *PEGEL* kann eingestellt werden, welche Pegel die Ausgangstreiber des Pins benutzen. Die hier möglichen Werte hängen vom benutzten FPGA ab und es ist daher ratsam, vor der Wahl eines Wertes die Dokumentation des FPGA-Herstellers zu konsultieren.

Bei dem hier verwendeten XC3S100E sind unter anderem folgende Pegel möglich:

LVTTL	3.3V low-voltage TTL
LVCMOS33	low-voltage CMOS mit 3,3V
LVCMOS25	low-voltage CMOS mit 2,5V
LVCMOS18	low-voltage CMOS mit 1,8V
LVCMOS15	low-voltage CMOS mit 1,5V
LVCMOS12	low-voltage CMOS mit 1,2V
PCI33_3	3V PCI bei 33 MHz
PCI66_3	3V PCI bei 66 MHz

Die verschiedenen Pegel unterscheiden sich jeweils in den akzeptierten Schwellenwerten. Die PCI-Pegel sind auf die PCI-Spezifikation abgestimmt und eignen sich daher zum Anschluss an einen PCI-Bus.

4 Grundelemente in VHDL

In diesem Abschnitt wird beschrieben, wie man grundlegende Elemente wie z.B. Register und Counter in VHDL definiert.

4.1 Logikgatter

Einfache Logikgatter können durch die entsprechenden boolschen Ausdrücke NOT, AND, OR, XOR, ... realisiert werden.
Diese können, wie in der booleschen Algebra üblich, kombiniert und geklammert werden.

Beispiel:

```
output <= (in1 and (in2 or not in3)) xor in4;
```

4.2 FlipFlops und Register

Ein FlipFlop wird in VHDL als spezielles Register mit nur einem Bit angesehen.
Ein Register kann durch ein zusätzliches Signal implementiert werden, welches als Speichereinheit dient.
Die Taktflankenerkennung geht über ein „event“, welches in einem Prozess abgefragt wird, der durch den Takt getriggert wird.
Register und FlipFlops müssen im Allgemeinen synchronisiert werden. Siehe dazu Kapitel 8.9 auf Seite 63

Beispiel: (ohne Synchronisation)

```
process(CLK)
begin
    if CLK'event and CLK = '1') then
        -- Bei steigender Taktflanke wird mem auf Eingang gesetzt
        mem <= eingang;
    end if;
end process;
ausgang <= mem;
```

4.3 Multiplexer

Ein (De-)Multiplexer lässt sich in VHDL durch einen Prozess mit einem if-else oder case-when Konstrukt beschreiben. Dabei wird der Ausgang entsprechend der Steuereingänge auf den jeweiligen Eingang geschaltet.

Beispiel:

```
process(steuervektor)
begin
    case steuervektor is
        -- wenn der steuervektor den Wert "00" hat,
-- wird der Ausgang auf eingang0 gesetzt
        when "00"   => ausgang <= eingang0;
        -- wenn der steuervektor den Wert "01" hat,
-- wird der Ausgang auf eingang1 gesetzt
        when "01"   => ausgang <= eingang1;
        when "10"   => ausgang <= eingang2;
        when "11"   => ausgang <= eingang3;
        -- undefined wenn steuervektor nicht definiert ist.
        when others => ausgang <= "XX";
    end case;
end process;
```

Man kann einen Multiplexer auch ohne einen Prozess mittels bedingter Zuweisung realisieren. Das folgende Beispiel beschreibt den gleichen Multiplexer ohne Prozess. Bei beiden Varianten wird durch die Synthese das selbe Ergebnis erzeugt.

Beispiel:

```
-- ausgang wird auf eingang0 gelegt, wenn der steuervektor "00" ist
ausgang <= eingang0 when steuervektor = "00"
           -- ausgang wird auf eingang1 gelegt, wenn der steuervektor "01" ist
           else eingang1 when steuervektor = "01"
           else eingang2 when steuervektor = "10"
           else eingang3 when steuervektor = "11"
           -- undefined wenn steuervektor nicht definiert ist.
           else "XX";
```

4.4 Counter

Counter wurden als Prozesse definiert, die ein Signal bei jedem Clockevent inkrementieren.
Ein Reseteingang kann vorteilhaft sein, um den Zähler wieder auf 0 zu setzen.
Counter sollten genau wie Register synchronisiert werden.

Beispiel: (ohne Synchronisation)

```
process(clk,reset)
begin
    if reset = '1' then
        cnt <= "000";
    elsif(clk'event and clk = '1') then
        cnt <= cnt + 1; -- cnt wird bei jedem clockevent um 1 hoch gezählt
    end if;
end process;
out1 <= cnt;
```

4.5 Busse

Die Busse werden durch Signale definiert, die in beide Richtungen senden und empfangen können (inout). Der FPGA kann nur unidirektionale Busse erstellen. Dies ist hier jedoch irrelevant, da in der Synthese die bidirektionalen Busse entsprechend durch unidirektionale simuliert werden. Es entstehen lediglich einige Warnungen während der Synthese.

4.6 Addierer

Der Addierer besteht aus einem Prozess, der beide Eingangsvariablen in der Sensitivity List stehen hat.
Im Prozess werden 5-bittige Hilfsvariablen definiert, deren ersten 4 Bit auf die Eingangssignale gesetzt werden. Mit den Hilfsvariablen wird dann gerechnet und zum Schluss wird aus der Ergebnishilfsvariablen das Carryflag und das Ausgangssignal gesetzt.

Beispiel:

```
entity add is
    Port(in1,in2         : in  std_logic_vector(3 downto 0);  -- Eingaben
         out1            : out  std_logic_vector(3 downto 0); -- Ergebnis
         carry, overflow : out std_logic); -- Carry- und Overflow-Flag
end add;

architecture add_logic of add is
begin

    process(in1, in2)
    -- 5-Bit Variablen zum rechnen.
    variable hilfin1, hilfin2, hilfakku : std_logic_vector(4 downto 0);
    begin

        -- höchstwertiges Bit wird auf 0 gesetzt.
        hilfin1(4)  := '0';
        hilfin2(4)  := '0';
        hilfakku(4) := '0';

        -- die restlichen Bits werden auf die jeweiligen Eingaben gesetzt.
        hilfin1(3 downto 0) := in1;
        hilfin2(3 downto 0) := in2;

        -- Die eigentliche Addition findet hier statt
        hilfakku := hilfin1 + hilfin2;

        -- Das Ergebnis wird in einen 4-Bit Vektor und das Carry-Flag geschrieben.
        out1     <= hilfakku(3 downto 0);
        carry    <= hilfakku(4);
        overflow <= ((hilfin1(3) xnor hilfin2(3)) and (hilfakku(3) xor hilfakku(4)));
    end process;

end add_logic;
```

4.7 ALU

Die ALU besteht aus mehreren Prozessen, die sich mit verschiedenen Aufgaben befassen. Um Überträge berücksichtigen zu können wurden Hilfssignale eingeführt, die 1 Bit breiter sind als die Ein- und Ausgangssignale (5 Bit statt 4 Bit).
Der erste Prozess kümmert sich um den arithmetischen Teil, wo ein Hilfssignal je nach anliegendem Befehl (Addieren, Subtrahieren, Inkrementieren...) auf einen Wert gesetzt wird (z.B. 0 für Identität, 1 für Inkrement oder das zweite Eingangssignal für die Addition). Im Port Map von add wird dieses Hilfssignal auf das erste Eingangssignal aufaddiert. Bei dieser Addition werden auch die Flags „Carry“ und „Overflow“ gesetzt.
Der zweite Prozess behandelt die logischen Operationen. Hier wird eine Hilfsvariable je nach Steuercode auf das Ergebnis der logischen Operation der beiden Eingänge oder das Ergebnis der Addition gesetzt.
Im dritten Prozess wird die Hilfsvariable geprüft und wenn sie 0 ist, wird das Zero-Flag gesetzt.

Parallel zu den Prozessen und der Addition wird das Flag „Sign“ (als höchstes Bit des Hilfssignals) und der Ausgang auf das Hilfssignal gesetzt.
Durch diesen Aufbau wird bei der Synthese ein Schaltnetz gebildet, das dem der ALU der MiniCPU sehr ähnelt.

```
entity alu is
    Port(in1,in2 : in    std_logic_vector(3 downto 0);   -- Eingänge
         control: in std_logic_vector(2 downto 0);       -- Steuereingang
         out1     : out std_logic_vector(3 downto 0);    -- Ausgang
         flags: out std_logic_vector(3 downto 0));       -- Flags
end alu;

architecture alu_logic of alu is
    signal hsig, adds: std_logic_vector(3 downto 0);-- Hilfssignale für den Adder
    signal in2compl : std_logic_vector(3 downto 0); -- 2er Komplement der Eingabe
    signal hilfsvar : std_logic_vector(3 downto 0); -- Puffer für den Ausgangswert
    -- (hilfsvar wird zum Berechnen der Flags benötigt)
begin

    in2compl <= (in2 xor "1111")+1;
    G1: add port map (in1, hsig, adds, flags(3), flags(1));

    process(control,in1,in2,in2compl)
    begin
        case control is
            when "000"  => hsig <= "0000";   -- identity
            when "001"  => hsig <= in2;      -- addition
            when "010"  => hsig <= "0001";   -- increment
            when "011"  => hsig <= in2compl; -- subtraction
            when "100"  => hsig <= "1111";   -- decrement
            when others => hsig <= "0000";   -- identity
        end case;
    end process;

    process(control,in1,in2,adds)
    begin
        case control is
            when "101"  => hilfsvar <= in1 and in2; -- and
            when "110"  => hilfsvar <= in1 or in2;  -- or
            when "111"  => hilfsvar <= in1 xor in2; -- xor
            when others => hilfsvar <= adds;        -- sum
        end case;
    end process;

    process(hilfsvar)
    begin
        if(hilfsvar = "0000") then -- ZERO
            flags(0)<='1';
        else flags(0)<='0';
        end if;
    end process;

    flags(2)<=hilfsvar(3); -- SIGN
    -- CARRY und OVERFLOW wird in add definiert
    --  flags(1) <= ((hilfin1(7) xnor hilfin2(7)) and (hilfsvar(7) xor hilfsvar(8)));
    out1<=hilfsvar;
end alu_logic;
```

4.8 RAM

Der RAM hat neben dem Lese- und Schreibflag und dem Adresseingang eine Datenleitung, die als Ein- und Ausgang genutzt wird. Daneben gibt es noch den Takt und die Takt-Enable Leitung. Der Speicher selbst wird über ein Array von Vektoren realisiert, welches als eigener Typ definiert ist. (Siehe Kapitel 3.5.1 Typendeklarationen auf Seite 17)
Addressiert wird das Array durch Konvertierung der Adresse zum Typ Integer, denn ein Array wird mit einem Integer adressiert, wohingegen die Adresse als Bitvektor vorliegt. Mit einem Prozess werden die Eingänge `read`, `write`, `address` und `data` abgefragt und die Änderungen entsprechend angewendet.
Ist weder das Read- noch das Write-Flag gesetzt oder sind beide gleichzeitig gesetzt, ist die Datenleitung hochohmig geschaltet (damit der Datenbus nicht beeinflusst wird). Das RAM ist eingebettet in einen RAM-Wrapper, der die Signale der MiniCPU an den RAM weiter leitet. Mittels des RAM-Wrappers kann der RAM jedoch auch programmiert bzw. geändert werden. Siehe Kapitel 6.2.4 RAM-Wrapper auf Seite 49.
Intern ist der RAM als Latch implementiert. Dies wird im Allgemeinen nicht empfohlen, ist in diesem Fall jedoch eine praktische Lösung, damit ein asynchroner Zugriff aus dem RAM-Wrapper auf das Speicherarray möglich ist.

Beispiel: RAM mit vordefiniertem Würfelprogramm

```
entity ram is
    Port(osc1   : in std_logic;                          -- Takt
         adr    : in std_logic_vector(3 downto 0);       -- Adresseingang
         r,w    : in std_logic;                          -- Lese- und Schreibe-Eingang
         data   : inout std_logic_vector(3 downto 0);    -- Datenbus
         memory : out Tmemory                            -- Speicherinhalt
    );
end ram;

architecture ram_cmb of ram is
    signal mem : Tmemory := (
--      Wuerfel:
        X"1", X"F",X"B",X"6",X"E",X"2",X"D",X"0",
        X"0",X"0",X"0",X"0",X"0",X"0",X"0",X"6"
    );
begin
    memory<=mem;
    process(osc1,adr,r,w,data,mem)
    begin
                if(r = '1' and w = '0') then
                    data <= mem(conv_integer(unsigned(adr)));
                elsif(r = '0' and w = '1') then
                    mem(conv_integer(unsigned(adr))) <= data;
                else
                    data <= "ZZZZ";
                end if;
    end process;
end ram_cmb;
```

4.9 ROM

Das Leitwerk, welches in der HADES-Implementation ein (P)ROM ist, wird durch einen Prozess implementiert, welcher den gerade anliegenden Befehlscode durch ein case Statement abfragt. Darin geschachtelt wird jeweils der aktuelle Befehlsschritt durch ein weiteres case Statement abgefragt und entsprechend alle Ausgänge gesetzt. Die doppelte Schachtelung ist nötig um beide Signale, den Befehlscode und den Befehlsschritt, zu berücksichtigen. Dies macht das Leitwerk leider recht unübersichtlich.

Beispiel: ADD-Befehl beim ersten Mikrotakt

```
architecture leitwerk_logic of leitwerk is
begin

    process(op,cnt,osc1, enable)
    begin
    if(osc1'event and osc1 = '1') then
        if enable='1' or reset='1' then
            case op is
                when "0000" => case cnt is -- NOP
                [...]
                when "0011" => case cnt is -- ADD
                    when "000"  => ac <= '0';    -- Befehl auf Bus
                              bc      <= '0';
                              outc    <= '0';
                              ing     <= '0';
                              alug    <= '0';
                              ramg    <= '1';
                              control <= "000";
                              stc     <= '0';
                              jz      <= '0';
                              jnz     <= '0';
                              arc     <= '0';
                              arw     <= '0';
                              pcc     <= '0';
                              ramw    <= '0';
                              irc     <= '0';
                    when "001"  => ac <= '0';    --IR übernimmt
                              [...]
                    when others => ac <= '0';
                              [...] -- Alle Signale auf 0 setzen
                end case;
                when "0100" => case cnt is -- SUB
                    [...]
                when others => ac <= '0';
                    [...] -- Alle Signale auf 0 setzen
            end case;
        end if;
    end if;
    end process;
end leitwerk_logic;
```

4.10 Entprellen von Tasten

Der Prozess zum Entprellen von Tastern ähnelt dem eines Counters. Auch hier wird ein Signal mit jedem Takt inkrementiert, jedoch wird bei einem bestimmten Wert (der abhängig vom Takt und vom gewünschten Verhalten gewählt wird) das Ausgangssignal einen Takt lang auf High gesetzt. Danach wird das Zählersignal wieder auf Low gesetzt (es sei denn es würde ohnehin direkt überlaufen, dann ist dieser Schritt nicht unbedingt nötig). Im Gegensatz zu einem echten Entpreller ist dies also nur eine Verzögerungfunktion, die bei einem Tastendruck den Ausgang einen Takt lang auf High setzt und eine gewisse Zeit wartet. Ist der Taster danach immer noch gedrückt, wird ein weiterer Takt lang das Ausgangssignal auf High gesetzt. Dadurch wird nicht nur das Prellen des Tasters abgefangen sondern zusätzlich auch eine mehrmalige Auswertung aufgrund des hohen Taktes vermieden.

Beispiel:

```
process(osc1)
variable count: std_logic_vector(16 downto 0) := (others => '0');
begin
    if(osc1'event and osc1 = '1') then
        if(count = "11111111111111111" and inpin = '1') then
            outpin <= '1';
        elsif(inpin = '1') then
            outpin <= '0';
            count  := count+1;
        else
            count  := (others=>'0');
            outpin <= '0';
        end if;
    end if;
end process;
```

5 4-Bit CISC MiniCPU

Ziel war es, die 4 Bit MiniCPU zu Lehrzwecken in VHDL auf einem FPGA zu implementieren. Die Vorgehensweise orientierte sich dabei im Wesentlichen an den Folien aus der Vorlesung „Technische Informatik C“, wobei größtenteils die gleichen Entwicklungsschritte vollzogen wurden. Das heißt vom Addierer über RALU (ALU mit Registern), die dann mit einem Bus und Leitwerk ausgestattet wurde, bis hin zur MiniCPU mit einfachen Sprungbefehlen.

Durch die Begrenzung auf 4 Bit sowohl für den Daten- als auch für den Adressbus sind nur 16 Befehle möglich und auch das RAM kann nur 16 Wörter (a 4 Bit) aufnehmen. Dadurch ist es jedoch möglich sämtliche Register, Busse und auch den kompletten Inhalt des RAM auf Siebensegmentanzeigen auszugeben.
Die Grundbausteine wie Register und Counter wurden als Module einmal definiert und können beliebig oft instanziiert werden.

5.1 Eigenschaften und Aufbau

Die MiniCPU besteht aus den folgenden Elementen:

5.1.1 Leitwerk

Da die MiniCPU eine CISC CPU implementiert, besteht ein Befehl aus 8 Mikrotakten. Dafür gibt es einen eigenen Counter, der vom Takt hochgezählt wird.
Der Ausgang des Counters führt zusammen mit dem aktuellen Befehl des Befehlsregisters ins Leitwerk.
Das Leitwerk setzt die Befehlscodes abhängig vom gerade anliegenden Mikrotakt auf die Steuerleitungen um. So wird der aktuelle Befehl aus dem RAM ins Befehlsregister geladen und dann je nach Befehl weitere Werte in den Akku bzw. in das Register B geschrieben und die Steuerleitungen der ALU gesetzt. Das Ergebnis einer Berechnung wird im Akku gespeichert. Zum Schluss wird immer der Programmcounter inkrementiert.

5.1.2 Rechenwerk

Das Rechenwerk besteht aus der ALU mit Akku und Register B (zusammen auch RALU genannt) sowie dem Statusregister. Sie ist für alle Arten von Berechnungen zuständig. Der Akku dient als zentrales Speicherelement, von wo aus Daten gespeichert und ausgegeben werden. Genauso werden alle Daten, die geladen oder eingelesen werden als erstes in den Akku gespeichert.
Es gibt 4 Statusflags:

- Carry - ist gesetzt, wenn die letzte Berechnung (Addition oder Inkrementierung) einen Übertrag ergab
- Sign - ist gesetzt, wenn der Wert negativ ist. Dann gilt das erste Bit als Vorzeichenbit

- Overflow - ist gesetzt, wenn eine Bereichsüberschreitung zwischen dem positiven und negativen Bereich auftritt
- Zero - ist gesetzt, wenn die letzte Berechnung 0 ergab.

