3 Entwurf, Simulation und Synthese von digitalen Strukturen

3.1 Programmierbare Logikschaltungen

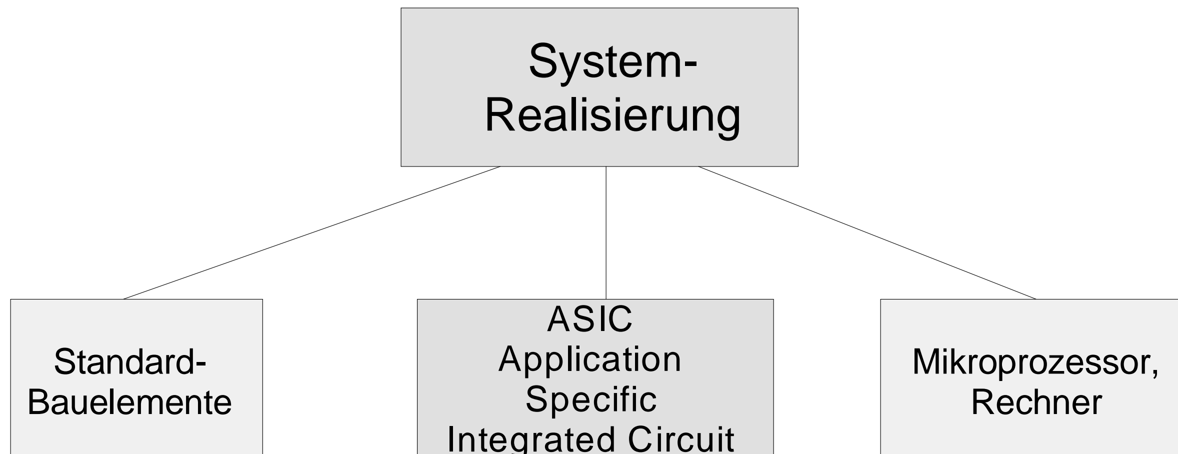


Bild 1: Übersicht ASIC, PLD, Prozessor

3.1.1 PLD-Strukturen

PLD (Programmable Logic Device) sind Anwenderspezifische Schaltungen (ASIC, Application Specific Integrated Circuit), die in ihrer logischen Funktion durch eine Programmierung beeinflusst werden können.

Dabei existieren zwei verschiedene Strategien:
- Viele Logikelemente mit relativ kleiner Komplexität und vielen, in der Regel eingeschränkt nutzbaren Verbindungsleitungen auf einem Chip. (Array-Struktur, auch als Channeled Array PLD bezeichnet)
- Wenige komplexe Logikelemente mit globalen, universell nutzbaren Verbindungselementen. (Multiple Array Struktur, auch als Segmented Block PLD bezeichnet)

3.1.1.1 AND-OR-Struktur

Die klassische PLD-Struktur besteht aus einer AND-OR-Struktur nach Bild 6. Die zweistufige Logik läßt sich einfach mit booleschen Gleichungen in disjunktiver Normalform beschreiben.

$Y = X_1\, X_2\, X_3 \ldots X_n \vee \overline{X}_1\, X_2\, X_3 \ldots X_n + \ldots + \overline{X}_1\, \overline{X}_2\, \overline{X}_3 \ldots \overline{X}_n$

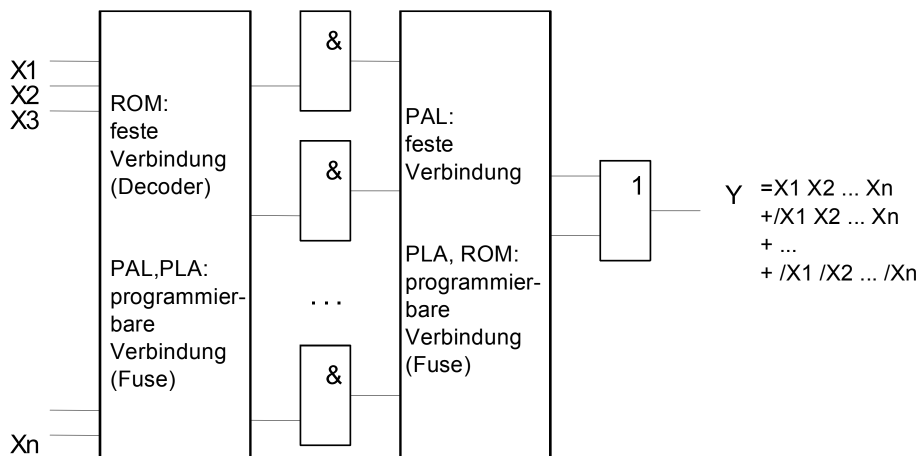


Bild 2: AND-OR-Struktur

Alle Eingänge werden direkt und invertiert über eine feste oder programmierbare Verbindung (Fuse) mit einer AND-Verbindung zusammengefaßt. Eine feste oder programmierbare Anzahl dieser AND-Verknüpfungen wird einer ODER-Verbindung zugeführt (zweistufige AND-OR-Logik). In Abhängigkeit von der Programmierbarkeit dieser Struktur unterscheidet man:

	ROM	PAL	PLA
AND	fest	programmierbar	programmierbar
OR	programmierbar	fest	programmierbar
Beispiele	PROM, EPROM	PAL, EPLD	PLA

