

7 Programmierbare Logik

7.1 Einleitung

Für die Entwurfsmethodik digitaler Schaltungen gilt es mehrere Dinge zu beachten. Die Wahl der Schaltungsstruktur wird durch den Technologiefortschritt beeinflusst. Wurde früher die Minimierung der Transistoranzahl (Chipfläche) angestrebt, sind heute oft kurze Entwurfszeit, Low-Power oder Geschwindigkeit wichtiger. Der Trend orientiert sich weg von chipflächenoptimierter Schaltung hin zu synthetisierbaren Strukturen, Ausnahmen bilden Komplexitäts- und geschwindigkeitsoptimierte Schaltungen.

7.2 Schaltnetze aus Multiplexern

Der Schaltnetzentwurf lässt sich mit Multiplexern (universal logic module ULM) vereinfachen, wenn weder Schaltgeschwindigkeit, Komplexität oder Verlustleistung kritisch sind. Ein Vorteil ist ihre hohe Programmierbarkeit (FPGAs). Mit Multiplexern für n Kontrollvariablen lassen sich alle Funktionen aus diesen Variablen bilden, für Funktionen mit mehr Variablen als die des Multiplexers erfolgt eine Kaskadierung von Multiplexern in zwei oder mehreren Ebenen (Datenselektor).

Der logische Entwurf mit Multiplexern erfolgt mit Hilfe des Erweiterungstheorems von Shannon. Die Funktion wird entsprechend der Multiplexerstruktur erweitert.

z. B. um x_1 und x_2 :

$$f(x_1, x_2, \dots, x_n) = \overline{x_1} \cdot \overline{x_2} \cdot f(0, 0, x_3, \dots, x_n) + \overline{x_1} \cdot x_2 \cdot f(0, 1, x_3, \dots, x_n) + \\ x_1 \cdot \overline{x_2} \cdot f(1, 0, x_3, \dots, x_n) + x_1 \cdot x_2 \cdot f(1, 1, x_3, \dots, x_n)$$

Beispiel 76. Abbildung einer Funktion auf einen 3-Variablen-Multiplexer:

$$f = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4 + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot x_3 \cdot x_4 + \\ \overline{x_1} \cdot x_2 \cdot x_3 \cdot \overline{x_4} + x_1 \cdot x_2 \cdot \overline{x_3} \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4}$$

Bei der kaskadierten Erweiterung werden die Teilterme entsprechend erweitert, wobei die Variablen die Kontrollfunktion für die nachfolgenden Ebenen übernehmen.

Beispiel 77. Implementierung mit 1-Variablen und 2-Variablen-Multiplexer:

$$f = \overline{x_1} \cdot [\overline{x_2} \cdot \overline{x_3} \cdot (x_4) + \overline{x_2} \cdot x_3 \cdot (\overline{x_4}) + x_2 \cdot \overline{x_3} \cdot (0) + x_2 \cdot x_3 \cdot (1)] + \\ x_1 \cdot [\overline{x_2} \cdot \overline{x_3} \cdot (0) + \overline{x_2} \cdot x_3 \cdot (0) + x_2 \cdot \overline{x_3} \cdot (x_4) + x_2 \cdot x_3 \cdot (\overline{x_4})]$$

Beispiel 78. Eine weitere Implementierungsvariante ist, alle Variablen mit Kontrollfunktion zu versehen (z. B. FPGA):

$$f = \overline{x_1} \cdot \overline{x_2} \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (1) + x_3 \cdot \overline{x_4} \cdot (1) + x_3 \cdot x_4 \cdot (0)] + \\ \overline{x_1} \cdot x_2 \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (0) + x_3 \cdot \overline{x_4} \cdot (1) + x_3 \cdot x_4 \cdot (1)] + \\ \overline{x_1} \cdot \overline{x_2} \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (0) + x_3 \cdot \overline{x_4} \cdot (0) + x_3 \cdot x_4 \cdot (0)] + \\ x_1 \cdot x_2 \cdot [\overline{x_3} \cdot \overline{x_4} \cdot (0) + \overline{x_3} \cdot x_4 \cdot (1) + x_3 \cdot \overline{x_4} \cdot (1) + x_3 \cdot x_4 \cdot (0)]$$

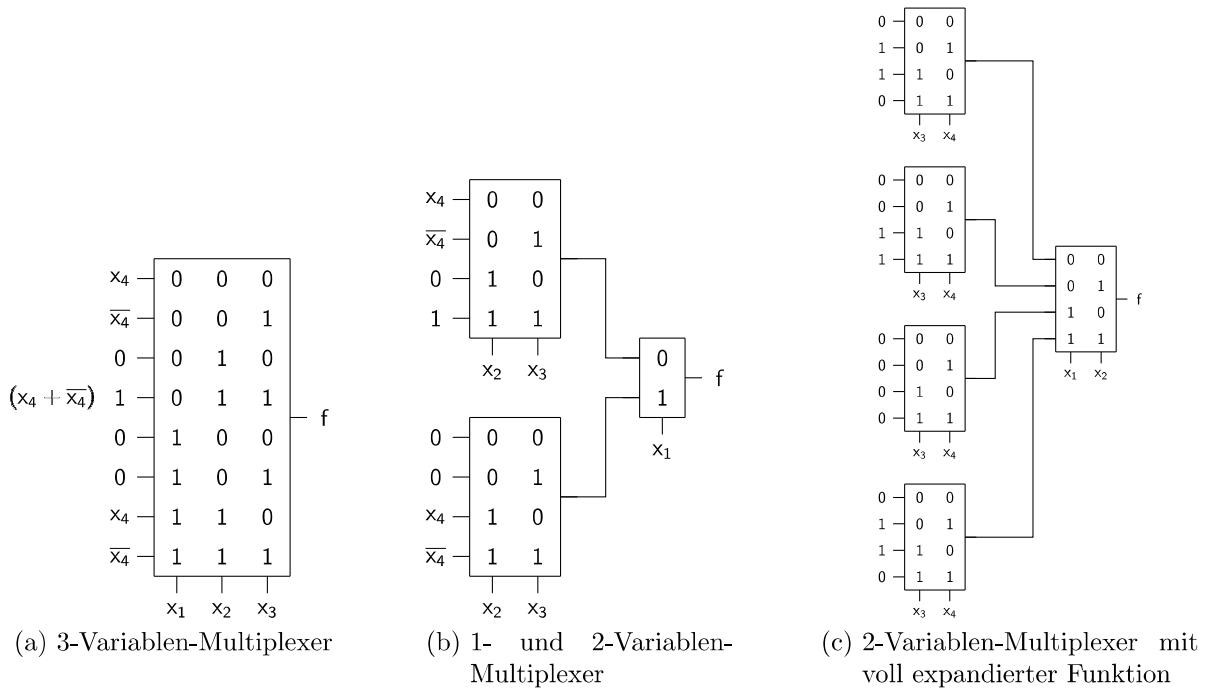


Abbildung 7.1: Implementierung der Funktion unter Verwendung verschiedener Multiplexervarianten

Implementierung von Funktionsbündeln

Stehen passende Multiplexer zur Verfügung, kann eine Minimierung entfallen, da die Funktionen vollständig expandiert zur Verfügung stehen.

