FPGA / INCA

Frank Ullmann

email : full@cadlab.de

15. November 1994

Seminar
der
Projektgruppe Systementwurf
WS 94/95 und SS 95

Inhaltsverzeichnis

1 Einleitung

Diese Seminararbeit befaßt sich mit dem Entwurf digitaler Schaltungen. Es werden zunächst einige Grundlagen erklärt, die z.B. für eine Schaltungsoptimierung benötigt werden. Hingewiesen sei in diesem Zusammenhang auf die *KV-Diagramme*, mit deren Hilfe man relativ einfach eine boolsche Funktion mit bis zu vier Variablen verkürzen kann.

Das darauffolgende Kapitel beschreibt dann die Unterschiede der anwenderprogrammierbaren Schaltungen, angefangen bei Speicherbausteinen, wie z.B. das ROM oder das EPROM, über PLAs (Programmierbare logische Felder), bis hin zu den Xilinx-Bausteinen. Da die zu benutzende Hardware in diesem Seminar auf diesen Xilinx-Bausteinen basiert, befaßt sich das ganze nächste Kapitel noch etwas intensiver mit dieser Art von anwenderprogrammierbaren Schaltungen.

Das Kapitel „ Virtual ASIC " setzt sich dann schließlich mit dem eigentlichen Entwurf einer digitalen Schaltung auseinander. Nachdem im vorherigen Kapitel die Hardware genauer erklärt wurde, wird hier nun auf die Software und deren Bedienung eingegangen. Da die genaue Beschreibung der Software den Rahmen dieser Seminararbeit sprengen würde, ist das Kapitel nur als eine Art Zusammenfassung aufzufassen. Für ein genaues Verstehen der Software ist es unabdingbar, sich das Manual anzueignen.

2 Entwurf digitaler Schaltungen

2.1 Theoretische Grundlagen

2.1.1 Grundlagen aus der Graphentheorie

Viele Probleme beim Entwurf digitaler Schaltungen lassen sich als graphentheoretische Probleme formulieren. Im allgemeinen bildet man hierbei die in einer Netzliste enthaltenen Gatter auf die Knoten eines Digraphen (*gerichteter Graph)* ab, während man die Verbindungen zwischen diesen einzelnen Gattern als Kante zwischen Knoten auffaßt.

Ein Graph G' = (V', E') heißt *Teilgraph* von G = (V, E), wenn V' $\subseteq$ V und E' $\subseteq$ E gilt. Gehört jede Kante von E auch zu E', so heißt G' *Untergraph* von G. Ein Graph heißt *vollständig*, wenn je zwei verschiedene Knoten von G durch eine Kante verbunden sind. Ein vollständiger Teilgraph eines Graphen ist gleichzeitig Untergraph und wird auch *Clique* genannt. Die Anzahl der mit einem Knoten v_i verbundenen Knoten wird *Grad* des Knotens genannt. *Zusammenhängend* ist ein Graph genau dann, wenn alle Paare von Knoten miteinander verbunden sind.

2.1.2 Grundlagen aus der Algebra

Mit Hilfe einiger Grundbegriffe aus der Algebra lassen sich viele Probleme beim Schaltungsentwurf auf eine kleinere Anzahl von Grundproblemen reduzieren. Die wichtigsten Grundbegriffe werden im folgenden kurz zusammengefaßt :

- Relation : Eine *Relation* ρ ist eine Teilmenge des kartesischen Produktes $M \times M, \rho \subseteq M \times M = \{(a,b) \mid a \in M, b \in M\}$. Zwei Elemente a und b stehen dann in Relation, auch a ρ b geschrieben, wenn das Paar (a, b) in der Menge ρ enthalten ist. Eigenschaften von Relationen :

Reflexivität :	$\forall a \in M :$	$a\rho a$
Symmetrie :	$\forall a, b \in M :$	$a\rho b \Rightarrow b\rho a$
Antisymmetrie :	$\forall a, b \in M :$	$a\rho b \wedge b\rho a \Rightarrow a = b$
Transitivität :	$\forall a, b, c \in M :$	$a\rho b \wedge b\rho c \Rightarrow a\rho c$

- Ordnungsrelation : Eine *Ordnungsrelation* ist reflexiv, antisymmetrisch und transitiv. Sie dient dazu, zwischen gewissen Elementen einer Menge M eine Reihenfolge festzulegen.

- Äquivalenzrelation : Eine *Äquivalenzrelation* ist reflexiv, symmetrisch und transitiv. Sie ermöglicht das Zerlegen einer Menge in ihre Äquivalenzklassen.

- Verträglichkeitsrelation : Diese Relation ist reflexiv und symmetrisch. Sie teilt ähnlich wie die Äquivalenzrelation eine Menge in Klassen auf. Alle Paare von Elementen aus einer Klasse stehen dann in Relation, der Unterschied zur Äquivalenzrelation besteht darin, daß nicht unbedingt jedes Element genau einer Klasse zugeordnet wird (da keine Transitivitäteigenschaft).

