24 Petri-Netze

Der endliche Automat (EA) erlaubt es, rein sequentielle Prozesse als Schaltwerk in Hardware zu implementieren. Parallelität durch mehrere miteinander kommunizierende EAs führt aber schnell zu unüberschaubaren Strukturen. Bereits 1961 veröffentlichte Petri eine Promotion über ein Modell, das parallele Prozesse beschreiben konnte, das sog. *Petri-Netz* [5] (s. a. [4]). Es hat neben Kanten (Zweigen) zwei Arten von Knoten: Plätze und Übergänge, jedoch können mehrere Plätze gleichzeitig Einfluß auf bestimmte Ereignisse haben. Bild 24.1 zeigt die grundlegende Idee dieses Modells. Die Plätze P_i (*Places*) können entweder frei, also unbelegt, oder durch eine Marke (*Token*) belegt sein. Die Plätze sind durch Kanten durch Übergänge T_i (*Transitions*) verbunden und umgekehrt. Zunächst stellen diese Übergänge Barrieren für den Datenfluß dar, wobei Datenfluß eine Verschiebung der vorhandenen Token im Netz bedeutet.

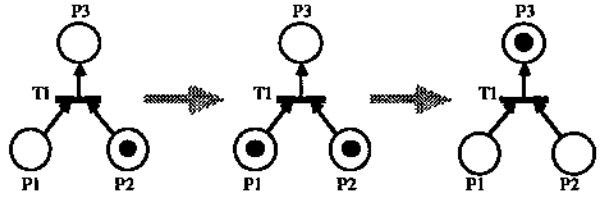


Bild 24.1: Zu den Grundlagen von Petri-Netzen.

Wenn jedoch alle Quellenplätze eines Überganges mit einem Token belegt sind und der Zielplätze des Überganges frei ist, so *zündet* der Übergang, d.h. alle Quelltoken werden gelöscht und der Zielplätze mit einem Token belegt. Wichtig ist für unsere folgenden Betrachtungen, daß der Zielplätze anschließend nur ein Token und nicht etwa drei enthält. Wir betrachten nämlich nur eine Untermenge der Petri-Netze, die sogenannten sicheren Petri-Netze, deren zwei Einschränkungen Fig. 24.2 zeigt. Jeder Platz kann im sicheren Netz niemals gleichzeitig Quelle und Ziel desselben Übergangs sein (Fig. 24.2 a) und - wie erwähnt - jeder Platz kann nur ein Token aufnehmen (Fig. 24.2 b), hat also nur die zwei möglichen Zustände *frei* und *belegt*..

24.1 VLSI Realisation von Petri-Netzen

1975 veröffentlichte Suhas Patil eine Arbeit über eine Hardware-Implementation von Petri-Netzen unter Benutzung der *Kolte-Array* -Methode [2]. Darauf basierend soll hier nun eine N-MOS VLSI-Schaltung für parallele Steuerwerke entstehen, die jedoch zusätzlich in der Lage ist, die bei parallelem Datenfluß entstehenden Zugriffskonflikte automatisch zu lösen. Diese Arbeit entstand 1986 an der Universität Kaiserslautern unter dem Namen *Patil Array* [1] [3] [6].

Kernstück der Realisierung einer solchen Struktur als nMOS-Schaltung ist die Beschreibung des Petri-Netzes durch eine Übergangsmatrix, in der für alle Übergänge Quell- und Zielplätze vermerkt sind. Fig. 24.4 zeigt ein Beispiel einer Beschreibung eines Petri-Netzes durch eine solche Matrix. Jede Reihe j der Matrix steht für einen Übergang T_j und jede Spalte i für einen Platz P_i. Ist der Platz P_i Ziel eines Überganges T_j, so wird an der so bezeichneten Stelle (P_i,T_j) ein "d" (*destination place*) eingetragen. Ein "s" (*source place*) an der Stelle (P_i,T_j)

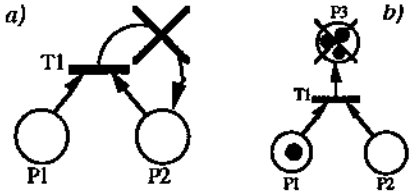


Fig. 24.2: Zur Definition sicherer Petri-Netze.

markiert P_i als Quellenplatz von T_j. Nicht vorhandene Verbindungen sind durch ("-") markiert. Da nur je eine der genannten Markierungen eingetragen wird, ist sichergestellt, daß ein Platz nicht gleichzeitig Quelle und Ziel eines Überganges sein kann.