Beispiel 79. *Funktionsbündel mit Multiplexern:*

$$\begin{aligned}
 f_1 &= \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \\
 &\quad x_1 \cdot \overline{x_2} \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot x_2 \cdot \overline{x_3} \cdot x_4 \\
 f_2 &= \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot x_4 + x_1 \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \\
 &\quad x_1 \cdot \overline{x_2} \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot \overline{x_3} \cdot x_4 \\
 f_3 &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4 + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot x_4 + \\
 &\quad x_1 \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + x_1 \cdot \overline{x_2} \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4}
 \end{aligned}$$

x_1, x_2 und x_3 werden als Kontrollvariable eingesetzt:

$$\begin{aligned}
 f_1 &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot (0) + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot (\overline{x_4}) + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot (\overline{x_4}) + \overline{x_1} \cdot x_2 \cdot x_3 \cdot (0) + \\
 &\quad x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot (0) + x_1 \cdot \overline{x_2} \cdot x_3 \cdot (1) + x_1 \cdot x_2 \cdot \overline{x_3} \cdot (1) + x_1 \cdot x_2 \cdot x_3 \cdot (0) \\
 f_2 &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot (0) + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot (0) + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot (1) + \overline{x_1} \cdot x_2 \cdot x_3 \cdot (0) + \\
 &\quad x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot (0) + x_1 \cdot \overline{x_2} \cdot x_3 \cdot (1) + x_1 \cdot x_2 \cdot \overline{x_3} \cdot (x_4) + x_1 \cdot x_2 \cdot x_3 \cdot (0) \\
 f_3 &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot (x_4) + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot (1) + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot (0) + \overline{x_1} \cdot x_2 \cdot x_3 \cdot (0) + \\
 &\quad x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot (0) + x_1 \cdot \overline{x_2} \cdot x_3 \cdot (1) + x_1 \cdot x_2 \cdot \overline{x_3} \cdot (\overline{x_4}) + x_1 \cdot x_2 \cdot x_3 \cdot (0)
 \end{aligned}$$

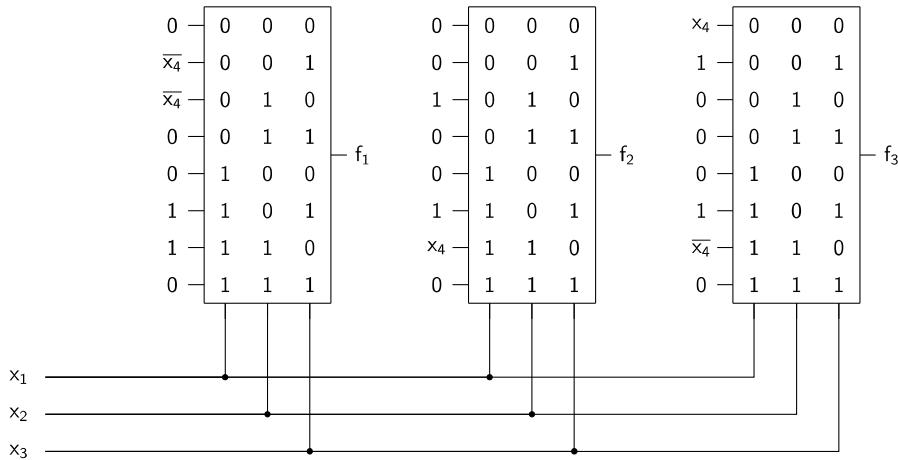


Abbildung 7.2: Implementierung des Funktionsbündels

7.3 Programmierbare Logik

7.3.1 Einleitung

Programmierbare Bausteine haben eine feste Struktur und können in ihrer Funktion speziell für eine Anwendung programmiert werden. Die programmierbare Logik nimmt bei

den Entwurfsstilen eine mittlere Stellung bezüglich Integrationsgrad und Entwicklungsdauer ein, wobei die Entwicklungskosten zum großen Teil von der Entwicklungsdauer beeinflusst werden. Die programmierbare Logik ist insbesondere wegen ihrer flexiblen Änderbarkeit attraktiv, ein Problem ist jedoch, dass die Programmierinfrastruktur die Integrationsdichte, Geschwindigkeit und Verlustleistung beeinträchtigt.

Programmierbare Schaltungen werden nach ihrer Struktur klassifiziert. Es gibt u.a.:

- Festwertspeicher (Read-Only Memories, ROMs): PROM, EPROM
- Schreib-/Lesespeicher: EPROM, EEPROM, EAPROM
- Programmierbare Logische Anordnungen (Programmable Logic Arrays, PLAs): PLA, PLD, GAL, PLM, FPLA, EPLD, Mehrfach-Array PLD
- strukturprogrammierbare Bausteine: FPGAs, PLSS (XILINX, ACTEL)

Programmierbare Array-Bausteine haben die Eigenschaft, dass die Komplexität wesentlich durch die Zahl der Anschlüsse und die Laufzeit begrenzt wird und dass der Logikentwurf der Synthese auf Gatterebene entspricht. Es gibt zwei verschiedene Möglichkeiten diese Bausteine zu programmieren. Bei irreversiblen Speichern (Festwertspeicher) werden Maskenprogrammierung und elektrische Programmierung eingesetzt, wobei die Maskenprogrammierung beim Halbleiterhersteller durchgeführt wird und nur bei großen Stückzahlen ($> 1M$) kostengünstig ist. Der Vorteil ist die hohe Schaltungskomplexität. Trotzdem werden maskenprogrammierte Bausteine heute weitgehend durch EPROM (OTP) ersetzt. Heutzutage ist jedoch die elektrische Programmierung eher vorzufinden.