2.1.3 Grundlagen aus der Logik

Basis der Realisierung digitaler Schaltungen ist eine boolsche Algebra $(B, \vee, \wedge)$, wobei stets $B = \{0, 1\}$ verwendet wird. Eine *boolsche Funktion* f bildet n binäre Variablen auf einen Wert 0 oder 1 ab :

$$f : \{0,1\} \rightarrow \{0,1,x\}, x \rightarrow f(x)$$

Das x in der Menge B führt man als sogenannten *don't care*–Wert ein, der symbolisiert, daß eine Variable einen beliebigen Wert annehmen kann. Bei der Spezifikation digitaler Schaltungen ist es häufig notwendig, eine Menge boolscher Funktionen als Implementierung zuzulassen. Mit Hilfe von *KV-Diagrammen (vgl. INFO-B)* ist es dann möglich, diese Funktionen zu optimieren, um den Schaltungsaufwand bei der Implementation zu minimieren. Es entstehen sogenannte *Bündelfunktionen*, die von mehreren Einzelfunktionen gemeinsam benutzt werden. Existiert z.B. der Term $x_1x_2x_3$ in mehreren Funktionen, so genügt es, wenn man die Schaltung, die diesem Term entspricht, lediglich einmal implementiert und alle betreffenden Funktionen auf diese Schaltung zurückgreifen können.

Um solche boolschen Funktionen mit Hilfe von Rechnern zu verarbeiten, kann man eine Funktion als *binären Entscheidungsbaum* darstellen, da sich KV-Diagramme nur schwer auf Computern realisieren lassen. Die Grundidee besteht darin, daß man die Funktion in alle n Eingangsvariablen aufspaltet und so einen binären Baum darstellt. Als Wurzel beginnt man mit x_1 (wobei $\overline{x}_1$ linker Teilbaum und x_1 rechter Teilbaum wird) und fährt entsprechend bis x_n fort. Als Blätter erhält man dann die 2^n Funktionswerte 0 oder 1. Reduziert wird dieser Baum dann dadurch, daß gemeinsame Teilbäume nur einmal dargestellt werden und die Aufspaltung nach einer Variablen unterbleibt, wenn die darzustellende Teilfunktion unabhängig von x_i ist.

2.2 Optimierungsziele

Zur Realisierung boolscher Funktionen ist häufig eine Funktionstabelle als Beschreibung gegeben. Man kann aus dieser Tabelle dann sehr schnell eine Strukturbeschreibung gewinnen, indem man alle zum Ausgabewert 1 zugehörigen Zeilen der Funktionstabelle als Produktterme interpretiert. Im allgemeinen ist diese Art der Implementierung (auch *krause Logik* genannt) jedoch nicht optimiert. Beim *manuellen Logikentwurf* werden die Optimierungen vom Benutzer mit Hilfe von boolscher Algebra selbst vorgenommen. Bei der *automatischen Logiksynthese* geschieht dies rechnergesteuert.

Optimert werden kann nach folgenden Gesichtspunkten :

- minimaler Realisierungsaufwand
- maximale Geschwindigkeit
- minimaler Leistungsverbrauch und
- geringer Entwurfsaufwand

Das Problem bei der rechnergesteuerten Minimierung ist die NP-Vollständigkeit. Um eine exakte Minimierung durchzuführen, benötigt man zwei Schritte. Zunächst werden für das spezifizierte Schaltnetz alle Primimplikanten bestimmt, danach wird eine minimale Überdeckung berechnet, was zu den NP-vollständigen Problemen gehört. Um aber diesen Algorithmus auch bei größeren Schaltungen anzuwenden, muß man die beiden Schritte so effizient wie möglich implementieren. Ist nicht die exakte Minimierung gefragt, so besteht eine Möglichkeit der Implementierung darin, iterativ aus einer gegebenen Lösung solange neue Lösungen zu generieren, bis keine Reduktion mehr möglich ist. Andere Verfahren brechen ihre Berechnung nach einer gewissen Zeit ab und geben die bis dahin erzeugten Lösungen aus.

2.3 Anwenderprogrammierbare Schaltungen

2.3.1 Speicherbausteine

Abbildung 1 veranschaulicht die logische Grundstruktur üblicher Speicherbausteine. Der Speicherbaustein besteht aus $2^n \cdot m$ Bits. Die Auswahl der 2^n Worte erfolgt über eine n-Bit-Adresse, welche der Adressdecoder intern so umwandelt, daß man jedes einzelne m-Bit-Wort am Datenausgang lesen kann.