Tabelle 2: Programmierbarkeit von PLD´s

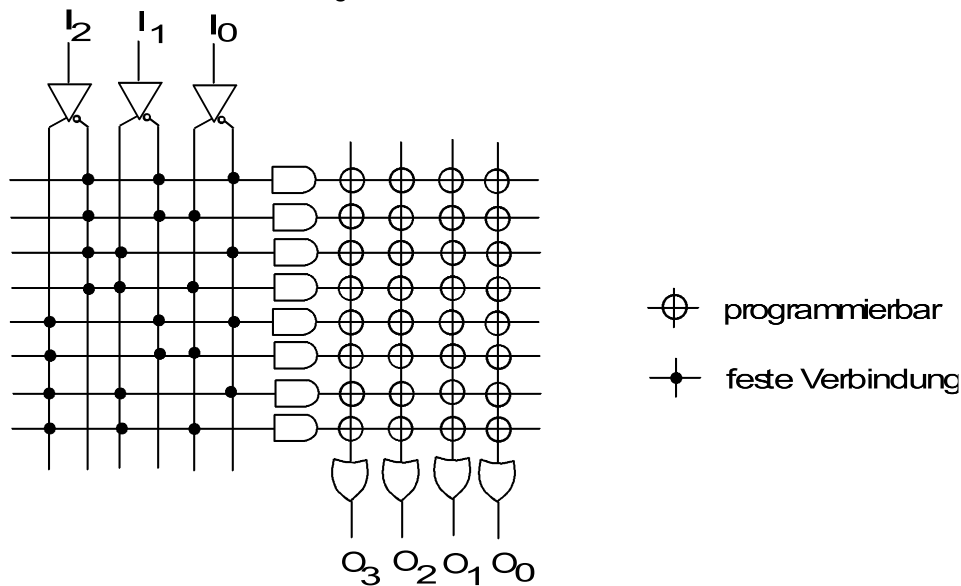


Bild 3 Grundstruktur eines PROM

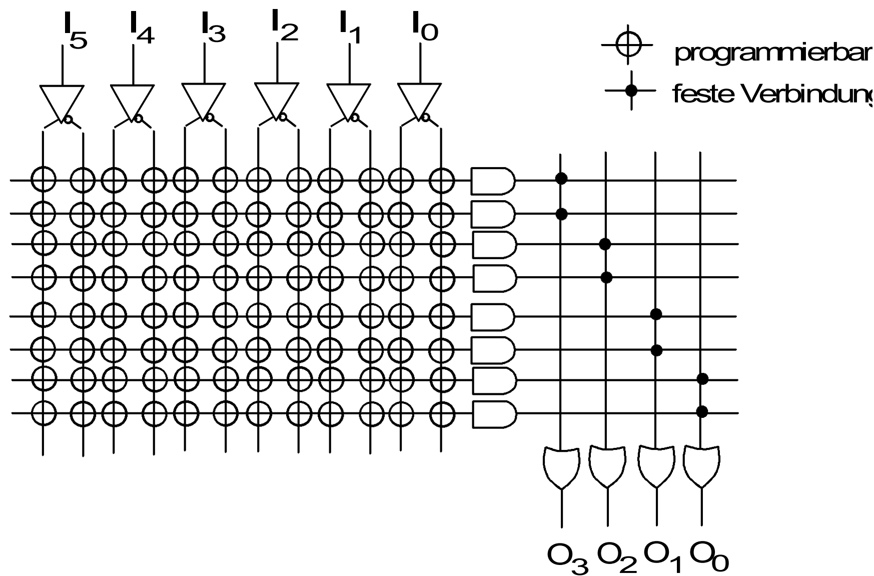


Bild 4 Grundstruktur eines PAL

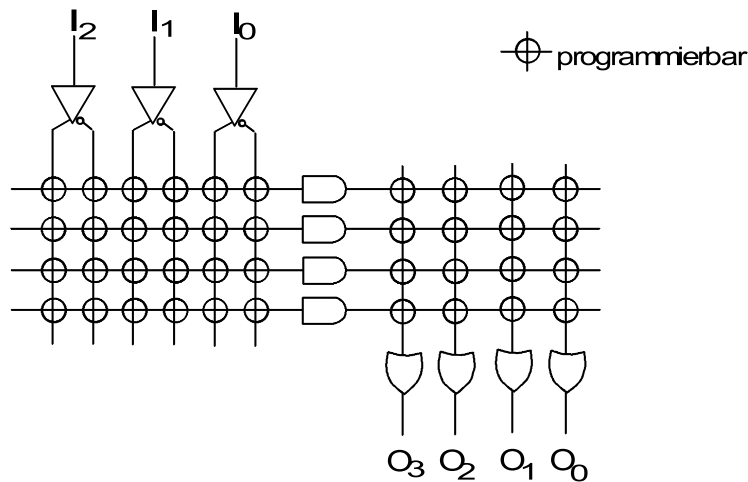


Bild 5 Grundstruktur eines PLA

Eine wesentliche Steigerung der Funktionalität von PLD´s in AND-OR-Struktur wurde durch die Einführung von Flipflops in Form von programmierbaren Register-Ausgangsmacrozellen (OLM: Output Logig Macrocell) in PAL- und PLA-Strukturen erreicht. Dabei finden meist D-Flipflops Verwendung, in EPLD´s aber auch JK-, RS- und T-Flipflops mit teilweise Reset- und Set- Eingängen (Bild 6).