Bei reversiblen Speichern (Schreib-/Lesespeicher) wird elektrisch geschrieben und entweder elektrisch oder durch UV-Licht gelöscht. Die elektrische Programmierung erfolgt beim Anwender im Labor und ist flexibel für kleinste Stückzahlen geeignet, die Nachteile sind der höhere Preis und die geringere Schaltungskomplexität und -geschwindigkeit.

7.3.2 ROM-Logik

Verschiedene Funktionsstrukturen haben verschiedene Einsatzbereiche, die Konjunktion ($f(x) = x_1x_2\bar{x}_3\dots x_n$) z.B. dient zur Realisierung von Mikroprogrammsteuerwerken, zur Programmierung einer Schaltnetzfunktion und zur Darstellung von Wertetabellen und mathematischen Funktionen.

Die Spalten einer ROM-Matrix stellen NOR-Gatter mit N Eingängen dar, die aus den Eingangsvariablen A, B und C dekodiert werden (1 aus n Code). Die Ausgangsfunktionen werden wegen der quadratischen Anordnung ebenfalls dekodiert. Eine ROM-Matrix mit N-Bit Adresse und M-Bit Ausgangscode benötigt demnach $M * N$ Speicherbits.

Ein Nachteil der ROM-Logik ist der Umstand, dass für alle N Zeilen und alle M Spalten Transistorplätze vorgesehen werden müssen, auch wenn sie zum Teil nicht benötigt werden. Meistens werden vom Schaltungsproblem auch nicht alle N Adressplätze (Zeilen) belegt, aufgrund der Redundanz ist die ROM-Logik also für die Implementierung einfacher Logikfunktionen ineffizient. Ein Vorteil liegt in der regelmäßigen Struktur der

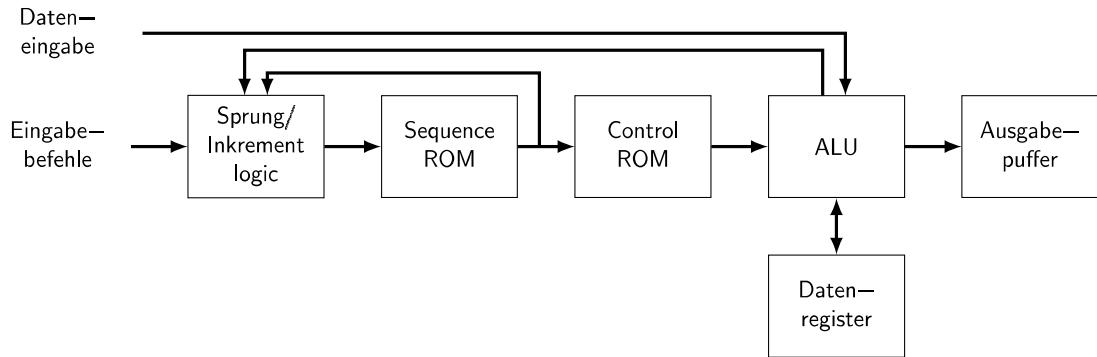


Abbildung 7.3: Blockschaltbild eines μ P mit zwei ROMs: für Folgeadresse (Sequence ROM) und Steuerwort (Control ROM)

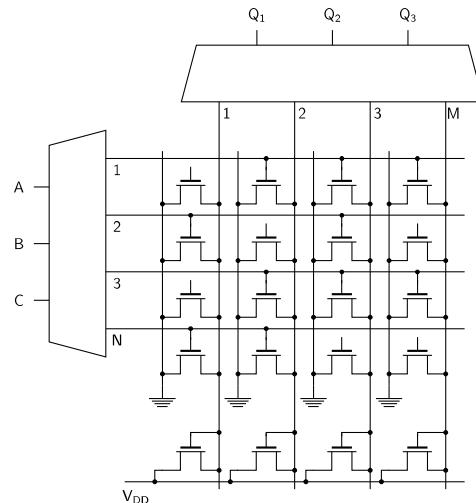


Abbildung 7.4: ROM-Matrix in MOS-Technik mit N Eingängen und M Ausgängen

ROM-Matrix, diese ermöglicht ein kompaktes Layout. Der Speicherinhalt kann leicht durch 1 bis 2 neue Masken programmiert werden, ohne dass eine Änderung der ROM-Architektur nötig wird.

Beispiel 80. $f_1 = \bar{x}yz + xyz; f_2 = \bar{x}z; f_3 = \bar{xy}z + \bar{x}z$

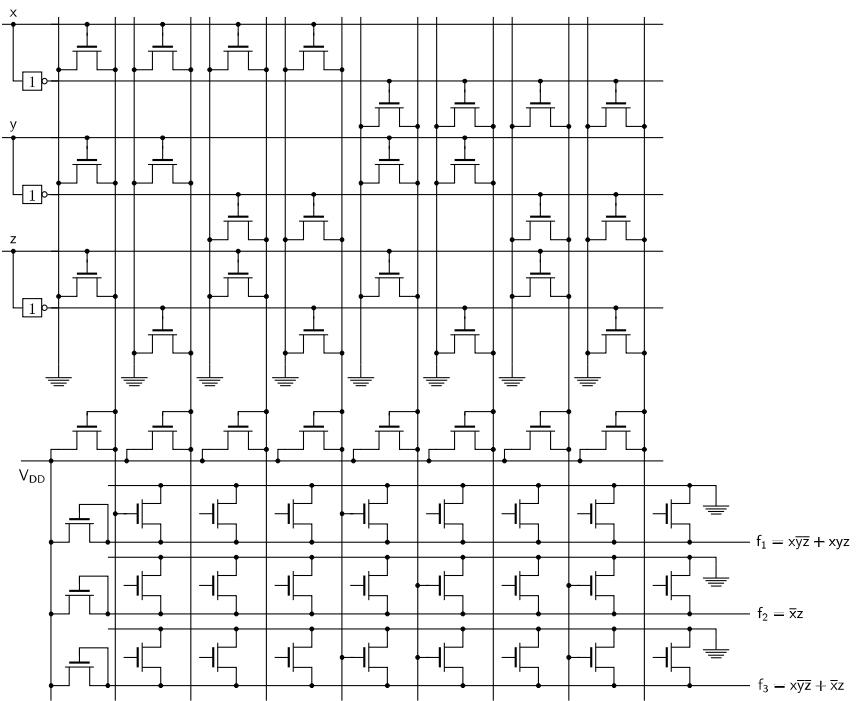


Abbildung 7.5: Beispiel einer ROM-Logik

7.3.3 Speicherzellen

Speicherzellen **irreversibler Festwertspeicher** sind aufgebaut aus Speicherelementen, die einer Kopplungsschaltung zwischen Wort- und Bitleitungen entsprechen, diese Koppelemente bestehen entweder aus Dioden, Bipolar- oder MOSFET-Transistoren.