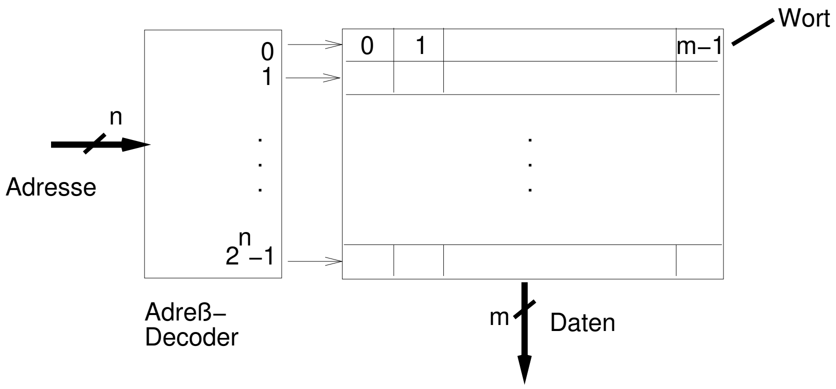

Abbildung 1: Grundstruktur von Speicherbausteinen

Ein Speicherbaustein besteht in der Regel nun aus n Pins für die Adresse, m Pins für die Daten *(meistens 8 Bit)* und 2 Pins für die Stromversorgung *(GND und Vcc)*. Weiterhin werden noch einige wenige Pins für die Steuerung des Bausteins benötigt. So zeigt die Abbildung 1 den Datenbus z.B. nur als unidirektional; zur Programmierung des Bausteins ist es jedoch notwendig, den Inhalt der Speicherzellen vorher festzulegen, was durch Aktivieren spezieller Signale und durch Anlegen der Daten an den Datenbus geschieht.

Es gibt nun verschiedene Typen von Speicherbausteinen :

- **ROMs** *(read-only memories)* sind Nur-Lese-Speicher. Sie werden vom Werk aus mit einer erhöhten Spannung und einer besonderen Betriebsart programmiert und können danach nicht wieder geändert werden.
- **PROMs** *(programmable ROMs)* können dadurch programmiert werden, daß in den einzelnen Speicherzellen selektiv Sicherungen durchgebrannt werden. Diese Programmierung kann im Gegensatz zum ROM auch vom Anwender durchgeführt werden.
- **EPROMs** *(erasable PROMs)* können innerhalb von wenigen Minuten mit ultraviolettem Licht gelöscht und danach wieder neu programmiert werden. Zu diesem Zweck befindet sich auf der Chip-Oberseite ein rundes Fenster, durch das das ultraviolette Licht direkt auf die eigentliche Chipfläche gelangen kann. Um ein unbeabsichtigtes Löschen des Speicherinhaltes zu vermeiden, verklebt man dieses Sichtfenster meistens mit einem kleinen Stück Klebeband.
- **EEPROMs** *(electrically erasable PROMs)* können elektrisch gelöscht und neu programmiert werden.
- **RAMs** *(random-access memories)* können auch während des Normalbetriebes verändert werden. Ihr Inhalt geht jedoch bei Abschalten der Versorgungsspannung verloren. Man unterscheidet zwischen statischen *(SRAMs)* und dynamischen *(DRAMs)* Speichern. Bei den dynamischen Speichern ist man gezwungen, den Inhalt der Speicherzellen in bestimmten Abständen wieder aufzufrischen. Man liest also den Inhalt jeder

einzelnen Speicherzelle aus und programmiert ihn sofort wieder neu ein. Bei den heutigen DRAMs geschieht dies allerdings schon halbautomatisch, sodaß z.B. in Computern die CPU damit nicht belastet wird. Im Gegensatz dazu bleibt bei SRAMs der Inhalt der Speicher nach einmaliger Programmierung erhalten. Ihr Nachteil liegt darin, daß die pysikalische Speicherdichte nicht so groß wie bei den DRAMs ist.

2.3.2 PLAs - Programmierbare logische Felder

Programmierbare logische Felder (PLAs, *programmable logical arrays*) gleichen im Prinzip den Speicherbausteinen. Abbildung 2 zeigt die Grundstruktur eines logischen Feldes ohne Programmierung.