5.1.3 Adresswerk

Das Adresswerk besteht aus dem Programmcounter (PC), dem Adressregister, dem Adressbus und einem Multiplexer für Sprünge. Der PC ist Counter und Register in einem und hat 3 Steuereingänge: Clock, Set und Reset. Durch den Clock (der vom Leitwerk kommt) wird normal inkrementiert. Durch Setzen des Set-Eingangs wird der an den Dateneingängen anliegende Wert übernommen, der vom Adressregister kommt. Der Reseteingang setzt den PC auf 0.
Der Multiplexer bekommt als Eingang das Zero-Flag des Statusregisters der ALU. Intern schaltet er zwischen 0, 1 dem Zero-Flag und dem negierten Zero-Flag anhand der Steuerleitung um. Der Ausgang des Multiplexers ist mit dem Set Eingang des PC verbunden. Die Steuerleitungen des Multiplexers werden vom Leitwerk gesetzt.
Der Eingang des Adressregisters ist mit dem Datenbus verbunden und der Ausgang wird gleichzeitig zum PC und über einen Tristate-Baustein in den Adressbus geleitet.
Der Adressbus führt von PC und Adressregister zum RAM und bestimmt, welche Adresse des RAMs als nächstes zum Lesen oder Schreiben angesprochen wird. Tristate Bausteine, die vom Leitwerk gesteuert werden, regeln ob der PC oder das Adressregister auf den Adressbus schreiben dürfen.

5.1.4 RAM

Der RAM ist mit dem Adress- und Datenbus verbunden. Gesteuert wird er vom Leitwerk. So liest oder speichert er je nachdem wie die Steuerleitungen gesetzt sind das Datum vom Datenbus von der (bzw. an die) Adresse, die am Adressbus anliegt. In der VHDL-Implementierung wird ein RAM-Wrapper vor den RAM geschaltet, der die Programmierung des RAMs ermöglicht. Siehe dazu Kapitel 6.2.4 auf Seite 49

5.2 Vorgehensweise

Um möglichst gut testen zu können, wurden in etwa die gleichen Schritte zur Entwicklung vorgenommen, wie in der Vorlesung. Dadurch konnten die gleichen Beispielprogramme zum Testen verwendet werden.
Das Projekt ließ sich in zwei Phasen aufteilen:
1. VHDL Definition mit Simulation
2. Beschaffung, Aufbau und Programmierung der Hardware
Für diese Vorgehensweise haben wir uns entschieden, weil wir bis zu dieser Studienarbeit noch keinerlei Erfahrung mit FPGAs oder VHDL hatten. Wir wollten erst ausprobieren wie alles funktioniert und ob unser Vorhaben in der Zeit die uns zur Verfügung stand realisierbar war, bevor wir in teure Hardware investierten.

Phase 1: VHDL Definitionen und Simulation

Das erste Modul war ein Inverter, der als „Hello World Programm“ fungierte.
Als nächstes wurde die ALU implementiert. Diese wurde im nächsten Schritt zur RALU weiter entwickelt und dann mit einem Bus versehen.
Danach musste das Leitwerk realisiert werden. Dies war recht umständlich, da es vergleichsweise aufwändig zu implementieren war und mit jedem weiteren Entwicklungsschritt geändert werden musste.
Nun kam das RAM dazu, wobei die Adressen noch von Hand über nach außen gelegte Adresspins gesetzt wurden.
RALU mit RAM und Leitwerk wurden mit dem Programcounter und Adressbus schnell zu einer ersten CPU aufgerüstet.
Das Adresswerk erlaubte nun, neben Programmcode auch Daten im RAM zu speichern.
Durch den Multiplexer wurden schließlich auch unbedingte und bedingte Sprünge möglich, so dass die Mächtigkeit eines „normalen“ Prozessors erreicht wird.
Die erste Phase verlief überraschend schnell und ohne wesentliche Komplikationen. Die CPU lief im ISE-Simulator nach nur wenigen Wochen problemlos und es war nun möglich, sich der Hardware zu widmen.

Phase 2: Aufbau und Programmierung der Hardware

Das FPGA-Board leitet alle relevanten Pins über zwei 40-polige Wannenstecker heraus. Daher war es die erste Aufgabe in diesem Bereich, ein Board zu entwickeln, welches die Pins aufteilt und entsprechend weiterführt. Daraus ist in mehreren Schritten die Hauptplatine entstanden. Die erste Version war noch auf einer Lochrasterplatine zum Testen aufgebaut.
Zu diesem Zeitpunkt wurde die Anzeigeplatine entwickelt um einzelne Register anzeigen lassen zu können. Dies war sehr wichtig, da die Anzeigeplatine die einzige Möglichkeit zum Debuggen war. Da es auch zu den Zielen gehörte, jedes Register und jeden Bus auf Siebensegmentanzeigen auszugeben, war die Entwicklung der Anzeigeplatine auch dahingehend ein wichtiger Schritt. Bei den ersten Tests der Anzeigeplatine wurde klar, dass normales Multiplexen der Anzeigen nicht möglich ist und ein Latch vor jede Anzeige geschaltet werden musste (Siehe auch Kapitel 8.7 auf Seite 62). Technisch gesehen wird Demultiplexing genutzt. Trotzdem wird diese Art der Ansteuerung von Anzeigen im Allgemeinen als Multiplexing bezeichnet.
Nach dieser kleinen Änderung wurden vier Anzeigeplatinen aufgebaut und auf eine Holzplatte geklebt. Zudem wurden mehrere Select-Leitungen auf die Lochrasterplatine herausgeführt um durch Umstecken bei Bedarf andere Register anzeigen zu können.
Zur Eingabe diente zu diesem Zeitpunkt noch ein Eingabegerät aus dem Hardwarepraktikum.
Leider hat es dann mehrere Wochen gedauert, die Grundfunktionen der CPU, die im Simulator problemlos lief, auf dem FPGA zum Arbeiten zu bewegen. Wie sich herausstellen sollte war die Ursache für die meisten dieser Probleme in der fehlenden Synchronisation bedingt (siehe dazu Kapitel 8.9 auf Seite 63).
Nachdem die Ursache der Probleme lokalisiert war lief die CPU dann innerhalb weniger Tage auch auf dem FPGA. Nun war es an der Zeit die Hauptplatine zu entwerfen und zu implementieren. Anschließend wurde die Ein-Ausgabeplatine erstellt, mit der von diesem Zeitpunkt an die MiniCPU gesteuert wurde. Die Flags wurden über 4 Ausgabepins des FPGA auf LED's ausgegeben und es wurde dafür auch ein entsprechendes LED-Board entwickelt.
Mit der Forderung nach paralleler Ein- und Ausgabe sollte die Ein-Ausgabeplatine entsprechend

geändert werden um die Ausgabe gleichzeitig auf einer Siebensegmentanzeige und auf 4 Pins auszugeben. Durch diese Änderung der Ein-Ausgabeplatine kam das Problem auf, dass zu wenige Ausgabepins des FPGA zur Verfügung standen. Somit wurde die parallele Ausgabe nach dem gleichen Prinzip wie die Anzeigeplatine gemultiplext. Auch die LED-Platine musste wegen der nun fehlenden Pins so geändert werden, dass die LEDs gemultiplext werden.

5.3 Befehlssatz

Der Befehlssatz besteht aus den folgenden 16 Befehlen:

Kürzel	Binär	Beschreibung
NOP	0000	No Operation - Keine Operation, nichts passiert
LDA	0001	Lädt das nächste Datum aus dem RAM in die ALU
STA	0010	Speichert das Datum der ALU in den RAM an die Adresse, die im nächsten Datum des RAMs angegeben ist
ADD	0011	Addiert den nächsten Wert im RAM mit dem Akku
SUB	0100	Subtrahiert den nächsten Wert im RAM vom Akku
INC	0101	Addiert 1 zum Akku
DEC	0110	Subtrahiert 1 vom Akku
AND	0111	und-verknüpft den nächsten Wert im RAM mit dem Akku
OR	1000	oder-verknüpft den nächsten Wert im RAM mit dem Akku
XOR	1001	verknüpft den nächsten Wert im RAM mit dem Akku antivalent
IN	1010	Liest von der Eingabe und schreibt den Wert in den Akku
OUT	1011	Gibt den Wert vom Akku aus
JMP	1100	Springt zur Adresse, die im nächsten Datum des RAMs angegeben ist
JZ	1101	Springt zur Adresse, die im nächsten Datum des RAMs angegeben ist, falls das Zero-Flag gesetzt ist
JNZ	1110	Springt zur Adresse, die im nächsten Datum des RAMs angegeben ist, falls das Zero-Flag nicht gesetzt ist
HLT	1111	Hält das Programm an

5.4 Takt

Es sind drei mögliche Taktquellen vorgesehen. Zwischen diesen kann man mit einem Takt-Umschalter jederzeit umschalten.
Die erste Möglichkeit ist den Prozessor von Hand zu Takten. Dazu gibt es einen (entprellten) Button, der bei jedem Druck eine Taktflanke generiert. So kann man nach jedem Takt alle Register, Busse und RAM-Adressen betrachten und ggf. erklären.
Als zweites gibt es einen Takt von 1 Hz, bei dem man live mitverfolgen kann, was sich im Prozessor abspielt ohne für jeden Takt einen Button drücken zu müssen.
Schließlich gibt es noch einen 100 Hz Takt. Damit kann man ein Programm schneller durchlaufen zu lassen um z.B. einen Würfel implementieren zu können. Auch um andere Geräte wie z.B. Mikrocontroller anzuschließen ist ein etwas schnellerer Takt oft sinnvoll.

5.5 Ein- und Ausgabe

Die Eingabe kann entweder über vier Eingabepins oder manuell mit den Tasten „Up“ und „Down“ erfolgen. Die Ausgabe wird immer gleichzeitig auf einer Siebensegmentanzeige und parallel auf 4 Pins ausgegeben.

Eingabe

Die Eingabe von Daten erfolgt über Inkrementieren bzw. Dekrementieren des Eingaberegisters solange, bis der gewünschte Wert erreicht ist. Danach kann man durch Takten den Ablauf des Programms fortsetzen. Alternativ kann die Eingabe auch parallel über 4 Pins erfolgen, so dass auch Mikrocontroller oder andere digitale Geräte angeschlossen werden können. Dabei können Eingaben auch bei einem Takt von 1 oder 100 Hz erfolgen.

Ausgabe

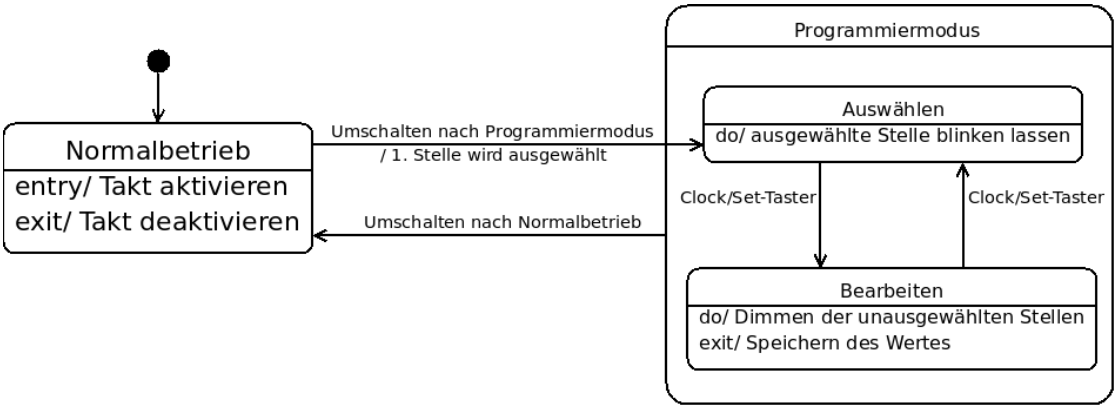
Die Ausgabe der Daten erfolgt über eine Siebensegmentanzeige. Gleichzeitig werden die Daten auch parallel auf 4 Pins ausgegeben.

5.6 Programmierung des RAM

Das RAM wurde in einen RAM-Wrapper eingebettet um dessen Programmierung über die Eingabe zu ermöglichen. Im Normalbetrieb leitet der RAM-Wrapper alle Signale direkt an den RAM weiter. Durch einen DIP-Schalter (Programmier-Umschalter) kann man den RAM-Wrapper so umstellen, dass die Up-, Down und Takt-Taster direkt abgegriffen werden. Der Takt wird nicht mehr zur eigentlichen CPU weiter geleitet, wodurch eventuelle Seiteneffekte durch gleichzeitigen Programmablauf während des Programmierens vermieden werden. Wenn der Programmier-Umschalter auf Programmieren steht, gibt es zwei Modi, die sich mit dem Takt-Taster umschalten lassen:

- Auswahlmodus - Hier kann mit den Up- und Down-Tasten die Stelle gewählt werden, die geändert werden soll.
- Bearbeitungsmodus - Hier kann mit den Up- und Down-Tasten der Wert an der ausgewählten Stelle geändert werden.

Ein UML-Zustandsdiagramm verdeutlicht die verschiedenen Zustände und Zustandsübergänge:



Der RAM wurde so implementiert, dass er direkt mit einem Programm initialisiert werden kann. Dadurch hat man eine weitere Möglichkeit ein Programm in den RAM zu schreiben indem man sich mehrere Binärdateien zur Programmierung des FPGA erzeugt. In den Dateien kann der RAM dann mit verschiedenen Programmen initialisiert werden. Dadurch braucht man den FPGA dann nur mit der entsprechende Binärdatei zu flashen um ein Programm zu laden.

5.7 Anzeige von Registern und Bussen

Alle Register und Busse sowie der gesamte Inhalt des RAM werden auf Siebensegmentanzeigen ausgegeben. Um Pins des FPGA zu sparen wurden die Anzeigen gemultiplext. Hierzu wurde eine Platine entwickelt, welche durch ein 10-adriges Flachbandkabel mit Spannung und Daten versorgt wird. Zusätzlich führt eine Leitung mit einem Select-Signal zu jeder Anzeigeplatine. Ein Latch speichert die Daten zwischen, bis das nächste Select-Signal das Latch aktualisiert. Mehr zur Anzeigeplatine Im Kapitel 7.2 auf Seite 54.

6 VHDL Code

Der VHDL-Code ist in 7 Dateien aufgeteilt: Sechs Dateien, in denen die verschiedenen Module definiert werden und eine Toplevel Datei, die als Topmodul fungiert.
Übersicht:

- toplevel.vhd - Hier wird die Hardware angesteuert und die Signale (ggf. entprellt) in die einzelnen Module geleitet.
- hardware.vhd - Hier sind Entpreller, Taktteiler und ähnliche Module definiert die für die Hardware zuständig sind.
- standardgatter.vhd - Beinhaltet Standardmodule wie Register, Counter und Tristatebausteine.
- komposita.vhd - Beinhaltet zusammengesetzte Module wie die ALU oder das Leitwerk.
- ram.vhd - Hier ist das RAM definiert. Diese Datei muss geändert werden, wenn man das RAM mit einem anderen Programm initialisieren möchte.
- minicpu.vhd - Definiert die MiniCPU
- minicomputer.vhd - Definiert den Minicomputer bestehend aus MiniCPU, Ein- und Ausgabe und RAM (-Wrapper)

Coding Guidelines

VHDL macht keine Unterscheidung zwischen Groß- und Kleinschreibung. Um den Code lesbar zu gestalten wurden einige minimale Richtlinien erstellt nach denen der Code geschrieben wurde.

- Alle VHDL-Ausdrücke werden klein geschrieben
- Alle Bezeichner werden klein geschrieben
- Alle Labels werden komplett groß geschrieben
- Hilfsvariablen und Hilfssignale tragen die gleichen Bezeichner wie das Originalsignal an den `_help` angehängt wird
- Signale, die nur dazu gedacht sind um die Werte der Register heraus zu führen, tragen ein `_out` am Ende
- Der 50 MHz Systemtakt wird immer mit `osc1` oder `clk` bezeichnet
- Das Enable Signal für `osc1` wird immer mit `enable` bezeichnet

6.1 Schemata

Da ein einzelnes Schema mit allen Modulen zu groß und unübersichtlich geworden wäre, wurden eigene Schemata für die wichtigsten Module erstellt.
Vektoren sind durch einen diagonalen Strich gekennzeichnet. Diese und die Bezeichner von Signalen können mehrfach an einem Signal angebracht sein um die Übersichtlichkeit zu erhöhen.

Manche Signale werden schaltungsbedingt invertiert, was nicht in den Schemata angegeben wird, da dort nur gezeigt werden soll wofür die Signale sind und wo sie hin führen. Ob sie nun invertiert werden oder nicht macht für die Logik keinen Unterschied.
Signale haben i.d.R. 3 Bezeichner, wenn sie von einem Modul in ein anderes geführt werden:

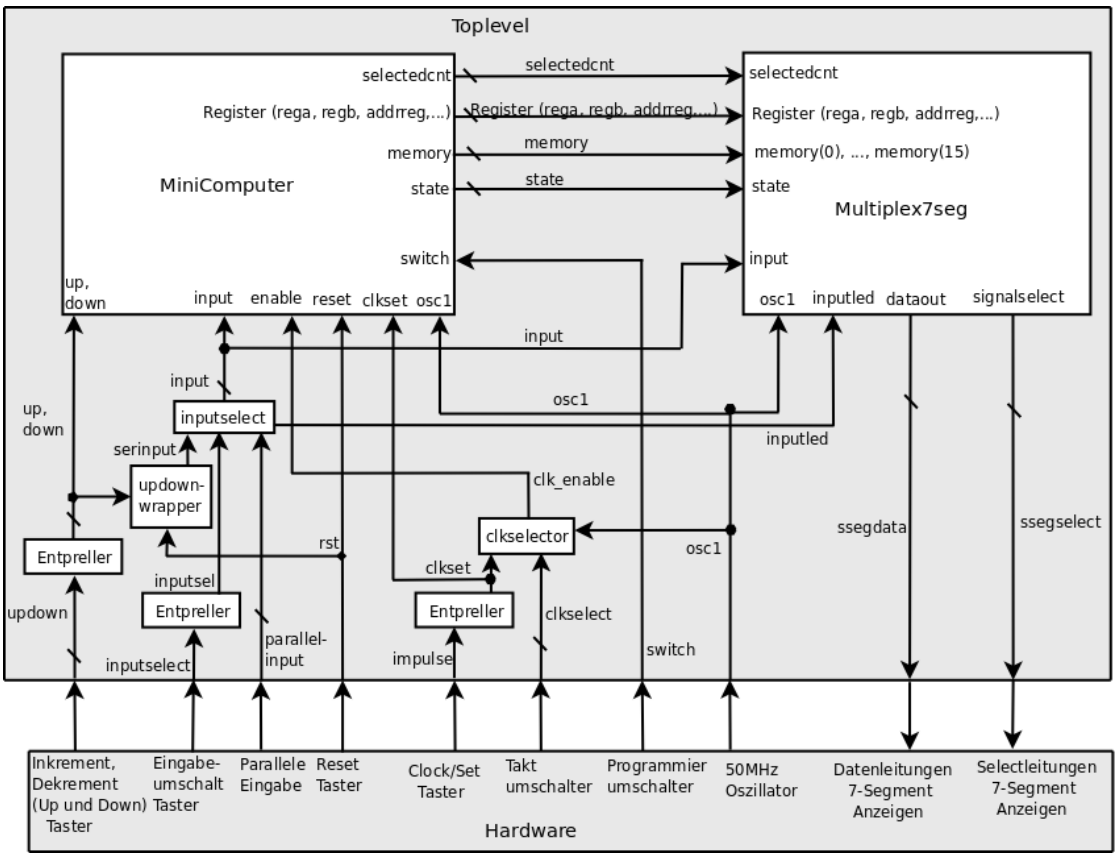
- im Quellmodul, der besagt wie das Signal dort bezeichnet wird
- im Modul, das gerade beschrieben wird
- im Zielmodul, der besagt wie das Signal dort bezeichnet wird

Einige Signale, die ein out-Signal im Port eines Moduls sind, können nicht als in-Signal ein anderes Modul (innerhalb des entsprechenden Moduls) genutzt werden, da sonst die Synthese fehlschlägt. Daher mussten Hilfssignale erstellt werden, wodurch einige Signale in den Schemata mehrere Bezeichner haben. Es wurde versucht, die Bezeichner so anzubringen, dass dieser Umstand deutlich wird.
Prozesse werden in den Schemata durch Kreise dargestellt. Die Prozesse sind nummeriert und werden im jeweiligen Text erklärt. Mehrere Prozesse in den Schemata können im VHDL Code durch einen einzelnen Prozess beschrieben werden. In den Schemata wird der Prozess dann in mehrere Prozesse aufgeteilt, wenn er für mehrere Aktionen zuständig ist. Dies soll die Übersichtlichkeit und Lesbarkeit der Schemata erhöhen (sonst würden teilweise sehr viele Signale von und zu einem Prozess laufen und es wäre schwer ersichtlich was der Prozess eigentlich macht.)
Bedingte Zuweisungen werden im Allgemeinen durch Multiplexer dargestellt. Signale, die lediglich zu Debug-Zwecken dienen wurden in den Schemata ignoriert.