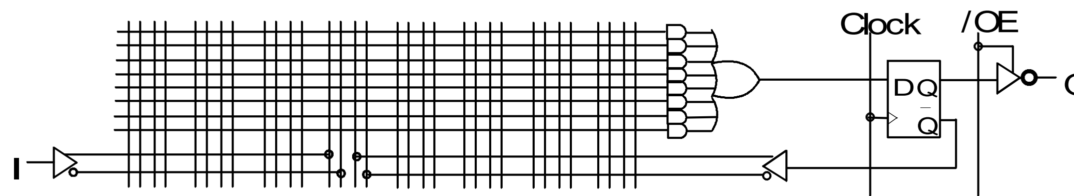


Bild 6 Register-PLD

Die Forderung nach

- einer möglichst hohen Anzahl von Makrozellen bei begrenzter Anzahl von I/O-Leitungen, wie sie in schnellen Mikroprogrammsteuerungen (Sequenzer, state machines) oder Strukturen mit internen Zählern oder Teilern benötigt werden, und nach
- programmierbarer Zuordnung von I/O-Zellen zu den internen Makrozellen

führten zu einer Anordnung mehrerer PAL- oder PLA-Grundstrukturen in Blöcken mit speziellen Verbindungswegen untereinander und zu den I/O-Leitungen. Diese "Multiple Array" Strukturen werden als „Enhanced“ PLD (EPLD) bezeichnet. Die großen Verbindungsmatrizen, genannt PIA (Programmable Interconmnect Array) lassen Punkt-zu-Punkt Verbindungen zwischen beliebigen PAL-Grundstrukturen (z.B. Macrocell/Expander Product Term Array, Generic Logic Block) zu (Bild 9).
Eine solche Struktur ist vorteilhaft, wenn die programmierbaren Verbindungselemente relativ groß und hochohmig sind, wie z.B. bei MOS Schalttransistoren in „floatig gate“ Technologie. Die Verbindungsmatrix beansprucht einen erheblichen Flächenanteil. Als Beispiel sei im Folgenden der EPLD MAX7064SLC44-10 gezeigt, der im Praktikum eingesetzt wird.

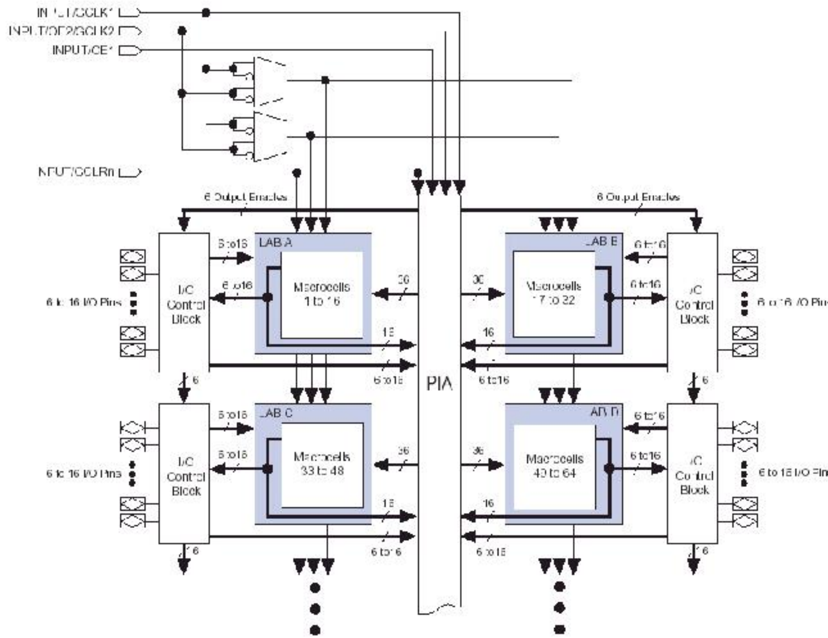


Bild 7 64 Makrozellen im EPLD mit Multiple-Array-Struktur: EPM7064SLC44-10

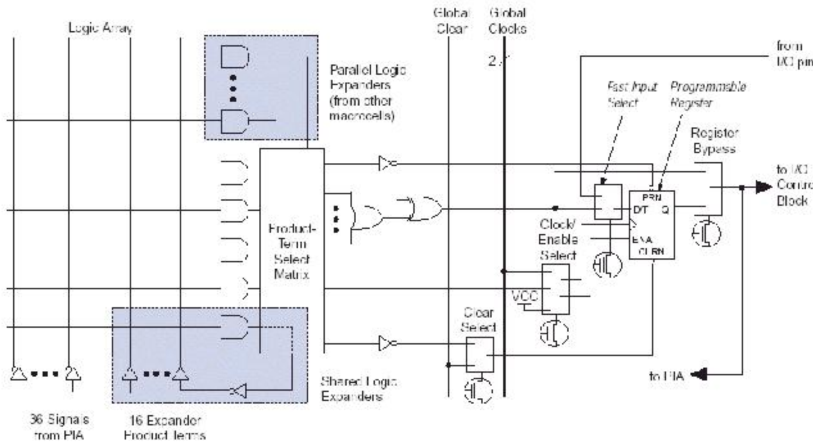


Bild 8 eine Makrozelle des EPLD EPM7064SLC44-10

Zu den Details sei auf das Datenblatt des EPM7064 verwiesen (Praktikum Digitale Systeme).

3.1.1.2 Array-Struktur

PLD´s in Matrix-Anordnung (Array-Struktur) besitzen eine zeilen- und spaltenweise Anordnung von CLBs (Configurable Logic Block).