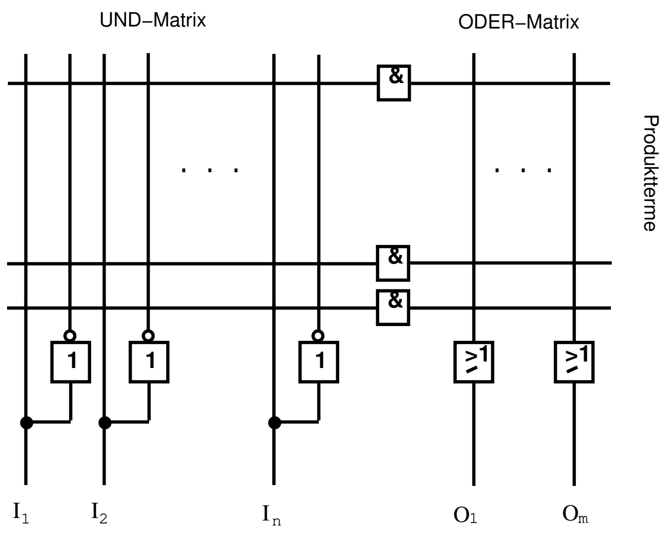

Abbildung 2: Grundstruktur von PLAs

Aus den Eingangsvariablen I_1 bis I_n wird zunächst jeweils das invertierte Signal gebildet. Damit können dann einzelne Signale miteinander zu Produkttermen konjunktiv verknüpft werden. Aus diesen Produkttermen werden dann durch disjunktive Verknüpfung die Ausgangssignale O_1 bis O_m erzeugt *(vgl. auch INFO-B-Skript, Kapitel „PLA - Programmierbare Logische Arrays“)*. Bei Speicherbausteinen werden im Vergleich dazu alle 2^n möglichen Produktterme erzeugt. Man erreicht dies, indem man die UND-Matrix binär von 0 bis 2^{n-1} einmalig programmiert, während die ODER-Matrix den eigentlichen Inhalt des Speicherbausteines repräsentiert. Die Eingänge entsprechen dann den Adressleitungen, die Ausgänge den Datenleitungen.

Bei anwenderprogrammierbaren PLAs *(FPLAs - field programmable logic arrays)* ist zu beachten, daß die Anzahl der verfügbaren Eingabe- und Ausgabesignale sowie der Produktterme fest vorgegeben ist. Elektrisch löschbare FPLAs werden auch unter dem Namen GAL *(generic array logic)* angeboten. Eine dritte Schaltungsart, das PAL *(programmable array logic)*, ermöglicht nur eine Programmierung der UND-Matrix, während die ODER-Matrix fest vorgegeben ist. So werden z.B. jeweils drei Produktterme fest durch ein ODER-Gatter mit einem Ausgang verbunden. Der Vorteil liegt darin, daß die Verzögerungszeiten kürzer als beim FPLA sind, da beim PAL nur ein programmierbares Feld durchlaufen werden muß.

2.3.3 PLDs

PLDs *(programmable logical devices)* kann man zwischen den PLAs und den FPGAs einordnen. Im Prinzip gleichen sie funktionell gesehen mehr den FPGAs, besitzen jedoch nicht deren Flexibilität. Von den PLAs unterscheiden sie sich physikalisch lediglich darin, daß die Anschlüsse nicht bei der Fertigung, sondern durch Programmierung geschaffen oder beseitigt werden. Weiterhin gibt es auch hier – ebenso wie beim EPROM – die Möglichkeit, den Inhalt der PLDs wieder zu löschen *(EPLD - erasable PLD)*

Kommerziell verfügbare PLDs besitzen jedoch eine größere Flexibilität als die beschriebenen Grundformen. Werden z.B. nicht nur die Eingänge der UND-Matrix bzw. die Ausgänge der ODER-Matrix als externe Anschlüsse benutzt, sondern auch Anschlüsse wie z.B. „zum UND-Feld", welche mit einem nicht zu einem Eingangspin verbundenen, vertikal verlaufenden Anschluß des UND-Feldes geführt werden, so sind weitere Funktionen verfügbar :

- Realisieren der negierten Funktion in der UND-/ODER-Struktur, d.h. Benutzung der konjunktiven statt der disjunktiven Form
- Zwischenspeichern der Ausgangssignale
- Deaktivierung der Ausgangssignale

2.3.4 PGAs - Programmierbare Logikbausteine

PGAs *(programmable gate arrays)* verbinden die Vorteile von PLDs (kurze Entwurfszeiten, anwenderprogrammierbar, Standardprodukt) mit den Vorteilen von Gate-Arrays (flexible Architektur, hohe Integrationsdichte). PGAs enthalten im Unterschied zu PLDs eine größere Anzahl von Grundelementen, die für sich betrachtet eine geringere Funktionalitat aufweist. Sie erlauben es, diese Grundelemente mehrstufig über eine komplexe Verbindungsstruktur zusammenzuschalten.

2.3.5 FPGAs - Field Programmable Gate Arrays

Die Entwicklung von CMOS-Technologien machte es möglich, FPGAs aus mehreren Tausend Gattern zu entwickeln, sodaß es möglich wurde, anwenderspezifische integrierte Schaltkreise (kurz ASICs) schnell und einfach herzustellen.