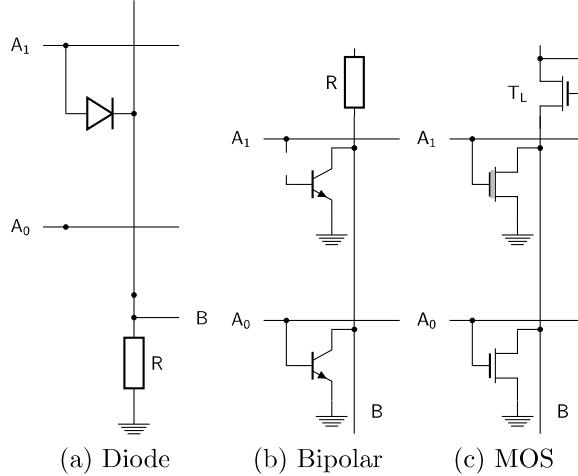


Abbildung 7.6: ROM-Matrizen basierend auf unterschiedlichen Technologien

Bei maskenprogrammierbare Speicherelemente sind die Speicherzellen also z.B. Dioden- oder Bipolartransistormatrizen. Das Datenmuster dieser Matix wird durch die Verdrahtung festgelegt, dies geschieht am Ende des Herstellungsprozesses, bei dem mit einer sog. Verdrahtungsmaske die Verbindung festgelegt wird. Bei einer Transistormatrix in MOS-Technik wird durch Maskenprogrammierung die Dicke der Gateisolierschicht festgelegt, wobei eine dicke Isolierschicht ein Schalten des Transistors trotz angelegter Spannung verhindert.

Elektrisch programmierbare Festwertspeicher (PROM) werden durch das Durchschmelzen (Zerstören) eines Widerstandes (fuse link) oder einer Diode programmiert. Ein spezieller Widerstand wird zur Basis des Bipolartransistors oder zur Diode in Reihe geschaltet. An der entsprechenden Speicherzelle wird an diesen Widerstand mit Hilfe eines Programmiergerätes eine hohe Spannung (ca. 10 V) angelegt, die den Widerstand bzw. die Diode wie ein Sicherungselement durchschmilzt.

Reversible Festwertspeicher besitzen die Eigenschaften, dass ihr Urzustand wiederhergestellt werden kann, das Löschen ist je nach Speichertechnologie auf zwei Arten möglich. Einerseits durch UV-Licht-Bestrahlung des Speicherchips, dabei wird die gesamte Speichermatrix auf einmal gelöscht, andererseits durch elektrische Impulse, durch die jede Zelle einzeln gelöscht werden kann.

Erasable PROM (EPROM) folgen dem 1-Transistor-FAMOS-Speicherelement in n-Kanal-Technik Speicherprinzip, bei dem (FAMOS = Floating Gate Avalanche Injection MOS) das Gate des Speichertransistors nach außen völlig isoliert in seinem Potential „schwebt“. Der Speichereffekt beruht auf einer Verschiebung der Schwellenspannung eines MOS-Transistors, d.h zum Programmieren wird der Source-Gate-Übergang des Spei-

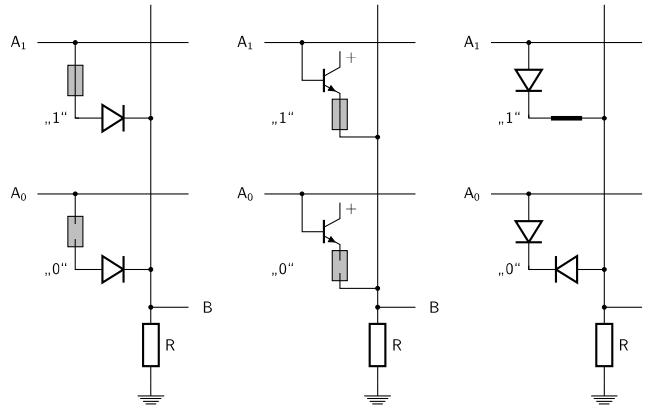


Abbildung 7.7: Elektrisch programmierbare Speicherelemente für Lesespeicher mit Schmelzsicherung

chertransistors mit hoher Spannung bis zum Avalanche-Durchbruch belastet. Der Speichertransistor arbeitet im programmierten und unprogrammierten Zustand als selbstsperrender Typ, die Löschung erfolgt mit UV-Licht. Die Bausteinkapazitäten liegen z.Z. bei 16 MBit.

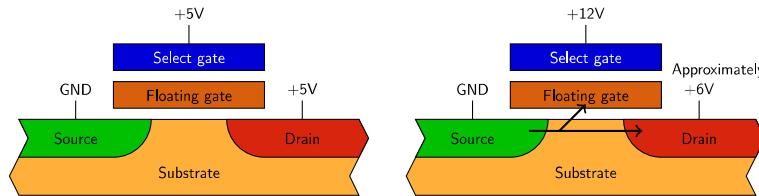


Abbildung 7.8: 1-Transistor-FAMOS-Speicherlement

Zur Neuprogrammierung im System wurden elektrisch löschräbare Speicher (EAROM = Electrically Alterable ROM bzw. EEROM = Electrically Erasable ROM) entwickelt, bei denen der Speichereffekt durch Speicherung von Ladung in einem schwebenden Gate (FAMOS) erreicht wird. Die Nachteile dieser Speicher sind hohe Programmierspannungen und längere Zugriffzeiten.