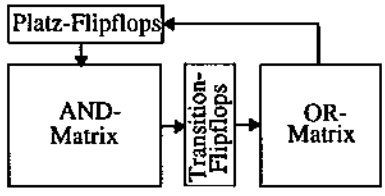
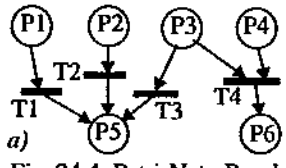


Fig. 24.3: Kolte-Array: Blockstruktur.

Durch die Übergangsmatrix werden zunächst nur die möglichen Übergänge beschrieben, eine konkrete Belegung des Netzes mit Token impliziert sie noch nicht. Die Funktion des Netzes beruht aber ja gerade auf dem "Bewegen" der Token durch Zündung von Übergängen. Nehmen wir also an, eine zusätzliche Markierung an jeder Spalte P_i zeigt, ob der entsprechende Platz belegt oder frei ist. Ein Markierung an jeder Reihe T_j soll den zugehörigen Übergang als "zündbar" ausweisen. Eine Tokenverschiebung im Netz verläuft dann in zwei Schritten:



b)

P1	P2	P3	P4	P5	P6	
s	-	-	-	d	-	T1
-	s	-	-	d	-	T2
-	-	s	-	d	-	T3
-	-	s	s	-	d	T4

Fig. 24.4: Petri-Netz-Beschreibung durch Übergangsmatrix.

Schritt 1: Markiere die Übergänge als 'zündbar', die folgende Bedingungen erfüllen: a) Alle Quellenplätze sind mit einem Token belegt, b) Der Zielplatz ist frei.

Schritt 2: Wechsle den Status aller Plätze, die mit einem als "zündbar" markierten Übergang verbunden sind: belegte Plätze werden frei und der freie Platz erhält einen Token.

Die beiden Schritte lassen sich auch als logische Funktion darstellen. Am Beispiel aus Fig. 24.4 bedeutet dies dann beispielsweise für Übergang T4 und Platz P5:

Schritt 1: T4 ist zündbar IF (P3 belegt) AND (P4 belegt) AND (P6 frei)

Schritt 2: P5 wechselt Status IF (T1 zündbar) OR (T2 zündbar) OR (T3 zündbar)

a)

b)

P1	P2	P3	P4	
s			d	T1
	s	s	d	T2

c)

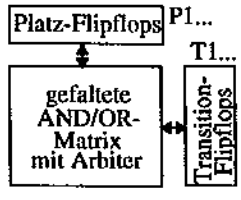


Fig. 24.5: Petri-Netz als Patil-Array: a-b) Beispiel, c) Blockdiagramm.

Analog verlaufen die Schritte dann jeweils für alle Plätze und Übergänge.

24.2 Grundlegendes Hardwarekonzept

Aus den vorangegangenen Überlegungen läßt sich nun direkt ein Hardwarekonzept ableiten, das Fig. 24.3 als Blockstruktur des Kolte-Arrays zeigt. Die Zustände der Plätze und Übergänge werden durch Flip-Flops festgehalten. Die Plätze werden über eine UND-Matrix mit