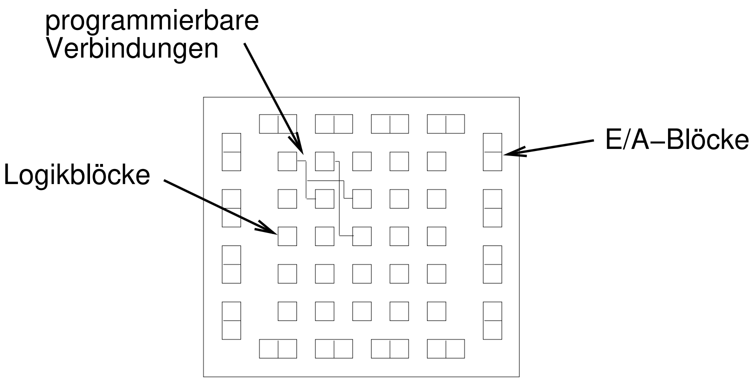

Abbildung 3: Grundstruktur von Xilinx-Bausteinen

Die Firma Xilinx entwickelte drei Generationen von FPGAs (siehe dazu auch nächstes Kapitel). Abbildung 3 zeigt die Grundstruktur von solchen Xilinx-FPGA-Bausteinen. Die

Funktionszellen im Kernbereich des Feldes (Logikblöcke) realisieren jeweils eine zweistufige SRAM- Speicherstruktur, dessen Inhalt vom Benutzer einprogrammiert wird. Zwischen den Zellen befinden sich programmierbare Verbindungen, die eine flexible Verbindung der Zellen untereinander ermöglicht. Im Randbereich befinden sich die Ein-/Ausgabezellen, die ebenso über die Verbindungen an die Logikblöcke gekoppelt sind.

Eine Zelle besteht (bei der XC 3000-Serie) aus neun Eingängen und zwei Ausgängen. Vernachlässigt man die Eingänge für die Multiplexer und Flipflops einer Zelle, so werden zwei Funktionen X und Y realisiert, die von 5 Eingängen A, B, C, D und E abhängig sind. Da der Speicher einer jeden Zelle jedoch nur aus 32 Bit besteht, existieren lediglich zwei mögliche Beschaltungen :

- 5 Eingänge, aber nur eine Ausgangsfunktion
- 4 Eingänge, aber dafür zwei Ausgangsfunktionen

Beispiel Es sei die logische Funktion

$$y = x_1\, x_2\, x_3\, x_4\, x_5\, x_6 \vee \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{x_4}\, x_5\, x_6$$

zu realisieren. Formt man diese Funktion durch Ausklammern zu

$$y = (x_1\, x_2\, x_3\, x_4 \vee \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{x_4})x_5\, x_6$$

um, benötigt man zu ihrer Realisierung zwei Funktionszellen (Abbildung 4).

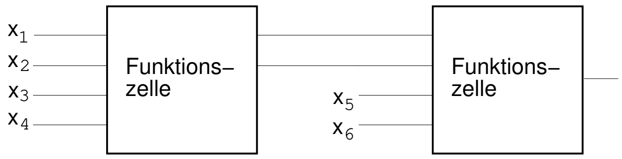

Abbildung 4: Beispiel für eine Funktionsrealisierung

3 Xilinx FPGAs

3.1 Produktinformation und -spezifikation

Die Xilinx FPGAs sind benutzerprogrammierbar und erhältlich in verschiedenen Variationen (verschiedene Größe, unterschiedliche Geschwindigkeiten, etc.). Die Grundstruktur entnehme man Abbildung 3.

Verglichen mit älteren Versionen, wurden bei der XC 4000-Serie die Logikblöcke wesentlich erweitert. So besteht ein Logikblock aus nunmehr 13 Ein- und 4 Ausgängen. Vier Eingänge

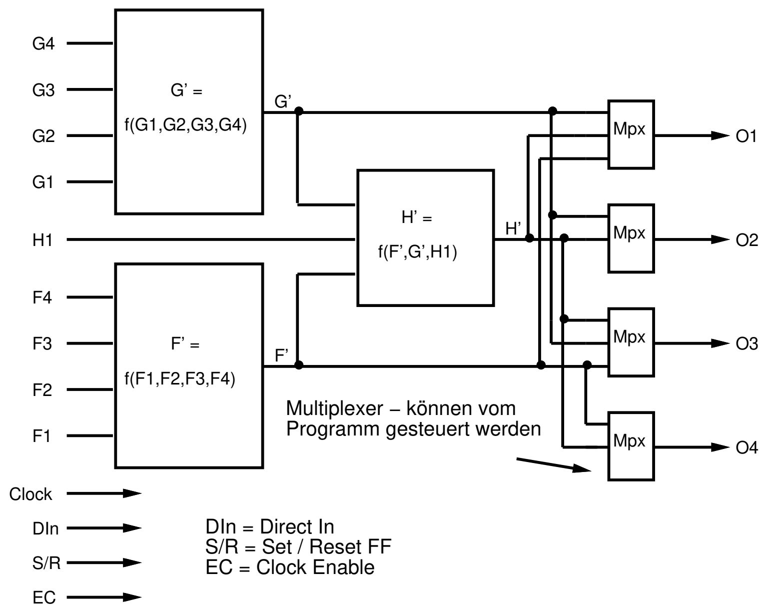

Abbildung 5: Stark vereinfachtes Blockdiagramm eines Logik-Blocks

sind für jeweils eine der insgesamt zwei Funktionsausgänge (F' und G') vorgesehen. Ein dritter Funktionsausgang (H') verknüpft weiterhin noch drei Eingänge, darunter die oben genannten beiden Funktionsausgänge, zu einem Ausgang (siehe auch Abbildung 5).