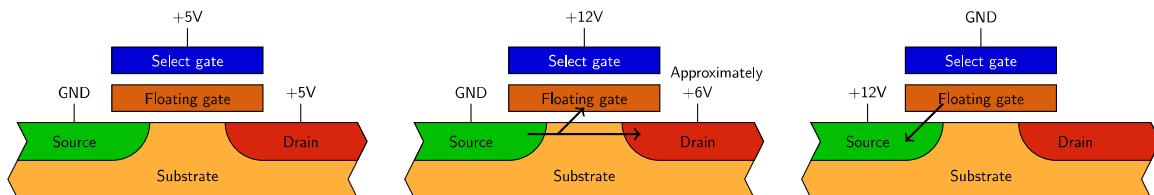


Abbildung 7.9: Programmierung einer FAMOS-EAPROM-Zelle

Die Vorteile der ROM-Logik liegen in leistungsfähigen Schaltnetzen mit mehrfachen Ein- und Ausgängen (Typische Anwendungen sind z.B. Codierer, Arithmetikschaltungen und ähnliches), der hohen Komplexität der Standardbausteine, und dass kein direkter Entwicklungsaufwand zur Minimierung der Logikfunktionen auf Gatterebene nötig

ist. Die Nachteile der ROM-Logik sind u.a. die langsamere Arbeitsgeschwindigkeit gegenüber irregulärer Logik (Kundenschaltung), diese Verzögerung wird verursacht durch eine zusätzliche Adressdecodierung und Matrixkapazitäten. Hinzu kommt eine recht schlechte Ausnutzung der Chipfläche, wegen unbenutzter Speicherbereiche und das Fehlen zusätzlicher Schaltungsmöglichkeiten, wie z. B. Negation oder Selektion von einzelnen Bits. Aufgrund dieser Nachteile ist die ROM-Logik weniger für kleine Schaltnetze geeignet, dafür aber für große kombinatorische Netzwerke mit regelmäßigen Strukturen (z.B. μP -Programme, Funktionstabellen).

7.3.4 Anwendungsbeispiele für ROM-Logik

Die algorithmische Berechnung arithmetischer Funktionen geschieht durch softwaremäßige Berechnung über Reihenentwicklungen, wenn keine Geschwindigkeit erforderlich ist:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

$$S_{k+1} = \frac{1}{2}(S_k + \frac{N}{S_k}) \text{ mit } \lim_{k \rightarrow \infty} S_k = \sqrt{N} \text{ (Newton-Raphson Verfahren)}$$

Die algorithmische Berechnung kann für technisch-wissenschaftliche Bereiche zu langsam sein, dann erfolgt die tabellarische Berechnung auf Hardwareebene. Die einfachste Realisierung ist die Verwendung eines ROM als Tabellenspeicher, Argument x als Adresse und Speicherinhalt als Ergebnis des Funktionswertes von x. Hierbei besteht aber der Nachteil, dass ein enormer Speicherbedarf bei gebräuchlichen Wortlängen benötigt wird, daher gibt es häufig Mischlösungen mit zusätzlicher Logik.

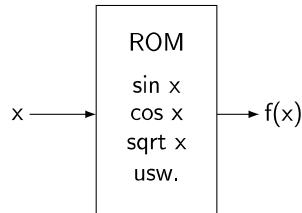


Abbildung 7.10: Struktur eines ROM-Schaltnetzes

7.4 Programmable Logic Array (PLA)

7.4.1 Überblick

Zur Klassifizierung programmierbarer Logikbausteine wird zwischen Array-Logikbausteinen wie PLA, PAL, PLD, GAL, PLM, FPLA, EPLD, Mehrfach-Array PLD und Strukturkonfigurierbaren Gate-Arrays wie XILINX, ACTEL unterschieden. Die physikalische Realisierung (Komplexität) programmierbarer Bausteine wird im wesentlichen durch die Zahl der Anschlüsse (Variablen) begrenzt. Ein Problem sind die mit wachsender Komplexität zunehmenden Leitungslaufzeiten. Der Logikentwurf wird von der Bausteinarchitektur geprägt, dies erfordert in der Regel eine firmenabhängige Entwurfsssoftware.

7.4.2 Schaltungsprinzip

Vergleich von ROM und PLA: ROM erfordert für n Adresseingänge einen Decodierer mit $2n$ Ausgängen, ein PLA (programmable logic array) wird dagegen direkt adressiert und besteht aus einer UND-Matrix mit einer kaskadierten ODER-Matrix. In der UND-Matrix werden die Eingangsvariablen in konjunktiver Form programmiert, in der ODER-Matrix erfolgt die Verknüpfung dieser Produkte. Diese 2-Ebenen-Programmierung erlaubt eine Informationsverdichtung mit Hardware-Einsparung gegenüber der ROM-Logik.

Funktionsstruktur DNF: $f(x) = x_1 \cdot x_2 \cdot \bar{x}_3 + x_4 \cdot x_5 \dots x_n + x_6 \cdot \bar{x}_7 + \dots$

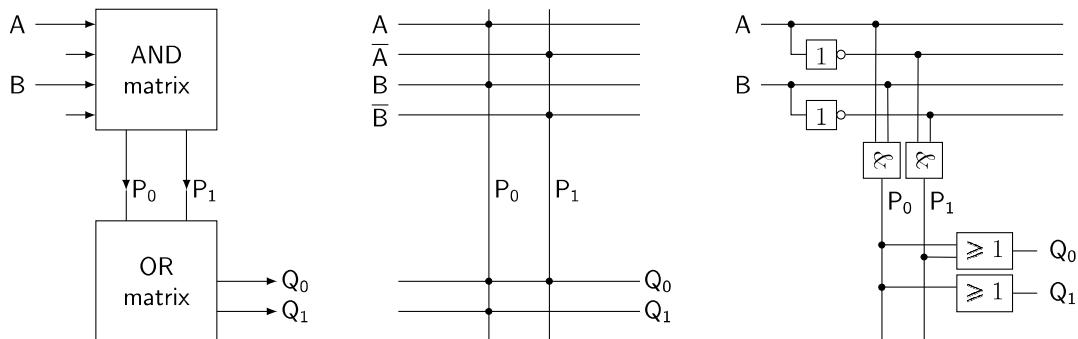


Abbildung 7.11: PLA: Blockdiagramm, Logikdiagramm, Logikschemata

Zur PLA-Realisierung werden für beide Matrizen NAND- oder NOR-Gatter gewählt, die Umwandlung in NAND-Funktionen ist direkt möglich. Eine NOR-Realisierung erfordert entweder negierte Ausgangsfunktionen oder die Programmierung der negierten Werte, NOR-Matrizen werden in MOS-Technik wegen der Parallelschaltung bevorzugt.