6.2 Beschreibung wichtiger Module

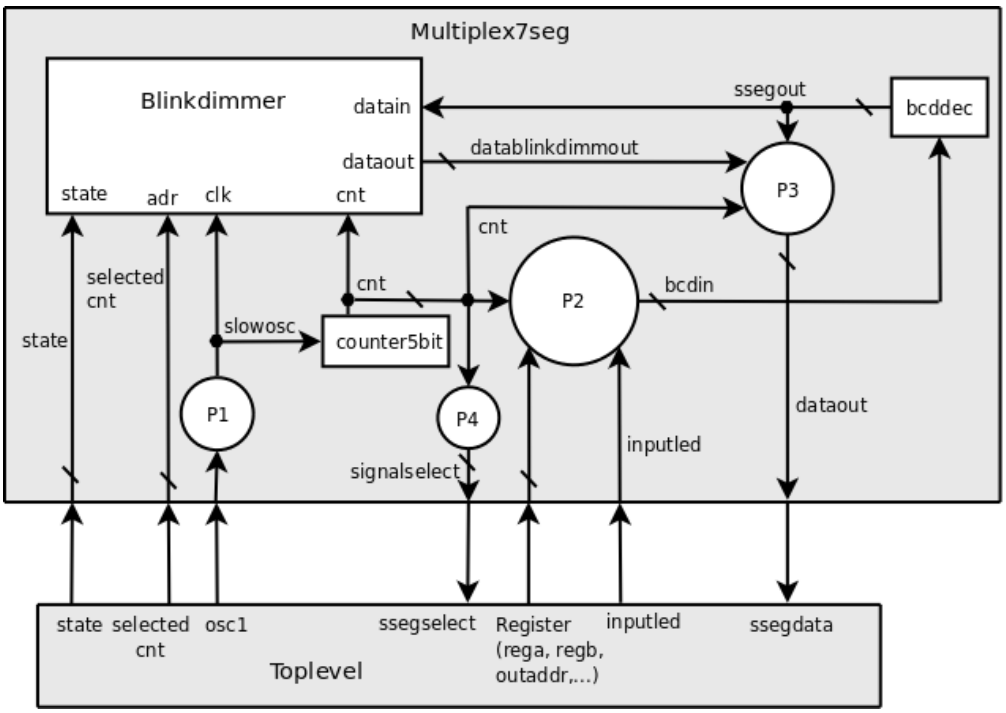
Im Folgenden werden die wichtigsten Module etwas genauer beschrieben

6.2.1 toplevel - Verbindung und Umgang mit der Hardware



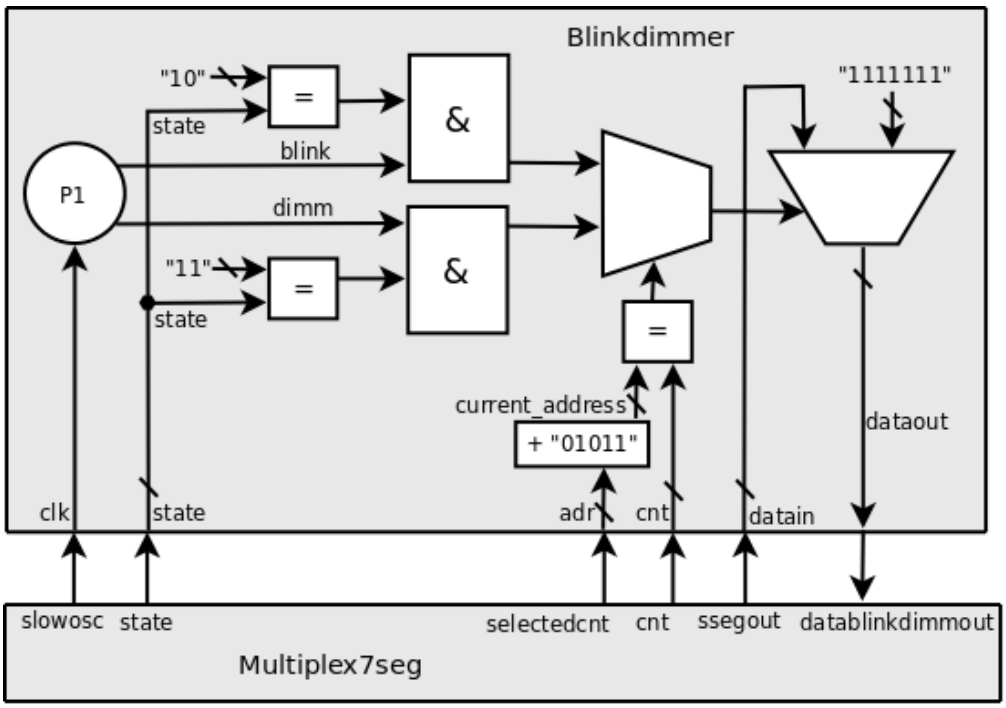
Im Toplevel wird die Verbindung zur Hardware hergestellt. Alle im Port definierten Signale entsprechen Pins des FPGA. In diesem Modul wird der gesamte Umgang mit der Hardware geregelt, wie das Entprellen von Tasten und die Auswahl des Taktes mittels des Takt-Umschalters (clkselector). Die Siebensegmentanzeigen werden dort durch das Moduls "multiplex7seg" angesteuert, das in einem eigenen Abschnitt näher beschrieben wird. Auch die Up und Down Taster werden ausgewertet und zu einem 4-Bit Signal verarbeitet, welches als Input für den MiniComputer dient. Gleichzeitig werden sie zur Programmierung des RAM an den MiniComputer übergeben. Der inputselector wählt anhand des Eingabe-Umschalters aus, ob die Up- und Down-Tasten oder der parallele Input verwendet werden.
Des Weiteren wird in der Toplevel-Datei der MiniComputer instanziiert, welcher der MiniCPU inklusive RAM und Ein- Ausgabe entspricht. Die Programmierung des RAM wird auch im MiniComputer vorgenommen, da dafür ein RAM-Wrapper genutzt wird, der den RAM selbst beinhaltet.

6.2.2 multiplex7seg - Ansteuerung der Siebensegmentanzeigen



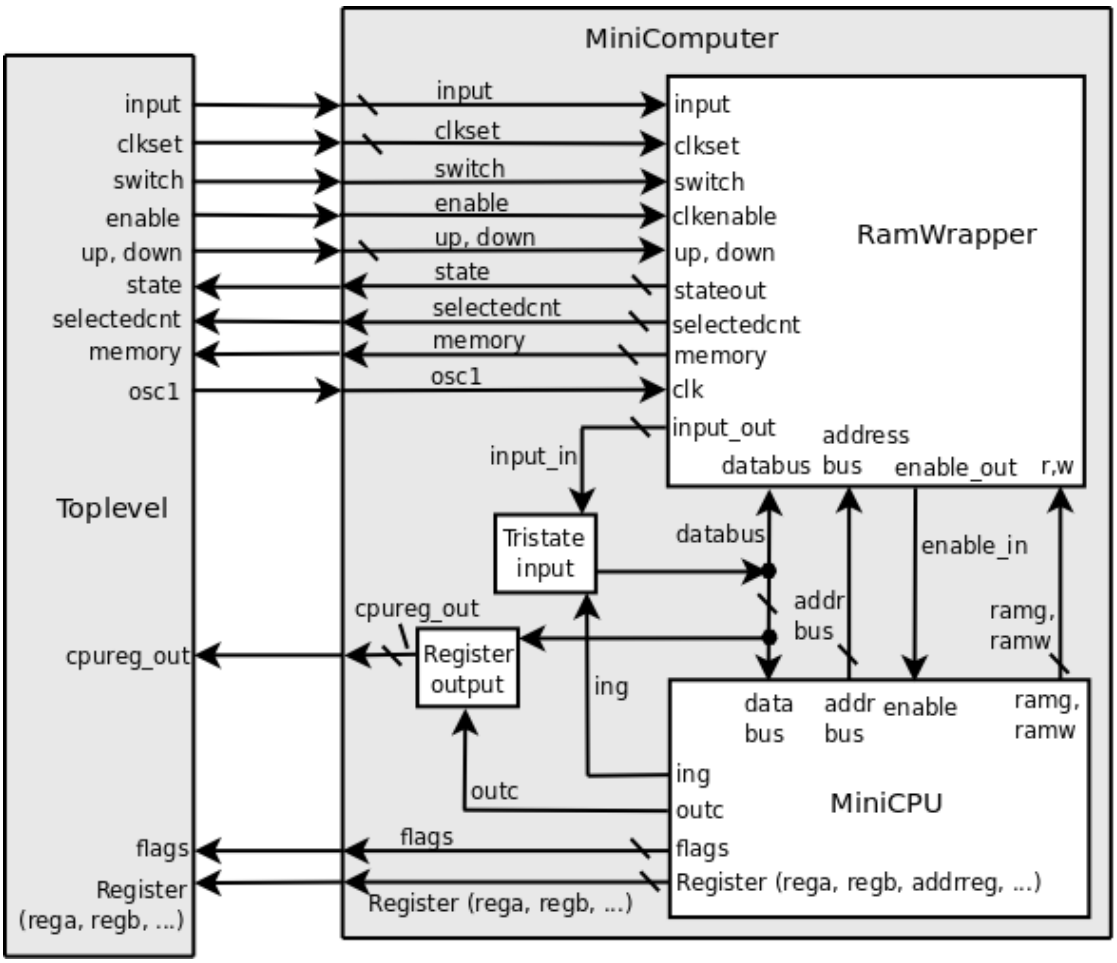
Das Modul "multiplex7seg" legt ein Datum auf die Datenleitung der Siebensegmentanzeigen und setzt gleichzeitig das entsprechende Select-Signal auf High. Dies wird immer wieder reihum für jede Siebensegmentanzeige wiederholt.
Das Modul besteht aus einem BCD-zu-Siebensegment-Decoder und einem Blink- und Dimmmodul, um die Anzeigen des RAMs beim Programmieren entsprechend dem Status blinken lassen oder dimmen zu können. Daneben wird ein 5-Bit Counter zum Durchlaufen der Anzeigen mittels eines verlangsamten Taktes (`slowosc`) hochgezählt. Ein Prozess setzt die Datenleitung der Anzeigen entsprechend des aktuellen Counter-Wertes auf den zugehörigen Wert.
Beschreibung der Prozesse:
P1: erzeugt aus dem Takt einen langsameren Takt (`slowosc`) um die Siebensegmentanzeigen zu steuern, da die Frequenz eines höheren Taktes die Spezifikation der Latches auf den Anzeigeplatinen übersteigt. Außerdem wird durch die geringere Frequenz der Einfluss von elektromagnetischer Einstrahlung reduziert.
P2: wählt anhand des `cnt`-Signals das Register aus, welches ausgegeben werden soll. Dieses wird in den BCD-zu-Siebensegment-Decoder geleitet.
P3: wählt anhand des `cnt`-Signals aus, ob ein Signal des RAM ausgegeben werden soll und nutzt in diesem Fall dem Status entsprechende gedimmte oder blinkende Signale.
P4: setzt das `ssegselect`-Signal entsprechend dem aktuellen Counterwert, so dass die Siebensegmentanzeige angesprochen wird, dessen Datum P2 ausgewählt hat.
P2-P4 werden zusammen in einem Prozess behandelt.

blinkdimmer - erzeugt blinkende oder gedimmte Ausgaben



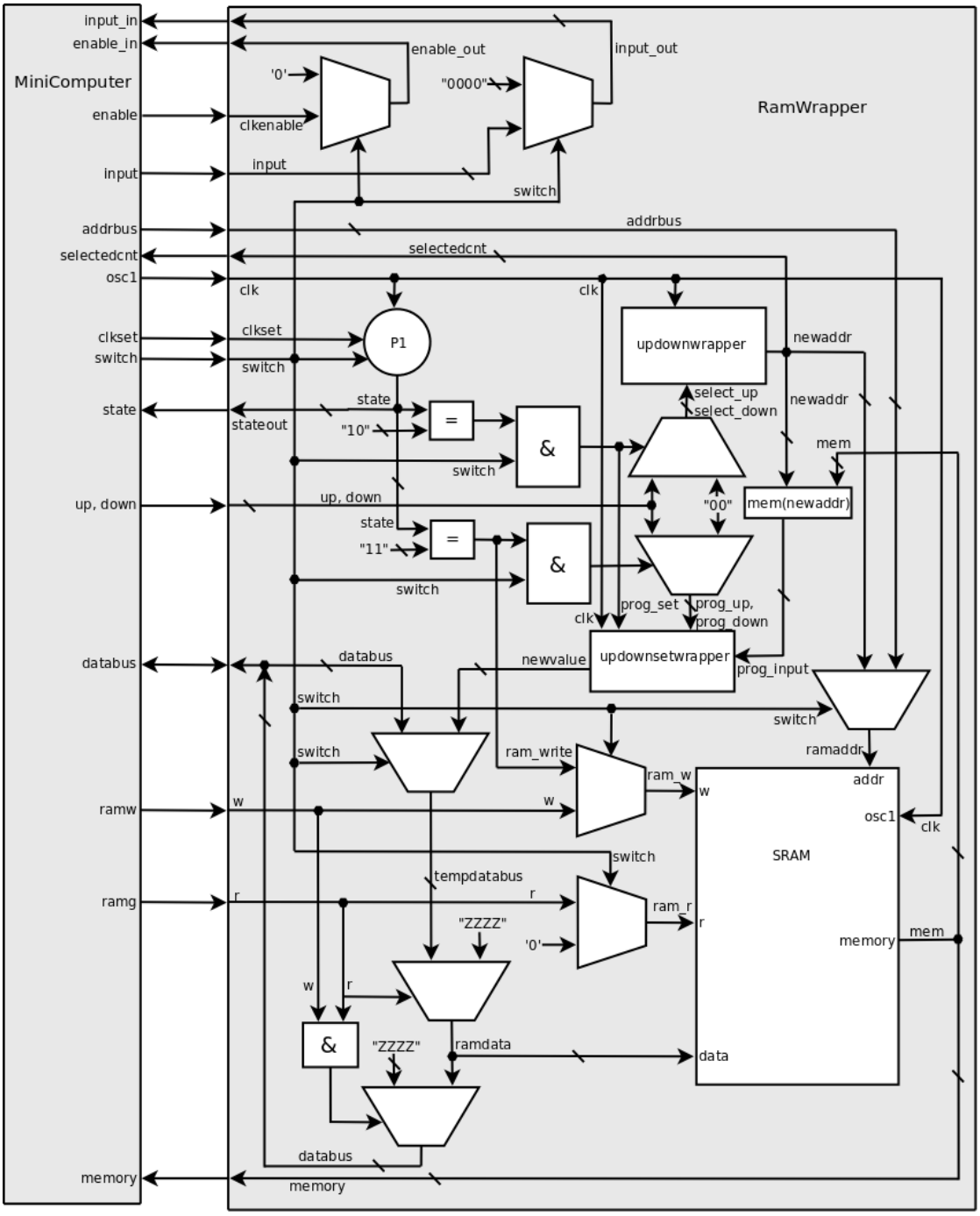
Abhängig vom State und der (im Programmiermodus) ausgewählten RAM-Stelle werden Signale so modifiziert, dass sie entweder blinken, gedimmt sind oder ungehindert passieren.
Der Prozess P1 erzeugt dafür die Signale `blink` und `dimm`, die pulsweitenmoduliert werden. Abhängig von diesen Signalen und des aktuellen Status werden die Signale der Siebensegmentanzeigen entweder weitergeleitet oder die Anzeige ausgeschaltet. Ob die gerade gewählte Anzeige blinken soll (im Auswahlmodus) oder alle bis auf die ausgewählte Anzeige gedimmt werden sollen (im Bearbeiten-modus) wird durch eine bedingte Zuweisung entschieden, die die gewählte Adresse mit der aktuell auszugebenden Adresse vergleicht.

6.2.3 minicomputer - RAM, MiniCPU und Ein- Ausgabe



Der MiniComputer ist der eigentliche Computer. Er besteht aus der MiniCPU, dem RAM-Wrapper (der das RAM beinhaltet) sowie einem Tristate Baustein für die Eingabe und dem Ausgaberegister. Das Signal `cpureg_out` gehört eigentlich mit zu den Registern, die an das Toplevel-Modul übergeben werden um sie auf Siebensegmentanzeigen auszugeben.