Häufigster Vertreter dieser Klasse von Schaltkreisen ist ein FPGA (Field Programmable Gate Array). Sie basieren auf einer Matrix von Funktionsblöcken, zu denen CLB-Logikblöcke, E/A-Blöcke sowie Verbindungsblöcke gehören (Bild 9). Die Funktion der CLB Blöcke und ihre Verbindung untereinander ist programmierbar. Die logische Verknüpfung von Signalen erfolgt in den CLB-Blöcken, die im Wesentlichen einen kombinatorischen Teil (über eine Logiktabelle frei programmierbar) und einen sequentiellen Teil (Flipflop) beinhalten.

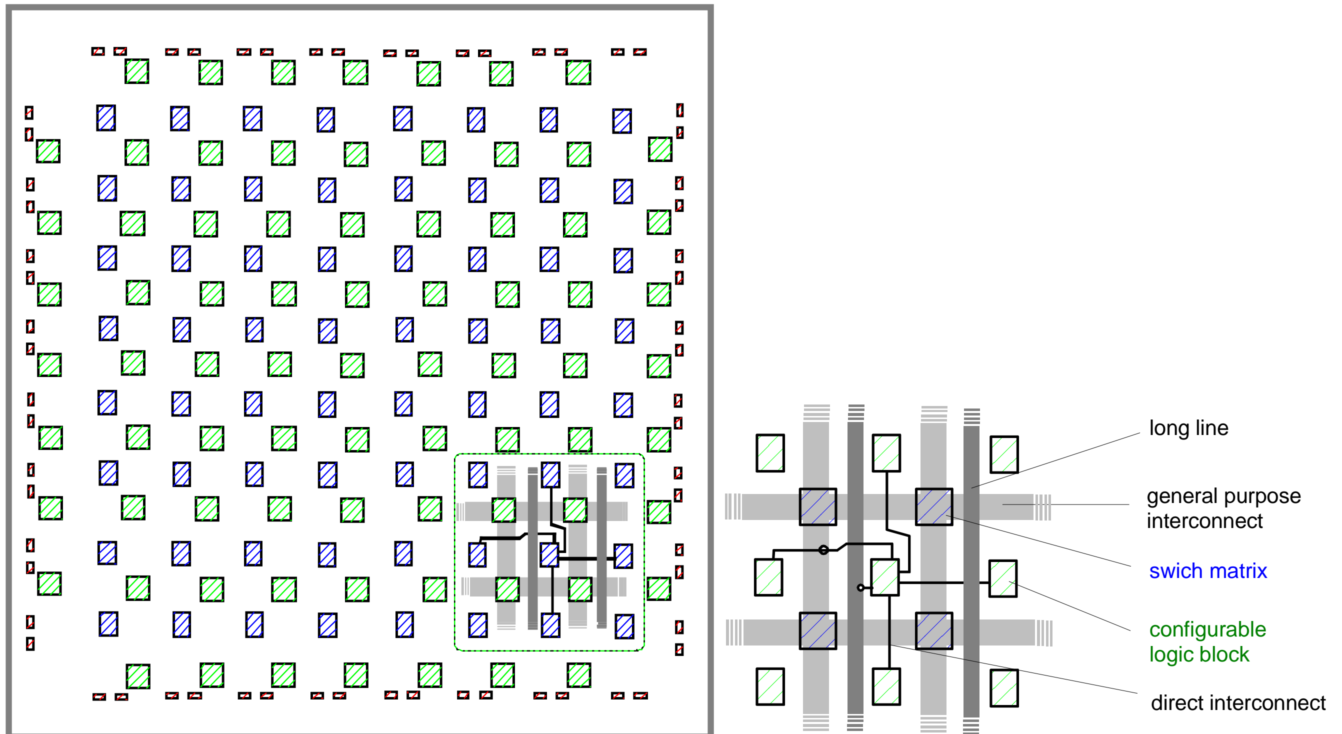


Bild 9 Grundaufbau eines PLD mit CLB

Die Verbindung zur Peripherie des Schaltkreises erfolgt an den Kanten des Schaltkreises über E/A-Zellen, die wahlweise (programmierbar) Tristate-Treiber , Eingangs- und Ausgangs-Register beinhalten. Die Verbindung der einzelnen CLB-Blöcke untereinander und zu den E/A-Zellen erfolgt über verschiedene lokale und globale Busse (general purpose interconnect, long lines), die horizontale und vertikale Verbindungen an den Kreuzungspunkten (switch matrix) zulassen.

3.1.2 Programmierung von PLD´s

3.1.2.1 Maskenprogrammierbar

Durch den Prozeß der Maskenprogrammierung im technologischen Herstellungsprozeß werden allgemein Gate-Arrays konfiguriert. Maskenprogrammierbare PLD´s in AND-OR-Anordnung (HAL, ROM) sind oft Versionen von elektrisch programmierbaren PAL´s bzw. PROM´s für große Stückzahlen.

3.1.2.2 Fuse-Programmierbar

Das hauptsächlich für ältere bipolare PLD´s verwendete Programmierverfahren ist die Nutzung von Schmelzsicherungen auf den Chipoberflächen (Laterale Fuse). Dabei wird ein wenige µm breiter Streifen aus einer Metallegierung mit hohem Wärmeleitwert (Nickel-Chrom, Titan-Wolfram) durch genau definierte Stromimpulse geschmolzen. Durch das Zurückziehen des geschmolzenen Metalles und die Bildung von Passivierungsschichten wird eine dauerhafte Trennung realisiert (Bild 10). Fuse-programmierbare PLD´s sind sehr schnell, da im unprogrammierten Zustand eine sehr niederohmige Verbindung besteht. Sie sind auf Grund des großen Flächenbedarfs der Fuse-Zellen und des großen Leistungsumsatzes während der Programmierung für kleine Integrationsdichten geeignet.