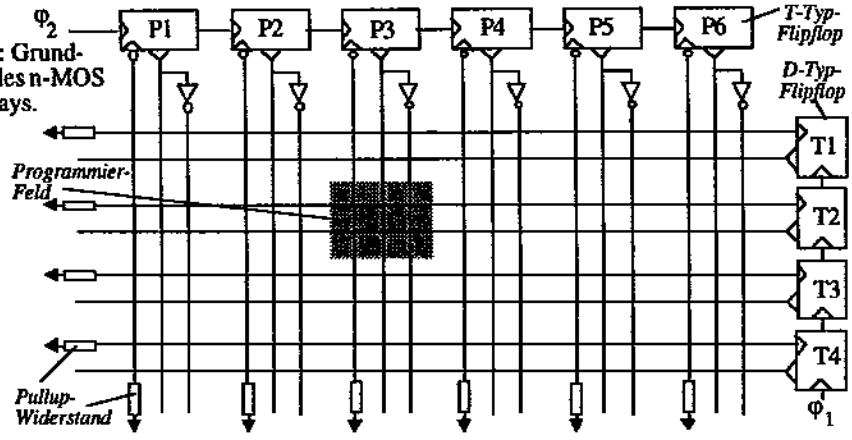


Fig. 24.6: Grundstruktur des n-MOS Patil-Arrays.

den Ubergangs-Flip-Flops verbunden, so daß in dieser Matrix festgelegt werden kann, welche Konjunktion von belegten Plätze zum Zünden der Übergänge in Schritt 1 erforderlich ist. Der Zustand der Übergänge wiederum bestimmt das Wechseln der Platzzustände. Dies war - wie in Schritt 2 gezeigt - für jeden Übergang eine Disjunktion der korrespondierenden zündbaren Übergänge und wird somit als ODER-Matrix realisiert. Um eine Analogie zw. Übergangsmatrix und Schaltung zu schaffen, werden wir die ODER-Matrix in die UND-Matrix hereinfalten. Dies spart Chipfläche und vereinfacht die Übertragung der Matrixbelegung zur Hardware, da - wie wir später sehen werden - die Einträge "s" oder "d" nur verschiedene Transistorplazierungen an der korrespondierenden Stelle der Schaltung erfordern.

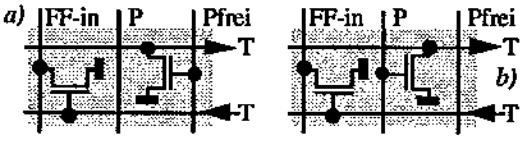


Fig. 24.7: Programmierte Zellen; a) "s" (source), b) "d" (destination).

Fig. 24.5 zeigt die Entstehung der endgültigen Struktur des Patil-Arrays. Aus dem Petri-Netz wird zunächst die Übergangsmatrix abgeleitet. Für jeden Platz besitzt die Schaltung ein Wechsel-Flip-Flop als Zustandsregister, das bei jedem Anlegen eines Steuersignals zwischen den beiden Zuständen "belegt/frei" hin- und herwechselt, so wie es immer in Schritt 2 erforderlich ist, wenn ein anliegender Übergang zündbar ist. Die Übergangsmarkierungen werden durch D-Flip-Flops realisiert, die man gezielt setzen oder löschen kann, um so Schritt 1 zu realisieren. Die Übergangsmatrix selbst ist in der gefalteten UND-/ODER-Matrix vorgegeben. Auf die Konfliktlösungslogik gehen wir später noch ein.

Fig. 24.6 zeigt die Grundstruktur des nMOS-Patil-Arrays. In diesem Bild ist noch keine einzige Verbindung zwischen den Plätzen und Übergängen hergestellt. Jeder Schnittpunkt zwischen den Flip-Flop-Leitungen - in Fig. 24.6 grau unterlegt - stellt jedoch eine "programmierbare" Zelle dar, in der Transistoren wie in Fig. 24.7 plaziert werden, um dort eine "s"- oder "d"-Markierung vorzunehmen. Die Platz-Flipflops wechseln ihren Status, wenn die invertierte Eingangsleitung "FF-in" auf *Low* geht. Da diese über einen *Pull-Up* Widerstand normalerweise auf *High* gehalten wird, passiert dieses erst, wenn mindestens ein anliegender Übergang T_j - gleich-