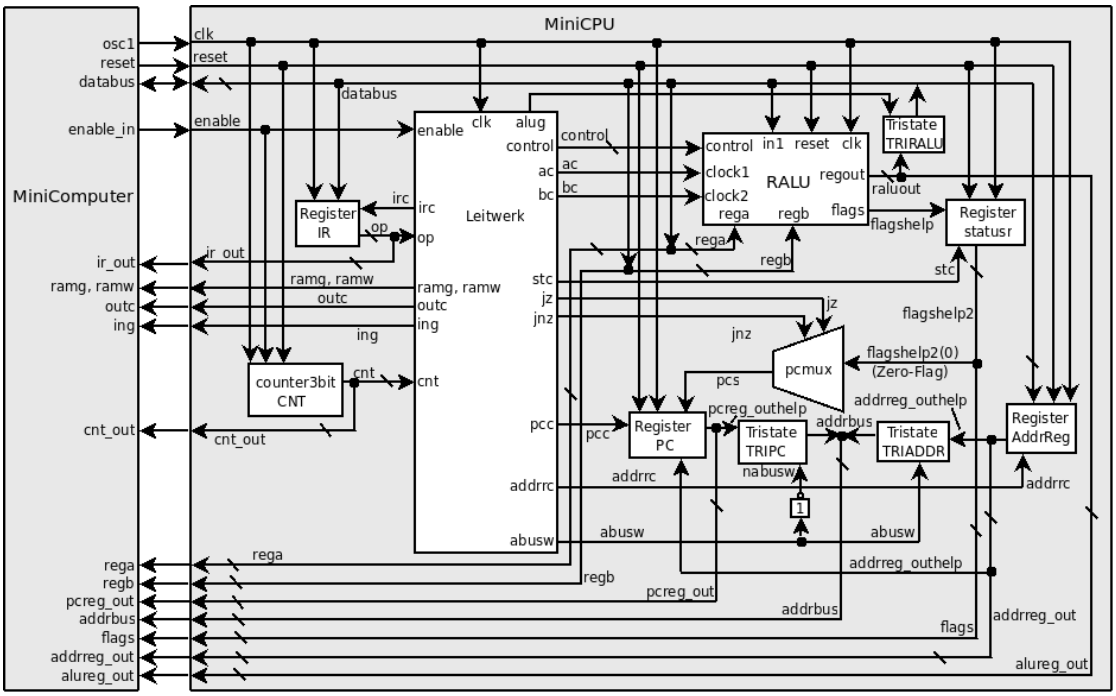
6.2.4 RAM-Wrapper



Der RAM-Wrapper wird zum Programmieren des RAM verwendet. Je nach Einstellung des Programmier-Umschalters (`switch`) wird das erste (höherwertige) Bit des Status (`state`) ge-

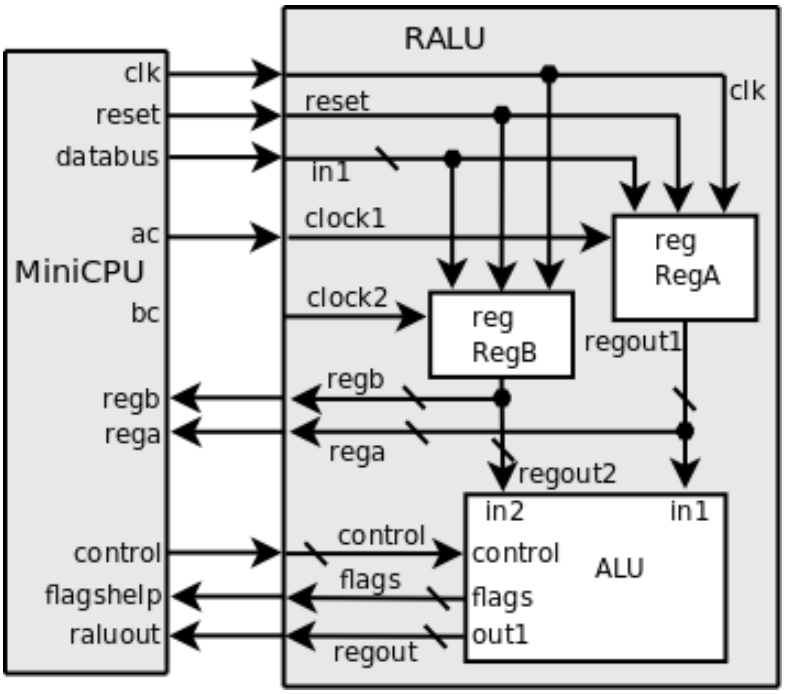
setzt, welches angibt, ob der Normalbetrieb oder der Programmiermodus aktiv ist. Dies wird im Prozess P1 gehandhabt. Mit dem Takt- bwz. Select-Taster (`clksel`) wird im Programmiermodus zwischen den beiden Modi „Auswählen“ und „Bearbeiten“ umgeschaltet, was im `state`-Signal das 2. Bit angibt.
Im Auswahlmodus werden die Up- und Down-Taster auf entsprechende Signale (`select_up` und `select_down`)gelegt und in den updownwrapper geleitet. Dieser berechnet die aktuell ausgewählte Adresse und legt sie auf das Signal `newadr`. Gleichzeitig wird das Set-Bit für den updownsetwrapper, der für den Bearbeiten-Modus zuständig ist, gesetzt, so dass dort der Wert der aktuellen Adresse gespeichert wird.
Im Bearbeiten-Modus werden die Up- und Down-Taster auch auf entsprechende Signale (`prog_up` und `prog_down`) gelegt und der updownsetwrapper erzeugt das Signal newvalue, welches dem neuen RAM-Wert entspricht.
Der Datenbus der MiniCPU sowie das Signal für den Datenbus des RAM werden abhängig der Read- und Write Signale so gesetzt, dass im Normalbetrieb die MiniCPU direkt mit dem RAM verbunden ist und im Programmiermodus der neue Wert ins RAM geschrieben wird. Hierfür werden eine Reihe von Hilfssignalen benötigt, die mittels bedingter Zuweisungen gesetzt werden damit der Datenbus nicht von mehreren Signalen aus beschrieben wird (dies führt sonst zu Problemen während der Synthese).
Die `enable`- und `input`-Signale für die MiniCPU werden im Normalbetrieb weiter geleitet und im Programmierbetrieb auf 0 gesetzt, so dass die MiniCPU während des Programmierens nicht weiter laufen kann.

6.2.5 MiniCPU



Die MiniCPU entspricht im Wesentlichen der MiniCPU aus der Vorlesung. Dieses Modul besteht größtenteils aus port-maps von Registern, RALU, Leitwerk und Tristate Bausteinen. Daneben gibt es noch den Multiplexer für den Adressbus. Die Register, die in diesem Schema zum MiniComputer herausgeführt werden, werden in den anderen Schemata zu „Register (`rega`, `regb`, `addrreg`,...)“ zusammen gefasst. Im Einzelnen sind dies: `rega`, `regb`, `ir_out`, `cnt_out`, `pcreg_out`, `addrbus`, `addrreg_out`, `alureg_out` und `flags`. Im MiniComputer kommt noch das Register `cpureg_out` hinzu. Aus dem Schema sowie dem VHDL-Code sollte die Verschaltung und Logik klar werden. Weitere Informationen zur MiniCPU findet man außerdem auf der Internetseite der technischen Informatik Koblenz: http://www.uni-koblenz.de/~physik/informatik/informatik.html

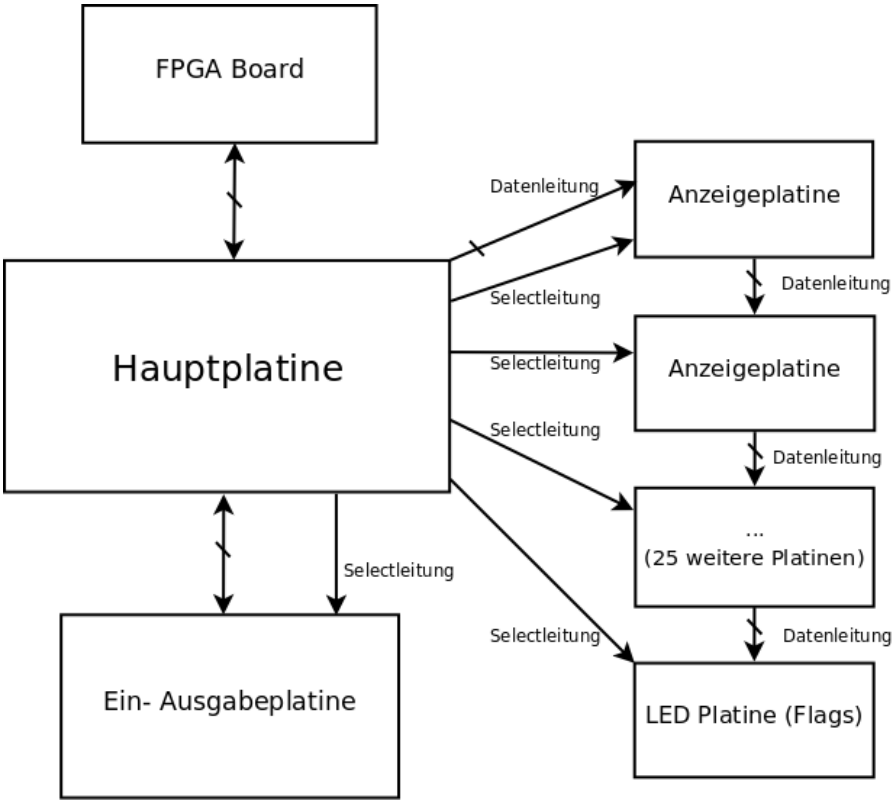
RALU - ALU mit Registern



Die Register-ALU beinhaltet die ALU sowie die beiden Register RegisterA (Accu) und RegisterB.

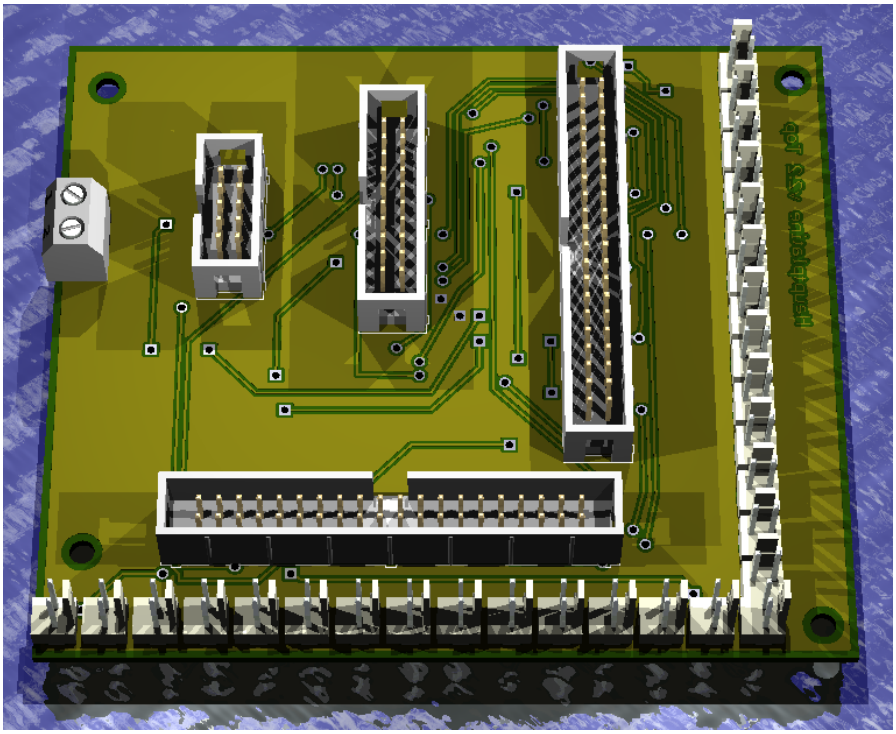
7 Platinen und Schaltungen zur Realisation

Neben der fertigen gekauften Platine mit dem FPGA werden noch einige weitere Boards für die Ein- und Ausgabe, die Siebensegmentanzeigen, die Flags und zum Herausführen von Pins des FPGA benötigt.
Die folgende Grafik veranschaulicht, welche Platinen es gibt und wie sie verbunden sind. Die Pfeile zeigen die Datenrichtung an (bei einem Doppelpfeil werden Daten in beide Richtungen übertragen). Vektoren sind durch einen diagonalen Strich gekennzeichnet.

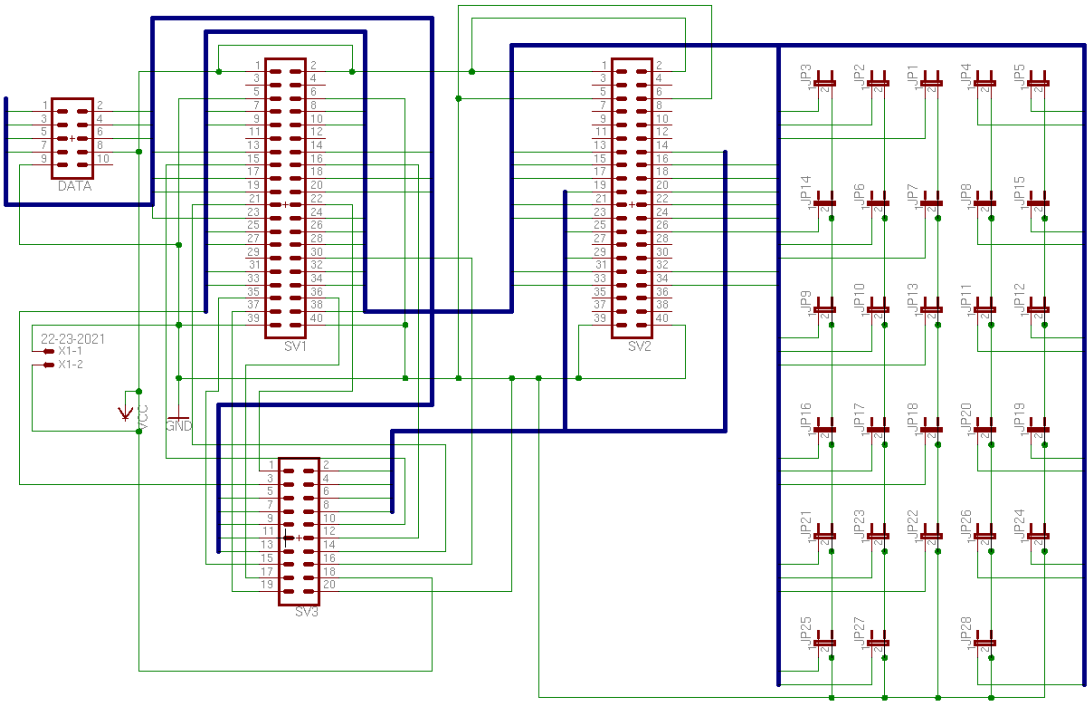


Die einzelnen Platinen werden im folgenden beschrieben.

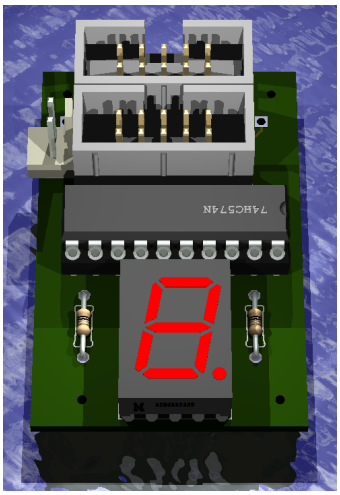
7.1 Hauptplatine



Das FPGA Board führt alle benötigten Pins des FPGA über 2 40-polige Flachbandkabel heraus. An diese Flachbandkabel wird nun die Hauptplatine angeschlossen, die als Verbindungsstück zum FPGA dient.
Die Hauptplatine wird an die 5V Stromversorgung angeschlossen, die an alle anderen Platinen (inklusive dem FPGA Board) weitergeleitet wird.
Des Weiteren werden sieben Output-Pins als Datenleitung zusammen mit Versorgungsspannung und Masse auf ein 10-adriges Flachbandkabel geleitet um die Siebensegmentanzeigen anzusteuern. Außerdem gibt es für jede Siebensegmentanzeige einen Select-Pin. Dabei fiel die Entscheidung auf einen 2-adrigen Steckadapter, wobei das Kabel der Select-Leitung mit einem Massekabel verdrillt wurde um Elektromagnetische Störungen zu unterdrücken. Es hat sich herausgestellt, das dies nicht bei jeder Leitung notwendig war.
Darüber hinaus ist auf der Hauptplatine ein 20-Poliger Wannenstecker verbaut, der über ein Flachbandkabel mit der Ein- Ausgabeplatine verbunden wird. Hierüber werden alle Ein- und Ausgaben zum bzw. vom FPGA weitergeleitet.

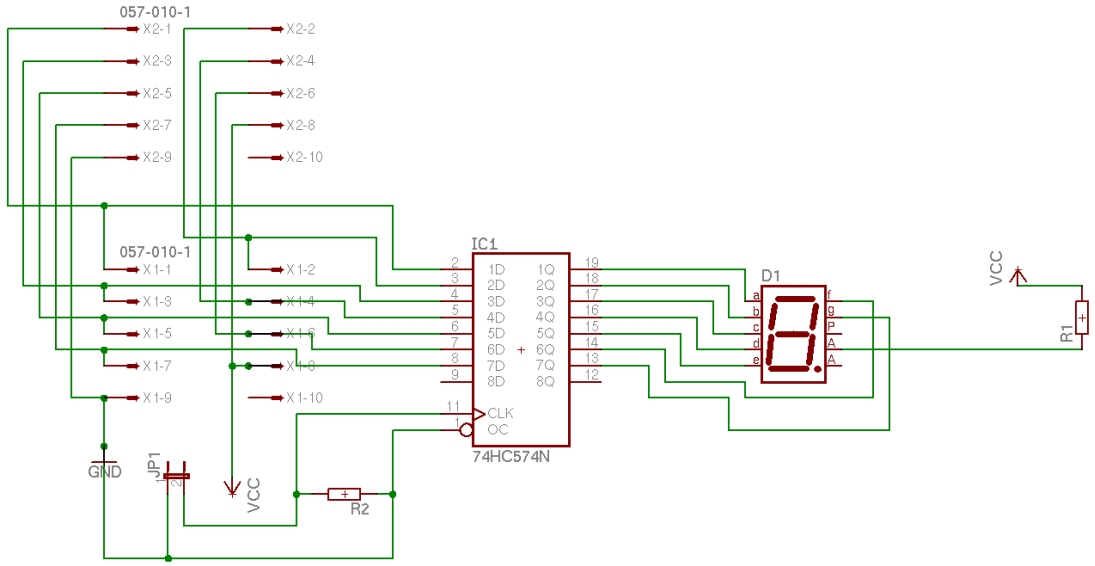


7.2 Anzeigeplatine

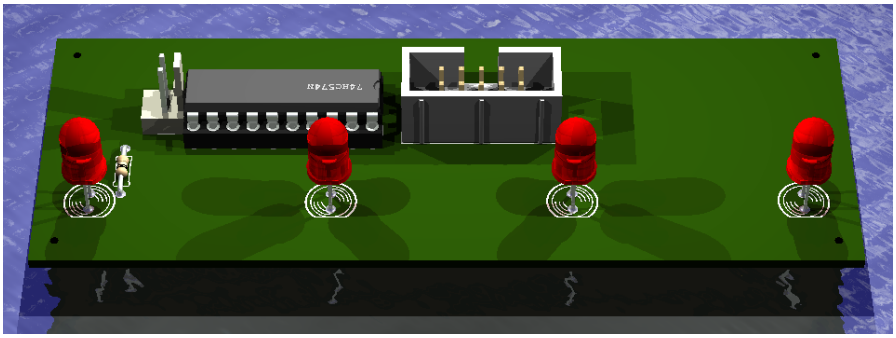


Die Anzeigeplatine dient der Anzeige eines 4-Bit Datums. Für das Projekt wurden 27 dieser Platinen benötigt. Über einen 10-poligen Wannenstecker werden die Daten sowie die Versorgungsspannung zugeführt. Ein weiterer Wannenstecker führt die gleichen Leitungen parallel wieder heraus. Ein Select-Signal wird über einen 2-poligen Steckadapter zugeführt. Die zweite Ader des Steck-

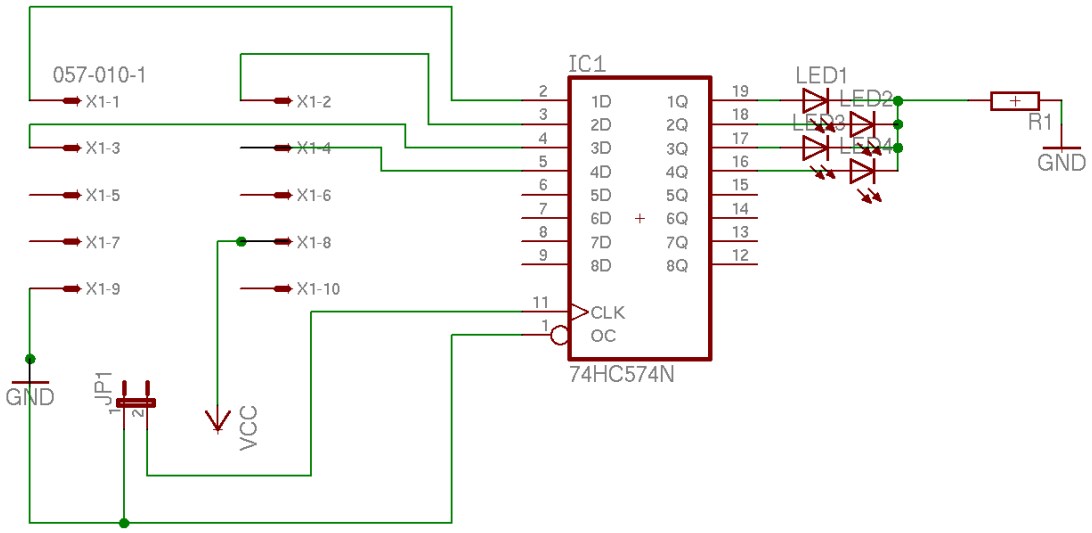
adapters ist auf Masse gelegt, um Störungen zu minimieren. Ein Latch wird vom Select-Signal getaktet und übernimmt die gerade anliegenden Daten zum richtigen Zeitpunkt. Die Ausgänge des Latches sind direkt an eine Siebensegmentanzeige angeschlossen. Die Anode führt dann über einen 27 Ohm Widerstand an die positive Versorgungsspannung (VCC). Die Konvertierung von Binär nach Siebensegment wird schon im FPGA vorgenommen.