In einem anderen Modus lassen sich diese Logik-Bausteine auch als kleiner Speicher nutzen. Die Eingabevariablen dienen dabei der Adresseingabe, während die Signale `DIn` und `S/R` als Dateneingabe zweckentfremdet werden. Man erhält so Speicherkapazitäten von 16 x 2 bzw. 32 x 1 Bit.

Die Verbindungen zwischen den einzelnen Logik-Blöcken unterteilen sich in drei Arten :

- **single-lenght lines** bilden eine Matrix aus horizontalen und vertikalen Verbindungslinien, die mithilfe von Transistoren zusammengeschaltet werden können.
- **double-length lines** gleichen den single-lenght lines, sind im allgemeinen nur genau doppelt so lang.
- **longlines** erstecken sich über das gesamte Gitter von Logik-Blöcken. Der Unterschied zu den single-lenght lines besteht darin, daß die longlines zwei weiter entfernte Signale direkt verbinden, um die Zeitverzögerung möglichst gering zu halten. Longlines werden aus diesem Grund meistens für zeitkritische Signale benutzt.

4 Virtual ASIC

4.1 Aufbau des Virtual ASICs

Abbildung 6 zeigt den Aufbau des Hardwareemulators *Virtual ASIC*. Der uns zur Verfügung stehende Emulator besitzt zwei solcher Boards, die über die Backplane (Anschlußpins an der Rückseite des Gehäuses) miteinander verbunden sind.

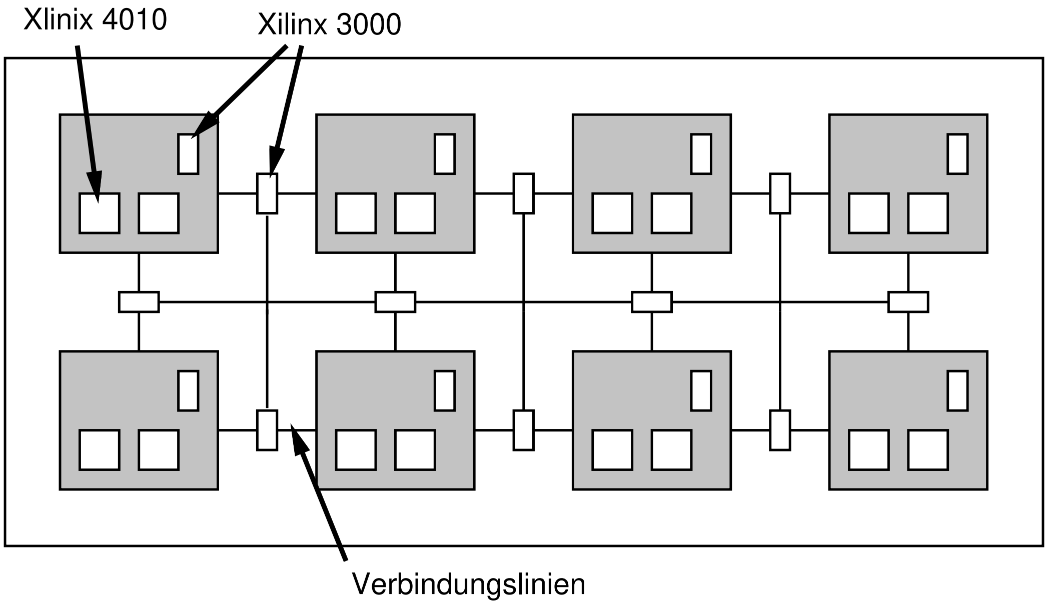

Abbildung 6: Aufbau des Hardware–Emulators

Jedes Board besteht nun aus acht Bausteinen, die wiederrum aus jeweils zwei Xilinx-4010-Chips und einem Chip aus der 3000er-Serie bestehen. Zehn weitere 3000er-Chips stellen die Verbindungen zwischen den acht Bausteinen her. Die beiden folgenden Tabellen zeigen die wichtigsten Daten der beiden Xilinx-FPGAs.