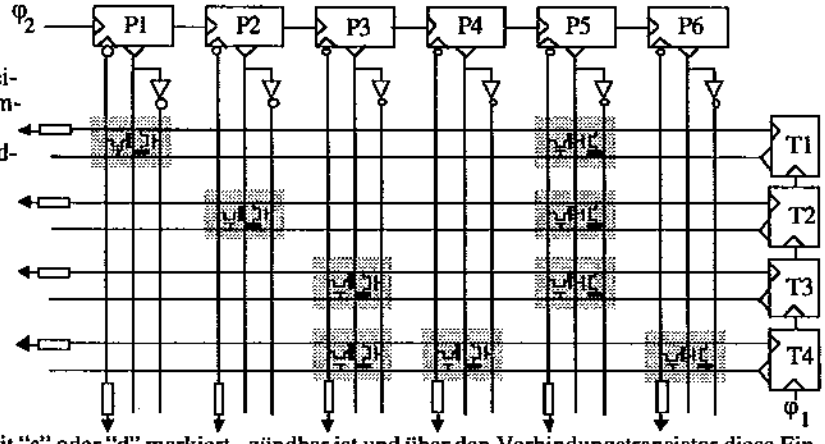


Fig. 24.8: Beispiel einer programmierten Patil-Grundstruktur.

gültig ob mit "s" oder "d" markiert - zündbar ist und über den Verbindungstransistor diese Eingangsleitung auf *Low* zieht.

Die Übergangs-Flip-Flops T_j werden nach einer NOR-Umformung der Funktion aus Schritt 1 beschaltet, d.h. ein *Pull-Up* Widerstand zieht die Eingangsleitung der D-Flip-Flops auf *High*, sofern keiner der Transistoren der programmierten Zellen diese auf *Low* hält.Für jeden Quellenplatz eines Übergangs plazieren wir also einen Transistor am Schnittpunkt der Eingangsleitung mit dem invertierten Ausgang (Pfrei) des Quellenplatz-Flip-Flops. Jeder Zielplatz wird mit einem Transistor am Schnittpunkt P_i-T_j versehen. Die Eingangsleitung eines Übergangs-Flip-Flops geht also nur dann auf *High* (d.h. der Übergang wird als zündbar markiert), wenn alle Quellenplätze mit Token belegt sind (also alle $Pfrei_s$=*Low*) und der Zielplatz frei ist (also P_d=*Low*). Ansonsten wird das Übergangs-Flip-Flop gelöscht, der Übergang ist nicht zündbar.

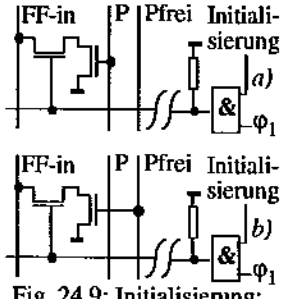


Fig. 24.9: Initialisierung; a) Kein Token, b) Token.

Die Flipflops werden in nicht-überlappenden zwei Phasen ϕ_1 und ϕ_2 getaktet. Mit ϕ_1 werden alle Übergänge in Abhängigkeit der Platzzustände als zündbar oder nicht markiert und mit ϕ_2 werden die Zustände aller mit zündbaren Übergängen verbundenen Plätze gewechselt. Fig. 24.8 zeigt die komplette Schaltung für das Petri-Netz in Fig. 24.4. Diese Schaltung repräsentiert nun zwar das Netzmodell, allerdings haben wir so noch keine Möglichkeit, eine Startbelegung mit Token vorzunehmen. Wir benötigen also noch eine Initialisierungsschaltung, mit der wir den Zustand der Platz-Flip-Flops beim Start vorbestimmen können. Zu diesem Zweck wird eine weitere Reihe hinzugefügt, die zum Zeitpunkt f_1 die Platzzustände auf "Token" oder "kein Token" vorbelegt.