Ein Vergleich mit Beispiel in der ROM-Logik zeigt eine erhebliche Aufwandsersparnis.

Es sind nur 27 gegenüber 72 Matrixpunkten erforderlich.

Die höhere Informationsdichte des PLA gegenüber dem ROM ist aber auch nachteilig. Im PLA ist die Anzahl der Zeilen und Spalten auf das Schaltungsproblem abgestimmt, so dass eine Logikänderung gleichzeitig eine Strukturveränderung des PLAs zur Folge haben. Da die PLA-Matrix kleiner ist als die ROM-Matrix, sind die Verzögerungszeiten kürzer.

Die Vorteile für PLAs gegenüber der Realisierung mit Standard-Gattern sind, dass keine zeitaufwendige Minimierung der Logik mehr nötig ist, aber eine Anpassung möglich ist, es bedarf auch keiner speziell entworfenen Logikgatter, außerdem ist die PLA-Struktur aufgrund ihrer einfacheren und regelmäßigeren Struktur leichter skalierbar.

Die Nachteile liegen in ihrer geringeren Flexibilität in der Programmierung gegenüber ROM, PLAs können keine komplexen Funktionen speichern (insbesondere keine mit vielen Produkttermen), ihre Struktur ist nicht geeignet für vorstrukturierte Hardware (Gate Arrays, Standardzellen, FPGAs) und eine Matrix verursacht längere Laufzeiten gegenüber optimierten Schaltnetzen.

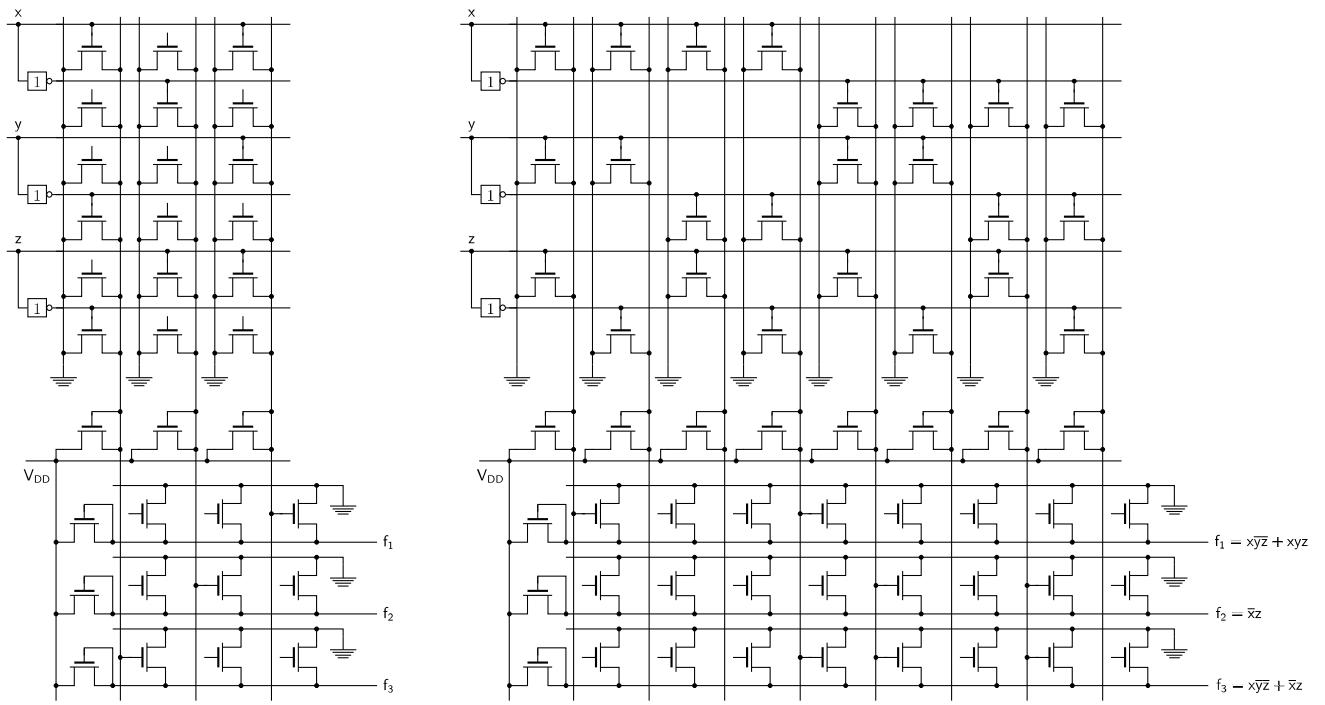


Abbildung 7.12: PLA-Beispiel

7.5 Andere programmierbare Logikschaltungen

Neben PLAs gibt es noch weitere Strukturen auf dem Markt, es handelt sich in der Regel um elektronisch programmierbare Schaltungen:

Array-Logikbausteine PLA und PAL

Der PLA ist der Vorläufer aller Array-Bausteine (PLA ab 1973, FPLA ab 1975) und besteht aus einem programmierbaren UND-Array und einem programmierbaren ODER-Array. Jeder Produktterm des UND-Arrays kann mit jeder Ausgangsfunktion verknüpft werden.

Der PAL entspricht einer Vereinfachung der PLA-Struktur (ab 1978) durch ein fixiertes ODER-Array, auch GAL und PLD genannt. Hierbei werden die Produktterme einer festen Struktur der Ausgangsfunktion zugeordnet, das ODER-Array wird durch eine einfache feste Verdrahtungsstruktur ersetzt. Die Kapazität der PAL-Bausteine liegt im Bereich der von PLAs.