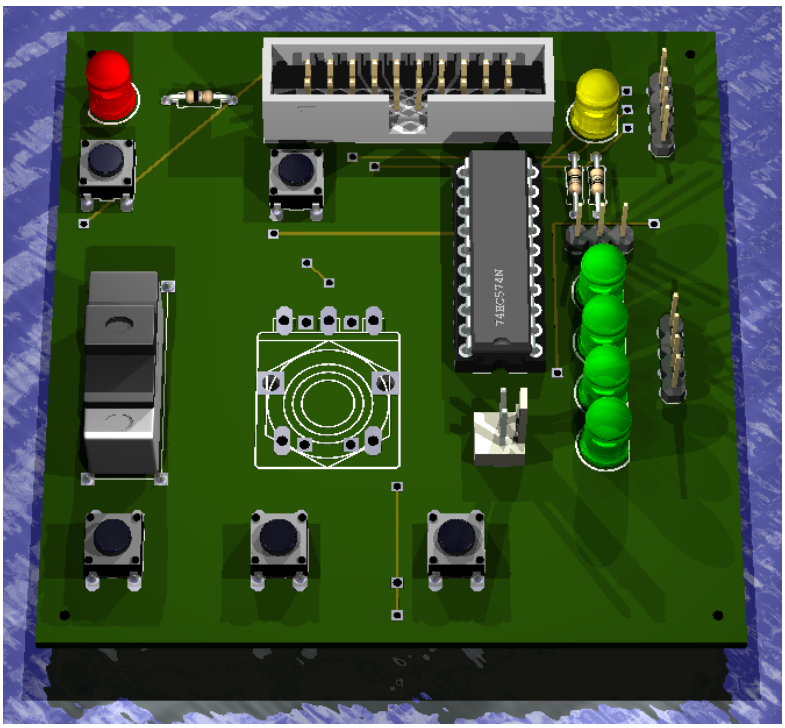
7.3 LED-Platine für Flags



Die LED-Platine nimmt die Daten für die Flags aus der gleichen Leitung wie die Anzeigeplatinen. Daher ist auch hier ein Latch und ein 2-poliger Steckadapter verbaut, die dem gleichen Zweck dienen wie bei den Anzeigeplatinen.
Die Flags werden durch einfache 5mm LEDs in rot angezeigt.
Die LEDs sind gesockelt, wodurch ein Austausch mit größeren LEDs bzw. solche mit einem größeren Abstrahlwinkel leicht möglich ist.



7.4 Ein- Ausgabeplatine

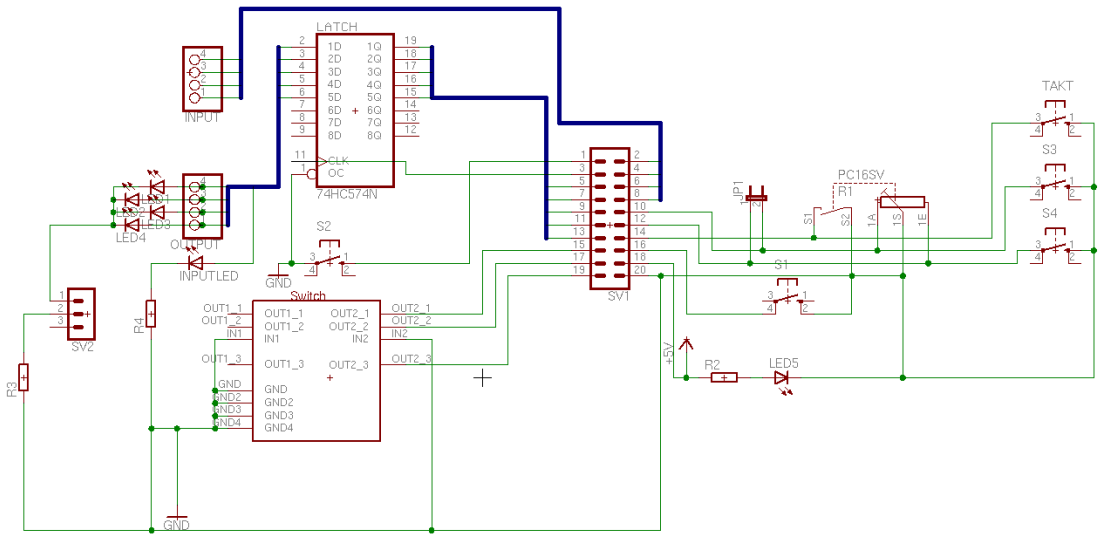


Die Ein- Ausgabeplatine ist mit einem 20-poligen Wannenstecker bestückt um Daten durch ein Flachbandkabel mit der Hauptplatine auszutauschen.
Auch die Stromversorgung wird über dieses Flachbandkabel zur Verfügung gestellt. Es sind 5

Taster verbaut, die die folgenden Funktionen übernehmen:

- Takt
- Inkrementieren der Eingabe
- Dekrementieren der Eingabe
- Reset
- Switchen der Eingabe von manuell auf parallel

Daneben gibt es 4 LED's, die parallel zur Ausgabe geschaltet sind. Diese können durch einen Jumper ein- oder ausgeschaltet werden. Eine Status-LED dient zur Anzeige ob die Eingabe parallel oder manuell entgegen genommen wird. Eine weitere LED zeigt an, ob die MiniCPU eingeschaltet ist. Zur parallelen Ein- und Ausgabe gibt es jeweils 4 Pins. Die parallele Ausgabe wird über die gleichen Datenleitungen ausgegeben, wie die Anzeigeplatinen. Daher ist auch ein Latch und ein 2-poliger Steckadapter auf dem Board um die Daten im richtigen Moment speichern zu können.
Ein Schiebeschalter mit 3 Stellungen dient zur Auswahl des Taktes (Manuell, 1Hz oder 100Hz). Schließlich wurde alles vorbereitet um einen Drehencoder aufzunehmen. Bei Bedarf kann dieser einfach eingelötet werden. Dazu muss jedoch der VHDL Code angepasst werden, damit die Drehungen richtig erkannt werden. Der Drehencoder kann auch zum manuellen Takten des Prozessors genutzt werden.



7.5 Pinbelegung des FPGA-Boards

Die folgende Tabelle Zeigt, welche Leitungen des FPGA an welche Pins der Wannenstecker führen und wie diese belegt sind. SL = Select Leitung, DL = Datenleitung, i = Dieser Pin ist nur als Eingang nutzbar

Bank2 (SV1)			**Bank1 (SV2)**		
WSL-Nr.	Pin-Nr.	Belegung	WSL-Nr.	Pin-Nr.	Belegung
1		V_{in}	1		V_{in}
2		V_{in}	2		V_{in}
3		VCC 3,3V	3		VCC 3,3V
4		VCC 3,3V	4		VCC 3,3V
5		GND	5		GND
6		GND	6		GND
7	2	SL_3	7	144	JTAG TDI
8	3	SL_2	8	109	JTAG TDO
9	4	SL_1	9	110	JTAG TCK
10	5	SL_4	10	108	JTAG TMS
11	13, 28	VCCIO3	11	121, 138	VCCIO
12	13, 28	VCCIO3	12	121, 138	VCCIO
13	8	DL	13	142	SL_{16}
14	7	DL	14 i	141	$Input_1$
15 i	12	Up	15	140	SL_{17}
16 i	10	Down	16	139	SL_18
17	15	DL	17	135	SL_20
18	14	DL	16	134	SL_19
19	17	DL	19 i	136	$Input_2$
20	16	DL	20	132	SL_21
21 i	18	CLK	21	131	SL_23
22 i	6	InputSelect	22	130	SL_22
23	21	DL	23	126	SL_26
24	20	SL_5	24	125	SL_24
25	23	SL_{14}	25 i	129	$Input_3$
26	22	SL_6	26	124	SL_25
27	26	SL_7	27	123	nicht verw.
28	25	SL_8	28	122	OSC1
29	31	VREF	29 i	120	$Input_4$
30 i	29	RST	30 i	119	nicht verw.
31	33	SL_9	31	117	SL_{27}
32	32	SL_{10}	32	116	SL_{15}
33	35	SL_{13}	33	113	SL_{28}
34	34	SL_{11}	34	112	SL_{29}
35 i	36	CLKSEL	35 i	114	nicht verw.
36 i	24	CLKSEL	36 i	111	nicht verw.
37 i	41	CLKSEL	37 i	107	nicht verw.
38	71	SL_{12}	38 i	106	nicht verw.
39		GND	39		GND
40		GND	40		GND

7.6 Aufbau auf der Spanholzplatte

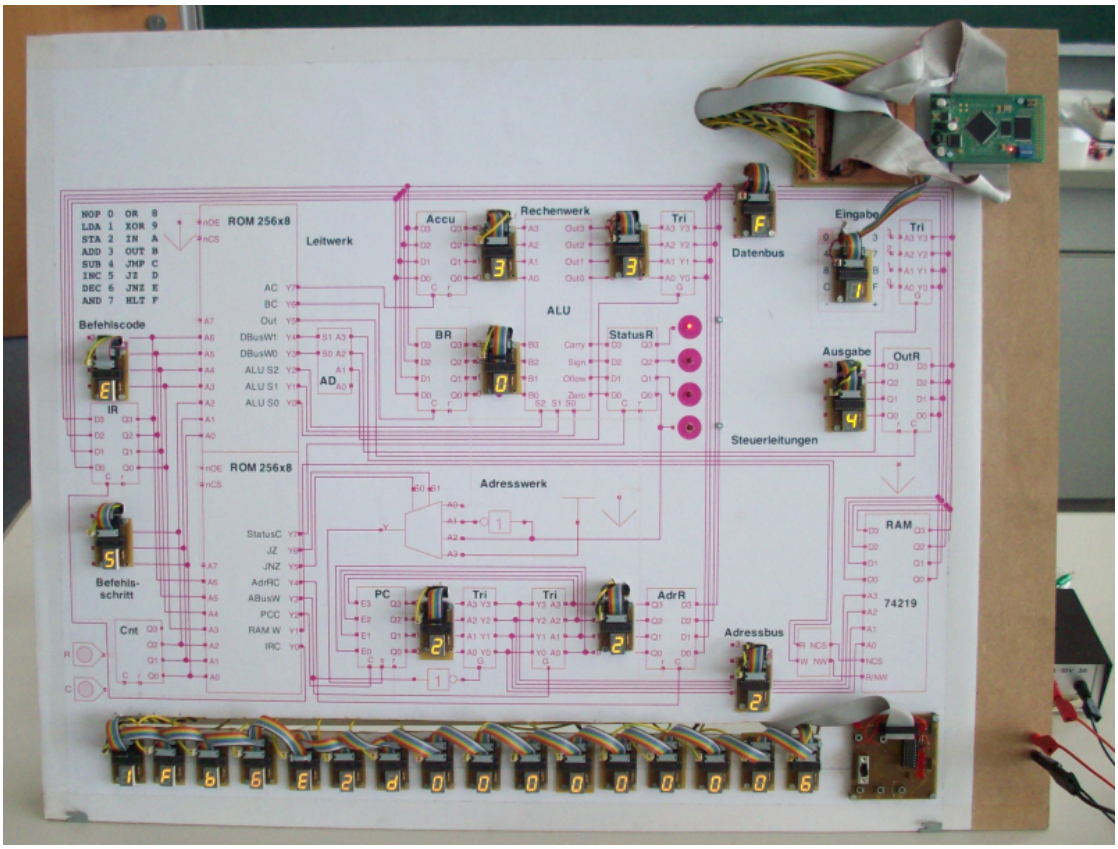
Der komplette Aufbau besteht aus einer Spanholzplatte von 70 x 90 cm (Höhe x Breite). Darauf wurde ein stark vergrößerter Ausdruck der MiniCPU aus der HADES-Schaltung geklebt. Auf

jede in Hades existierende Siebensegmentanzeige wurde nun eine Anzeigeplatine angebracht, so dass die Anzeigen an den gleichen Positionen sind wie auch in HADES. Unterhalb der MiniCPU wurden zusätzlich die 16 Siebensegmentanzeigen angebracht, die den Inhalt des RAM anzeigen.

Das FPGA-Board ist direkt neben der Hauptplatine in der rechten oberen Ecke der Spanholzplatte angebracht. Neben der Hauptplatine ist ein Loch in die Spanplatte gebohrt, durch das sämtliche Selectleitungen für die Siebensegmentanzeigen sowie das Flachbandkabel zur Verbindung mit der Ein- Ausgabeplatine geführt werden. Über jede Siebensegmentanzeige wurde ein Schlitz in die Platte gesägt, durch den die Datenleitung und die Selectleitung an die Anzeige geführt werden kann.

Die LED-Platine wurde so entwickelt, dass die LEDs genau in die entsprechenden Punkte der LEDs im HADES-Bild passen. Sie könnten noch durch größere LEDs ersetzt werden, die einen größeren Abstrahlwinkel besitzen damit man sie besser erkennen kann. Die Ein- Ausgabeplatine wurde in der rechten unteren Ecke neben zwei Bananenbuchsen für die Stromversorgung angebracht.

Durch diesen Aufbau soll der Vergleich der in VHDL implementierten MiniCPU mit der HADES Simulation erleichtert werden.



8 Fehler und Probleme

Bei der Umsetzung der MiniCPU in Hardware sind leider einige Probleme aufgetreten. Während es in der Simulation noch vergleichsweise einfach ist, Fehler zu lokalisieren, da sämtliche Signale angezeigt werden können, ist dies in der Realität nicht so einfach möglich. Es kommt auch vor, dass ein Modul in der Simulation einwandfrei funktioniert und auf der Hardware nicht zum Laufen zu bekommen ist.

8.1 Verify

Die Verify Funktion des ISE ergibt eine Fehlermeldung, weswegen man nicht sicher gehen kann, dass das Programm korrekt in den FPGA (bzw. das EEPROM) übertragen wurde. Dieses Problem ließ sich auch bis zum Schluss nicht lösen. Jedoch traten Fehler beim Übertragen nur sehr selten auf und wurden dann immer auf Grund eines Abbruchs der Übertragung als Fehler angezeigt.

8.2 Drehencoder

Unsere erste Idee für die Eingabe war die Nutzung eines Drehencoders. Dieser hätte gleichzeitig auch als Taster für den manuellen Takt gedient.
Leider hat der getestete Drehencoder beim Drehen zwischen zwei Einrastpunkten teilweise undefinierte Signalpegel gezeigt. Dadurch war es nicht möglich eine eindeutige Erkennung zu gewährleisten. Entweder wurden bei zu langsamen Drehen mehrere Takte gezählt oder bei zu schnellen Drehen der Takt nicht erkannt.
Deswegen wurden Taster zum Inkrementieren und Dekrementieren sowie zum manuellen Takten der MiniCPU verwendet.

8.3 Fehler der MiniCPU

Input Datenbus

Bei den ersten Tests auf der Hardware gab es das Problem, dass der Input immer auf dem Datenbus lag. Dies sollte aufgrund der Tristatebausteine nicht passieren. Um den Rest der MiniCPU testen zu können, haben wir den Input vorerst abgeschaltet. Doch auch dann traten Fehler in der MiniCPU auf. Erst durch die Synchronisation wurden diese Fehler behoben und auch der Input wurde danach nur zu den gewünschten Zeitpunkten auf den Datenbus gelegt.

Zufallswerte auf dem Datenbus und im RAM

Auch bei abgeschaltetem Input lagen immer wieder zufällige Werte auf dem Datenbus. Gleichzeitig veränderten sich zufällig Werte im RAM.
Diese Probleme wurden auch durch die fehlende Synchronisation ausgelöst. Nach der Synchronisation des gesamten Systems waren die Fehler behoben.

8.4 Probleme beim Erkennen und Entprellen von Tastendrücken

Die Erkennung von Tastendrücken und deren Entprellung wurde mittels eines Counters implementiert, der bei Druck des Tasters das Signal nur einen Takt lang durch lässt und dann bis zum Überlauf des Counters wartet.
Problematisch bei diesem Ansatz war, dass der Counter entweder zu lange bis zum Überlauf gebraucht hat, so dass schnelle Tastendrücke hintereinander nicht möglich waren oder zu kurz, so dass bei einem Tastendruck 2 Takte durch kamen und somit ein Sprung entstand. Dieses Problem konnte durch eine zusätzliche Variable gelöst werden. Erreicht der Counter den Wert der Variablen, wird das Signal einen Takt lang auf High gesetzt und der Counter auf 0 gesetzt. Dadurch war eine viel genauere Justierung der Empfindlichkeit möglich.

8.5 Probleme beim Programmieren des RAM

Der erster Ansatz zur Programmierung des RAMs war ein RAM-Arbiter, der die vorhandenen Busse und Counter nutzte und lediglich auf das RAM umleitete. So wurde die Eingabe direkt auf den Datenbus gelegt und der Adressbus über einen Counter angesprochen, der vom CPU-Takt inkrementiert wurde. Gleichzeitig wurde der Write-Eingang des RAMs fest auf High gelegt.
Somit konnte man mittels der Eingabe den Wert in der aktuellen Adresse des RAMs ändern und mit dem CLK-Taster die Adresse ändern. Dies erschien zunächst eine elegante Möglichkeit zu sein, da möglichst viele Komponenten der MiniCPU verwendet wurden um den RAM zu programmieren. Es wurde lediglich ein zusätzlicher Counter benötigt.
Dabei traten jedoch unerwartete Seiteneffekte auf, wobei zufällige Stellen des RAMs auf die aktuelle Eingabe geändert wurden (und nicht nur die aktuell ausgewählte Stelle). Beim Umschalten des Programmier-Umschalters auf Normalbetrieb änderte sich zusätzlich jedes Mal die erste Stelle des RAMs.
Diese Lösung war auch schlecht bedienbar, da man um eine Stelle im RAM zu ändern den gesamten RAM bis zu dieser Stelle neu programmieren musste.
Aus diesen Gründen wurde dann ein anderer Ansatz gewählt und ein RAM-Wrapper geschrieben, der den RAM beinhaltet. Im Normalbetrieb werden nun alle Signale direkt ins RAM geleitet. Im Programmiermodus werden die Taster direkt abgegriffen und in eine Programmierlogik geleitet um den RAM komfortabel programmieren zu können. Die genaue Funktionsweise des RAM-Wrappers ist im Kapitel 6.2.4 auf Seite 49 beschrieben.