Xilinx–Daten

	XC3000	XC4000
Anzahl Gatter	n.a.	10000
Logik–Block–Matrix	16 x 20	20 x 20
Anzahl Logik–Blöcke	320	400
Anzahl Flip–Flops	928	1120
Maximaler Speicher	0	12800
Anzahl IO-Bits	144	160
Funktionsgeneratoren	2	3
Anzahl Eingänge pro Logikzelle	5	9
Anzahl Ausgänge pro Logikzelle	2	4

Quelle : Xilinx - The Programmable Logic Data Book

Xilinx–Leistung

		XC3000		XC4000	
16 Bit Addierer	max. Geschwindigkeit	30 ns	41 LB	20.5 ns	9 LB
	min. Bauelemente	50 ns	30 LB	20.5 ns	9 LB
16 Bit Zähler	max. Geschwindigkeit	30 MHz	27 LB	40 MHz	8 LB
	min. Bauelemente	20 MHz	16 LB	40 MHz	8 LB
16 Bit Dekodierer		15 ns	4 LB	12 ns	0 LB

Quelle : Xilinx - The Programmable Logic Data Book

4.2 Entwicklungsschritte

Die Firma InCA *(Integrated Circuit Applications)* entwickelte ein Softwarepaket, mit dem es möglich wurde, die Programmierung von FPGAs stark zu vereinfachen. Das Paket läuft auf SUN Workstations genauso wie auf PCs. Der Entwicklungsprozeß verläuft im allgemeinen nach folgenden Gesichtspunkten :

- **Schematischer Entwurf**
- **Simulation**
- **Fertigung einer Netzliste :** wird benötigt, um den Schaltungsentwurf in ein für das Softwarepaket lesbaren Code umzuwandlen. Außerdem findet bei diesem Entwicklungsschritt eine Art Prüfung auf allgemeine Schaltungsfehler statt (z.B. wenn zwei Ausgänge eines Gatters miteinander verbunden sind).
- **Partionieren :** Hierbei wird die Schaltung in kleinere Blöcke aufgeteilt, um sie dann in das FPGA-Board zu implementieren. Es wird zwischen mehreren Strategien unterschieden :
  - Benutzerdefiniertes Partionieren
  - Blockweises Partionieren
  - Vollautomatisches Partionieren

  Die Aufgabe des Partionierers ist es, die gesamte Schaltung in die einzelnen Logik-Blöcke des FPGA-Boards aufzuteilen. Diese Logik-Blöcke erscheinen dann im dem `browser`- Programm als Icon.
- **Compilieren :** Beim Compilieren wird der bisherige Code in einen binär verschlüsselten Code umgewandelt, der dann direkt in das Board programmiert werden kann.

4.3 `browse` — Die grafische Entwicklungsumgebung

4.3.1 Allgemeines

Das Programm `browse` ist eine grafische Entwicklungsumgebung, die auf SUN Workstations läuft und folgende Funktionen besitzt :

- **translation.** Schaltungsentwürfe können von verschiedenen Formaten in das Format konvertiert werden, welches von `browse` benötigt wird.
- vertiert werden, welches von `browse` benötigt wird.
- **validation.** Es ist möglich, Schaltungen auf spezielle Fehler zu untersuchen, um ein einwandfreies Funktionieren der Emulation zu gewährleisten.

- **partitioning.** Partionieren ist notwendig, um die Schaltung später auf das FPGA-Board kopieren zu können.
- **global placements.** Einmal partioniert muß eine Schaltung auf physikalische FPGA-Resourcen festgelegt werden.
- **global routing.** Nach dem Plazieren müssen die Verbindungen zwischen den einzelnen Logik-Blöcken hergestellt werden.
- **FPGA compilation.** Der wichtigste Prozeß – das Compilieren.

Gestartet wird das Programm wie folgt :

```
browse -top <design_file>
```

Der Parameter <design_file> bestimmt die Datei, die eingelesen werden soll, wobei es sich um eine vorher abgespeicherte Datei oder um eine Netzlisten-Datei handeln kann.

Vor der Benutzung des Programmes ist es jedoch erforderlich, einige Environment-Variablen zu setzen :

```
setenv XILINX $INCA_SOFT/xilinx
setenv XACT $INCA_SOFT/xilinx
setenv XACT_MEMSIZE 12
setenv XLM_ENABLE_LM_LICENSE_FILE 1
setenv LM_LICENSE_FILE /usr/local/data/license.dat
```

Einfacher ist es, wenn man sich die Datei setup.inca, die sich in meinem public–Verzeichnis (~ full/public) befindet, in seine eigene .cshrc–Datei einbindet. Zu beachten ist noch, daß die Software nur auf einer X11–Umgebung und auch nur auf Farbbildschirmen läuft. Außerdem ist diese Software lizensiert und nur auf folgenden Rechnern verfügbar :

elba	eros	gizeh	here	haegar2
kiew	korsika	pollux	ringo	wembley

Ist man also auf einem anderen Rechner eingeloggt, so ist es zwingend erforderlich, sich mittels rlogin <rechnername> auf einen der oben genannten Rechner einzuloggen. Um dann auch die X11–Software laufen zu lassen, muß man unbedingt folgende Enviroments setzen :

xhost +<rechnername> **vor dem rlogin--Befehl !**

und dann nach dem rlogin-Befehl und dem Einloggen auf dem fremden Rechner :

```
setenv DISPLAY <eigener rechnername>:0.0
```

Dazu ein kleines Beispiel. Angenommmen, man ist momentan auf dem Rechner paris eingeloggt und will die InCA–Software benutzen. Folgende Schritte sind dann notwendig :

```
xhost +pollux
rlogin pollux
```
hier nun das eigene Passwort eingeben
```
setenv DISPLAY paris:0.0
```

4.3.2 Die Menüs

Dies soll lediglich eine kurze Zusammenfassung der wichtigsten Befehle sein. Eine ausführliche Beschreibung findet man in [INCA].