Weitere Unterschiede gibt es im Logikentwurf. Bei PLAs können Produktterme mehrfach von verschiedenen Ausgangsfunktionen genutzt werden, bei PALs ist das nicht möglich, für jede Ausgangsfunktion müssen eigene Produktterme implementiert werden. Daher benötigen PAL-Entwürfe in der Regel mehr Produktterme als PLA-Entwürfe. Beide ermöglichen eine zweistufige Implementierung jeder booleschen Funktion (UND/ODER, disjunktive Normalform), vorausgesetzt der Baustein enthält genügend

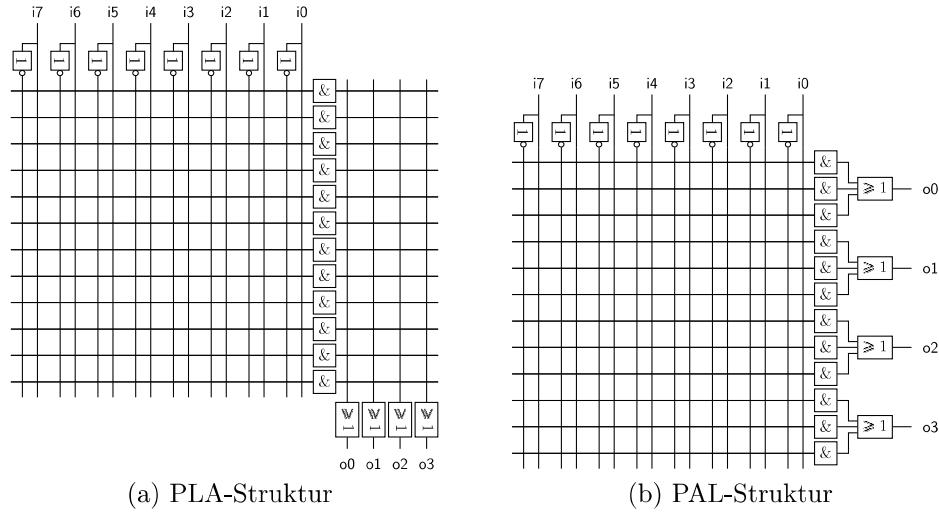


Abbildung 7.13: Strukturen programmierbarer Logikschaltungen

Produktterme.

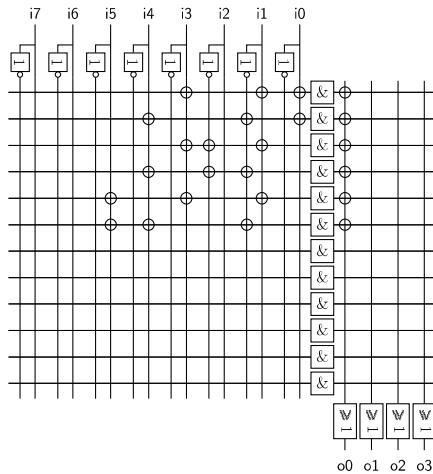


Abbildung 7.14: PLA-Beispiel

Ähnlich der Struktur von PLAs, PALs und PGAs ist die Struktur von programmierbaren Logik-Sequenzern. Flipflops können vom D- oder JK-Typ sein und auch Rückkopplungen sind möglich.

7.6 Programmierbares Gate Array (FPGA)

Benutzerprogrammierbare Gate Arrays bieten eine hohe Flexibilität, da auch Leitungssegmente programmiert werden können. Field Programmable Gate Arrays basieren auf dem Logikblock-Konzept. Ein Logikblock besteht aus einer kleinen Anzahl von Logikgattern, die flexibel programmierbar sind. Die Logikfunktion wird in Multiplexern

(Look-up tables, LUTs) oder RAM-Blöcken abgelegt, zusätzlich können noch Register existieren. Jedes FPGA enthält eine größere Anzahl von Logikblöcken (100 bis 500), jeder Block ist mit den anderen Blöcken und den I/O-Ports durch programmierbare Verknüpfung verbunden. Dadurch ist eine Verknüpfung der Logikblöcke untereinander sowie mit den I/O-Ports möglich. Es gibt verschiedene Programmierarten: EPROM/EEPROM, Schmelzsicherungen, SRAM.

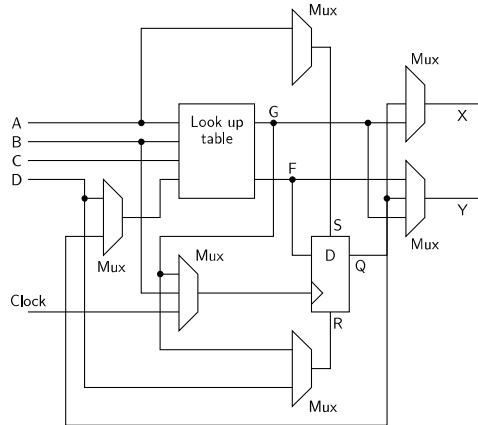


Abbildung 7.15: Struktur eines konfigurierbaren Logikblocks (Xilinx)

Die EPROM/EEPROM-Programmierung ist identisch mit der von PLDs, bei denen durch Floating-Gate-MOS-Transistor die zu programmierende Schalterverbindung gesteuert wird. Durch das Löschen des EPROM ist eine Wiederprogrammierung des FPGA's möglich.

Die Programmierung mittels Schmelzsicherungen ist ähnlich der Programmierung mit EPROMs, mit der Ausnahme, dass kein Löschen möglich ist. Zur Programmierung sind keine MOS-Transistoren erforderlich, so dass eine hohe Packungsdichte, geringe Laufzeiten und ein niedriger Preis erzielt werden können.

Bei der SRAM-Programmierung wird der programmierbare Schalter durch eine 1Bit-SRAM-Speicherzelle kontrolliert, bei Spannungsverlust verliert die SRAM-Zelle die Information und das FPGA seine Konfiguration. Zur Aufrechterhaltung der Konfiguration wird der Speicherinhalt in ein externes ROM/EPROM geladen, das dann mit einer speziellen Power-On-Routine das FPGA konfiguriert wird. Ein FPGA benötigt keine speziellen Programmiergeräte (nur für ROM/EPROM), allerdings Software zur Synthese. Die flexible Infrastruktur zur Programmierung verteuert das FPGA gegenüber dem PLD und die Software zur Synthese ist aufwendiger und damit teuer, außerdem sind die Verzögerungszeiten uneinheitlich und erst nach Plazierung und Verdrahtung bestimmbar.

Beispiel 81. Konfiguration einer NOT-Funktion

Der Logikblock hat 8 Eingänge und einen Ausgang, besteht aus drei 2:1 Multiplexern und einem OR-Gatter, die NOT-Funktion wird mit einer 1 am A und S_0 Eingang und einer 0 am B Eingang festgelegt, die Variable X erzeugt so am S_A Eingang die NOT-Funktion am Y-Ausgang.

Funktion	Gatterbedarf	benötigte Eingänge
NOT	9/14	4/8
2-AND	9/14	4/8
2-OR	14/14	6/8

Tabelle 7.1: Auslastungseffizienz einzelner Funktionen

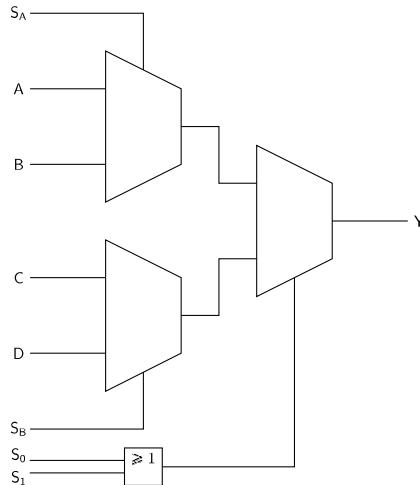


Abbildung 7.16: Struktur eines einfachen kombinatorischen Logikblocks

In einem Logikblock (XILINX CMOS LCA) besteht jede Zelle aus einem Schaltnetz und einem Speicherelement. Die Zellen können als jede Funktion von 4 Variablen oder als zwei Funktionen von 3 Variablen programmiert werden und werden als konfigurierbare Logikblöcke (CLB configurable logic block) bezeichnet.

Zellverknüpfung

Die Verknüpfung der Logikblöcke und E/A-Module geschieht über horizontal und vertikal verlaufende Verbindungen miteinander. Beim kleinsten Baustein (Actel A1010) mit 295 Logikblöcken erfolgt z.B. die Verknüpfung über 22 horizontale und 13 vertikale Spuren. Die Verbindung wird mittels Schmelzsicherungen/Transistoren hergestellt, die nach der Programmierung eine niederohmige (ca. 50Ω) Verbindung ergeben.

E/A Module

Die externe Schnittstelle wird von E/A-Modulen übernommen, E/A-Module lassen sich zu vier Konfigurationen programmieren: Eingang, Ausgang, Tri-state-Ausgang, bidirektionales Buffer.

Die LCA Architektur (Logic Cell Array) besteht aus einer Matrix identischer Logikzellen, die von Ein-/Ausgabezellen umgeben ist. Das Verbindungsnetzwerk besteht aus Leitungssegmenten, die horizontal und vertikal entlang in Kanälen zwischen den Zellen

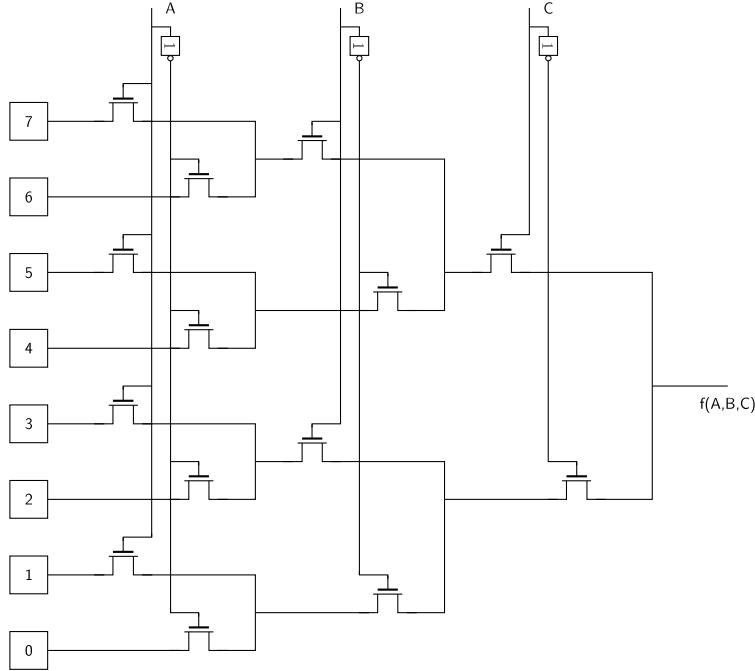


Abbildung 7.17: Look-up Funktionsgenerator mit 8 Speicherzellen und 8 zu 1 Multiplexer

verlaufen. Die Programmierung der Verdrahtung erfolgt durch die Schaltermatrix, die sich an den Kreuzungspunkten zwischen den Reihen und Spalten befindet.

Programmierung

Die Logikzellen, Peripheriezellen und die Verdrahtung sind vom Benutzer programmierbar. Die Information zur Konfiguration der Logik und Verdrahtung wird im Speicher gehalten. Die Programmierung des Speichers erfolgt mittels eines Entwurfssystems, das die Schaltungseingabe, Makrobibliothek, Simulation und Schaltungsumsetzung unterstützt. Die Plazierung und Verdrahtung erfolgt automatisch oder interaktiv manuell, dafür ist spezielle Entwurfsssoftware erforderlich, die den entsprechenden IC konfiguriert. Die Schaltungskonfiguration kann im EPROM/ROM auf der Platine oder im Rechner gespeichert werden.

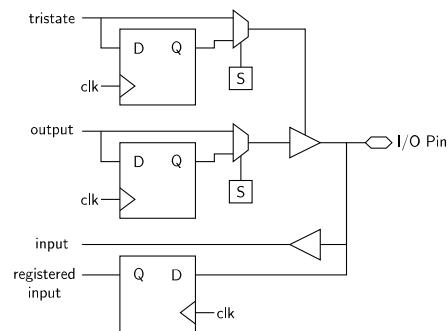


Abbildung 7.18: Prinzipieller Aufbau einer Ein-/Ausgabeezelle

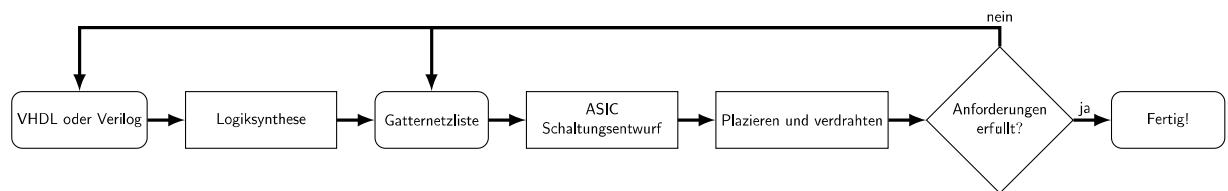


Abbildung 7.19: Programmierungs-Workflow