8.6 Probleme durch elektromagnetische Einstrahlung

Auf dem unserem Hardwarelabor gegenüberliegenden Dach befindet sich eine Mobilfunkantenne, die sehr wahrscheinlich für eine recht hohe elektromagnetische Einstrahlung bei unserem Aufbau verantwortlich ist.
Bei der Ansteuerung der Siebensegmentanzeigen traten unerklärliche Fehler auf. Ein zum Testen erstelltes Modul zeigte, dass die Methode der Ansteuerung grundsätzlich funktioniert. Allerdings kam es sporadisch vor, dass einzelne Anzeigen nicht die Werte ausgaben, die anliegen sollten. Indem man die Hand um die Signalleitungen bewegte, konnte man diesen Effekt beeinflussen, wodurch sich schließen ließ, dass der Fehler durch Einstrahlung/Induktion von entweder den anderen Signalleitungen oder möglicherweise der Mobilfunkantenne herrührten. Um diesen Einfluss zu mindern haben wir einige Signalleitungen der Anzeigen mit masseführenden Kabeln verdrillt und die Taktfrequenz des Moduls reduziert.
Ein weiteres Problem durch elektromagnetische Strahlung ergab sich bei der Verbindung der Ein-Ausgabeplatine mit der Hauptplatine. Ist das Verbindungskabel zu lang, ändert sich die Eingabe, ohne das ein Taster gedrückt wird.

8.7 Displayprobleme

Probleme beim Multiplexen

Am Anfang sollten die Anzeigeplatinen nur aus der Siebensegmentanzeige bestehen. Dabei sollten die Datenleitungen über alle Anzeigen geteilt werden und jede Anzeige separat über die gemeinsame Anode aktiviert werden. Beim Testen sind allerdings die Anzeigen schwarz geblieben, obwohl die Leitungen richtig angesteuert wurden. Dies lässt sich darauf zurückführen, dass beim Multiplexen mit 27 Anzeigen jede Anzeige auch nur $\frac{1}{27}$ der Zeit aktiv ist und somit die Spannung nicht mehr ausreicht, um die Anzeige zum Leuchten zu bringen. Daher wurde zusätzlich ein Latch auf jeder Anzeigeplatine platziert und die Anzeigen an die Ausgänge der Latches angeschlossen. Die Selectleitungen des Multiplexers sind an den Takteingang eines jeden Latches angeschlossen, so dass das Latch jedes Mal übernimmt, wenn die Selectleitung aktiv ist.
Ein weiterer Fehler ist aufgetreten, bei dem die Anzeige an der ersten Signalleitung zusätzliche Clock-Signale von anderen Anzeigen bekommen hat. Dieser Fehler ist nach dem Aufräumen des Projektverzeichnisses wieder verschwunden. Die genaue Ursache des Fehlers ließ sich somit nicht mehr feststellen.

Fehler im Datenblatt

Ein Pin des FPGA war im Datenblatt des Boardherstellers falsch angegeben, wodurch eine Anzeige keinen Wert anzeigen kann. So musste die Schaltung und das Layout der Hauptplatine entsprechend geändert werden.

8.8 Verwendung des Flash Speichers zur Programmierung

Das FPGA Board besitzt einen Flashspeicher, der sich zur Programmierung des FPGAs nach einer Unterbrechung der Stromversorgung eignet. Dadurch braucht man den FPGA nicht jedes Mal neu zu Programmieren. Leider besitzt der verwendete FPGA (Xilinx Spartan 3E XC3S100E) keine Möglichkeit den auf dem FPGA-Board bestückten Flash über die JTAG-Schnittstelle zu beschreiben. Man hätte einen Mikrocontroller entsprechend Programmieren müssen um die Binärdatei in den Flashspeicher zu schreiben. Alternativ hätte man auch eine VHDL Schaltung realisieren können, die den Flash beschreibt.
Leider wären beide Varianten selbst ein Projekt für sich, welches den Rahmen dieser Studienarbeit gesprengt hätte. Daher musste leider auf die Nutzung des Flashspeichers verzichtet werden.

8.9 Synchronisation

Beim Übergang von der reinen Simulation zum Betrieb auf dem FPGA können Probleme auftreten, da die Simulation nicht den FPGA simuliert, sondern nur das Verhalten des VHDL-Codes eines Moduls. Beim FPGA selbst treten z.B. Laufzeitunterschiede auf, die in der Simulation nicht erkannt werden können. Laufzeitunterschiede können durch verschiedene Leitungslängen und -eigenschaften hervorgerufen werden. Sie führen zu unvorhersagbarem Verhalten des FPGAs.

Fehler, die durch Laufzeitunterschiede verursacht wurden, kann man durch Synchronisation beheben: Flipflops und Register sollten innerhalb des FPGAs alle über den „globalen“ Takt des FPGA (Systemtakt) getaktet werden. Erreicht wird dieses Verhalten durch das Einführen eines zusätzlichen Signals, welches den Takt aktiviert (im Folgenden Takt-Enable Signal genannt). Dieses Signal wird in die entsprechenden Module geleitet und mit dem Systemtakt und-verknüpft. Durch diese Vorgehensweise übernimmt ein Register oder Flipflop nur dann das eingehende Signal, wenn der Systemtakt und das Takt-Enable Signal einen High-Pegel besitzen.

Somit wird das FlipFlop (bzw. das Register) mit dem Systemtakt synchronisiert. Dies bedeutet, dass selbst wenn die Taktsignale der Flipflops zu unterschiedlichen Zeiten ankommen, sie doch zur gleichen Zeit übernehmen, weil der Systemtakt des FPGAs über ein spezielles Taktnetz geleitet wird. [5] Es sollte darauf geachtet werden, dass kein nicht-Takt-Signal über das Taktnetz geleitet wird, da bei diesem u. U. versucht wird, das Signal über Verzögerungselemente phasengleich zu halten, was allerdings nur bei gleichmäßigen Taktsignalen funktioniert und bei anderen Signalen zu unerklärbaren Fehlern führen kann.

Ein weiteres Problem ensteht, wenn das Takt-Enable Signal länger als einen Systemtakt lang auf High gesetzt ist, da dadurch der anliegende Wert bei jedem Systemtakt erneut übernommen werden würde. Daher wurde eine Variable eingeführt, die dafür sorgt, dass der anliegende Wert nur einmal übernommen wird. Das folgende Beispiel soll dies etwas verdeutlichen:

Beispiel:

4-Bit Register ohne Synchronisation:

```
entity reg is
  Port(
    CLK   : in   std_logic;
    reset : in   std_logic;

    D : in  std_logic_vector(3 downto 0);
    Q : out std_logic_vector(3 downto 0)
  );
end regreset;

architecture reg_logic of reg is
signal mem : std_logic_vector(3 downto 0);
begin
  process(CLK, reset)

  begin
      if reset='1' then
        mem<="0000";
      elsif CLK'event and CLK = '1' then

        mem <= D;




      end if;
  end process;
  Q<=mem;
end regreset_logic;
```

4-Bit Register mit Synchronisation:

```
entity regsync is
  Port(
    CLK    : in   std_logic;
    reset  : in   std_logic;
    enable : in   std_logic;
    D : in  std_logic_vector(3 downto 0);
    Q : out std_logic_vector(3 downto 0)
  );
end regsync;

architecture regsync_logic of regsync is
signal mem : std_logic_vector(3 downto 0);
begin
  process(CLK, reset)
  variable senable: std_logic :='1';
  begin
    if reset='1' then
      mem<="0000";
    elsif CLK'event and CLK = '1' then
      if enable='1' and senable='1' then
        mem <= D;
        senable:='0';
      elsif enable='0' and senable='0' then
        senable:='1';
      end if;
    end if;
  end process;
  Q<=mem;
end regsync_logic;
```

Die Variable `senable` aktiviert das Takt-Enable Signal einen Takt lang und wird sobald der anliegende Wert übernommen wurde auf 0 gesetzt. Erst wenn das Takt-enable Signal auf 0 ist, wird das Signal `senable` wieder auf 1 gesetzt und der Anliegende Wert kann beim nächsten High-Pegel des Takt-Enable Signals übernommen werden.

9 Fazit

9.1 Umfang und Aufwand

Der Aufwand der Arbeit war erheblich höher als erwartet. Nach ca. 15 Arbeitstagen lief die MiniCPU bereits im Simulator. Dieser Teil ging sehr schnell.
Durch die anfangs fehlende Synchronisation der MiniCPU bzw. des MiniComputers traten jedoch auf der Hardware immer wieder unerklärbare Fehler auf. Erst durch das einheitliche Synchronisieren des gesamten Systems wurden diese behoben. Dies zu erkennen brauchte auch eine gewisse Zeit.
Die Entwicklung der Platinen nahm dann auch recht viel Zeit in Anspruch. Zum Einen waren die Ergebnisse beim Ätzen der Platinen teilweise sehr schlecht, so dass wir viele Ätzvorgänge wiederholen mussten. Zum Anderen mussten wir die Platinenlayouts mehrmals ändern und Fehler korrigieren. Gerade die Hauptplatine, die doppelseitig geätzt wurde erforderte eine gewisse Geschicklichkeit beim Belichten um das Layout passgenau aufzutragen.
Auch die Entwicklung des RAM-Wrappers und der restlichen hardwarespezifischen Module war sehr Zeitaufwändig. Dort tauchten bis zum Schluss immer wieder unerwartet Probleme auf.
Allgemein war auch der Umstand, dass wir VHDL erst mit diesem Projekt erlernten dafür verantwortlich, dass einige Module ungünstig implementiert wurden und später umgeschrieben werden mussten.

9.2 Kritik an VHDL

VHDL hat aus unserer Sicht eine teilweise recht umständliche Syntax.
Bei der Deklaration einer Entity und einer Architecture wird diese mit „is“ eingeleitet. Bei einer Component hingegen wird kein „is“ benötigt. Diesen Unterschied muss man sich merken, da er oft zu Syntaxfehlern bei der Synthese führt. Der Grund für diese Unterscheidung ist uns nicht ersichtlich.
Auch die Notationsunterschiede zwischen Bits und Vektoren, die mit einfachen oder doppelten Anführungszeichen gekennzeichnet werden, sind aus unserer Sicht unnötig.
Ähnliches gilt für die end-Bezeichner. So reicht es bei Components „end component“ zu schreiben, bei Architectures jedoch muss man mit „end *NAME*“ den Namen der Architecture angeben.
Grundsätzlich wären geschweifte Klammern anstatt der „begin“ und „end“ Schlüsselwörter wesentlich einfacher zu handhaben, da auch einige Editoren geschweifte Klammern besser unterstützen.
Eigenartig bei der Port-Deklaration fanden wir, dass im Gegensatz zu allen anderen Ausdrücken die Deklaration des letzten Signals nicht mit einem Semikolon abgeschlossen wird.
Sehr umständlich empfanden wir, dass die Bibliotheken für jedes Modul innerhalb einer Datei erneut importiert werden müssen und dies nicht dateiweit geschehen kann.
Die hier aufgeführten Umstände sind nach unserer Meinung unnötig und erschweren das Entwickeln in VHDL dahingehend, dass man sich alle Ausnahmen merken muss. Somit kommen leicht sehr viele Syntax-Fehler zustande. Diese müssen dann bei der Synthese einzeln behoben werden.

9.3 Ziele

Bis auf die Tatsache, dass es uns nicht möglich ist den Flash-Speicher auf dem FPGA-Board zu nutzen haben wir alle unsere Ziele erreicht.
Aufgrund der Funktionsweise eines FPGA konnten wir die MiniCPU aus der Vorlesung zwar nicht 1 zu 1 nachbauen, jedoch blieben wir sehr nahe am Design derselbigen. Die Unterschiede liegen vor allem bei der Synchronisation, dem Umstand, dass die Busse intern zu unidirektionalen Bussen umgewandelt werden und der Tatsache, dass wir einen RAM-Wrapper erstellen mussten um den RAM programmieren zu können. Darüber hinaus sollten alle Register und Busse auf Siebensegmentanzeigen heraus geführt werden. Dies veränderte das interne Layout auch gegenüber dem eigentlichen Layout der MiniCPU.
Dies alles sind jedoch gewollte bzw. unvermeidliche Änderungen am Design der MiniCPU. Von der Funktionsweise und von der Bedienung her sind unsere Arbeit und die MiniCPU aus der Vorlesung sehr ähnlich.

10 Anhang

A Installation von Software und Treibern

In diesem Abschnitt wird die Vorbereitung des Systems beschrieben.

Download und Installation von ISE unter Linux

Das "Webpack ISE" ist die kostenlose Entwicklungsumgebung für FPGAs von Xilinx. Man kann sie unter http://www.xilinx.com/ise/logic_design_prod/webpack.htm beziehen.
Sie unterscheidet sich von der kostenpflichtigen Variante "ISE Foundation" dadurch, dass beim ISE Webpack einerseits die Unterstützung für die größeren FPGA-Modelle fehlt und andererseits nur die 32-Bit Varianten der Software verfügbar sind. Beide Varianten der Software sind sowohl für Linux als auch für Windows verfügbar.
Die Installationsdatei ist 3GB groß und kann direkt ausgeführt werden, nachdem man mit dem Befehl `chmod +x ise` das Ausführungsbit der Datei gesetzt hat.
Der Installationsvorgang ist sehr unkompliziert und unterscheidet sich kaum von dem eines normalen Windows-Programms.
Die Cable Driver brauchen nicht installiert zu werden, da es unter Linux einfacher ist, einen alternativen Treiber für den Programmieradapter zu installieren.
Nach der Installation können noch zusätzlich ungefähr 1GB an Updates installiert werden.
Danach ist ISE fertig installiert und man kann anfangen VHDL (oder Verilog) zu programmieren und einen FPGA (oder CLPD) zu simulieren.
Zum Flashen einer fertigen Binärdatei auf den FPGA kann man iMPACT benutzen, welches ebenfalls im ISE Paket enthalten ist.

Installation der Treiber

Zum Betreiben des JTAG Parallelportkabels unter Linux ist für gewöhnlich ein sog. WinDriver erforderlich. Dieser ist jedoch schwierig zu installieren, da er als Kernelmodul installiert werden muss.
Für die verwendeten Debian- und Ubuntu-Systemen wurde ein alternativer Treiber verwendet, welcher den WinDriver emuliert und als Library installiert wird. Diesen Treiber kann man unter http://www.rmdir.de/~michael/xilinx/ beziehen.
Um den Treiber zu kompilieren wird die Bibliothek libusb-dev benötigt. Unter Debian Systemen kann man diese per `apt-get install libusb-dev` installieren.
Nach dem Download muss der Treiber kompiliert und installiert werden (`make` und anschließendes kopieren der Datei libusb-driver.so nach `/usr/lib`). Vor dem Starten von iMPACT ist darauf zu achten, dass der Treiber mit dem Befehl

```
export LD_PRELOAD=/usr/lib/libusb-driver.so iMPACT
```

geladen wird.
Um als normaler User den FPGA flashen zu können benötigt man Schreibrechte auf den Parallelport. Diese kann man durch den Befehl

```
chmod oug+rwx /dev/parport0
```

(als root ausführen) erreichen.

Startscript

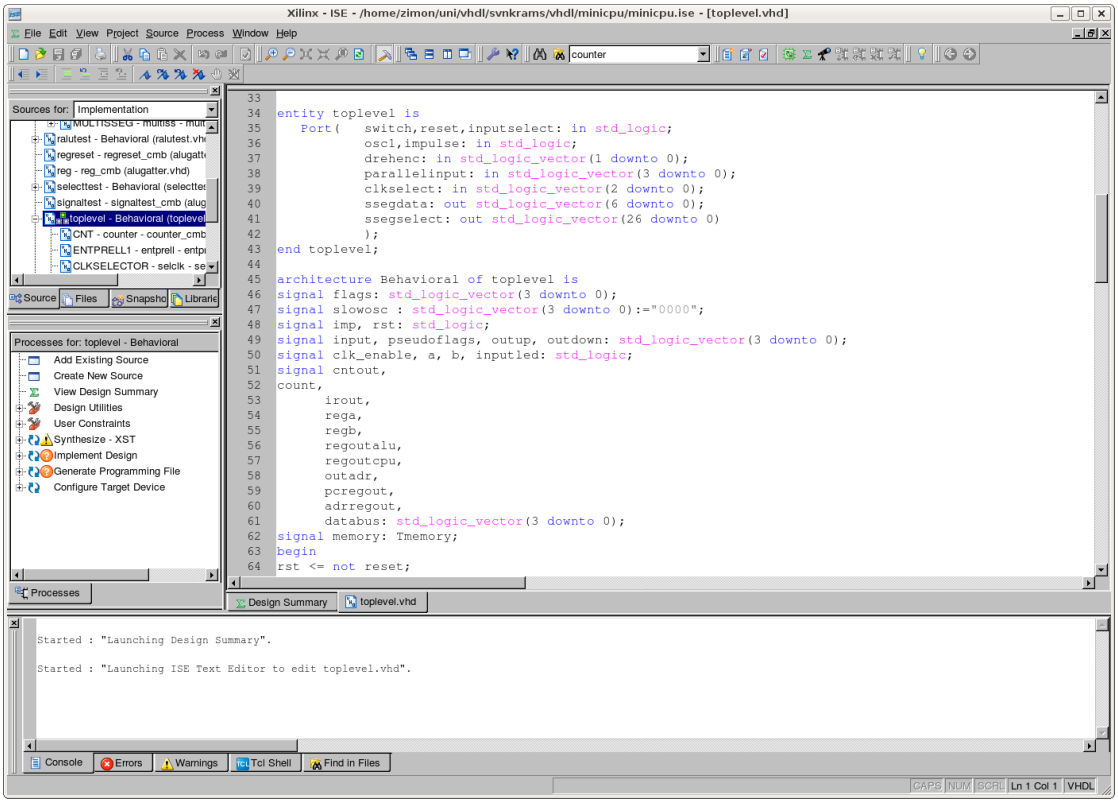
Da unter Ubuntu nach jedem Neustart des Systems die Rechte des Parallelports neu gesetzt und nach jedem Login die Treiber neu geladen werden müssen, wurde ein Startscript erstellt, welches diese Vorgänge automatisiert und ISE sowie iMPACT startet. Das folgende Script kann man nun irgendwo auf der Festplatte als `startise.sh` speichern und mit `chmod +x startise.sh` als ausführbare Datei kennzeichnen. Beim Ausführen wird man nach seinem Passwort gefragt um die Rechte des Parallelport setzen zu können (Der User benötigt Rechte um das System zu administrieren, also um `sudo` bzw. `gksu` auszuführen). Der Pfad zum ISE Verzeichnis muss entsprechend geändert werden.

```
#!/bin/bash

ISE_DIR=/Pfad/zu/ISE # Hier Pfad zu ISE einfügen

gksudo 'chmod oug+rw /dev/parport0'

export LD_PRELOAD=/usr/lib/libusb-driver.so
$ISE_DIR/bin/lin/ise &
$ISE_DIR/bin/lin/impact &
```