Das File-Menü Mit dem file-Menü kann man, wie sollte es auch anders sein, seine Daten komplett bzw. teilweise sichern und wieder laden.

WRITE DESIGN... Speichert das momentane Dokument.

READ DESIGN... Lädt ein vorher gespeichertes Dokument.

SAVE PART... Speichert nur einen (vorher angeklickten) Block ab.

RESTORE PART... Lädt einen einzelnen Block.

SHELL... Ruft eine UNIX-Shell mit dem Verzeichnis eines vorher angeklicken Blocks auf.
QUIT Beendet das Programm.

Das view-Menü Dieses Menü wird benutzt, um das Format der Darstellung auf dem Bildschirm zu beeinflussen.

ARRANGE CELLS... Alle Icons werden in genormten Abständen neu gezeichnet.

SHOW PHYSICAL... Zeigt die pysikalischen Verbindungen zwischen Blöcken. Nützlich nach dem Befehl *Route...*, um sich das Ergebnis anzuschauen.

SHOW CONNECTIONS... Positioniert die angeklickten Icons auf die linke Seite des Bildschirms, die Icons, die mit diesen verbunden sind, auf die rechte Seite und alle übrigen Icons werden an den unteren Bildrand geschoben.

SHOW PORT NAMES... Zeigt die Signal- und Port-Namen, die mit den selektierten Icons in Verbindung stehen an.

SHOW NETS... Zeigt die Verbindungen des selektierten Icons an.

PROCESS MANAGER... Zeigt alle momentan laufenden Prozesse an.

LOG... Übersicht über alle bisherigen Statusmeldungen des Programms.

Das edit-Menü

CREATE... Kreiert ein neues Icon.

REMOVE... Löscht ein selektiertes Icon.

FIND... Sucht Icons.

EDIT CONSTRAINTS...

CREATE CONSTRAINTS...

EDIT LAYOUT... Zeigt das Layout des FPGAs grafisch an.

EDIT DESIGN... Das selektierte Icon wird als .CIF-Datei abgespeichert und kann dann mit einem Texteditor editiert werden.

Das Properties-Menü

TOOLS... Verändert einige Eigenschaften des Programmes.

INSTANCE... Zeigt die Eigenschaften des selektierten Icons an.

Das Tools-Menü

CLUSTER... Verbindet Icons miteinander.

ROUTING... Startet den Router.

PARTITION... Startet den automatischen Partionierer.

ESTIMATE... Startet die Überprüfung des Schaltung auf evtl. Fehler.

COMPILE... Compiliert die Schaltung und erstellt eine Datei, die von dem Virtual ASIC benötigt wird.

DOWNLOAD... Programmiert den Virtual ASIC.

4.3.3 Die Vorgehensweise mit dem `browse`–Programm

Voraussetzung bei der Benutzung des Programmes ist eine Datei, die in dem .edif–Format vorliegt. Dieses Format benutzt z.B. auch das `Synopsys`–Programm. Um nun den Inhalt der Datei grafisch auf dem Bildschirm darzustellen, muß man die Datei in das für das Programm `browse` eigene Format umwandeln. Dies geschieht mit dem Menü PROPERTIES...TOOLS..., womit aus der .edif–Datei eine .cif–Datei erzeugt wird. Die nun dargestellten Gatter werden in eine oder (bei größeren Schaltungen) in mehrere Partitionen verschoben. Dazu legt man vorher mit EDIT...CREATE...PARTITION die benötigte Anzahl von Partitionen an, welche man dann wiederrum in sog. *Cards* verschiebt. Diese Cards werden mit EDIT...CREATE...CARDS...VA-II-4010 angelegt und repräsentieren den Xilinx–4010–Chip auf dem Hardwareemulator.

Nach diesem Schritt kann man dann seine Schaltung routen und compilieren lassen. Ist alles erfolgreich abgeschlossen, ist man endlich in der Lage, seine Daten mittels des Download–Befehls in den Virtual ASIC zu laden.

Literatur

[Xilinx] *Xilinx - The Programmable Logic Data Book*, 1994

[FPGA] H. Grünbacher und R.W. Hartenstein. *FPGA - Field Programmable Gate Arrays*, Seite 11 - 25

[INCA] *InCA - Manual*

[Esch] B. Eschermann. *Funktionaler Entwurf digitaler Schaltungen.* Springer-Lehrbuch, 1993.

[Pals] PAL Device Data Book, Advanced Micro Devices, 1990