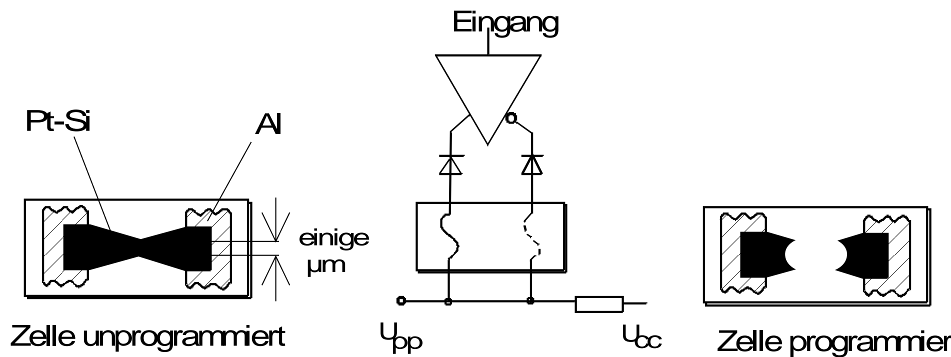
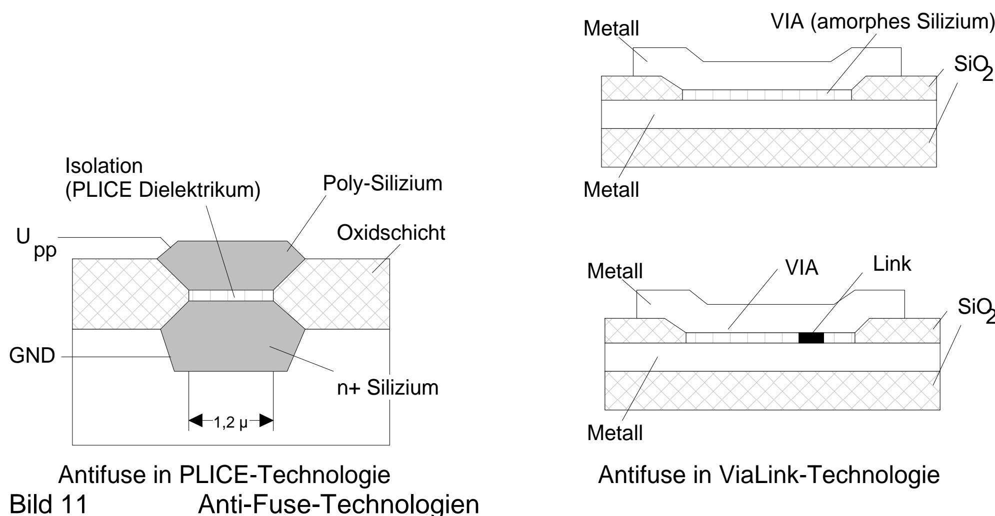


Bild 10 Fuse-Technologie

Beispiele: PAL, EPLD, bipolare PROM

3.1.2.3 Anti-Fuse-Programmierbar

Im Gegensatz zur Fuse-Technologie wird bei der Antifuse-Technik eine dauerhafte Verbindung innerhalb einer Isolationsschicht geschaffen. Die Programmierung erfolgt durch Anlegen einer hohen Spannung an U_{pp}. Anti-Fuse-Zellen sind sehr klein und im programmierten Zustand niederohmig, sodass eine große Anzahl von Schaltelementen auf einem Chip untergebracht werden können. Sie werden insbesondere in PLD´s in Array-Anordnung, z.B. FPGA´s eingesetzt.



Antifuse in PLICE-Technologie Antifuse in ViaLink-Technologie

Bild 11 Anti-Fuse-Technologien

3.1.3 Floating Gate, UV-Löschbar

Die Nutzung der Floating-Gate-Technologie erlaubt bei CMOS- PLD´s die reversible elektrische Programmierung. Durch das Anlegen einer hohen Programmierspannung gelangen hochbeschleunigte (heiße) Elektronen auf ein vollständig isoliertes Gate (Floating Gate). Das Löschen der Verbindungen erfolgt durch UV-Licht und besonders gestaltetem Gehäuse (Quarzglasfenster). Es werden auch OTP (One Time Programmable) - Versionen angeboten, die im billigeren lichtundurchlässigem Kunststoffgehäuse gefertigt werden.
Da EPROM-Zellen im on-Zustand relativ hochohmig sind, treten hohe Verzögerungszeiten pro Schaltelement auf. Sie werden daher insbesondere in EPLD´s mit Multiple Array-Strukturen eingesetzt.

Beispiele: EPROM, EPLD

3.1.4 elektrisch Löschbar

Sind die Floating-Gate-Oxidschichten so dünn (ca. 10...20 nm), dass bei realisierbaren Feldstärken ein Tunneleffekt der Elektronen einsetzt, so ist auch das Flash-Prinzip (Programmierung durch heiße Elektronen, löschen durch Tunneleffekt) oder die E²-Technik (Programmierung und Löschung durch Tunneleffekt) zur Programmierung von CMOS-PLD´s nutzbar.
Sie können im preiswerten Plastgehäuse gefertigt werden.
Bezüglich der elektrischen Eigenschaften und der Anwendbarkeit gelten die zu den UV-löschbaren PLD´s getroffenen Aussagen.
Beispiele: EEPROM, Flash-EPROM, Flash-ROM, EPLD

3.1.5 RAM programmierbar

Eine weiter Programmiermöglichkeit ist die Nutzung von RAM-Zellen zur Konfiguration der internen Logik- und Ein-/Ausgangsstufen. Da es sich um eine flüchtige Speicherung der Programmierungsinformation handelt, werden Betriebsarten vorgesehen, die eine effektive Programmierung von einzelnen oder mehreren kaskadierten Bauelementen ermöglichen. Beispielsweise können FPGA´s im *Master Mode* ihre Konfigurationsdaten aus einem parallel oder seriell angeschlossenen EPROM oder Flash-ROM auslesen. Dazu existieren ein interner Oszillator und ein Adresszähler. Im *Serial Master Mode* werden die Daten seriell über eine Daten- und Taktleitung aus einem Speicher mit internem Adreßzähler ausgelesen; in Verbindung mit weiteren FPGA´s, die im *Serial Slave Mode* betrieben werden, ist eine Kaskadierung mehrerer Bausteine möglich.
Beispiele: FPGA

3.2 Entwicklungswerkzeuge von PLD-Schaltungen

PLD-Schaltkreise benötigen für Ihre Programmierung auf jeden Fall Unterstützung durch entsprechende Programmiergeräte und Programmieralgorithmen. Es liegt daher nahe, auch den gesamten Schaltungsentwurf (Design) mit PLD-Schaltkreisen mit Programmunterstützung zu gestalten.

3.2.1 Fuse-Matrixbeschreibung

Die Programmierung von PLD´s erfolgt im Allgemeinen durch die Angabe der Zeilennummer und der einzelnen Programmierbits für die programmierbaren Verbindungen (fuses bzw. floating gates).
Der JEDEC Standard legt fest, wie diese Informationen im ASCII-Textformat abgelegt werden, so daß eine definierte Schnittstelle zwischen dem Designentwurf und den Programmiereinrichtung besteht.
Zur Programmierung von EPROM´s hat sich dagegen historisch der de-facto-Standard mit der INTEL-HEX-Code Beschreibung durchgesetzt, welcher etwas kompakter ist.

3.2.2 PLD-Assembler

Die Umsetzung von Booleschen Gleichungen in die Programmiervorschrift erfolgt mit einem PLD-Assembler (z.B. PLDASM). Da das JEDEC-File PLD-spezifisch erstellt werden muß, ist eine $DEVICE Angabe in der Quelldatei und eine entsprechende Bausteinbibliothek seitens des PLD-Assemblers notwendig.
Die PLD-Beschreibung auf PLDASM-Niveau hat folgende Grenzen:
- Die Minimierung der Produktterme, wie sie aufgrund der begrenzten OR-Verknüpfung in PAL´s notwendig ist, muß durch den Programmierer selbst erfolgen.
- Die Portierung eines Entwurfs auf einen anderen PLD-Typ erfordert oftmals einen Neuentwurf.
- Eine funktionale Simulation des Schaltungsentwurfs ist nicht möglich.

Vorteile des PLDASM bestehen in:
- der relativ einfachen Software-Wartung, es sind schnell neue PLD-Typen oder geänderte PLD-Eigenschaften implementierbar
- preiswerte Variante bei seltenen und einfachen PLD-Entwürfen

```
this is a simple logic made with the 18n8

Device 18n8

 pin1    = 1
 pin2    = 2
 pin3    = 3
 pin4    = 4
 pin5    = 5
 pin6    = 6
 pin7    = 7
 pin8    = 8
 pin9    = 9
 pin11    = 11
 pin12    = 12
 pin13    = 13
 pin14    = 14
 pin15    = 15
 pin16    = 16
 pin17    = 17
 pin18    = 18
 pin19    = 19
```

```
start

 pin15   /=  pin1 * pin2;

 pin14   /=  pin3 * pin4;

 pin13   /=  pin5 * pin6;

 pin16   /=  pin7 * pin8;

 pin17   /=  pin9 * pin11;

 pin18   /=  pin11 * pin12;

 pin19   /=  pin1 * pin2;

end
```

Bild 16: Darstellung einer einfachen Logik in PLDASM-Notation

3.2.3 CAE-Software

Alle führenden PLD-Hersteller bieten grundsätzlich auch die entsprechende Entwicklungssoftware an. In der Regel werden damit Schaltungsentwicklungen mit PLD´s des jeweiligen Anbieters unterstützt. Darüberhinaus bieten verschiedene Firmen auch universelle Entwicklungssoftware an, die durch ein umfassendes Bibliothekskonzept an die gängigen PLD-Typen angepaßt wird.

3.2.3.1 Universalprogramme

Entwurfssysteme, die eine Entwicklung von Schaltkreisen wie Gate-Arrays oder Standardzellenschaltkreise zum Ziel haben, bieten Möglichkeiten der Verifikation des Designentwurfes mittels programmierbarer Logik. Für den Fall der Implementierung auf PLD bzw. FPGA werden zahlreiche Tools angeboten, die eine Konvertierung der Gate-Array-Logikstruktur auf eine zweistufige AND-OR-Struktur oder auf eine Array-Struktur ermöglichen. Auf Grund der günstigeren Übereinstimmung des Zeitverhaltens und der größeren möglichen Gatter- bzw. Blockdichte werden EPLDs und FPGAs in Arraystruktur dabei bevorzugt.

In die Gruppe der firmenneutralen universell einsetzbaren Entwicklungssoftware gehört z.B.:

Symplify	FPGA Design	Synopsys
Symphony HLS	High Level Synthesis	Synopsys
Allegro	FPGA System Planner	Cadence
ORCAD Capture	FPGA PCB Tool	Cadence
Precision RTL	FPGA Synthese	Mentor
Leonardo Spectrum	FPGA Synthese	Mentor
FPGA Design	PCB Designtool	Altium

3.2.3.2 Spezialprogramme

PLD-Hersteller bieten oftmals Spezialprogramme zur Programmierung ihrer PLD-Typen an. Vorteile sind:

- sehr gute Ausnutzung der Ressourcen der PLD´s
- schnelle Anpassung an neue Typen desselben Herstellers
- Simulation unter Berücksichtigung der Signallaufzeiten bausteinspezifischer Ressource (z.B. zur Taktgenerierung)

Beispiele firmenspezifischer Programmpakete sind z.B.:

PLUS	von Altera (MAX+PLUS, Quartus)
ISE, EDK	von Xilinx
Libero IDE	von Actel

3.3 Simulation von digitalen Strukturen

3.3.1 Simulationsmodell

Es gibt verschiedene Ansätze zu Modellierung digitale Strukturen:

- mathematisches Modell
  $Y = \overline{X}_0$
- elektrisches Modell
  $$U_y = U_s(1 - e^{-t/RC})$$
- SPICE Modell

```
V1 1 0 PULSE 0V 1V 1ns 1ps 1ps 1ns 2ns *500MHz
V2 3 0 DC 1V
M1 2 1 0 0 nmos
M2 3 1 2 3 pmos
C1 2 0 0.01pF
```

- HDL (Hardware Description Language) Timing-Modell

```
X0 <= '0', '1' AFTER 1 ns;
Y  <= NOT X0 AFTER 1 ns;
```

Diese Modelle lassen sich auf modernen Digitalrechnern nach folgender Methode ausführen bzw. lösen:

- Formale Eingabe als Sprache, Notation, Grafik,..
- Extraktion der mathematisch/logischen Gleichungen.
- Lösung der Gleichungen, das ist u.U. mehrfach nötig.
- Speicherung und Darstellung der Ergebnisse.

Da viele Komponenten (z.B. Transistoren, digitale Gatter usw.) simuliert werden müssen, ergibt sich ein hoher Rechen- und Speicherbedarf. Er ist besonders hoch bei der numerischen Lösung von linearen und nichtlinearen Gleichungssystemen (mathematisches bzw. Spice-Modell), und wird geringer, wenn die Gatter (z.B. der Inverter) abstrakter beschrieben werden, z.B. im HDLTimingmodell.

Im Folgenden soll die Timing-Simulation (am HDL Beispiel) genauer erläutert werden:

Eine Timing-Simulation erfolgt in folgenden Schritten:

1. Aufstellung der Gleichungen und Ihrer Abhängigkeiten zu einem bestimmten Zeitpunkt:
   verallgemeinert: alle Prozesse werden aufgelistet: *process list*

2. Speicherung aller Signale zu bestimmten Zeitpunkten
   verallgemeinert: alle Signale werden aufgelistet: *signal list*

3. Organisation der Reihenfolge der Berechnung
   - periodisch (z.B. im Takt einer ns): *cycle driven*
   - bei Bedarf (also immer, wenn eine neue Signalkonstellation vorliegt): *event driven*

4. Speicherung bzw. Ausgabe der Signale
   - bei *cycle driven* Simulation werden in jedem Takt nur die Signale, nicht die Zeiten abgelegt. Längere Zeit unveränderliche Signale werden u.U. mehrfach abgelegt. Durch zusätzliches Speichern der Taktnummer und Beschränkung auf die Zeiten, in denen sich die Signale ändern, lässt sich das vermeiden.
   - bei *event driven* Simulation werden immer die Zeiten (Events) und die Signalwerte abgelegt. Die Zeit muss dazu hinreichend klein gestuft dargestellt werden (z.B. als natürliche Zahl mit 32 oder 64 bit und einer physikalischen Einheit fs, ps, ms, s, ...)

HDL Beispiel1:

```
X0  <= '0', '1' AFTER 1 ns;   -- Eingangssignal
Y   <= NOT X0;                -- Inverter
```

Y wird berechnet, wenn sich X0 ändert: bei 0 ns und bei 1 ns
Der berechnete Wert wird sofort in die *signal list* eingetragen

	0ns	1 ns
X0:	0	1
Y:	1	0

HDL Beispiel2:

```
X0  <= '0', '1' AFTER 1 ns;   -- Eingangssignal
Y   <= NOT X0 AFTER 1 ns;     -- Inverter
```

Y wird berechnet, wenn sich X0 ändert: bei 0 ns und bei 1 ns
Die berechneten Werte werden in die *signal list* jeweils 1 ns später eingetragen,
ggf. wird die Liste erweitert (hier: auf 2 ns). Treten keine Änderungen mehr auf, wird die Simulation beendet, die Signale bleiben unverändert X0=1 und Y =0.

	0ns	1 ns	2 ns
X0:	0	1	1
Y:	U	1	0

HDL Beispiel3:

```
                                -- Timing-Modellierung …
X1 <= '0', '1' AFTER 2 ns;     -- des Zeitverlaufs von X1
X2 <= '0', '1' AFTER 5 ns;     -- der Zeitverlaufs von X2
Y  <= NOT X0 AFTER 1 ns;       -- eines Inverters
X0 <= X1 AND X2 AFTER 1 ns;    -- eines AND Gatters
```

Eine *event-driven* Simulation würde so erfolgen:

1. Aufstellung der Gleichungen und Ihrer Abhängigkeiten:
NOT ist von X0 abhängig und aktualisiert Y
AND ist von X1 und X2 abhängig und aktualisiert X0

2. Speicherung aller Signale X0, X1, X2, Y für t=0ns, 1ns, usw.

	0ns	
X0:	U	U: noch unbelegt
X1:	0	
X2:	0	
Y:	U	(wegen X0=U kann Y noch nicht berechnet werden)

3. Organisation der Reihenfolge der Berechnungen
X0 wird berechnet (0) und für die Zeit 1 ns gespeichert
Da bei 1 ns X0 von U auf 1 wechselt, kann Y bei 2 ns aktualisiert werden.
Speicherung der Signale für weitere Zeitpunkte, an denen Änderungen auftreten.
Bei 3 und 4 ns finden keine Änderungen statt – es wird keine *signal list* für diese Zeiten abgelegt.
4. Speicherung bzw. Ausgabe der Signale:

	0ns	1ns	2ns	5 ns	6 ns	7 ns
X0:	U	0	0	0	1	1
X1:	0	0	1	1	1	1
X2:	0	0	0	1	1	1
Y:	U	U	1	1	1	0

Da die Zeit immer eine Adresse bzw. Bezeichner für die *signal list* ist, darf sie nicht als reelle Zahl, sondern muss immer als Ganze Zahl gespeichert werden.

3.3.2 Simulationswerkzeuge

3.3.2.1 MaxPlusII (Altera)

In MaxPlusII werden die Eingangssignale durch einen Waveform-Editor grafisch erzeugt,
Es erfolgt eine Event-Driven Simulation der Gleichungen, die in der Report-Datei (*.rpt) abgelegt sind.

VHDL Beispiel 3:

```
                    -- Timing-Modellierung …
Y  <= NOT X0;       -- (AFTER 1 ns) entfällt,
X0 <= X1 AND X2;    -- da die Gatter im Altera-Chip
                    -- Verzögerungszeiten haben
```

Die Report-Datei beinhaltet die konkrete Realisierung
Timing-Informationen `(AFTER 1 ns)` werden ignoriert bzw. durch konkrete Zeiten ersetzt. Der im Praktikum verwendete Schaltkreis EPM7064SLC44-10 hat 10 ns Verzögerungszeit pro Logikzelle (logic cell). Deshalb werden die Eingänge in größeren Zeitschritten als 1 ns geändert (hier 20 ns und 50 ns). Für die detaillierte Timingsimulation muss weiterhin die STEP SIZE auf 10 ns oder kleiner gesetzt werden.
Die Altera-Logikzellen haben eine Invertierungsmöglichkeit über ein XOR Gatter:
Y = X0 XOR 1 = X0 $ VCC (mit $ = XOR, VCC=1)

Report-Datei:

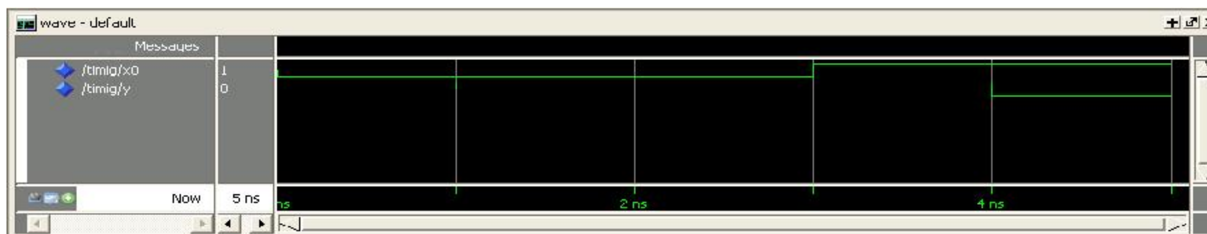
```
** EQUATIONS **
X1        : INPUT;
X2        : INPUT;
-- Node name is 'Y'
-- Equation name is 'Y', location is LC049, type is output.
 Y        = LCELL( _EQ001 $  VCC);
  _EQ001 =  X1 &  X2;
```

Waveform-Datei:

3.3.2.2 Modelsim (Mentor)

VHDL Beispiel 2:

```
X0  <= '0', '1' AFTER 1 ns;   -- Eingangssignal
Y  <= NOT X0 AFTER 1 ns;      -- Inverter
```



VHDL Beispiel 3:

```
                              -- Timing-Modellierung …
X1 <= '0', '1' AFTER 2 ns;    -- des Zeitverlaufs von X1
X2 <= '0', '1' AFTER 5 ns;    -- der Zeitverlaufs von X2
Y  <= NOT X0 AFTER 1 ns;      -- eines Inverters
X0 <= X1 AND X2 AFTER 1 ns;   -- eines AND Gatters
```

VHDL Simulatoren belegen bei Beginn der Simulation alle Signale in der *signal list* mit U, zum Zeitpunkt 0 ns werden Zuweisungen von X1, X2 auf 0 daher in einem Zusatzschritt (delta-Zyklus) berechnet. Solche delta-Zyklen werden zu einem bestimmten Zeitpunkt so oft eingefügt, bis keine Signaländerungen in der aktuellen *signal list* mehr auftreten.

Signal Listing (Modelsim-Simulator)

list

ns	delta	/timing/x0	/timing/x1	/timing/x2	/timing/y
0	+0	U	U	U	U
0	+1	U	0	0	U
1	+0	0	0	0	U
2	+0	0	1	0	1
5	+0	0	1	1	1
6	+0	1	1	1	1
7	+0	1	1	1	0

Waveform (Modelsim-Simulator)