B Grundlagen zur Bedienung der Software

B.1 Webpack ISE

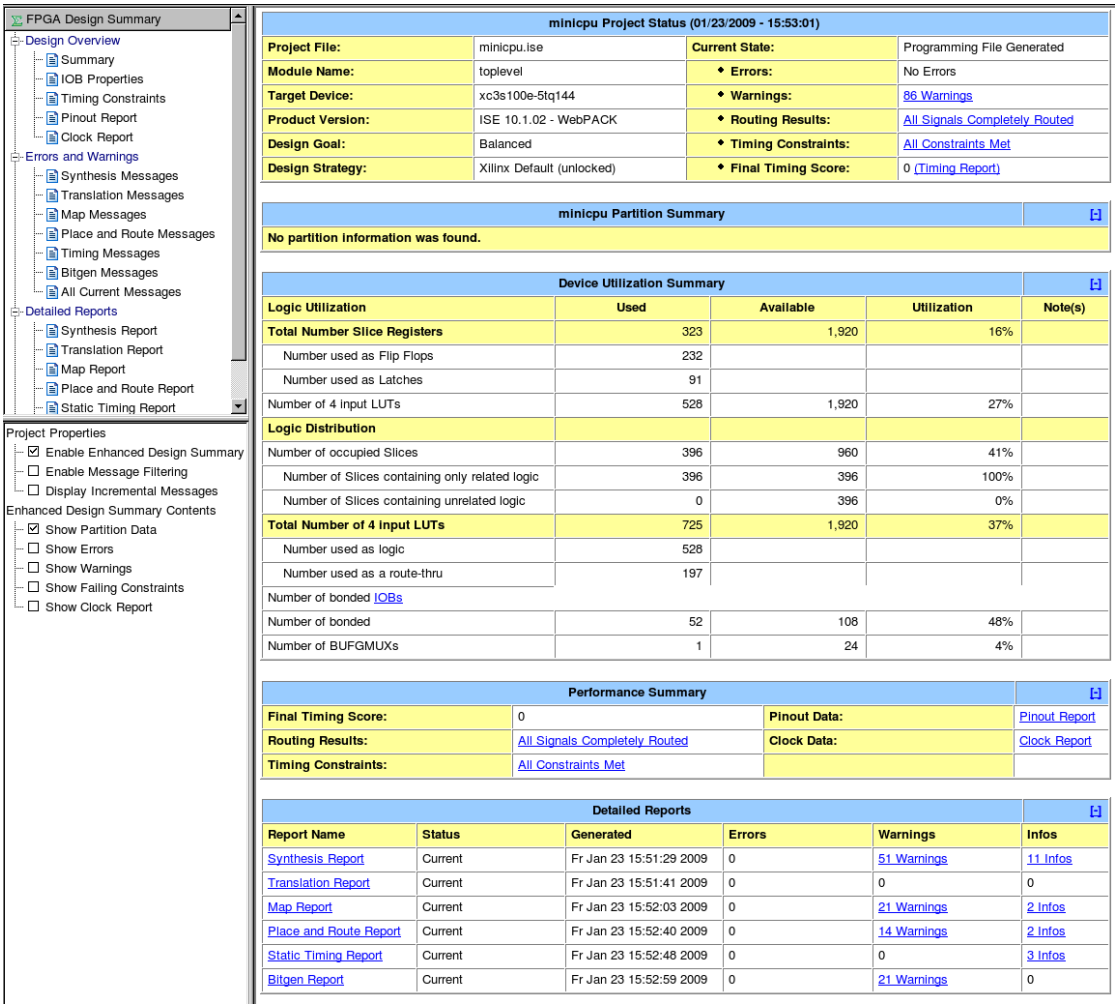
Benutzeroberfläche



Das Fenster von ISE ist in 4 Bereiche eingeteilt:

- Rechts oben befindet sich der Editor, mit dem man die Quelltexte bearbeiten kann. Er unterstützt auch Tabs.
- Unten ist das Ausgabefenster von Konsolen-, Warn- und Fehlermeldungen. Dieses ist auch in Tabs gegliedert.
- Das Sourcefenster links oben ist eine Art Dateibaum, wo alle Module und deren Untermodule aufgeführt sind. Dabei korrespondiert die Sicht nicht unbedingt mit der tatsächlichen Dateihierarchie, sondern es wird angezeigt, welche Module voneinander abhängen.
- Links unten ist das Prozessfenster. Es beinhaltet eine Liste der verschiedenen Aktionen, die bei dem Modul durchführbar sind, das im Bereich darüber ausgewählt wurde.

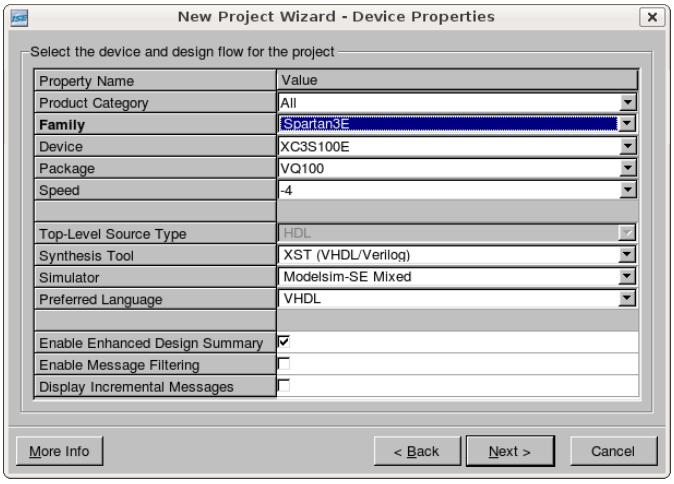
Über das Prozessfenster kann man sich das Design Summary anzeigen lassen, welches auch beim Öffnen eines Projekts angezeigt wird. Dieses zeigt eine Statistik über die verwendeten LUTs[1], Slices[1], FlipFlops (nach der ersten Synthese) sowie weitere Daten an.



Neues Projekt erstellen

Um ein neues Projekt zu erstellen, klickt man unter „File“ auf „New Project...“. Daraufhin wird man gefragt, wie das Projekt heißen soll und wo es gespeichert werden soll.
Im nächsten Schritt werden Angaben zur Hardware und zur verwendeten Sprache (in unserem Fall VHDL) gemacht. Für unsere Hardware sind die Angaben wie im Screenshot zu sehen vorzunehmen:

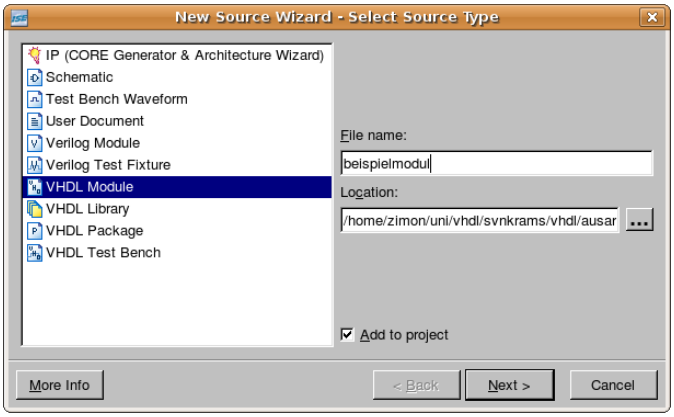
[1]Siehe auch die entsprechenden Erklärungen im Glossar auf Seite 79



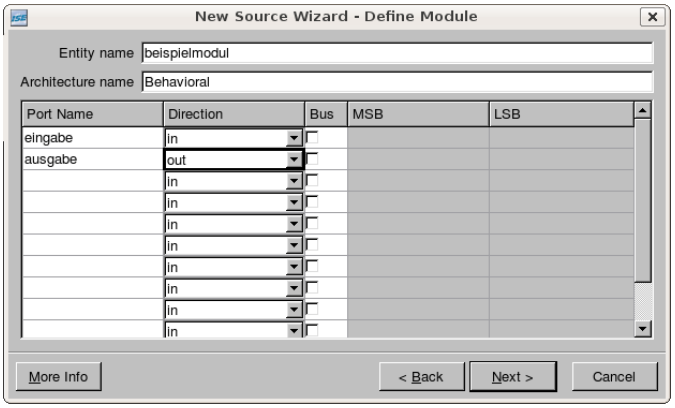
In den nächsten beiden Schritten können Quelldateien erstellt bzw. existierende Quelldateien hinzugefügt werden. Dies kann jedoch auch zu einem späteren Zeitpunkt geschehen. Zum Schluss wird noch eine Zusammenfassung aller eingegebenen Angaben angezeigt. Ein Klick auf „Finish“ erstellt dann das Projekt und lädt es.

Module und Pakete hinzufügen

Um ein Modul oder ein Paket zu erstellen klickt man im Prozessfenster auf „Create new Source“ und gibt im Fenster, welches sich daraufhin öffnet, den Dateinamen sowie die Art der neuen Datei an. Für ein Modul wählt man hier „VHDL Module“ (wie im Screenshot zu sehen), für ein Paket „VHDL Package“. Der Haken bei „Add to project“ muss dabei gesetzt sein.



Im nächsten Schritt kann man die Schnittstelle, also die Ein- und Ausgabesignale definieren. Diese kann man jedoch auch später direkt im VHDL Code definieren bzw. ergänzen.



Schließlich wird noch eine Zusammenfassung angezeigt. Mit „Finish“ wird die Datei mit dem Grundgerüst des VHDL-Moduls erstellt.

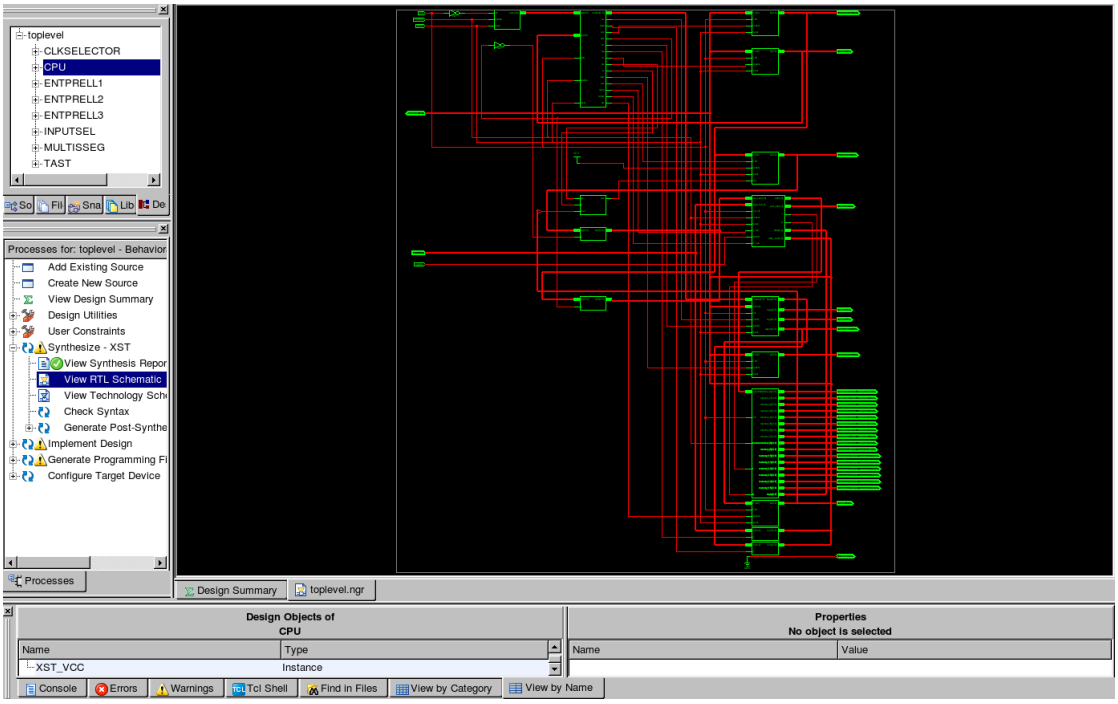
Man kann mehrere Module erstellen und diese dann abwechselnd zu Topmodulen[1] deklarieren (rechte Maustaste auf das Modul und dann „Set as Topmodule“ auswählen). Dies ist sehr praktisch um einzelne Module zu testen.

Um eine existierende Quelldatei dem Projekt hinzuzufügen klickt man im Prozessfenster auf „Add Existing Source“ und wählt die Datei aus, die hinzugefügt werden soll.

Synthetisieren, Implement Design, Programmdatei erstellen

Zur Synthese (und allen weiteren Schritten) wird das Topmodul im Sourcefenster ausgewählt. Im Prozessfenster kann man nun „Synthesize“, „Implement Design“ oder „Generate Programming File“ auswählen. Ein Doppelklick auf den jeweiligen Eintrag genügt. Wählt man einen Schritt aus ohne den Vorherigen ausgeführt zu haben, so wird dieser automatisch vorher erledigt.
Bei der Synthese wird zuerst ein Syntaxcheck durchgeführt. Danach werden die logischen Verknüpfungen interpretiert und eine Netzliste aus Gattern bzw. Logikelementen erstellt.
Nach der Synthese kann man sich über den Punkt „View RTL Schematic“ (dies ist ein Unterpunkt von Synthesize) das Logische Schema anzeigen lassen.

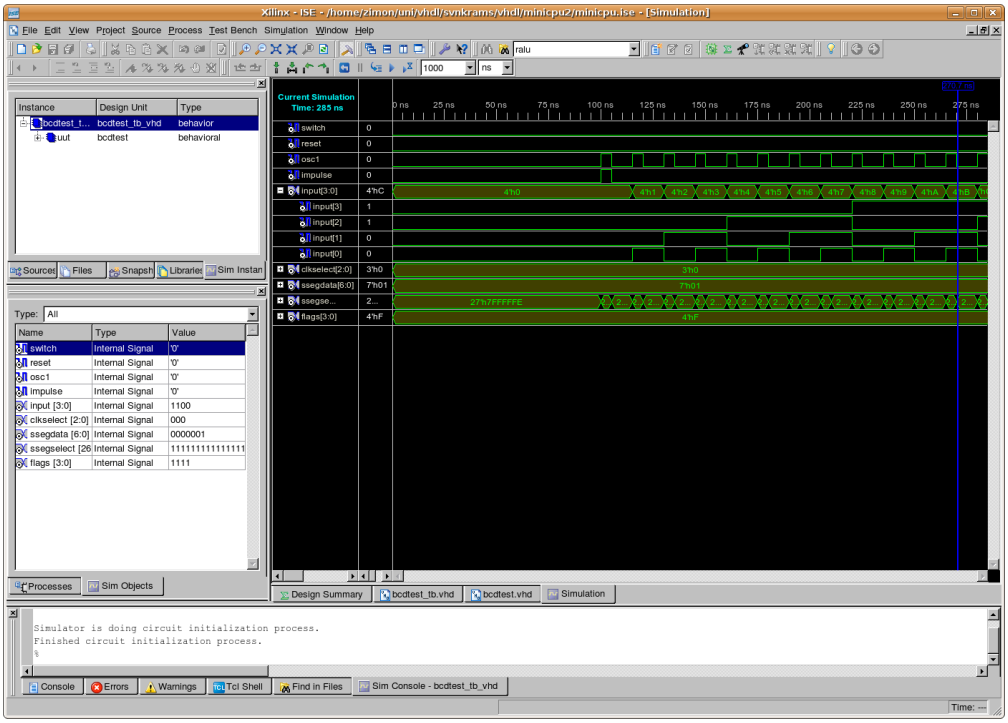
[1] Siehe auch die Erklärung im Glossar auf Seite 79



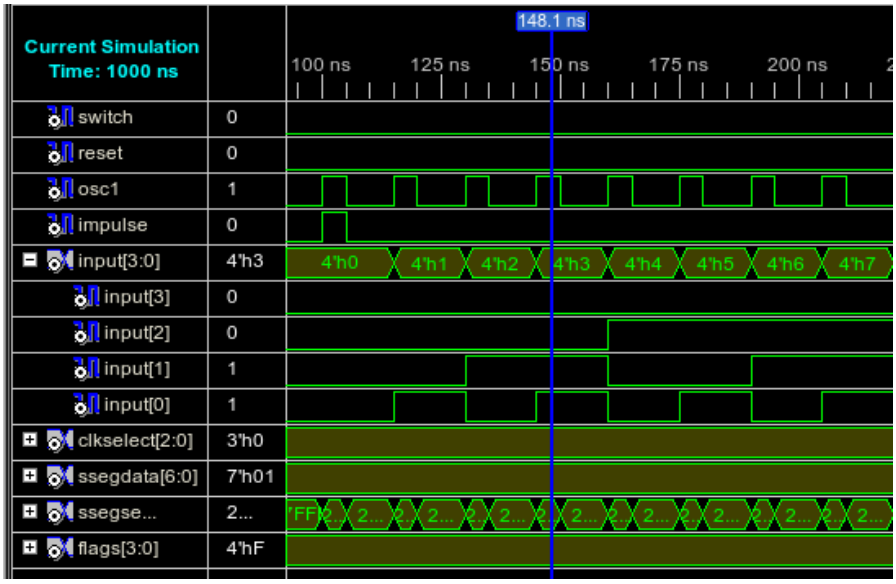
Bei „Implement Design“ werden die Logikgatter günstig angeordnet und verschaltet. Hier wird auch die UCF-Datei berücksichtigt, die definiert welche Signale auf welchen Pins herausgeführt werden sollen. Hier können noch benutzerdefinierte Bedingungen zum Timing festgelegt werden. „Generate Program File“ erstellt schließlich die Binärdatei.

Simulation

Um das Projekt zu simulieren, reicht es, die Synthese einmal ausgeführt zu haben. Man erstellt eine Testbench indem man eine neue Datei zum Projekt hinzufügt und als Typ „VHDL Test Bench“ wählt. Es wird das Grundgerüst der Testbench anhand des Topmoduls erstellt. Hinter den Kommentar „`-- Place stimulus here`“ kann man nun die Eingaben belegen. Die Erstellung einer Testbench ist im Kapitel 3.10 auf Seite 24 genauer beschrieben. Nun sollte man spätestens im Sourcefenster ganz oben „Sources for“ von „Implementation“ auf „Behavioral Simulation“ umstellen. Wählt man im Sourcefenster die neue Datei aus, so hat man nun die Möglichkeit im Prozessfenster unter „Xilinx ISE Simulator“ den Punkt „Simulate Behavioral Model“ auszuwählen. Ein Doppelklick darauf startet die Simulation.
Es öffnet sich ein neues Tab und man befindet sich im Simulationsmodus:



Nun werden auf der X-Achse die Zeit und auf der Y-Achse die Signalpegel aufgetragen. Jedes (Input- und Output-) Signal ist in einer Zeile eingezeichnet.