Fig. 24.9 zeigt die zwei verschiedenen Beschaltungen der Platz-Flip-Flops, die zur Initialisierung der Platz als unbelegt oder belegt erforderlich sind. Soll der Platz P_i unbelegt sein, so wird der Zustand gewechselt, falls P_i belegt ist, d.h. die entsprechende Flip-Flop Eingangsleitung "FF-in" wird genau dann auf *Low* gezogen, wenn ein Initialisierungssignal zum Zeitpunkt

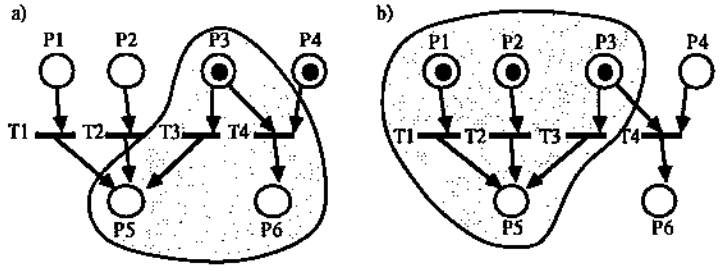


Fig. 24.10: Zugriffskonflikte: a) Ein Platz für mehrere Übergänge (SSMD), b) Mehrere Übergänge für einen Platz (MSSD).

f_1 vorliegt und P=*High* ist. Entsprechend soll der Zustand gewechselt werden, falls P_i ein Token erhalten soll, aber zum Initialisierungszeitpunkt Pfrei=*High* gilt.

24.3 Übergangsprioritäten

In einem Petri-Netz kann es aufgrund der parallelen Aktionen zu zwei Konflikten kommen, die in der bisher entwickelten Schaltung nicht berücksichtigt wurden. Das erste Problem taucht auf, wenn mehrere Übergänge sich einen gemeinsamen Quellenplatz teilen (*Single Source, Multiple Destination - SSMD*). Fig. 24.10 a verdeutlicht dieses Problem: P3 ist *single source* und P5 mit P6 sind *multiple destination*. Sowohl T3 als auch T4 könnten zünden, allerdings kann das Token von P3 nur für eine Zündung verwendet werden, so daß eine Entscheidung getroffen werden muß, welcher Übergang nun zündet.

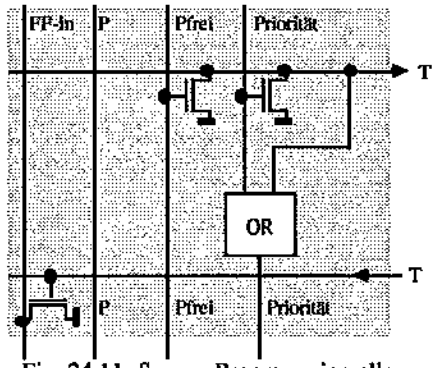


Fig. 24.11: *Source*-Programmierzelle mit Prioritätssignalleitung.

Die zweite Problem-Art entsteht, wenn mehrere zündbare Übergänge die gleichen Ziel-Plätze haben, der allerdings nur ein Token aufnehmen kann (*Multiple Source, Single Destination - MSSD*). In Fig. 24.10 b beispielsweise sind die Übergänge T1, T2 und T3 zündbar, aber der Platz P5 kann nur das Token von einem dieser Übergänge annehmen. P1, P2 und P3 sind *multiple source*, hingegen P5 ist *single destination*. Auch hier ist also eine Entscheidung erforderlich, welcher Übergang zum Zuge kommt.

Die Schaltung soll nun um eine Konfliktlösungslogik erweitert werden, die den Übergängen verschiedene Prioritäten zuordnet. Von mehreren aufgrund desselben Quellenplatzes zündbaren Übergängen zündet dann letztendlich nur der mit der höchsten Priorität. In dem obigen Beispiel würden die Token von P3 und P4 über T4 zu P6 übergehen. Wäre P4 unbelegt, so wie in Fig. 24.10 b, dann hätte T3 die nächsthöhere Priorität und das Token von P3 würde T3 zünden und dann in den Platz P5 wechseln.

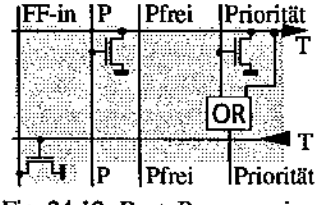


Fig. 24.12: *Dest.*-Programmierzelle mit Prioritätssignalleitung.

Eine Möglichkeit, solche Prioritäten zu berücksichtigen, ist eine Prioritätssignalleitung, die parallel zu den Platzsignalleitungen verläuft, so wie es Fig. 24.11 zeigt. Die Priorität

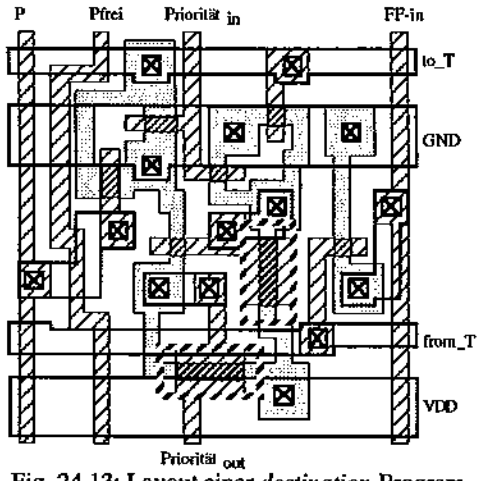


Fig. 24.13: Layout einer *destination*-Programmierzelle für einen Patil-Array-Generator.

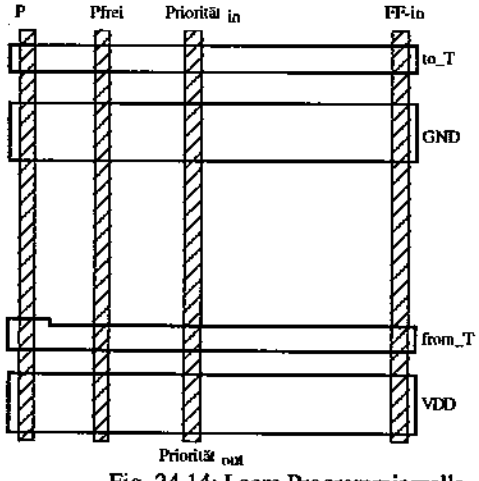


Fig. 24.14: Leere Programmierzelle.

der Übergänge nimmt von oben nach unten ab. Die Eingangsleitung T der Übergangs-Flip-Flops kann nur auf *High* gehen, wenn die Prioritätsleitung *Low* ist. Dies ist in der obersten Reihe zunächst immer der Fall, daher ist dort die höchste Priorität. Wird ein Übergang als "zündbar" markiert, so geht dieses Signal über das ODER-Gatter in den *source*-Programmierzellen auf die Prioritätsleitung und blockiert alle darunter liegenden Übergänge. Die Prioritätsleitung wird in der *destination*-Program-mierzelle analog zur *source*-Programmierzelle beschaltet (s. Fig. 24.12).

Damit sind die notwendigen Grundlagen geschaffen, um das Layout für die verschiedenen Programmierzellen zu entwerfen. Achtet man dabei darauf, daß alle Programmierzellen die gleiche Fläche benötigen und die Anschlüsse für wiring-by-abutment lagerichtig angeordnet sind, so kann man einen Patil-Array-Generator bauen. Analog zu PLA-Generatoren erzeugt dieser das Layout zu einem Patil-Array aus einer Personality-Matrix, die angibt, an welchen Stellen *source-*, *destination-* oder leere Zellen zu plazieren sind. Fig. 24.15 zeigt das NMOS-Layout einer *source*-Programmierzelle, deren Außenanschlüsse für wiring-by-abutment zu benachbarten Programmierzellen ausgelegt sind.

Um das Design kompakt zu halten, wurde das Layout der *destination*-Programmierzelle in Fig. 24.13 von Grund auf neu gemacht, so daß sich größere Unterschiede ergeben, als man vom Transistorschaltbild vermuten könnte.

Der Nachteil eines solchen Layout-Generators sind die hohen Kosten für die Fertigung eines ganzen ICs für jede neue Personality-Matrix. Eine erste Verbesserung wäre ein Layout für ein maskenprogrammierbares Patil-Array, bei dem die Personalisierung nur noch auf der obersten Metallebene und den entsprechenden Kontaktierungen erfolgt. Dies würde zwar eine größere Fläche für die einzelnen Programmierzellen bedeuten, da in jeder Programmierzelle alle Transistoren schon vorgesehen sein müßten, die dann durch Kontaktierungen zu VDD oder dem entsprechenden Signal entweder abgeschaltet oder genutzt werden. Der Vorteil läge in den ge-

ringeren Stückkosten, da auf teilgefertigten Wafers nur noch die Metallisierung und Kontaktierung als Prozeßschritte durchzuführen wären. Die flexibelste Lösung, verschiedene Petri-Netze in Patil-Arrays zu implementieren wird im nächsten Abschnitt dargestellt.

24.4 Frei programmierbares Patil-Array

Obwohl nun das Problem der Zugriffskonflikte gelöst ist, bleibt die Flexibilität des Patil-Arrays durch das feste Plazieren von "s"- und "d"-Zellen doch beschränkt. Eine flexiblere Lösung stellt das vom Benutzer frei programmierbare Patil-Array dar, wie es Fig. 24.16 am Beispiel einer Programmierzelle zeigt. Statt unterschiedlichen, fest verdrahteten "s"- und "d"-Zellen und frei bleibenden restlichen Matrixplätzen gibt es nun eine Zelle, aus der die gesamte Matrix gebildet wird. Während einer Initialisierungsphase kann jede einzelne dieser Zellen als *source-*, *destination-* oder neutrale Zelle definiert werden, so daß der Benutzer im Rahmen der Array-Kapazität beliebige Petri-Netze so oft er möchte modellieren kann.

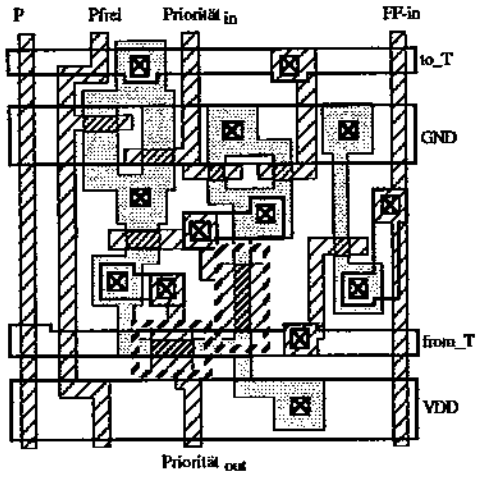


Fig. 24.15: Layout einer *source*-Programmierzelle für einen Patil-Array-Generator.

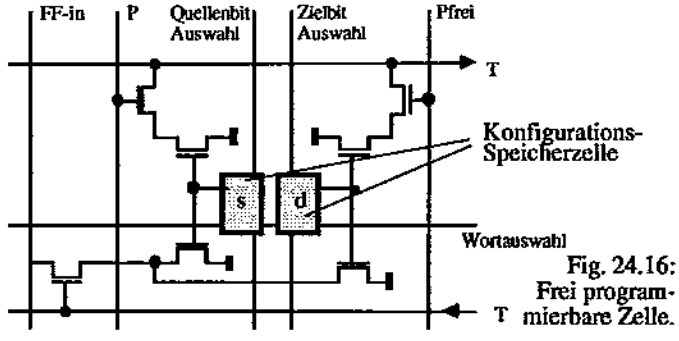


Fig. 24.16: Frei programmierbare Zelle.

Während der Konfigurationsphase wird zunächst über die Wortauswahlleitung die gewünschte Zelle ausgewählt. Je nachdem ob nun eine "1" in der "s"- oder "d"-Konfigurationsspeicherzelle gespeichert wird, verhält sich diese Zelle wie eine "s"- oder "d"-Zelle. Einer "0" in beiden Speicherzellen entspricht dann ein unbelegter Platz in der Übergangstabelle. Eine "1" in beiden Zellen ist nicht erlaubt, da dann eine Zelle gleichzeitig *source* und *destination* zu einer Transition wäre. Um Fehler zu vermeiden, sollte die Programmiersoftware diesen Fall entweder abfangen oder gleich unmöglich machen ihn einzugeben.

In Fig. 24.16 wurde die Prioritätssignalleitung zum einfachen Verständnis des Prinzips weggelassen.Eine vollständige frei programmierbare Patil-Array-Zelle zeigt Fig. 24.17. Die Prioritätsleitung zur nächstunteren Zelle berechnet sich dabei nach folgender Logikgleichung:

$$Prioritaet_{out} = Prioritaet_{in} + S \cdot T + D \cdot T$$

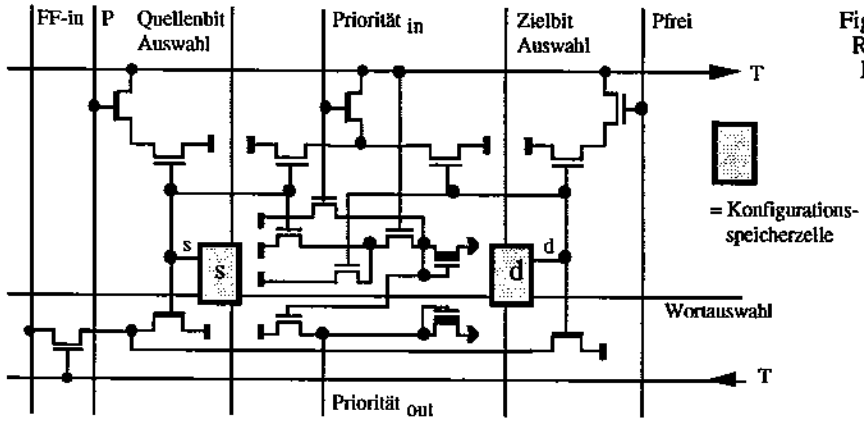


Fig. 24.17: RAM mit Priorität.

Sobald die Prioritätsleitung gesetzt ist, können diese und nachfolgende Transitionen nicht zünden. Also muß eine gesetzte Prioritätsleitung weitergereicht werden oder eine ungesetzte Prioritätsleitung gesetzt werden, falls an einer *source*-Zelle oder einer *destination*-Zelle die Transition zündete. Dies entspricht genau der oben angegebenen Logikgleichung.

24.5 Abschließende Betrachtung

Durch die Hardware-Implementation von Petri-Netzen als Patil-Array erschließt sich der VLSI-Designer die einfache Modellierung paralleler Steuerwerksprozesse mit selbständiger Zugriffskonfliktlösung, deren Realisation durch parallel arbeitende EA's nicht praktikabel wäre. Für die Massenproduktion kundenspezifischer Patil-Chips steht ihm ein einfaches Grundmuster eines Patil-Arrays zu Verfügung, das er durch Bestücken mit unterschiedlichen Zellen aus einer Bibliothek fest verdrahten kann. Ein beliebig oft frei programmierbares Array bietet weitere Flexibilität, falls dieses erforderlich ist, wie etwa in Forschung und Entwicklung. Für beide Verfahren wurde in Kaiserslautern das Entwurfs- und Simulationswerkzeug *PADS* entwickelt.

24.6 Literatur

[1] R. W. Hartenstein, A. Hirschbiel, M. Weber: Patil Array; Report, CS Department, Univ. of Kaiserslautern, 1987

[2] R. Hartenstein, A. Hirschbiel, M. Weber: Patil Array: a self-prioritizing array to implement Petri nets; Int'l EUROMICRO Symposium; Brussels, Belgium, 1985.

[3] S. Patil: Micro-Control for Parallel Asynchronous Computers; (R. Hartenstein, R. Zaks:) Microarchitecture of Computer Systems; North-Holland, Amsterdam / New York 1975

[4] J. L. Peterson: Petri Nets: ACM Computing Surveys, Vol 9, No. 3, September 1977

[5] W. Reisig: Petri-Netze: Eine Einführung; 2. Auflage; Springer-Verlag; Berlin; 1986

[6] K. Schmidt: An NMOS Implementation of a Patil-Array; Projektarbeit; Universität Kaiserslautern, Fachbereich Informatik, 1987