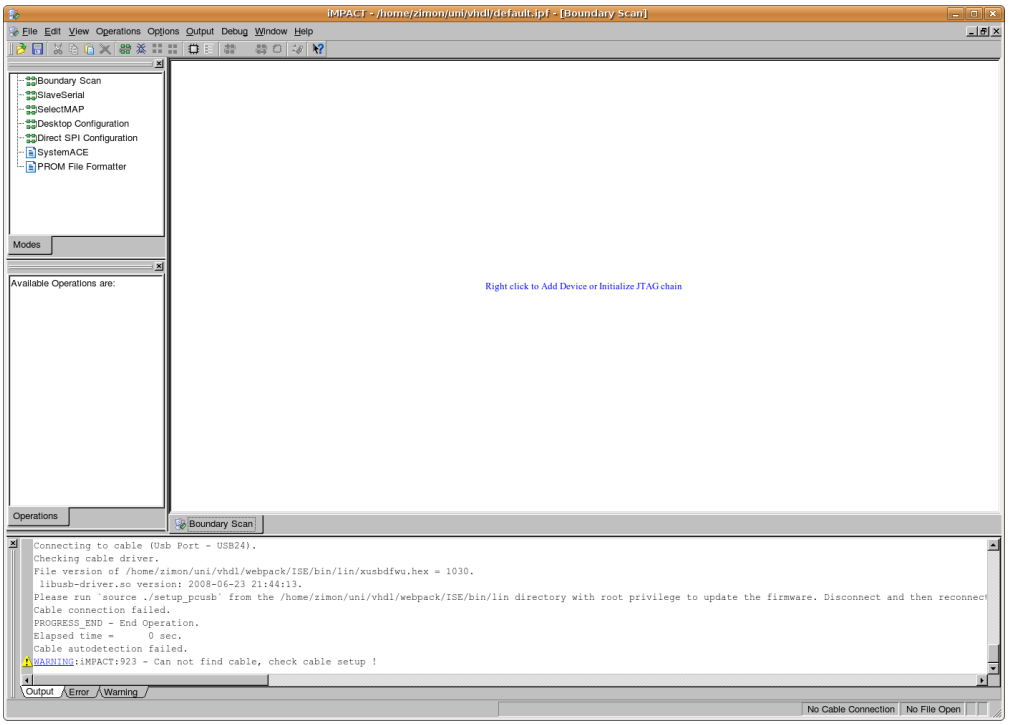
Den blauen Zeitstreifen kann man beliebig verschieben. Er zeigt die verstrichene Zeit an diesem Punkt in Nanosekunden an. Gleichzeitig wird in der 2. Spalte der Wert jedes Signals angezeigt.

Diese Anzeige bezieht sich auf den Zeitpunkt, den der Zeitstreifen markiert.

B.2 iMPACT

iMPACT ist ein Programm um die generierte Programmierdatei auf den FPGA zu flashen. Es gehört zu ISE und wird bei der Installation automatisch mit installiert.

Benutzeroberfläche

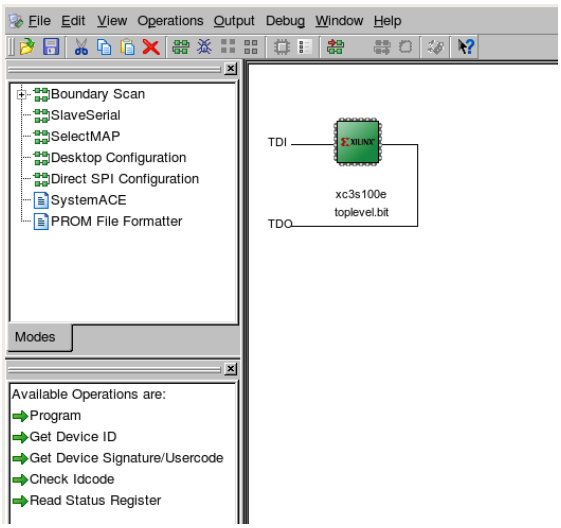


Die GUI von iMPACT ist ähnlich aufgebaut wie die von ISE und besitzt ebenfalls vier Fenster:

- Rechts oben befindet sich das Hauptfenster, in dem angeschlossene Geräte angezeigt werden.
- Unten ist das Ausgabefenster von Konsolen, Warn- und Fehlermeldungen. Dieses ist wie bei ISE in Tabs gegliedert.
- Statt dem Sourcefenster in ISE gibt es eine Auswahl, was für ein Device auf welche Weise programmiert werden soll (JTAG, SPI, ...)
- Links unten ist das Prozessfenster. Es beinhaltet eine Liste der verschiedenen Aktionen, die durchgeführt werden können.

Grundlegende Arbeitsschritte

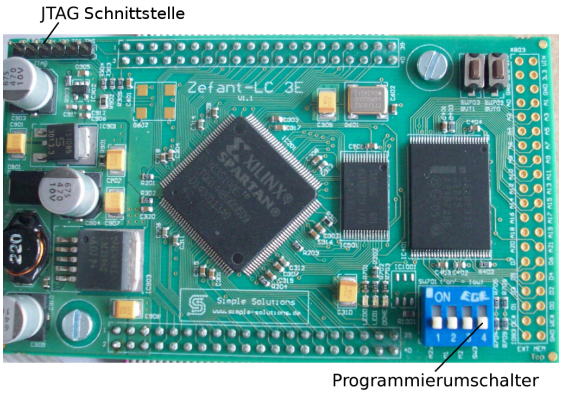
Am einfachsten ist es, erst dann iMPACT zu starten, wenn das JTAG-Kabel angeschlossen und der FPGA mit Strom versorgt ist. Dann erscheint direkt ein grünes Symbol, welches den FPGA darstellt. Ist das Symbol nicht zu sehen, kann man einen Boundary Scan durchführen.
Dafür klickt man im linken oberen Fenster doppelt auf „Boundary Scan“. Nun kann man mit der rechten Maustaste im Hauptfenster das Kontextmenü aufrufen und dort „Initialize Chain“ auswählen. Dadurch sollte der angeschlossene FPGA erkannt werden.

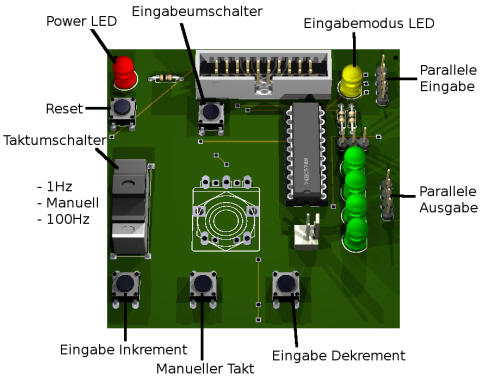


Mittels „Assign New Configuration File...“ im Kontextmenü (rechte Maustaste auf das grüne FPGA Symbol) kann man nun die .bit Datei auswählen, die zuvor über ISE über den Punkt „Generate Programming File“ erstellt wurde.
Programmieren kann man den FPGA mit der gewählten Datei nun über den Menüpunkt „Program“ des Kontextmenüs. Nun erscheint ein Dialog in dem man weitere Einstellungen vornehmen kann. Die Verify-Funktion hat bei uns leider nicht funktioniert, weswegen wie diese immer abgewählt ließen (siehe Kapitel 8.1 auf Seite 60). Ein abschließendes OK startet nun den Programmiervorgang.
Wurde die .bit Datei geändert (z.B. weil man noch Änderungen im VHDL Code vorgenommen und die Datei neu erstellt hat) entfällt ein erneutes „Assign New Configuration File...“. Jedoch wird man dann darauf hingewiesen, dass sich die Datei geändert hat.

C Bedienungsanleitung

Hier eine Schritt für Schritt Anleitung zur Bedienung der MiniCPU.





Vorbereitung

Als erstes muss die Stromversorgung hergestellt werden. Dazu werden die Bananenbuchsen an eine 5V Stromversorgung angeschlossen.
Als nächstes wird der FPGA geflasht. Dafür wird die Verbindung zum PC mittels des JTAG-Programmieradapters hergestellt. Dieser wird an den Parallelport des PC's und auf der anderen Seite an die JTAG Schnittstelle des FPGA-Boards angeschlossen.
Nun startet man iMPACT (wobei der Treiber vorher wie in Kapitel A beschrieben geladen werden muss). Beim Starten wird man gefragt, ob man eine iMPACT-Projektdatei laden möchte. Es ist nicht verkehrt, eine solche Datei beim ersten Start zu erstellen und später bei jedem Start zu laden, da dann die richtige .bit Datei automatisch geladen wird. iMPACT sollte das angeschlossene Board selbstständig erkennen. Ansonsten kann man den in Kapitel B.2 beschriebenen Boundary Scan durchführen. Vor dem Flashen empfiehlt es sich den Takt-Umschalter auf „Manuell“ (mittlere Position) zu stellen.
Zum Flashen klickt man nun mit der rechten Maustaste auf das grüne Symbol im rechten oberen Fenster und wählt „Assign New Configuration File...“. Nun wählt man z.B. die „wuerfel.bit“ (oder eine andere .bit Datei, die durch „Generate Programming File“ in ISE generiert wurde) aus.
Jetzt kann der FPGA geflasht werden indem man wieder mit der rechten Maustaste auf das grüne Symbol klickt und „Program“ auswählt. Im sich öffnenden Fenster brauchen keine weiteren Einstellungen vorgenommen werden (es ist lediglich darauf zu achten, dass die Verify Funktion deaktiviert ist, da diese nicht funktioniert). Ein Klick auf „OK“ genügt.

Programmausführung

In der Programmdatei „wuerfel.bit“ wird der RAM bereits mit dem Würfelprogramm initialisiert. Um es auszuführen kann man nun einfach auf den Taster für den manuellen Takt drücken. Ein vorheriger Reset ist jedoch empfehlenswert um sicher zu stellen, dass alle Register 0 sind. Jeder Tastendruck wird als ein Takt gewertet.
Mit dem Takt-Umschalter kann auch ein 1Hz oder 100Hz Takt eingestellt werden.

Ein- und Ausgaben

Ausgaben werden automatisch auf die entsprechende Siebensegmentanzeige ausgegeben. Parallel dazu werden alle Ausgaben auch auf den 4 Ausgabepins ausgegeben.
Die Eingabe erfolgt durch die Eingabe-Inkrement und -Dekrement Taster. Mittels des Eingabe-Umschalters kann man die Eingabe zwischen parallel (mittels der 4 Input-Pins) oder manuell umschalten. Bei paralleler Eingabe leuchtet die Eingabe-LED.

Programmieren des RAM

Um ein anderes Programm in den RAM zu schreiben muss der Programmier-Umschalter umgeschaltet werden (DIP-Schalter 4 am FPGA-Board - ganz rechts). Nun befindet man sich im Modus „Auswahl“ und kann mit den Tasten Up und Down die Speicherstelle wählen, die geändert werden soll. Die jeweils aktuell gewählte Stelle blinkt dabei. Nach dem Umschalten des Programmier-Umschalters ist immer die erste Stelle ausgewählt. Mit dem Set bzw. Clock Taster kann man nun in den „Bearbeiten“ Modus wechseln, woraufhin alle anderen als die gewählte Speicherstelle gedimmt werden. Mit den Tasten Up und Down kann man nun den Inhalt der Speicherstelle ändern. Mit einem erneuten Druck auf den Set/Clock Taster wird der Wert gespeichert und man gelangt in den „Auswahl“ Modus zurück. Wenn alle Stellen des RAM den gewünschten Wert besitzen kann man den Programmier-Umschalter wieder zurück stellen. Nun drückt man noch einmal Reset. Dann kann das neue Programm ausgeführt werden.

D Glossar

- Antifuse: FPGA-Architektur, bei der beim Programmieren die Verbindungen durch das „durchschmelzen“ von Fuses aus amorphem Silizium hergestellt werden. Der FPGA ist dadurch nur einmal programmierbar, hält aber die Programmierung auch nach dem Ausschalten. [3]
- Anzeigeplatine: Eine Platine mit Latch und Siebensegmentanzeige zur Darstellung des Inhalts von Registern und Bussen.
- Architecture: Ein VHDL-Schlüsselwort, welches die Beschreibung der Logik eines Moduls einleitet.
- Auswahl-Modus: Ein Modus des Programmiermodus, welcher der Auswahl einer RAM-Stelle dient.
- Bearbeiten-Modus: Ein Modus des Programmiermodus, der das Bearbeiten einer ausgewählten RAM-Stelle ermöglicht.
- bedingte Zuweisung: Eine VHDL-Anweisung, die anhand eines Signals entscheidet, was einem anderen Signal zugeordnet werden soll (wie ein Multiplexer).
- BlinkDimmer: Modul um Signale für Siebensegmentanzeigen blinken zu lassen und dimmen zu können.
- Case-when: Eine VHDL-Anweisung in Prozessen, die dem switch-case Konstrukt aus C entspricht.
- CISC: Complex Instruction Set Computer - ein Befehlssatz mit komplexen Befehlen, die den Prozesor längere Zeit auslasten als bei modernen RISC (Reduced Instruction Set Computer) Befehlssätzen.
- CLB: Complex Logic Block - Bestandteil des FPGA, der benutzt wird um Logikschaltkreise zu implementieren.
- Component: VHDL-Schlüsselwort, das die Signatur eines Moduls (inklusive Port-Deklaration) einleitet um dieses als Komponente exportieren zu können.
- Datenleitungen: 7 Leitungen, die parallel mit Latches auf allen Siebensegmentanzeigen sowie den LEDs der LED-Platine und der Ein- Ausgabeplatine verbunden sind.
- DCM: Digital Clock Manager - Bestandteil des FPGA, der Taktsignale erzeugen und in der Phase verschieben kann.
- (E)EPROM: (Electrically) Erasable Programmable Read Only Memory - Wiederbeschreibbarer Speicher, wird auch in manchen FPGAs verwendet.
- Ein- Ausgabeplatine: Eine Platine mit Tastern, LEDs und Pins für sämtliche Ein- und Ausgaben.
- Eingabe-Umschalter: Ein Taster auf der Ein- Ausgabeplatine um zwischen serieller (manueller) und paralleler Eingabe zu wechseln.
- Entity: VHDL-Schlüsselwort, das die Signatur eines Moduls (inklusive Port-Deklaration) einleitet.
- Entpreller: Ein Modul, das vermeidet, einen Tastendruck aufgrund des hohen Taktes mehrfach zu zählen und auch das Prellen von Tastern ignoriert.
- FPGA: Field Programmable Gate Array - Der Baustein, auf dem wir die MiniCPU implementiert haben.

- Flags: 4 Bits, die Informationen über die letzte Operation der ALU bzw. dessen Ergebnis beinhalten.
- Floorplan: Der Floorplan eines FPGA bestimmt die Verteilung der programmierten Logik auf die Logikressourcen (CLBs, Multiplizierer, IOBs, etc.) des FPGA.
- Hauptplatine: Eine Platine, die direkt mit dem FPGA Board verbunden ist und die Signale zu allen anderen Platinen weiter leitet.
- iMPACT: Xilinx Software zum Flashen von FPGAs (ist ein Teil von ISE).
- IOB: Input Output Block - Bestandteil des FPGA, der einen Pin des FPGA zu der programmierbaren Logik verbindet.
- ISE: Xilinx Entwicklungsumgebung zum entwickeln von VHDL Code.
- JTAG: Ein Standard zum Testen, Debuggen und Programmieren von elektronischer Hardware.
- Komponente: ein Modul, welches mit einer Component-Deklaration zum Export vorbereitet wurde.
- LED-Platine: Platine mit Latch und 4 LEDs zur Anzeige der Flags.
- LUT: Lookup Table - SRAM-Zelle, die in SRAM-basierten FPGAs zur Erzeugung von Logikfunktionen genutzt wird.
- MiniComputer: Modul mit MiniCPU, RAM(Wrapper) sowie Ein- und Ausgabe.
- MiniCPU: Modul der MiniCPU, wie sie in der Vorlesung „Digitaltechnik“ vorgestellt wurde.
- Modul: Eine Art Klasse in VHDL, die beliebig oft instanziiert werden kann. Beispiele: Register, Counter
- Multiplex7seg: Modul, welches für die Ansteuerung der Siebensegmentanzeigen zuständig ist.
- Netzliste: Eine Liste von Logikgattern, die während der Synthese aus dem VHDL Code erstellt wird.
- Normalbetrieb: Modus in dem der MiniComputer läuft.
- Package: Eine Sammlung von Komponenten.
- Port: VHDL-Schlüsselwort, welches die Definition aller Ein- und Ausgänge einer Entity oder Component einleitet.
- Process: VHDL-Anweisung für einen Prozess, mit dem Anweisungen sequentiell statt parallel ausgeführt werden. Hier können auch Variablen, Schleifen, if- und case Anweisungen genutzt werden.
- Programmiermodus: Modus in dem das RAM Programmiert werden kann. Wird in Bearbeiten- und Auswahl-Modus unterteilt.
- Programmier-Umschalter: Ein DIP-Schalter auf dem FPGA Board um zwischen dem Normalbetrieb und dem Programmiermodus umzuschalten.
- RAM-Wrapper: Modul, welches das RAM enthält und Möglichkeiten zu dessen Programmierung bietet.
- Schaltmatrix: Bestandteil eines FPGA, der andere Bestandteile mit der Routing Infrastruktur verbindet.

- Selectleitungen: Je eine Leitung, die zum Latch einer jeden Siebensegmentanzeige (sowie LEDs der LED- und Ein- Ausgabeplatine) führt um den aktuellen Wert auf der Datenleitung zu übernehmen.
- Sensitivity List: Liste von Signalen, deren Änderung den zugehörigen Prozess triggern (auslösen).
- Signal: Eine Leitung eines Moduls. Alle Ein- und Ausgaben sind Signale.
- Slice: Bestandteil eines CLBs in einem FPGA.
- SRAM: Static Random Access Memory: Statischer RAM, wird in vielen FPGAs verwendet.
- Synthese: Der Prozess, bei dem aus HDL-Code Netzlisten erzeugt werden.
- Takt-Umschalter: Ein Switch, mit dem man zwischen den Taktquellen „manueller Takt“, 1Hz und 100Hz umschalten kann.
- Taktnetz: Infrastruktur innerhalb eines FPGAs bestehend aus Leitungen zur Übertragung von Taktsignalen.
- Testbench: Ein Prozess, der das Verhalten während einer Simulation beschreibt.
- Toplevel: Topmodul der gesammten Implementation, welches für die Verbindung und Verwaltung der Hardware zuständig ist.
- Topmodul: Oberstes Modul in VHDL, welches die gesammte Funktionalität beinhaltet und dessen Eingangs- und Ausgangssignale direkt mit der Hardware verschaltet sind.
- Type: VHDL-Anweisung zur Erstellung eigener Datentypen.
- UCF Datei: Datei in der die Zuordnung von Pins des FPGA zu Signalen im VHDL-Code vorgenommen wird. Dort werden auch Signalpegel und Pullups definiert.
- Variable: VHDL-Konstrukt, in dem ein Wert gespeichert werden kann. Sie kann auch mehrmals zugewiesen werden, kann aber nur innerhalb eines Prozess deklariert werden.
- Vektor: Ein mehrbittiges Signal bzw. eine mehrbittige Variable.
- VHDL: Die von uns genutzte Hardwarebeschreibungssprache.

Literatur

[1] Altium Limited: *Accolade VHDL Reference Guide*, 2004. Seiten 82–84. Erreichbar unter http://ece.wpi.edu/~wrm/Courses/EE3810/geninfo/Welcome%20to%20the%20VHDL%20Language.pdf.

[2] FPGA4FUN: *FPGA - Unterschiede zu CPLDs*. Website, 2009. Erreichbar unter http://www.fpga4fun.com/FPGAinfo1.html; .

[3] Hertwig, A. und R. Brück: *Entwurf digitaler Systeme*, Kapitel 5.3 Field Programmable Gate Arrays, Seiten 113–121. Hanser Verlag, 2000.

[4] Microkontroller.net: *FPGA*. Website, 2009. Erreichbar unter http://www.mikrocontroller.net/wikisoftware/index.php?title=FPGA&oldid=34616; [Online; Stand 31. März 2009] .

[5] Reichardt, J. und B. Schwarz: *VHDL-Synthese*. Oldenbourg, 2007.

[6] Wikipedia: *Field Programmable Gate Array – Unterschiede zu CPLD— Wikipedia, Die freie Enzyklopädie*, 2009. http://de.wikipedia.org/w/index.php?title=Field_Programmable_Gate_Array&oldid=58327479; [Online; Stand 31. März 2009].

[7] Xilinx: *Datasheet*. Seite 27. Erreichbar unter http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf.