mit dem direkten und dem verzögerten negierten Takt als Eingabesignale. Bei einer steigenden Taktflanke sind beide Eingänge des UND-Gatters für eine kurze Zeit t_{d1} gleichzeitig »1«, so dass das nachfolgende UND-Gatter den gewünschten Impuls erzeugt. Für $T = 1$, $T = 0$ und auch nach der fallenden Taktflanke bleibt das Freigabesignal inaktiv.

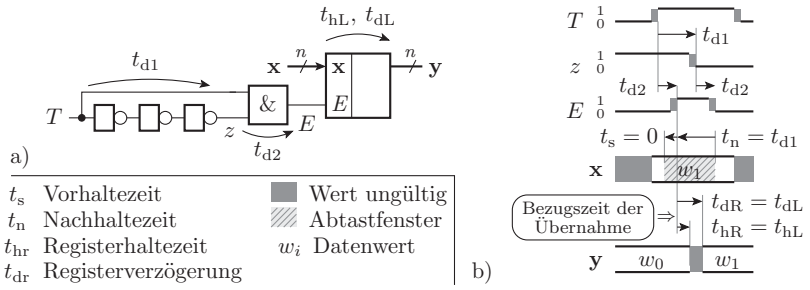


Abb. 4.48. Gepulstes Latch a) Schaltung b) Zeitverhalten

Eine korrekte Datenübernahme ohne Glitches am Latch-Ausgang setzt voraus, dass das Eingabesignal während der gesamten Dauer, die das Freigabesignal aktiv ist, d.h. im Zeitfenster t_{d2} bis $t_{\mathrm{d1}} + t_{\mathrm{d2}}$, stabil und gültig ist. Zur Beschreibung der Gesamtschaltung mit dem Registermodell aus Abschnitt 1.4.1 ist es notwendig, den Bezugszeitpunkt für die Datenübernahme auf die um t_{d2} verzögerte steigende Flanke des Übernahmeimpulses zu verschieben. Das äquivalente Registermodell hat dann eine Vorhaltezeit $t_{\mathrm{s}} = 0$ und eine Nachhaltezeit $t_{\mathrm{n}} = t_{\mathrm{d1}}$. Die Registerverzögerung ist gleich der Latch-Verzögerung und die Registerhaltezeit gleich der Latch-Haltezeit.

Die Registernachbildung mit einem gepulsten Latch hat schaltungsbedingt eine Nachhaltezeit größer null. Das hat für die Register-Transfer-Funktionen zwischen den Registern den Nachteil, dass eine Mindesthaltezeit in der Größenordnung der Nachhaltezeit zu fordern ist (Abb. 4.49). Die Synthese unterstützt die Zusicherung von Mindesthaltezeiten normalerweise nicht, so dass

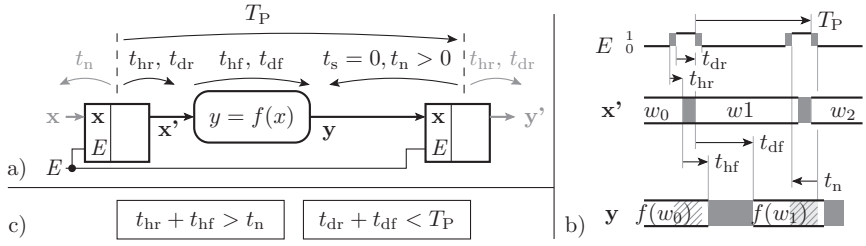


Abb. 4.49. Register-Transfer-Funktion mit gepulsten Latches a) Schaltung b) Signalverläufe c) zuzusichernde Zeitbedingungen

manuelle Nacharbeit bei der Festlegung der Schaltungsstruktur und der Leitungsführung erforderlich werden kann.

4.3.5 Register aus Master-Slave-Flipflops

Ein Master-Slave-Flipflop ist eine taktflankengesteuerte Speicherzelle, die aus zwei taktzustandsgesteuerten D-Flipflops zusammengesetzt ist. Das erste D-Flipflop – der Master – übernimmt die Daten vor der Übernahmeflanke und speichert sie in der Takthälfte nach der Übernahmeflanke. Das zweite D-Flipflop – der Slave – übernimmt den zwischengespeicherten und damit stabilen Wert aus dem Master in der zweiten Takthälfte (Abb. 4.50). Das Eingabesignal des gesamten Master-Slave-Flipflops, das über den Master und den Slave zum Ausgang weitergeleitet wird, muss nur während eines kleinen Zeitfensters vor der Übernahmeflanke einen gültigen und stabilen Wert haben. Die Nachhaltezeit ist null. Die Gesamtschaltung ist zwar deutlich aufwändiger als ein funktionsgleiches gepulstes D-Flipflop. Dafür sind Master-Slave-Flipflops laufzeitunkritisch und Schaltungen mit ihnen einfacher zu entwerfen.

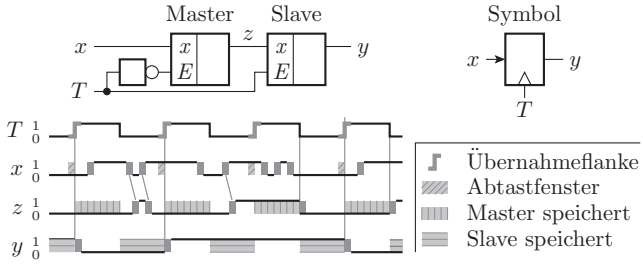


Abb. 4.50. Master-Slave-Flipflop

Ein Register fasst mehrere Master-Slave-Flipflops mit einem gemeinsamen Takteingang zusammen. Die einzelnen Speicherzellen sind räumlich voneinander getrennt, so dass die aktiven Taktflanken sie mit einem geringen Taktversatz erreichen (vgl. Abschnitt 1.4.5). Die Eingabedaten müssen um die Vorhaltezeit vor dem frühestmöglichen Zeitpunkt, zu dem die aktive Taktflanke eine Registerzelle erreichen kann, bis zur Nachhaltezeit nach dem spätestmöglichen Zeitpunkt, zu dem die aktive Taktflanke eine Registerzelle erreichen kann, stabil anliegen. Die Registerausgabe ändert sich frühstens nach der Zellenhaltezeit nach dem frühestmöglichen Zeitpunkt der Taktflanke. Der neue Wert ist spätestens nach der Zellenverzögerungszeit nach dem spätestmöglichen Zeitpunkt der Taktflanke am Registerausgang verfügbar (Abb. 4.51).

Bei einem Taktversatz ist irgendein Zeitpunkt innerhalb des Intervalls, in dem die aktive Taktflanke die Zellen erreichen kann, als Bezugszeitpunkt zu definieren. Mit dem frühestmöglichen Zeitpunkt als Bezugszeitpunkt sind die Vorhaltezeit und die Haltezeit des Registers gleich der Vorhaltezeit und der

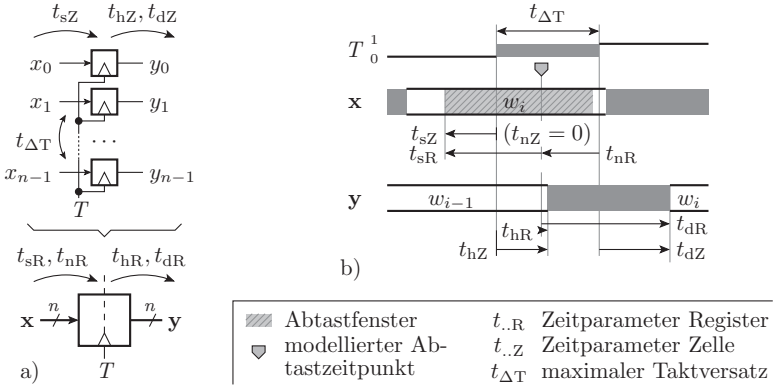


Abb. 4.51. Register mit Taktversatz a) Schaltung b) Zeitverläufe

Haltezeit der einzelnen Speicherzellen. Die Nachhaltezeit ist gleich dem maximalen Taktversatz[10] und die maximale Verzögerungszeit gleich der Summe aus dem Taktversatz und der Verzögerungszeit einer Speicherzelle:

$$\begin{aligned} t_{sR} &= t_{sZ} \qquad t_{hR} = t_{hZ} \\ t_{nR} &= t_{\Delta T} \qquad t_{dR} = t_{\Delta T} + t_{dZ} \end{aligned} \tag{4.34}$$

(t_{sR} – Vorhaltezeit des Registers; t_{sZ} – Vorhaltezeit einer Speicherzelle; t_{hR} – Haltezeit des Registers; t_{hZ} – Haltezeit einer Speicherzelle; t_{nR} – Nachhaltezeit des Registers; $t_{\Delta T}$ – maximaler Taktversatz; t_{dR} – Verzögerungszeit des Registers; t_{dZ} – Verzögerungszeit einer Speicherzelle).

Eine Nachhaltezeit ungleich null lässt sich schlecht simulieren (vgl. Abb. 1.47 in Abschnitt 1.4.2). Mit einem Taktversatz kleiner der Haltezeit lässt sich der Bezugszeitpunkt um den Taktversatz nach hinten verschieben, ohne dass die Haltezeit einen nicht simulierbaren negativen Wert annimmt. Die Vorhaltezeit vergrößert sich, die Halte- und die Verzögerungszeit verringern sich gegenüber Gleichung 4.34 um den maximalen Taktversatz:

$$\begin{aligned} t_{sR} &= t_{sZ} + t_{\Delta T} \qquad t_{hR} = t_{hZ} - t_{\Delta T} \\ t_{nR} &= 0 \qquad\qquad\quad t_{dR} = t_{dZ} \end{aligned} \tag{4.35}$$

Die Nachhaltezeit ist null. Das ist das bevorzugte Verhaltensmodell für ein Register.

Freigabe- und Initialisierungseingang

Abbildung 4.52 zeigt die Synthesebeschreibung für ein Register mit einem Initialisierungs- und einem Freigabeeingang (vgl. Abschnitt 2.1.3). Beide Zu-

[10] plus der Zellennachhaltezeit, die für Master-Slave-Flipflops null ist

satzfunktionen werden oft benötigt. Das rechtfertigt eine genauere schaltungstechnische Betrachtung. Die zustandsgesteuerte Übernahme des Anfangswertes wird bei einem Register aus Master-Slave-Flipflops mit initialisierbaren Slaves nachgebildet. Die Setz- und Rücksetzeingänge der Slave-Flipflops haben Vorrang vor der Übernahme vom Dateneingang (vgl. Abschnitt 4.3.3, Abb. 4.46), so dass die Initialisierung sofort und unabhängig vom Takt, d.h. asynchron, erfolgt. Die zusätzliche Freigabesteuerung kann unterschiedlich realisiert werden. In Abb. 4.52 b wird die bedingte Übernahme mit einem Multiplexer nachgebildet, der wahlweise den Ist-Zustand oder das abzutastende Signal an den Registereingang weiterleitet. In Abb. 4.52 c blockiert das inaktive Freigabesignal des Registers die Datenweitergabe vom Master zum Slave. Das Freigabesignal E^* muss während der gesamten Slave-Übernahmephase stabil und gültig sein. Dazu wird es in Abb. 4.52 c in dieser Taktphase in einem zusätzlichen D-Flipflop gespeichert. Das zusätzliche D-Flipflop wird für das gesamte Register nur einmal benötigt, während in Abb. 4.52 b jedes Bit seinen eigenen Multiplexer haben muss.

```
signal x, y: STD_LOGIC_VECTOR(n-1 downto 0);
signal E, I, T: STD_LOGIC;
...
process(I, T):
begin
 if I='1' then
   y <= aw;
 elsif E='1' and RISING_EDGE(T) then
   y <= x;
 end if;
end process;
```

a)

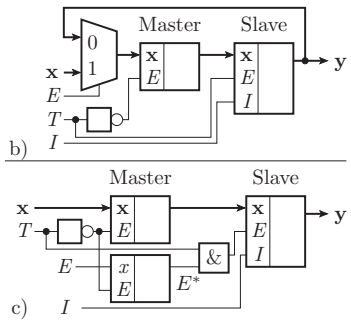


Abb. 4.52. Register mit Freigabe- und Initialisierungseingang a) Synthese-Beschreibung b) Schaltung mit Eingabemultiplexer c) Schaltung mit Übernahmeblockierung für den Slave

4.3.6 Taktversorgung

Taktsignale legen die Zeitpunkte und Zeitfenster für die Datenübernahme in die Register und Latches fest. Sie müssen zeitgenau und Glitch-frei sein. In einer synchronen Schaltung werden alle Taktsignale von einem Haupttakt abgeleitet, der mit einem frequenzstabilen Oszillator erzeugt wird.

Oszillator

Ein Oszillator ist eine Schaltung, die ein periodisches Signal bereitstellt. Prinzipiell ist diese Funktion mit dem Ringinverter aus Abschnitt 4.2.3 realisierbar. Aber die Periodendauer des von einem Ringinverter erzeugten Signals ist

meist zu ungenau. Zeitgenaue Oszillatoren verwenden einen Quarz oder einen anderen mechanischen Schwinger als frequenzbestimmendes Element. Abbildung 4.53 a zeigt eine gebräuchliche Oszillatorschaltung mit einem Quarz. Der Quarz verhält sich elektrisch wie ein Schwingkreis. Er bildet die Rückkopplung des ersten Inverters, der hier als Verstärker arbeitet. Die beiden Kapazitäten dienen zur Phasenverschiebung. Der zweite Inverter verbessert die Signalform. Die relative Abweichung zwischen der Ist- und der Soll-Frequenz eines solchen Oszillators liegt in der Größenordnung von 10^{-5} bis 10^{-4}. Das ist für den Takt einer digitalen Schaltung ausreichend genau.

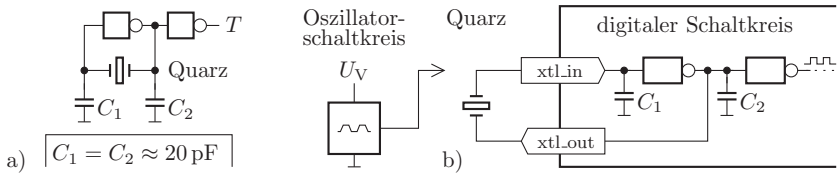


Abb. 4.53. Quarzstabilisierter Taktgenerator a) Schaltung b) Schaltkreis mit Quarz- oder Oszillatoranschlüssen

Quarze und andere mechanische Schwinger gibt es als Einzelbauteile oder komplett mit Beschaltung als Oszillatorschaltkreise. Hochintegrierte Schaltkreise, die einen frequenzstabilen Takt benötigen, haben meist die Inverter für den Oszillator mit auf dem Chip, so dass extern nur noch der mechanische Schwinger angeschlossen werden muss. Alternativ kann an dem Schaltkreisanschluss, der auf den Eingang des ersten Inverters führt, auch ein externes Taktsignal von einem Oszillatorschaltkreis oder einer anderen Taktquelle eingespeist werden (Abb. 4.53 b).

Taktnetze

Taktsignale müssen die Takteingänge von Hunderten oder Tausenden von angeschlossenen Speicherzellen treiben. Die große Anzahl von Lasten und der geringe zulässige Taktversatz erfordern spezielle Treiber und spezielle Leitungsführungen (Abb. 4.54). In programmierbaren Logikschaltkreisen gibt es spezielle Taktverteilernetze. In der Spartan- und Virtex-Familie erfolgt die Einspeisung von Taktsignalen in diese Netze mit BUFG-Treibern. Der Entwurf der Taktnetze erfordert einen weit tieferen Einblick in das elektrische Verhalten der Gatter und Leitungen als alle anderen Entwurfsaufgaben in der Digitaltechnik zusammen.

Taktteiler

Takte mit einer geringeren Frequenz werden mit Taktteilern erzeugt. Ein Taktteiler ist ein autonomer Automat mit dem Ausgabetakt als Zustandssignal

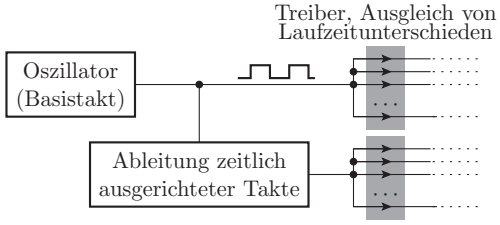
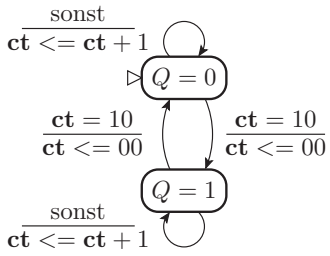


Abb. 4.54. Schaltung zur Taktversorgung

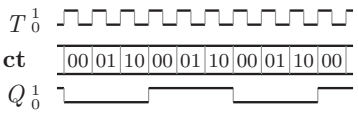
und einem Zähler. In jedem Zustand wird nach einer bestimmten Anzahl von Zählschritten der Zustand gewechselt und der Zähler zurückgesetzt. Bei einem Tastverhältnis von eins zu eins genügt eine Übergangsfunktion, die das Zustandssignal immer nach der entsprechenden Taktanzahl invertiert. Abbildung 4.55 zeigt die VHDL-Beschreibung, den Operationsablaufgraph und die Signalverläufe für einen Taktteiler mit dem Teilerverhältnis eins zu sechs. Die Abtastung, die eine Definition des Taktes als Zustandssignal einschließt, ist notwendig, um Glitches auf dem Ausgabetakt auszuschließen, und minimiert gleichzeitig den Taktversatz zum Basistakt.

a)
```
signal ct: tUnsigned(1 downto 0);
signal I, T, Q: STD_LOGIC;
...
process(T, I)
begin
 if I='1' then
  Q <= '0';
  ct <= "00";
 elsif RISING_EDGE(T) then
  if ct="10" then
   ct <= "00";
   Q <= not Q;
  else
   ct <= ct + "1";
 end if;
end if;
end process;
```

b)



c)



⇒Web-Projekt: P4.3/Taktteiler.vhdl

Abb. 4.55. Taktteiler a) VHDL-Beschreibung b) Operationsablaufgraph c) Signalverläufe

Phasenregelkreise (PLL)

Komplexe Taktversorgungsschaltungen benutzen Phasenregelkreise (PLL – **p**hase **l**ocked **l**oop). Ein Phasenregelkreis besteht aus einer Quelle für den Referenztakt, einem spannungsgesteuerten Oszillator (VCO – **v**oltage **c**ontrolled **o**scillator), einem Phasenvergleicher, einem Integrator und optional zwei Frequenzteilern (Abb. 4.56). Der Referenztakt T_{Ref} muss eine konstante Frequenz haben. Bei einem spannungsgesteuerten Oszillator lässt sich die Frequenz über das Steuerpotenzial φ_{Ctl} einstellen. Der Phasenkomparator vergleicht die aktiven Flanken der beiden Takte und steuert einen Integrator an. Gemeinsam mit dem Integrator erhöht bzw. verringert der Phasenkomparator das Steuerpotenzial φ_{Ctl} so, dass sich nach einer gewissen Zeit die Flanken der heruntergeteilten Takte T_1und T_2 zeitlich angleichen. Man spricht dann vom Einrasten des Regelkreises, der von dem Phasenkomparator mit einem Ready-Signal quittiert wird. Im eingerasteten Zustand erzeugt der spannungsgesteuerte Oszillator einen Takt T_{VCO} mit einer Frequenz von exakt

$$f_{\mathrm{VCO}} = \frac{Q}{P} \cdot f_{\mathrm{Ref}} \tag{4.36}$$

(P – Teilerwert für den Referenztakt; Q – Teilerwert für den VCO-Takt; f_{Ref} – Frequenz des Referenztaktes). Schaltungen mit Phasenregelkreisen in der Taktversorgung sind erst betriebsbereit, wenn der Regelkreis eingerastet ist. Nach dem Zuschalten der Versorgungsspannung dauert das einige Hundert Takte. Erst dann kann der Rest der Schaltung initialisiert werden und seinen normalen Betrieb aufnehmen. Wenn diese Reihenfolge verletzt wird, kann es zu Fehlfunktionen kommen.

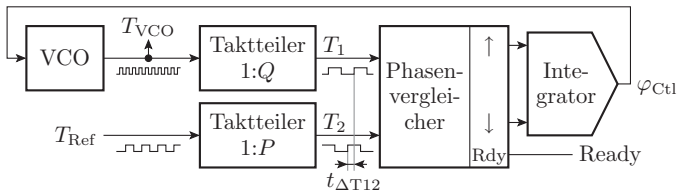


Abb. 4.56. Takterzeugung mit einem Phasenregelkreis

Phasenregelkreise werden zur Frequenzvervielfachung und zur Korrektur von Taktversätzen eingesetzt. Zur Erzeugung von Takten mit einem ganzzahligen Vielfachen der Frequenz des Basistaktes wird der P-Teiler nicht benötigt ($P = 1$). Mit dem Q-Teiler wird der gewünschte Vervielfachungsfaktor eingestellt. Mit $P > 1$ und $Q > 1$ lassen sich nicht ganzzahlige Vielfache der Basistaktfrequenz einstellen. Eine andere Anwendung von Phasenregelkreisen ist die Korrektur oder Erzeugung von Taktversätzen. Abbildung 4.57 a zeigt einen Phasenregelkreis mit einem Johnson-Zähler als Taktteiler. Der Schaltungstakt

T_0, der am Eingang des Johnson-Zählers abgegriffen wird, regelt sich so ein, dass er in Frequenz und Phase mit dem Referenztakt übereinstimmt. An den übrigen Ausgängen des Johnson-Zählers können frequenzgleiche, je um eine Achtelperiode phasenversetzte Takte abgegriffen werden (Abb. 4.57 b). Einer dieser phasenversetzten Takte dient in der Beispielschaltung als Ausgabetakt T_{out}. Zu ihm soll das Ausgabesignal $\mathbf{y}_{\text{out}}$ eine vorgegebene Vorhaltezeit $\geq t_{\text{s}}$ haben (Abb. 4.57 c). Die Vorhaltezeit ist in der Beispielschaltung die Differenz zahlreicher unterschiedlicher Verzögerungszeiten. Eine Synthese kann normalerweise ein Vorhaltezeit-Constraint dieser Art schwer einhalten. In der Beispielschaltung ist das aber kein Problem. Denn es ist hier vorgesehen, nach der fertigen Platzierung und Verdrahtung die Vorhaltezeit durch eine geringfügige lokale Verdrahtungsänderung, nach der der Ausgabetakt vom richtigen Zählerausgang abgegriffen wird, zu korrigieren.

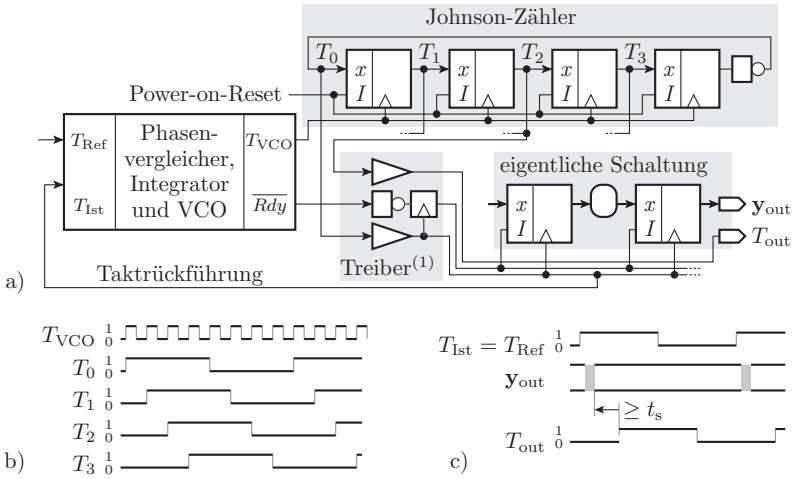


Abb. 4.57. PLL-basierte Erzeugung phasenverschobener und phasenkorrigierter Taktsignale a) Schaltung b) phasenverschobene Ausgabesignale des Johnson-Zählers c) Datenausgabe mit einer vorgegebenen Vorhaltezeit zum Ausgabetakt ((1) – Treiber für eine große Anzahl von Lasten)

4.3.7 Zusammenfassung und Übungsaufgaben

Daten werden in einer digitalen Schaltung bitweise gespeichert, entweder in einer kleinen Kapazität oder in einer bistabilen Schaltung. Die Signalwerte in Kapazitäten müssen nach der garantierten Haltezeit aufgefrischt werden. Eine bistabile Schaltung – ein Ring aus zwei invertierenden Gattern – behält den gespeicherten Wert bis zum nächsten Schreibvorgang oder bis zum Abschalten der Versorgungsspannung. Für das Schreiben der Daten gibt es unterschiedliche Ansteuerschemata. Das klassische RS-Flipflop ist laufzeitkritisch und nicht

für zu synthetisierende Entwürfe geeignet. Auch D-Flipflops und Latches, die ihre Daten zustandsgesteuert übernehmen, sind für die Synthese problematisch. Das ideale taktflankengesteuerte Speicherelement ist das Master-Slave-Flipflop, das aus zwei D-Flipflops besteht. Der Master übernimmt die Daten vor der aktiven Taktflanke und speichert sie nach der aktiven Taktflanke. Der Slave übernimmt die im Master gespeicherten Daten und speichert, während der Master die nächsten Daten übernimmt. Bei einem Master-Slave-Flipflop mit initialisierbarem Slave hat die Initialisierung Vorrang vor der Datenübernahme und das Initialisierungssignal muss zum Takt ausgerichtet sein.

Der Takt einer digitalen Schaltung wird in der Regel mit einem Quarz oder einem anderen mechanischen Schwinger stabilisiert, der eine hohe Zeitgenauigkeit sichert. Auch Taktteiler, Taktvervielfachungsschaltungen und Schaltungen zur Phasenkorrektur sind Spezialschaltungen zur Bereitstellung zeitgenauer Signale. Bei einer Taktversorgung mit Phasenregelkreisen ist auf die Initialisierungsreihenfolge zu achten. Weiterführende und ergänzende Literatur siehe [20, 21, 30, 36, 47, 52].

Aufgabe 4.8

Gegeben sind die Transistorschaltung und der Eingabesignalverlauf in Abb. 4.58.

a) Welche Funktion haben die Gatter G1 bis G4?
b) Beschreiben Sie die Funktionsweise der Schaltung und skizzieren Sie die Signalverläufe der Zwischensignale und des Ausgabesignals für den gegebenen Eingabesignalverlauf. Die Verzögerungszeiten seien in der Skizze vernachlässigbar klein und die Haltezeit, wenn der Gatterausgang hochohmig ist, sei viel größer als die Taktperiode.
c) Um welchen Typ von Speicherzelle handelt es sich?

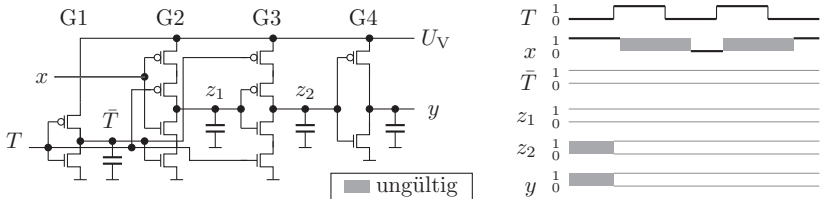


Abb. 4.58. Schaltung und Eingabesignalverlauf zu Aufgabe 4.8

Aufgabe 4.9

Wie müssen die Teilerverhältnisse P und Q für den Phasenregelkreis in Abb. 4.56 gewählt werden, um mit einem quarzstabilisierten 50MHz-Referenztakt einen 300MHz-Takt zu erzeugen?

4.4 Schreib-Lese-Speicher

Für die Speicherung größerer Informationsmengen in digitalen Schaltungen werden Blockspeicher eingesetzt. Blockspeicher sind bottom-up entworfene, hochgradig optimierte Schaltungen mit einer wesentlich höheren Transistordichte, als der, die mit einer freistrukturierten Schaltung mit Latches und Registern erzielbar ist. Blockspeicher bestehen aus einer zweidimensionalen Speichermatrix, die den Hauptteil der Fläche einnimmt, umgeben von einer Ansteuerschaltung (Abb. 4.59).

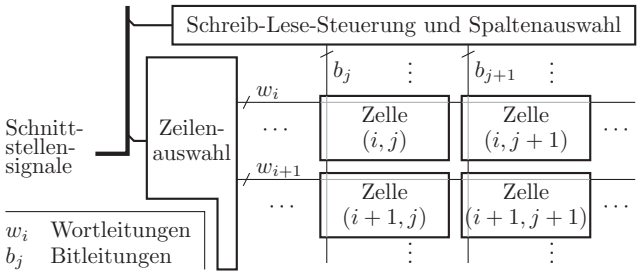


Abb. 4.59. Blockspeicher

Die Funktion eines Blockspeichers wird in erster Linie von der Funktion der Zellen bestimmt. Abbildung 4.60 zeigt eine Übersicht über die wichtigsten Speicherarten. Es wird zwischen Festwertspeichern und Schreib-Lese-Speichern unterschieden. Festwertspeicher können nur einmal oder nur mit großem Zeitaufwand beschrieben werden, behalten ihre Daten auch ohne Versorgungsspannung über Jahre und werden in einem separaten Abschnitt behandelt. Schreib-Lese-Speicher haben eine Schreibzeit in der Größenordnung der Lesezeit. Statische Speicher behalten die gespeicherten Daten, bis die Versorgungsspannung abgeschaltet wird. Dynamische Speicher verwenden Kapazitäten als Speichermedium. Sie haben eine sehr hohe Speicherdichte, verlieren ihre Daten aber ohne Auffrischen nach wenigen Millisekunden. Die meisten Blockspeicher sind so aufgebaut, dass in jedem Zeitschritt nur der Zugriff auf einen Speicherplatz möglich ist. Mehrportspeicher, bei denen mehrere Speicherzellen zeitgleich wahlfrei adressiert werden können, benötigen spezielle

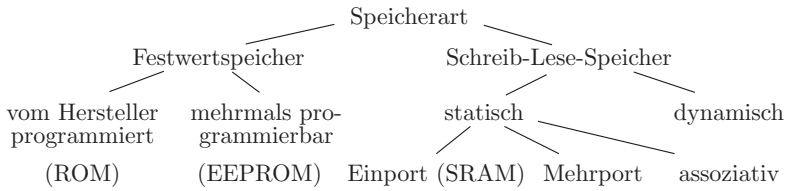


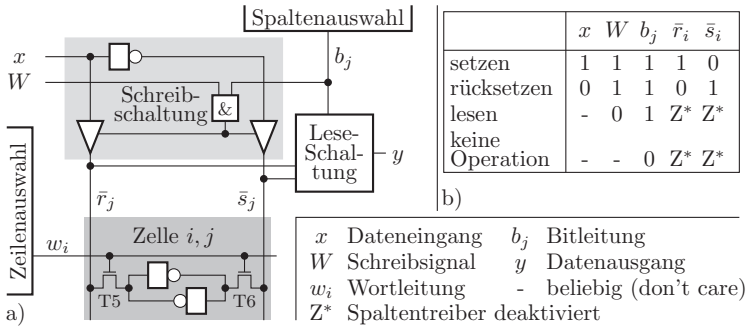
Abb. 4.60. Einteilung der Blockspeicher

Speicherzellen. Das gilt auch für Assoziativspeicher, die in einem Zugriffsschritt für ein Eingabedatenwort die Adresse, unter der es gespeichert ist, suchen und ausgeben können.

4.4.1 SRAM

Die Abkürzung RAM ist ein Akronym für **r**andom **a**ccess **m**emory (Speicher mit wahlfreiem Zugriff) und wird als Bezeichnung für Schreib-Lese-Speicher mit wahlfreiem Zugriff verwendet[11]. Das vorangestellte »S« steht für »**s**tatisch« und bedeutet, dass die Speichermatrix aus statischen Speicherzellen besteht.

Ein SRAM besteht aus einer Matrix bistabiler Speicherzellen umgeben von einer Ansteuerschaltung für die Spalten- und Zeilenauswahl und die Schreib-Lese-Steuerung. Die kleinste bistabile les- und beschreibbare Speicherzelle benötigt sechs Transistoren. Vier Transistoren bilden den Inverterring (vgl. Abschnitt 4.3.2). Die übrigen beiden Transistoren T5 und T6 dienen zur Zellenauswahl. Wenn die Wortleitung w_i einer Speicherzelle i, j aktiv ist, schalten beide Auswahltransistoren ein. Das Steuersignalpaar $\bar{r}_j, \bar{s}_j$ wählt dann zwischen vier möglichen Aktionen: setzen (»1« schreiben), rücksetzen (»0« schreiben), lesen und keine Operation (Abb. 4.61).



	x	W	b_j	$\bar{r}_i$	$\bar{s}_i$
setzen	1	1	1	1	0
rücksetzen	0	1	1	0	1
lesen	-	0	1	Z*	Z*
keine Operation	-	-	0	Z*	Z*

b)

x	Dateneingang	b_j	Bitleitung
W	Schreibsignal	y	Datenausgang
w_i	Wortleitung	-	beliebig (don't care)
Z*	Spaltentreiber deaktiviert		

Abb. 4.61. SRAM a) Schaltung b) Ansteuerung der Speicherzellen

In Abb. 4.61 erfolgt die Zeilenauswahl über die Wortleitungen w_i und die Spaltenauswahl über die Bitleitungen b_j. Bei jedem Zugriff sind genau eine Wortleitung und eine Bitleitung aktiv. In allen anderen Zeilen sind die Auswahltransistoren ausgeschaltet und in allen anderen Spalten sind die Treiber der Schreibschaltung für die Steuersignale $\bar{r}_j, \bar{s}_j$ hochohmig und die Leseschaltungen deaktiviert, so dass keine Operation ausgeführt wird.

Für die ausgewählte Zelle schaltet die Wortleitung w_i die Auswahltransistoren ein. Die Bitleitung aktiviert, wenn das Schreibsignal aktiv ist, die Treiber der Steuersignale $\bar{r}_j, \bar{s}_j$. Diese übernehmen den direkten und invertierten

[11] Auch die meisten Nur-Lese-Speicher sind wahlfrei adressierbar.

Eingabewert. Bei einer zu schreibenden »1« wird der Zellenzustand über das low-aktive Setzsignal $\bar{s}_j$ auf »1« und bei einer zu schreibenden »0« wird er über das low-aktive Rücksetzsignal $\bar{r}_j$ auf »0« gesetzt. Wenn das Schreibsignal W inaktiv ist, bleiben die Spaltentreiber inaktiv. Die Signale $\bar{r}_j, \bar{s}_j$ übernehmen die schwachen Werte der ausgewählten Speicherzelle und leiten sie an die Leseschaltung weiter, die den Datenwert übernimmt und zum Ausgang des Schaltkreises durchstellt.

Abbildung 4.62 zeigt die komplette Transistorschaltung der 6-Transistor-Zelle und ein Beispiel für eine mögliche geometrische Anordnung. Die Transistoren haben Minimalabmessungen und werden auf minimaler Fläche angeordnet. Die zeilen- und spaltenweise Verdrahtung erfolgt in mehreren Ebenen von Metallleiterbahnen. Zur Erzeugung einer Zellenmatrix aus vielen Zeilen und Spalten wird die Zelle einfach Tausende oder Millionen Male kopiert.

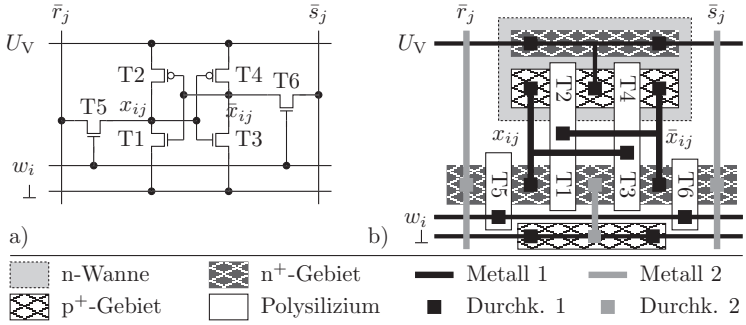


Abb. 4.62. SRAM-Zelle a) Transistorschaltung b) geometrische Anordnung

Eine wichtige Kenngröße eines Schreib-Lese-Speichers ist die Zugriffszeit. Das ist die Dauer eines Speicherzugriffs. Sie liegt bei großen Schreib-Lese-Speichern mit mehreren Megabit Speicherkapazität in der Größenordnung von 10 ns bis 100 ns und ist um Zehnerpotenzen größer als die Verzögerungszeit eines einzelnen Gatters. Benötigt wird diese lange Zeit für die Adressdecodierung und das Treiben der Zeilen- und Spaltenleitungen, die eine linear mit der Spalten- bzw. Zeilenanzahl zunehmende Lastkapazität haben. Aus Letzterem folgt, dass die Zugriffszeit eines Speichers mit der Anzahl der Speicherzellen zunimmt. Kleine Blockspeicher sind deutlich schneller als große.

Für die Simulation wurde das RAM-Modell in Abschnitt 3.4.3 in eine Kernfunktion und eine Schnittstellenfunktion unterteilt. Die Kern-Lesefunktion ähnelt der einer kombinatorischen Schaltung. Wenn das Schreibsignal inaktiv ist, wird das Ausgabesignal nach einer Adressänderung nach der Haltezeit t_{hy} ungültig. Wenn die neue gültige Adresse anliegt, wird nach der Verzögerungszeit t_{dy} der gespeicherte Inhalt ausgegeben. Ein korrekter Schreibvorgang setzt voraus, dass die Adresse hinreichend lange vor und nach der Aktivierung des Schreibsignals stabil und gültig ist. Der Schreibimpuls muss eine Mindestlän-

ge haben und die zu schreibenden Daten müssen hinreichend lange vor der Deaktivierung des Schreibsignals stabil und gültig sein (Abb. 4.63). Bei einer Verletzung der Zeitbedingungen werden ungültige Werte gespeichert.

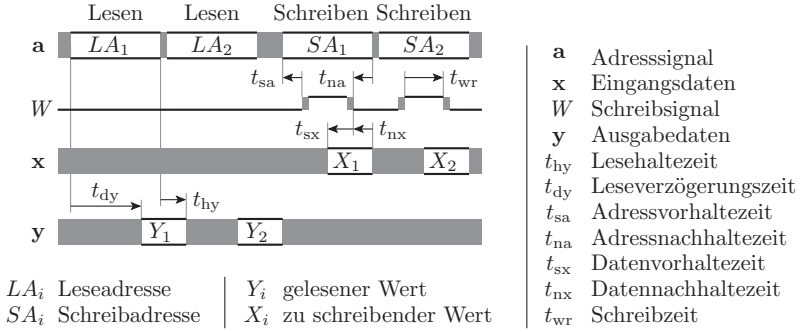


Abb. 4.63. Signalverläufe zum Lesen und Schreiben für die SRAM-Kernfunktion

Simulationsmodell für einen Speicherschaltkreis

Große Blockspeicher sind oft eigenständige Schaltkreise. In diesem Abschnitt wird ein vereinfachtes Simulationsmodell für einen asynchronen SRAM-Schaltkreis entwickelt. Ziel ist die Vermittlung eines skizzenhaften Einblicks in diese Art der Modellbildung und die Entwicklung eines Beispielmodells für den darauf folgenden Unterabschnitt.

Der Beispielschaltkreis ist ein IS61LV25616AL [8]. Er hat achtzehn Adresseingänge, sechzehn bidirektionale Datenanschlüsse, eine Speicherkapazität von vier Megabit und eine Zugriffszeit von 10 ns. Die Operationsauswahl soll ausschließlich über die drei Steuersignale

- Ausgabeaktivierung: $\overline{oe}$ (**o**utput **e**nable, low-aktiv),
- Schreibaktivierung: $\overline{wr}$ (**wr**ite, low-aktiv) und
- Schaltkreisauswahl: $\overline{cs}$ (**c**hip **s**elect, low-aktiv)

erfolgen. Die hier nicht berücksichtigten Byte-Auswahlsignale seien ständig aktiv. Die bidirektionalen Datenanschlüsse bilden einen Bus, über den bei einem Schreibzugriff die zu schreibenden Daten zum Schaltkreis geschickt und bei einem Lesezugriff die gelesenen Daten abgeholt werden.

Intern hat der Speicherschaltkreis, ohne dass das explizit im Datenblatt steht, eine Matrixstruktur wie in Abb. 4.61, die in eine Schnittstellenschaltung eingebettet ist. Letztere erzeugt aus den im Datenblatt beschriebenen Anschlusssignalverläufen die Signalverläufe für die Kernfunktion. Mit diesem Hintergrundwissen kann das Simulationsmodell aus dem bereits behandelten Simulationsmodell für die Kernfunktion und einem noch zu entwickelnden

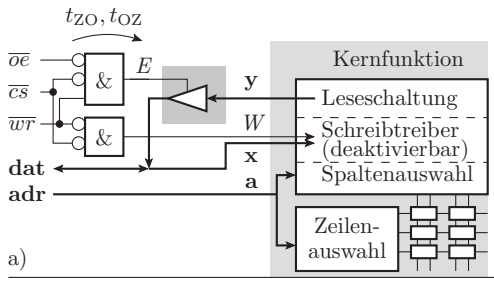
Simulationsmodell für die Schnittstellenschaltung zusammengesetzt werden. Für die Schnittstellenschaltung lässt sich aus dem Datenblatt Folgendes rekonstruieren. Das Ausgabesignal **y** der Kernfunktion ist über einen deaktivierbaren Treiber mit den bidirektionalen Datenanschlüssen **dat** des Schaltkreises verbunden. Das Freigabesignal E des Treibers wird nur aktiviert, wenn das Schaltkreisauswahlsignal $\overline{cs}$ und das Ausgabefreigabesignal $\overline{oe}$ aktiv und das Schreibsignal $\overline{wr}$ inaktiv ist:

$$E = cs \wedge oe \wedge \overline{wr}$$

Das Schreibsignal W der Kernfunktion wird nur aktiviert, wenn das Schaltkreisauswahlsignal $\overline{cs}$ und das Schreibsignal $\overline{wr}$ des Schaltkreises aktiv ist (Abb. 4.64):

$$W = cs \wedge wr$$

Im nächsten Schritt ist das Zeitverhalten nachzubilden. Aus den im Datenblatt für den Lesezyklus dargestellten Signalverläufen ist ablesbar, dass die Leseverzögerung $t_{\mathrm{dy}} \leq 10\,\mathrm{ns}$ und die Lesehaltezeit $t_{\mathrm{hy}} \geq 2\,\mathrm{ns}$ beträgt.



Schaltkreisanschlüsse:
- $\overline{oe}$ Ausgabefreigabesignal
- $\overline{cs}$ Schaltkreisauswahlsignal
- $\overline{wr}$ Schreibsignal
- **adr** Adresssignal
- **dat** bidirektionales Datensignal

Verzögerung des Ausgabetreibers:
- t_{OZ} Deaktivierungsverzögerung
- t_{ZO} Aktivierungsverzögerung (im Beispiel unten $2 \times 4\,\mathrm{ns}$)

b)

```
signal adr: tUnsigned(17 downto 0);
signal dat, y: STD_LOGIC_VECTOR(15 downto 0);
signal n_oe, n_cs, n_wr, W, E: STD_LOGIC;
...
W <= not n_cs and not n_wr;                        -- 1. Gatter
E <= 'X', not n_oe and not n_cs and n_wr after 4 ns; -- 2. Gatter
ram(W=>W, a=>adr, x=>dat, y=>y, thy=>2 ns,         -- Kernfunktion
 tdy=>10 ns, tsa=>0 ns, tna=>0 ns, twr=>8 ns, tsx=>6 ns);
process(oe, y)                                     -- Ausgabetreiber
begin
 if  E='1' then dat <= y;
 elsif E='0' then dat <= (others=>'Z');
 else  dat <= (others=>'X');
 end if;
end process;
```

⇒Web-Projekt: P4.4/IS61.vhdl

Abb. 4.64. Verhaltensmodell eines asynchronen SRAM-Schaltkreises a) Ersatzschaltung aus Kernfunktion und Schnittstellenschaltung b) VHDL-Beschreibung

Die Aktivierung und Deaktivierung des Ausgabetreibers dauert zwischen null und 4 ns. Die Adresse benötigt keine Vor- und keine Nachhaltezeit. Für die Eingabedaten sind eine Vorhaltezeit von $t_{\mathrm{dx}} \geq 6\,\mathrm{ns}$ und eine Nachhaltezeit $t_{\mathrm{nx}} \geq 0$ gefordert. Das Schreibsignal $\overline{wr}$ muss mindestens für 8 ns aktiv sein und die Adresse für mindestens 10 ns stabil anliegen. Die Aktivierungs- und Deaktivierungsverzögerung des Treibers sind in Abb. 4.64 b als Halte- und Verzögerungszeit des Gatters, das das Freigabesignal E erzeugt, modelliert. Die übrigen Zeitparameter werden der nebenläufigen Prozedur »ram(...)« aus Abschnitt 3.4.3, Abb. 3.25 übergeben, die das Verhalten der Kernfunktion nachbildet. Zum Aufdecken möglicher Modellschwächen sind Testbeispiele aus dem Datenblatt abzuleiten.

Das entwickelte Schaltkreissimulationsmodell ist noch nicht ganz exakt. Für die Adresse kontrolliert die Prozedur »ram(...)« nur, dass sie bei einem Schreibzugriff für die Summe aus der Mindestschreibimpulsbreite, der Adressvorhaltezeit und der Adressnachhaltezeit

$$t_{\mathrm{wr}} + t_{\mathrm{sa}} + t_{\mathrm{na}} = 8\,\mathrm{ns} + 0 + 0$$

stabil anliegt. Laut Datenblatt soll sie aber mindestens für 10 ns stabil anliegen. Das erfordert eine kleine Änderung in der Prozedur »ram(...)«. Laut Simulationsmodell hat das Signal $\overline{cs}$ dieselbe zeitliche Wirkung auf den deaktivierbaren Ausgabetreiber wie das Signal $\overline{oe}$, was auch nicht genau den Datenblattangaben entspricht. Für unsere Zwecke soll das Simulationsmodell in Abb. 4.64 b jedoch so genügen, wie es ist.

Speichercontroller

Der Schaltkreis aus dem vergangenen Abschnitt soll in eine synthesefähige Beschreibung eingebettet werden. Dazu muss er in Register eingerahmt werden, die die Eingabe- und Ausgabesignale abtasten und dabei zeitlich am Takt ausrichten. Das abgetastete Eingabedatensignal ist über einen deaktivierbaren Treiber an die bidirektionalen Datenanschlüsse des Schaltkreises anzuschließen. Für die Einstellung der geforderten Zeitversätze und Zeittoleranzfenster der Steuersignale in Bezug auf die abgetasteten Adress- und Datensignale gibt es zwei Lösungsansätze:

- Einstellung mit abgestimmten Verzögerungszeiten und
- Erzeugung mit einer sequenziellen Schaltung.

In Abb. 4.65 werden die Steuersignalverläufe mit abgestimmten Verzögerungszeiten eingestellt. Aus den Abtastwerten der externen Steuersignale W (**w**rite, schreiben) und R (**r**ead, lesen) werden mit einer kombinatorischen Schaltung die drei Schaltkreisansteuersignale $\overline{cs}$, $\overline{wr}$ und $\overline{oe}$ sowie ein Steuersignal E_{W} für die Aktivierung des Schreibtreibers und ein Freigabesignal E_{R} für das Leseregister erzeugt. Die kombinatorischen Funktionen für die Erzeugung der fünf Steuersignale sind einfach, aber ihre Gatternachbildungen müssen genau

aufeinander abgestimmte Einschalt- und Ausschaltverzögerungen haben. Die Ausschaltverzögerung der Schaltung für die Bereitstellung des Schreibsignals $\overline{wr}$ muss z.B. mindestens so groß wie die Adressverzögerung und seine Einschaltverzögerung darf maximal so groß wie die Adressverzögerung sein. In dieser Rechnung sind alle Verzögerungen, auch die auf der Baugruppe, zu berücksichtigen. Während eines Adresswechsels zwischen zwei Schreibvorgängen ist normalerweise das Schreibsignal zu deaktivieren. Der Beispielschaltkreis hat jedoch eine Adressvorhalte- und eine Adressnachhaltezeit null. Wenn sich alle Adressbits am Schaltkreis fast gleichzeitig ändern, kann auf die Deaktivierung verzichtet werden. Wie groß »fast gleichzeitig« ist, steht leider nicht im Datenblatt. Das Zeitfenster dafür ist offenbar sehr klein. Enge Zeittoleranzen erfordern, auch wenn die Schaltung nur aus wenigen Gattern besteht, einen professionellen Handentwurf. Für die Synthese der umgebenden Schaltung sind die manuell gefundene Schaltung sowie ihre Platzierung und Verdrahtung mit Constraints festzuschreiben (vgl. Abschnitt 2.1.5). Der Entwurf ist viel schwieriger, als es die Schaltung in Abb. 4.65 vermuten lässt.

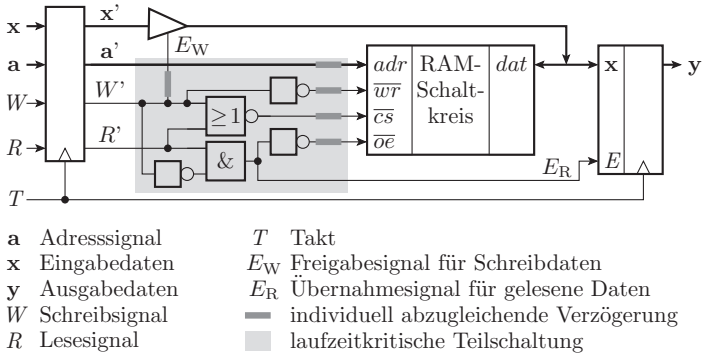


Abb. 4.65. Speichercontroller für einen asynchronen Schreib-Lese-Speicher mit individuell angepassten Verzögerungszeiten

Die Alternative zu einem Handentwurf der laufzeitkritischen Schaltungsteile ist die Erzeugung der Steuersignale mit einer sequenziellen Schaltung. Abbildung 4.66 a zeigt die Prinzipschaltung. Die Adresse und die Daten werden genauso wie in der ersten Lösung abgetastet und die abgetasteten Eingabedaten werden genauso über einen deaktivierbaren Treiber an die bidirektionalen Datenanschlüsse des Schaltkreises angeschlossen. Neu ist der Moore-Automat, der die Steuer- und Freigabesignale erzeugt.

Für den Automatenentwurf sind als erstes die Zeitverläufe der zu erzeugenden Steuersignale zu spezifizieren. In Abb. 4.66 b ist jeder Speicherzugriff in vier Taktphasen unterteilt. Die abgetastete Adresse **a**' soll jeweils in allen vier Phasen stabil anliegen und das Schaltkreisauswahlsignal $\overline{cs}$ soll in allen vier Phasen aktiv sein. Um bei einem Schreibzugriff eine ausreichende

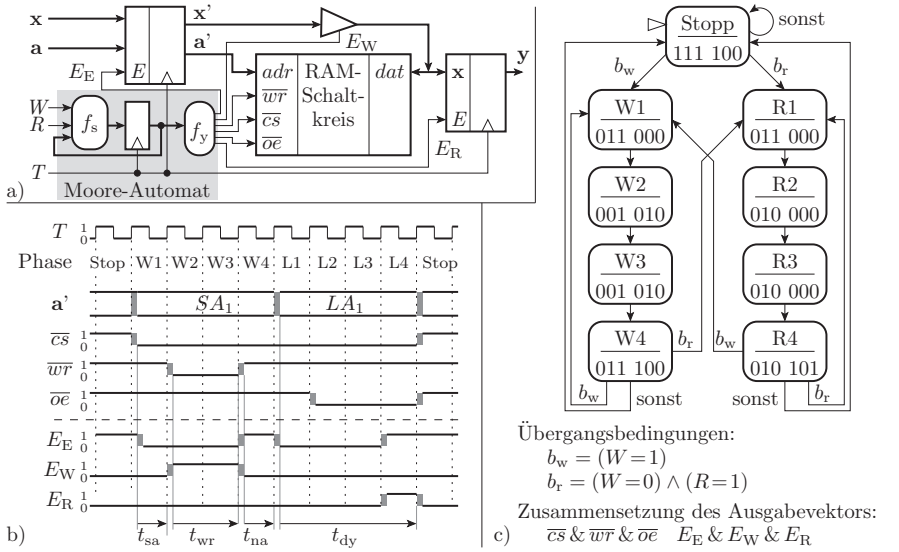


Abb. 4.66. Speichercontroller für einen asynchronen Schreib-Lese-Speicher mit einem Automaten zur Erzeugung der Steuersignale a) Gesamtschaltung b) zu erzeugende Steuersignalverläufe c) Zustandsgraph der Steuerung

Adressvorhalte- und Adressnachhaltezeit zu garantieren, wird das Schreibsignal $\overline{wr}$ erst einen Takt nach der Adressabtastung aktiviert und einen Takt vor der nächsten Adressabtastung deaktiviert. Das Freigabesignal E_W für den Schreibtreiber soll erst ab der zweiten Taktphase aktiviert werden, damit der Ausgabetreiber im RAM-Schaltkreis nach einem Lesezyklus genügend Zeit hat, sich zu deaktivieren. Das Ausgabefreigabesignal des Speicherschaltkreises $\overline{oe}$ und das Freigabesignal E_R für das Leseregister bleiben während des gesamten Schreibzugriffs inaktiv. Bei einem Lesezugriff bleiben das Speicherschreibsignal $\overline{wr}$ und das Freigabesignal E_W für den Schreibtreiber durchgängig inaktiv. Das Ausgabefreigabesignal des Schaltkreises $\overline{oe}$ soll, damit der Treiber für die zu schreibenden Daten genügend Zeit zur Deaktivierung hat, erst im zweiten Takt aktiviert werden. Das Freigabesignal E_R für das Leseregister ist nur im letzten Takt vor Abschluss der Leseoperation zu aktivieren.

Im nächsten Entwurfsschritt wird aus den Steuersignalverläufen in Abb. 4.66 b der Zustandsgraph in Abb. 4.66 c konstruiert. Nach der Initialisierung und wenn kein Speicherzugriff erfolgt, soll sich der Automat im Zustand »Stopp« befinden, in dem alle Steuersignale außer dem Freigabesignal E_E für das Eingaberegister inaktiv sind. Wenn das externe Schreibsignal W im Stopp-Zustand oder am Ende eines Schreib- oder Lesezylusses aktiv ist, wird anschließend die Zustandsfolge »W1« bis »W4« durchlaufen, die die Steuersignale für den Schreibvorgang erzeugt. In Analogie dazu wird in diesen Zuständen, wenn das Schreibsignal W inaktiv und das Lesesignal R aktiv ist,

die Zyklusfolge »R1« bis »R4« für die Erzeugung der Steuersignalfolge für den Lesevorgang durchlaufen. Im letzten Schreib- und im letzten Lesezustand wird jeweils wie im Stopp-Zustand das Steuersignal E_{E} für die Abtastung der Eingabeadresse und der Eingabedaten aktiviert, damit sich sofort der nächste Schreib- oder Lesezyklus anschließen kann.

Der nächste Entwurfsschritt ist die Abschätzung der minimalen Taktperiode. Die kritischste Zeitbedingung für den Beispielschaltkreis ist, dass der zwei Takte lange Schreibimpuls eine Mindestbreite von $t_{\mathrm{wr}} \geq 8\,\mathrm{ns}$ haben muss. Daraus folgt für die Taktperiode eine Mindestlänge von $T_{\mathrm{P}} \geq 4\,\mathrm{ns}$. Relativ kritisch ist auch die Einhaltung der Datenvorhaltezeit von $t_{\mathrm{sx}} \geq 6\,\mathrm{ns}$ für die in Summe mit der Aktivierungsverzögerung des Eingabetreibers auch nur zwei Taktperioden eingeplant sind. Wenn der Eingabetreiber des zu entwerfenden Controllers eine längere Aktivierungsverzögerung als 2 ns hat, muss die Taktperiode entsprechend länger als 4 ns gewählt werden.

Der weitere Entwurf vom Zustandsgraphen bis zur fertigen Simulations- und Synthesebeschreibung erfolgt in der bereits mehrfach vorgeführten Weise und hat Rezeptcharakter (siehe Abschnitt 1.6). Auch wenn die Schaltung in Abb. 4.66 wesentlich komplizierter aussieht als die in Abb. 4.65, ist die zweite Schaltung einfacher zu entwerfen. Denn sie ist nicht laufzeitkritisch und damit synthesefähig. Allerdings ist die zweite Schaltung mit dem Automaten deutlich aufwändiger und langsamer als die erste.

4.4.2 Mehrportspeicher

Ein zeitgleicher Zugriff auf mehrere Speicherplätze eines Blockspeichers verlangt Zellen, die über mehrere Sätze von Steuerleitungen unabhängig voneinander gelesen und beschrieben werden können. In Abb. 4.67 a ist die 6-Transistor-Zelle aus Abb. 4.62 um einen zweiten Port erweitert. Der zweite Port besteht auf der Zellenebene aus dem zusätzlichen Auswahltransistorpaar T7 und T8. Jeder Port besitzt eigene Zeilen- und Spaltenauswahlsignale, die von einer Port-eigenen Ansteuerschaltung erzeugt werden (Abb. 4.67 b). Nach außen hin verhalten sich die einzelnen Ports eines Mehrportspeichers wie separate Speicher, nur dass der Zugriff auf dieselbe Speichermatrix erfolgt. Ein

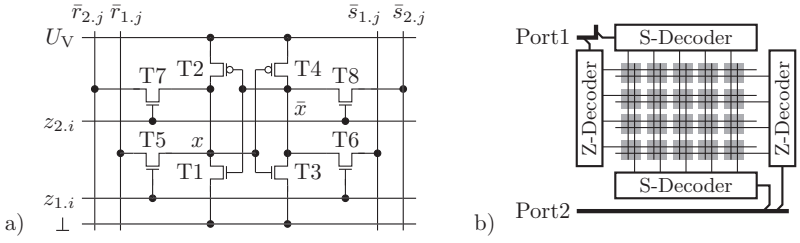


Abb. 4.67. 2-Port-Speicher a) Speicherzelle b) Gesamtstruktur (S-Decoder – Schreib-Lese-Steuerung und Spaltenauswahl; Z-Decoder – Zeilenauswahl)

gleichzeitiges Beschreiben derselben Zelle mit unterschiedlichen Werten invalidiert den gespeicherten Inhalt und ist nach Möglichkeit durch eine geeignete Ansteuerschaltung auszuschließen. Eine typische Anwendung für Mehrportspeicher ist die Kopplung von Rechnern.

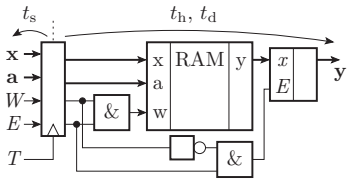
Synthesebeschreibungen mit 1-Port- und Mehrportspeichern

Abbildung 4.68 a zeigt noch einmal die synthesefähige Beschreibung für den 1-Port-Speicher aus Abb. 3.26 in Abschnitt 3.4.3. Wenn das Schreibsignal aktiv ist, wird das adressierte Feldelement beschrieben. Sonst, wenn das Lesesignal aktiv ist, wird der Wert des adressierten Feldelements gelesen. Für beide Operationen steht maximal die Dauer einer Taktperiode zur Verfügung.

a)

```
signal RAM is tRAM(0 to 2**a'LENGTH - 1);
signal T, W, E: STD_LOGIC;
signal x, y: tDaten;
signal a: tAdresse;
 ...
process (T)
begin
 if RISING_EDGE(T) then
   if W='1' and E='1' then
    RAM(Int(a)) <= x;
   elsif E='1' then
    y <= RAM(Int(a));
   end if;
 end if;
end process;
```

b)



c)

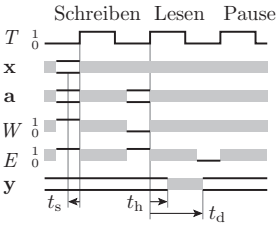


Abb. 4.68. Synthesebeschreibung eines 1-Port-Speichers a) VHDL-Beschreibung b) erforderliche Beschaltung der Kernfunktion c) Ansteuerung

Abbildung 4.68 b zeigt die zugehörige Beschaltung der Kernfunktion. Die Adress-, Steuer- und Eingabedatensignale müssen, damit der gesamte Blockspeicher nach außen laufzeitrobust ist, abgetastet auf die Kernfunktion geführt werden. Ein nachgeschaltetes Ausgabe-Latch lässt die Lesewerte bei einem Lesezugriff passieren und hält die Ausgabe während der Schreibzugriffe konstant. Die internen Verzögerungszeiten sind so laufzeitkritisch, dass der Speicher gemeinsam mit seiner Schnittstelle von der Synthese nur als fertig vorentworfener und verdrahteter Block verwendet werden kann. Die Vorhalte- und die Nachhaltezeiten zur aktiven Taktflanke für die Adresse, die Eingabedaten und die Steuersignale sind die des eingangsseitigen Abtastregisters und damit sehr klein. Die Haltezeit t_h und die Verzögerungszeit t_d eines Lesezugriffs sind etwas größer als die der Speicherkernfunktion und damit sehr groß (Abb. 4.68 c).

Die Synthesebeschreibung eines Mehrportspeichers ist sehr ähnlich zu der eines 1-Port-Speichers. Die Speichermatrix und die einzelnen Schreib- und Lesezugriffe werden genauso beschrieben. Der einzige Unterschied ist, dass in der Beschreibung eines Mehrportspeichers die Möglichkeit existiert, dass auf mehrere Speicherelemente zeitgleich zugegriffen wird. Schaltungstechnisch ist bei einem synchronen Mehrportspeicher jeder Port wie der des 1-Port-Speichers in Abb. 4.68 b zu beschalten.

Im nachfolgenden Beschreibungsbeispiel wird in jedem Takt auf den Speicherplatz mit der Adresse **a** geschrieben und gleichzeitig der Inhalt der Vorgängeradresse gelesen:

```
process (T)
begin
 if RISING_EDGE(T) then
   RAM(Int(a)) <= x;
   y <= RAM(Int(a) - 1);
 end if;
end process;
```

Das erfordert einen 2-Port-Speicher. Eine weitere Leseoperation würde die erforderliche Port-Anzahl auf drei erhöhen. Auch wenn zwei zeitlich nicht zueinander ausgerichtete Prozesse auf dasselbe Bitvektorfeld schreibend oder lesend zugreifen, muss der Speicher mindestens zwei getrennte Ports haben. Umgekehrt ist es eine wichtige Optimierungsrichtlinie, in der Synthesebeschreibung die maximal mögliche Anzahl der zeitgleichen Speicherzugriffe, die die Portanzahl festlegt, so gering wie möglich zu halten.

FIFO mit 1- und 2-Port-Speicher

Abbildung 4.69 zeigt eine Schaltung zur Rechnerkopplung mit einem FIFO-Speicher[12]. Das Initialisierungssignal *I* stellt den Anfangszustand »FIFO leer« her. Rechner 1 schreibt Daten in den FIFO und Rechner 2 holt die Daten in derselben Reihenfolge aus dem FIFO ab.

In Abschnitt 3.4.4 wurde ein FIFO als ein Datenobjekt aus einem Blockspeicher, einem Lesezeiger, einem Schreibzeiger, einem Flag für »voll«, einem Flag für »leer« und je einer Bearbeitungsmethode zum Initialisieren, zum Schreiben eines Datenobjekts und zum Lesen eines Datenobjekts modelliert. Die Methode »init« setzt die beiden Zeiger auf null, das Flag »leer« auf TRUE und das Flag »voll« auf FALSE. Die Schreibmethode wird nur ausgeführt, wenn der FIFO nicht voll ist. Sie löscht das Flag »leer«, schreibt die Daten auf den nächsten freien Platz und stellt den Schreibzeiger weiter. Die Lesemethode entnimmt, wenn der FIFO nicht leer ist, den ältesten Wert und stellt den Lesezeiger weiter:

[12] Ein FIFO-Speicher bildet eine Warteschlange nach, in die nacheinander Daten geschrieben und aus der die Daten in derselben Reihenfolge wieder abgeholt werden können (vgl. Abschnitt 3.4.4).

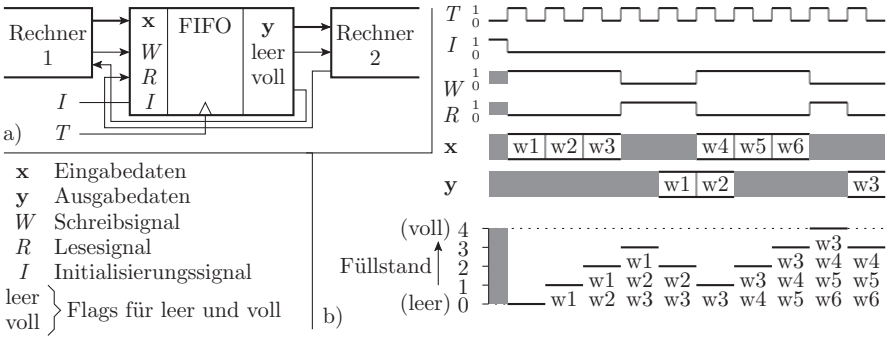


Abb. 4.69. FIFO-Speicher a) Einsatz zur Kopplung von zwei Rechnern b) Funktionsweise an einem Beispielablauf

```
-- Package-Vereinbarungen für ein FIFO-Modell
constant N_Adr: POSITIV := ...; -- Anzahl der Adressbits
subtype tAdrIdx is range 0 to 2**N_Adr - 1;
subtype tDaten is ...;
type tSpeicher is array (tAdrIdx) of tDaten;
type tFIFO is record
 Speicher: tSpeicher;          -- Speichermatrix
 sz, lz  : tAdrIdx;            -- Schreib- und Lesezeiger
 leer, voll : BOOLEAN;         -- Statussignale
end record;
procedure init(signal fifo: inout tFIFO);
procedure write(x: tDaten; signal fifo: inout tFIFO);
procedure read(signal y: out tDaten; fifo: inout tFIFO);
```

⇒WEB-Projekt: P3.4/FIFO_pack.vhdl

Im folgenden Verhaltensmodell wird der FIFO in jedem Taktschritt genau wie in Abb. 4.69 nur entweder beschrieben oder gelesen:

```
signal x, y: tDaten;
signal fifo: tFIFO;
...
if init='1' then init(fifo);
elsif RISING_EDGE(T) then
 if   W='1' and not voll then write(x, fifo);
 elsif R='1' and not leer then  read(y, fifo);
 end if;
end if;
```

Da in keinem Takt auf mehrere Speicherplätze zugegriffen wird, würde die Synthese für den Blockspeicher des FIFO-Objekts einen 1-Port-Speicher wählen. Bei einer geringfügigen Modifikation, die ein zeitgleiches Lesen und Schreiben nicht mehr ausschließt,

```
...
elsif RISING_EDGE(T) then
 if W='1' and not voll then write(x, fifo); end if;
 if R='1' and not leer then  read(y, fifo); end if;
end if;
```

muss die Synthese einen synchronen 2-Port-Speicher einsetzen.

4.4.3 Assoziativspeicher

Ein Assoziativspeicher ist ein normal beschreibbarer und lesbarer Speicher mit einer Zusatzfunktion für den parallelen Vergleich. Bei der Vergleichsoperation werden die Eingabedaten in einem Schritt mit den gespeicherten Bitmustern aller Speicherzeilen verglichen. Abbildung 4.70 zeigt die Schaltung einer Zelle. Die Transistoren T1 bis T6 bilden eine normale RAM-Zelle und die Transistoren T7 bis T10 bilden die Schaltung für den bitweisen Vergleich. Bei der Vergleichsoperation werden die Spaltenleitungen wie beim Schreiben angesteuert:

$$\bar{r}_j = d_j;\ \bar{s}_j = \bar{d}_j \tag{4.37}$$

(d_j – Bitwert für Spalte j). Die Zeilenauswahlsignale bleiben jedoch alle inaktiv, so dass der Zelleninhalt nicht verändert wird. Das Netzwerk aus den Transistoren T7 bis T10 hat die Funktion

$$\begin{aligned} f_{ij} &= (\bar{r}_j \wedge \bar{x}_{ij}) \vee (\bar{s}_j \wedge x_{ij}) \\ &= (d_j \wedge \bar{x}_{ij}) \vee \left(\bar{d}_j \wedge x_{ij}\right) \end{aligned} \tag{4.38}$$

Die Parallelschaltung der beiden Transistorpaare sperrt genau dann, wenn der Zellenwert mit dem Eingabewert übereinstimmt. Die Vergleichsschaltungen aller Spalten j einer Zeile i sind parallel geschaltet und damit ODER-verknüpft:

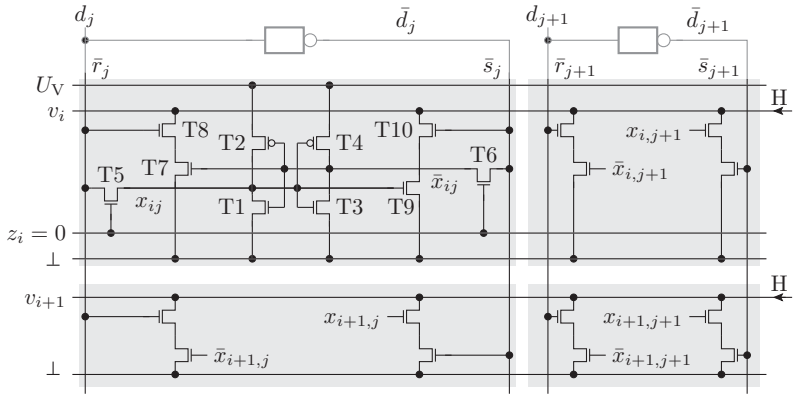


Abb. 4.70. Aufbau der Speichermatrix eines Assoziativspeichers

$$f_i = \bigvee_{j=1}^{N_S} f_{ij} \tag{4.39}$$

(N_S – Spaltenanzahl). Das Gesamtnetzwerk einer Zeile i ist genau dann gesperrt ($f_i = 0$), wenn die Eingabebits aller Spalten übereinstimmen. Ein Pull-Up-Element erzeugt in diesem Fall den Vergleichswert $v_i = 1$. Die Gesamtausgabe besteht in der Regel aus zwei Informationseinheiten. Das ist zum einen ein binäres Signal »hit«, das aktiviert wird, wenn das Suchmuster im Speicher steht, und zum anderen die Adresse der ersten Zeile, in der das Suchmuster gefunden wird.

Für die Modellierung eines synchronen Assoziativspeichers bietet sich genau wie für den FIFO im Vorabschnitt eine objektorientierte Modellierung an. Das Modell der Speichermatrix und die Schreib- und die Lesemethode unterscheiden sich nicht von der eines einfachen Schreib-Lese-Speichers und seien wie die in Abschnitt 3.4.3 definiert:

```
subtype tDaten is STD_LOGIC_VECTOR(... downto 0);
type tMem is array (NATURAL range <>) of tDaten;
subtype tAdr is tUnsigned(... downto 0);

function read(Mem: tMem; adr: tAdr) return tDaten;
procedure write(x: tDaten, adr: tAdr,
                                signal Mem: inout tMem);
```

Zusätzlich benötigt das Modell eines Assoziativspeichers eine Suchmethode zur Bestimmung der Adresse der ersten Speicherzeile, in der der Eingabewert steht:

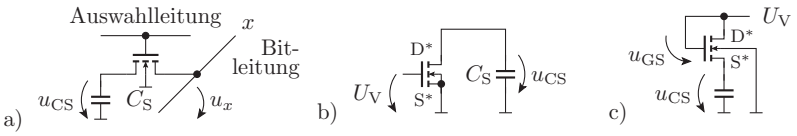
```
procedure search(x: tDaten; signal adr: out tAdr;
  signal hit: out STD_LOGIC; signal Mem: inout tMem) is
begin
 for idx in Mem'RANGE loop
  if Mem(idx)=x then
   hit <= '1';
   adr <= to_tUnsigned(idx, adr'LENGTH);
   return;
  end if;
 end loop;
 hit <= '0';
 adr <= (others=>'0');
end procedure;
```

⇒WEB-Projekt: P4.4/AsRAM_pack.vhdl

Auch wenn der Vergleich in der Hardware parallel erfolgt, muss er im Simulationsmodell sequenziell nachgebildet werden.

4.4.4 Dynamische Speicher (DRAM)

Dynamische Speicher (DRAM – **d**ynamic **r**andom **a**ccess **m**emory) besitzen die kleinsten Speicherzellen und die höchste Speicherdichte. Jede Speicherzelle besteht aus einer winzigen Kapazität C_S, die über einen NMOS-Transistor mit einer Bitleitung verbunden ist (Abb. 4.71 a). Der Preis des einfachen Aufbaus und des geringen Flächenbedarfs der Zellen ist eine deutlich kompliziertere Funktionsweise und Ansteuerung.



*Der Source ist bei einem NMOS-Transistor immer der Kanalanschluss mit dem niedrigeren und der Drain der mit dem höheren Potenzial.

Abb. 4.71. a) DRAM-Zelle b) Schreiben einer »0« c) Schreiben einer »1«

Beschreiben einer Speicherzelle

Zum Beschreiben der Speicherzelle wird auf der Bitleitung die Spannung zur Darstellung des Logikwertes angelegt

$$u_x = \begin{cases} 0 & \text{für } x = 0 \\ U_\mathrm{V} & \text{für } x = 1 \end{cases} \tag{4.40}$$

und der Transistor eingeschaltet. Beim Schreiben einer »0« arbeitet der Transistor ganz normal als Low-Side-Schalter (Abb. 4.71 b). Der Source, d.h. der Kanalanschluss mit dem niedrigeren Potenzial, ist der Leseleitungsanschluss und hat das Potenzial 0 V. Die Lesegeschwindigkeit errechnet sich nach demselben Modell wie die Ausschaltzeit t_aus eines Inverters (Gleichung 4.20). Beim Aufladen der Lastkapazität hat die Kapazitätsseite des Kanals das niedrigere Potenzial und bildet den Source. Die Gate-Drain-Spannung ist null, so dass der Transistor während des gesamten Aufladevorgangs im Abschnürbereich arbeitet (Abb. 4.71 c). Die Spannung über der Speicherkapazität strebt nicht gegen die Versorgungsspannung, sondern nur gegen

$$u_\mathrm{CS} \leq U_\mathrm{V} - U_\mathrm{TN} \tag{4.41}$$

(U_TN – Einschaltspannung des Auswahltransistors). Der Aufladestrom ist deutlich kleiner als beim Aufladen über einen PMOS-Transistor mit vergleichbaren Parametern, so dass das Schreiben einer »1« vergleichsweise lange dauert [30].

Lesen

Der Lesevorgang ist noch komplizierter. Vor dem Lesen wird die Ladung auf der Bitleitung gelöscht:

$$Q_{\mathrm{Cx}}^{(-)} = C_{\mathrm{x}} \cdot U_{\mathrm{x}}^{(-)} = 0$$

Die Speicherkapazität hat vor dem Lesen die Ladung

$$Q_{\mathrm{CS}}^{(-)} = C_{\mathrm{S}} \cdot \begin{cases} 0 & \text{für eine gespeicherte »0«} \\ U_{\mathrm{V}} - U_{\mathrm{TN}} & \text{für eine gespeicherte »1«} \end{cases} \tag{4.42}$$

Anschließend wird der Transistor geöffnet. Es kommt zum Ladungsausgleich. Die gespeicherte Ladung geht dabei nicht verloren, sondern verteilt sich auf beide Kapazitäten:

$$Q_{\mathrm{CS}} + Q_{\mathrm{Cx}} = Q_{\mathrm{CS}}^{(-)} \tag{4.43}$$

Im stationären Zustand nach dem Einschalten des Transistors sind die Spannungsabfälle über beiden Kapazitäten gleich:

$$U_{\mathrm{CS}}^{(+)} = U_{\mathrm{x}}^{(+)} \tag{4.44}$$

Die Ausgabespannung auf der Bitleitung strebt gegen

$$U_{\mathrm{x}}^{(+)} = \frac{C_{\mathrm{S}}}{C_{\mathrm{S}} + C_{\mathrm{x}}} \cdot \begin{cases} 0 & \text{für eine gespeicherte »0«} \\ U_{\mathrm{V}} - U_{\mathrm{TN}} & \text{für eine gespeicherte »1«} \end{cases} \tag{4.45}$$

Die Kapazität C_{x} der Bitleitung ist um mindestens zwei Zehnerpotenzen größer als die Speicherkapazität C_{S}, so dass der Potenzialunterschied zwischen einer gelesenen »0« und einer gelesenen »1« nur wenige Millivolt beträgt (Abb. 4.72 a). Die Auswertung der Lesepotenziale auf den Bitleitungen erfolgt nach den Grundprinzipien des Analogentwurfs »Symmetrie und Kompensation«. In einer vollkommen symmetrischen Anordnung werden immer paarweise zwei

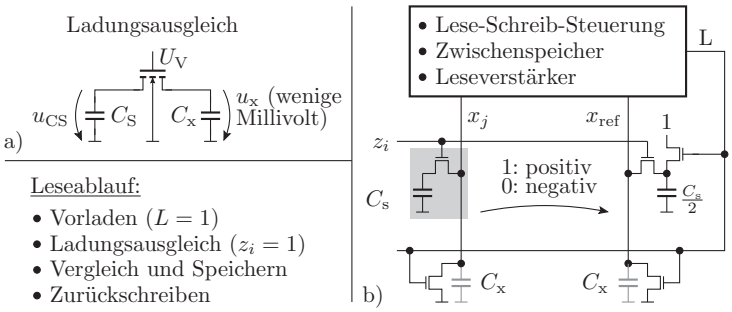


Abb. 4.72. Lesen einer DRAM-Zelle a) Ersatzschaltung für den Ladungsausgleich b) symmetrische Anordnung zur Auswertung der Lesepotenziale

Zellen gelesen, eine richtige und eine aufgeladene Dummy-Zelle mit der halben Kapazität. Wenn nach dem Ladungsausgleich das Potenzial der Leseleitung der richtigen Zelle größer als das der Leseleitung der Dummy-Zelle ist, wird eine »1« erkannt, sonst eine »0« (Abb. 4.72 b).

Beim Lesen wird der gespeicherte Wert zerstört, so dass jede Zelle nach dem Lesevorgang neu beschrieben werden muss. Der komplette Lesezyklus besteht praktisch aus vier Schritten:

- Entladen der Leseleitungen und Aufladen der Dummy-Zellen,
- Ladungsausgleich,
- Bestimmung der Logikwerte auf den Leseleitungen und Übernahme in den Zwischenspeicher und
- Zurückschreiben der gelesenen Inhalte.

Ein DRAM hat noch mindestens eine weitere Betriebsart, das Auffrischen (engl. refresh). Die Daten in den Speicherzellen bleiben nur wenige Millisekunden gültig (vgl. Abschnitt 4.3.1). Deshalb ist es notwendig, dass jede Speicherzelle innerhalb von wenigen Millisekunden einmal gelesen und der gelesene Inhalt zurückgespeichert wird. Damit das zeitlich möglich ist, erfolgt das Auffrischen und damit auch das Lesen nicht zellen-, sondern zeilenweise. Die Zeilenanzahl bestimmt die Bitleitungskapazität C_x. Sie darf, damit die Potenzialunterschiede auf den Bitleitungen beim Lesen noch sicher ausgewertet werden können, die Größenordnung Hundert bis Tausend nicht überschreiten. Dadurch gilt für alle DRAMs unabhängig von ihrer Organisation und Speichergröße, dass mindestens alle Hundert bis Tausend Speicherzugriffe ein Auffrischzyklus einzufügen ist.

Ansteuerung

DRAMs haben heute praktisch alle eine programmierbare synchrone Schnittstelle, die die komplexen internen Zeitabläufe verbirgt. Die Ansteuerung erfolgt über Befehle. Abbildung 4.73 zeigt einen vereinfachten Operationsablaufgraphen für die Ansteuerung eines DDR2-RAMs. Nach Zuschalten der Versorgungsspannung ist die Schnittstellenschaltung des Speichers zu initialisieren, bevor sie für Datenzugriffe bereit ist. Das beinhaltet die Initialisierung der Regelkreise für die interne Takterzeugung und dauert einige Hundert Takte (vgl. Abschnitt 4.3.6). Im Zustand »bereit« akzeptiert die Schnittstellenschaltung zwei Kommandos:

- ACT (**act**ivate): Lesen einer Speicherzeile in den Zwischenpuffer und
- SFR (**ref**resh): Lesen, Verstärken und Zurückschreiben.

Das Activate-Kommando benötigt zusätzlich die Zeilenadresse, das Refresh-Kommando entnimmt die Zeilenadresse einem internen Zähler, der nach jedem Refresh weitergeschaltet wird. Beide Operationen dauern mehrere Takte.

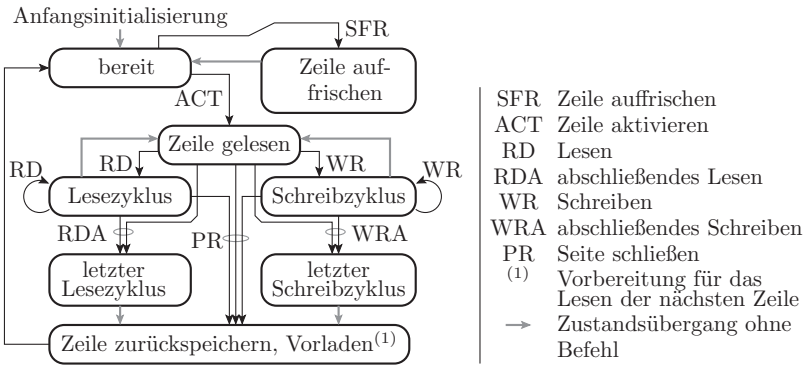


Abb. 4.73. Vereinfachter Operationsablaufgraph für die Ansteuerung eines DDR2-SDRAMs

Im Zustand »Zeile gelesen« wird nur noch mit dem Zwischenpuffer gearbeitet. Dieser verhält sich ähnlich wie ein wahlfrei adressierbarer Schreib-Lese-Speicher. Wenn eine Zeile im Zwischenpuffer steht, kann der Speicherschaltkreis folgende Kommandos ausführen:

- RD, RDA: Lesen, abschließendes Lesen,
- WR, WRA: Schreiben, abschließendes Schreiben und
- PR (**pr**echarge) Seite schließen.

Mit den Schreib- und Lesebefehlen ist zusätzlich die Spaltenadresse zu übergeben. Nach den Abschlussoperationen RDA, WRA und PR werden die Werte aus dem Zwischenspeicher zurück in die Speicherzeile geschrieben, die Bitleitungen entladen und die Dummy-Zellen aufgeladen. Danach ist der Schaltkreis wieder für eine Auffrischoperation oder die Aktivierung einer anderen Zeile bereit.

Die Lese- und Schreibzugriffe auf einen DRAM erfolgen in der Regel Burst-orientiert. Bei jedem Zugriff werden die Daten für z.B. vier oder acht aufeinanderfolgende Adressen übertragen. In Abb. 4.74 wird zuerst mit dem Activate-Befehl und der Übergabe der Zeilenadresse der Lesevorgang einer Speicherzeile gestartet. Das dauert typisch zwei bis drei Takte. Anschließend wird mit einem Lesebefehl und der Übergabe einer Spaltenadresse eine Leseoperation gestartet. Danach vergehen weitere zwei bis drei Takte, bevor der Schaltkreis eine Datenfolge schickt. Die Daten wechseln sowohl nach der fallenden als auch nach der steigenden Flanke, so dass in jeder Taktperiode zwei Datenworte übertragen werden. Letzterem verdankt das Schnittstellenprotokoll seinen Namen (DDR – **d**ouble **d**ate **r**ate). Der Zeitablauf einer Schreiboperation ist ähnlich. Nach Übersenden des Schreibkommandos und der Spaltenadresse vergeht eine definierte Anzahl von Takten, nach denen die zu schreibenden Daten mit der doppelten Taktrate auf den Datenbus zu legen sind.

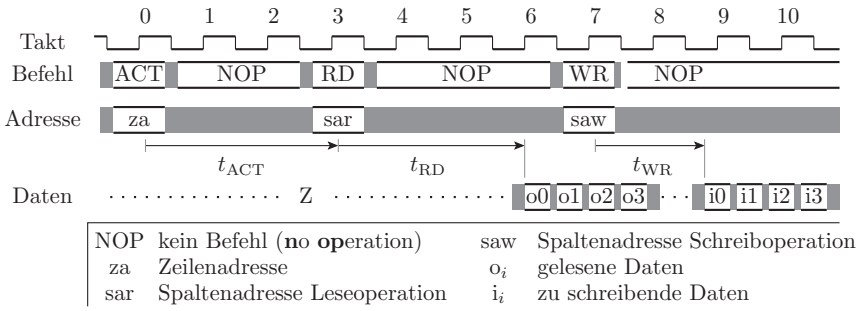


Abb. 4.74. Datenaustausch über eine DDR2-Schnittstelle

Ein einzelner Lese- oder Schreibzugriff auf einen DRAM-Schaltkreis dauert mehrere Takte, aber es ist im günstigsten Fall fast möglich, in jedem Takt zwei Datenworte zu übertragen. Der hohe Datendurchsatz lässt sich jedoch nur nutzen, wenn die Daten in größeren Blöcken gelesen und gespeichert werden. Schaltungen mit DRAMs als Speicher für große Datenmengen besitzen deshalb meist schnelle Zwischenspeicher, die aus den DRAM-Schaltkreisen blockweise geladen und deren Daten einzeln weiterverarbeitet werden. Umgekehrt werden die Daten in den schnellen Zwischenspeichern einzeln berechnet und blockweise in die DRAMs zurückgeschrieben. In Rechnern sind diese Zwischenspeicher die Cache-Speicher. Weiterführende und ergänzende Literatur siehe [36, 47].

4.5 Festwertspeicher

Festwertspeicher, abgekürzt ROM (**r**ead **o**nly **m**emory), können nur einmal oder nur mit großem Zeitaufwand beschrieben werden und behalten ihre Daten auch ohne Versorgungsspannung über Jahre. Das Speicherelement ist heute meist ein einzelner Transistor, der entweder ein- und ausschaltbar ist oder nur einen der beiden Schaltzustände besitzt. Bei einem NOR-ROM sind die Transistoren, die die Speicherzellen bilden, parallel geschaltet. Die deaktivierten Transistoren schalten nicht ein. Die Zeilenauswahlschaltung steuert pro Spalte nur einen Transistor mit »1« und alle anderen mit »0« an. Die gesamte Parallelschaltung ist leitend, wenn der ausgewählte Transistor einschaltet. Die Pull-Up-Elemente der Spalten wandeln die Schaltzustände in Logikwerte um (Abb. 4.75). Es gibt noch weitere Organisationsformen.

Die Deaktivierung der Transistoren kann bei der Herstellung oder beim Anwender erfolgen, nur einmal oder mehrmals möglich sein etc. [30]. Das Verhaltensmodell eines Festwertspeichers ist eine initialisierte Bitvektor-Konstante, deren Elemente mit der Adresse als Index gelesen werden (vgl. Abschnitt 3.4.3). Die Zugriffszeit ist wie bei anderen Blockspeichern um Zehnerpotenzen größer als die Verzögerung eines Logikgatters. Denn auch nur lesbare

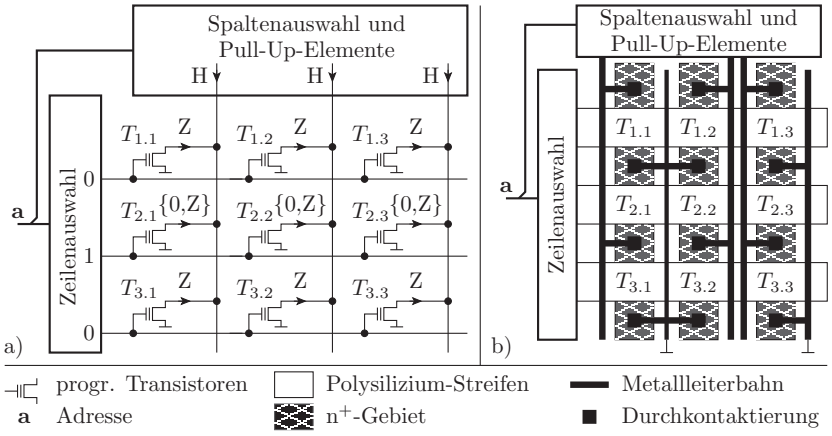


Abb. 4.75. Festwertspeicher a) Schaltung b) geometrische Anordnung

Speicher besitzen Zeilen- und Spaltenleitungen mit großen Lastkapazitäten, die von verhältnismäßig schmalen Transistoren angesteuert werden.

4.6 Programmierbare Logikschaltkreise

Ein programmierbarer Logikschaltkreis besteht aus programmierbaren Logikblöcken, programmierbaren Verbindungsnetzwerken und programmierbaren Eingabe-Ausgabe-Schaltungen (Abb. 4.76). Größere programmierbare Logikschaltkreise enthalten zusätzlich konfigurierbare Taktversorgungsschaltungen, Blockspeicher, Rechenwerke und Prozessorkerne. Die Funktion wird über Konfigurationsspeicher festgelegt. Kostengünstige programmierbare Logikschaltkreise können inzwischen Schaltungen nachbilden, die in nicht programmierbaren Schaltkreisen aus Millionen von Transistoren bestehen. Eingesetzt werden programmierbare Logikschaltkreise in Prototypen, Kleinserien, studentischen Praktika und auch zunehmend, um Hardware umprogrammieren zu können.

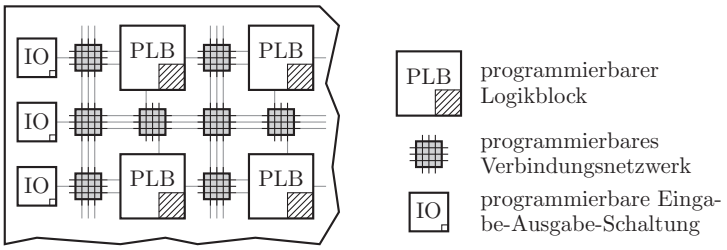


Abb. 4.76. Grundstruktur eines programmierbaren Logikschaltkreises

4.6.1 Programmierbare Tabellenfunktionen

Eine programmierbare Tabellenfunktion ordnet jedem Eingabewert einen individuell einstellbaren Ausgabewert zu. Auf diese Weise lässt sich jede kombinatorische Funktion nachbilden. Die Schaltung ähnelt einem Speicher mit wahlfreiem Zugriff, einem Festwertspeicher oder einem RAM. Der Zeilendecoder – eine UND-Matrix – aktiviert für jede Eingabevariation genau eine der 2^n Zeilenauswahlsignale. Die Zeilenauswahlsignale werden als Produktterme bezeichnet. Die ODER-Matrix, die für jedes Ausgabebit auswählt, bei welchen aktivierten Produkttermen eine »1« ausgegeben wird, ist programmierbar (Abb. 4.77).

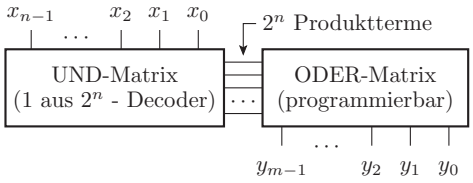


Abb. 4.77. UND-ODER-Matrix zur Programmierung von Tabellenfunktionen

Abbildung 4.78 zeigt eine mögliche Realisierung. Die schwarzen Querstriche an den Kreuzungspunkten der Zeilen- und Spaltenleitungen in der UND-Matrix sind normale NMOS-Transistoren, die bei einer »1« am Gate einschalten. An den anderen Kreuzungspunkten befindet sich entweder kein oder ein deaktivierter Transistor, der auch bei einer »1« am Gate ausgeschaltet bleibt (vgl. Abschnitt 4.5). Das PMOS-Netzwerk ist durch Pull-Up-Elemente ersetzt. Die Treiber zwischen den Ausgängen der UND-Matrix und den Eingängen der ODER-Matrix sind Puffer zur Verringerung der Verzögerungszeiten (vgl. Abschnitt 4.2.7). Die ODER-Matrix besteht aus programmierbaren Transistoren. Um bei der Auswahl einer Zeile am Ausgang eine »0« auszugeben, ist

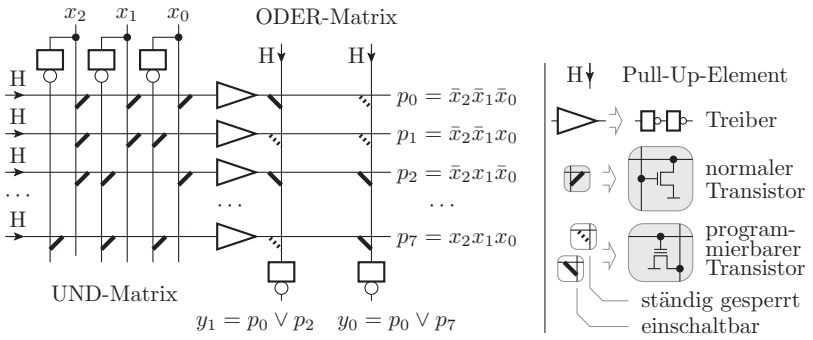


Abb. 4.78. Logikschaltung mit programmierter ODER-Matrix

der zugeordnete Transistor zu deaktivieren. Die Inverter an den Ausgängen bewirken, dass bei einem eingeschalteten (ausgewählten, nicht deaktivierten) Transistor eine »1« ausgegeben wird.

Eine Tabellenfunktion mit n Eingängen und m Ausgängen benötigt $m \cdot 2^n$ Programmierelemente. Eine einzelne Tabellenfunktion ist entsprechend nur für die Nachbildung von Schaltungen mit wenigen Eingängen geeignet. Größere Schaltungen werden aus mehreren Tabellenfunktionen zusammengesetzt. Das erfordert zusätzlich ein programmierbares Verbindungsnetzwerk. Dieses ist eine Matrix aus deaktivierbaren Treibern oder Transfergattern mit Programmierstellen an den Steuereingängen. Abbildung 4.79 zeigt als Beispiel die Aufspaltung der fünfstelligen Logikfunktion

$$y = x_1 x_2 \left(x_3 \vee x_4 \vee x_5\right) \tag{4.46}$$

in zwei kleinere Tabellenfunktionen mit je nur drei Eingängen:

$$\begin{aligned} &\text{Tabellenfunktion 1} : z = x_3 \vee x_4 \vee x_5 \\ &\text{Tabellenfunktion 2} : y = x_1 x_2 z \end{aligned} \tag{4.47}$$

Bei der Technologieabbildung mit Tabellenfunktionen als Bausteine werden die minimierten logischen Ausdrücke oder Entscheidungsdiagramme in Teilfunktionen mit der Eingangsanzahl des Tabellentyps und einem Ausgang aufgeteilt.

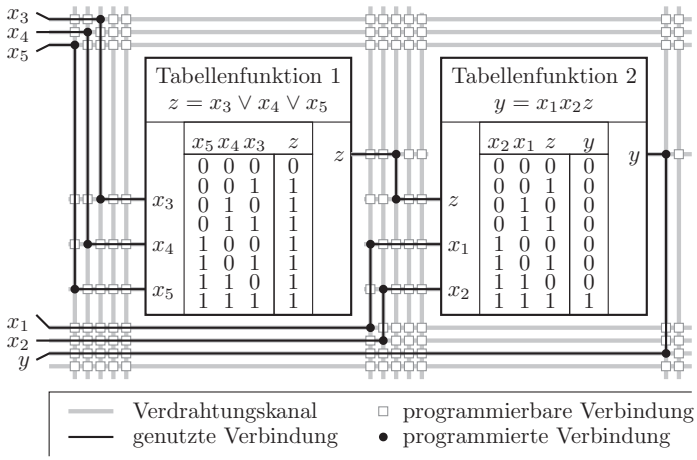


Abb. 4.79. Programmierte Logikschaltung mit zwei Tabellenfunktionen

4.6.2 Programmierbare UND-Matrix

Statt der ODER-Matrix kann auch die UND-Matrix programmierbar ausgeführt sein. Die ODER-Matrix ist dann die ODER-Verknüpfung aller Pro-

duktterme (Abb. 4.80). Die nachzubildende Funktion wird mit Hilfe von KV-Diagrammen oder des Verfahrens von Quine und McCluskey minimiert (vgl. Abschnitt 2.2). Die begrenzende Ressource für die Größe der programmierbaren Funktionen ist hier die Eingangsanzahl der UND-Matrix (Größenordnung 8 . . . 16) und die maximale Anzahl der Produktterme (Größenordnung 4 . . . 16).

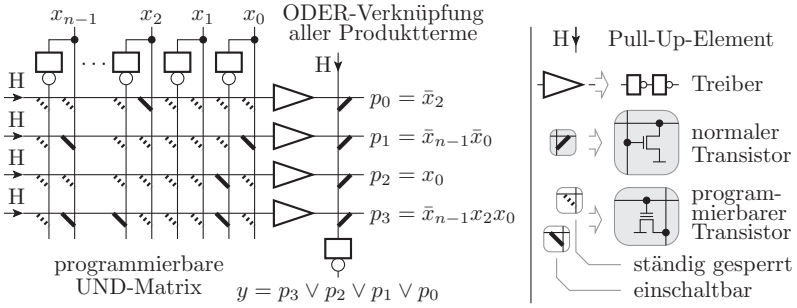


Abb. 4.80. Logikschaltung mit programmierter UND-Matrix

4.6.3 Weitere programmierbare Elemente

Eine typische Erweiterung einer Logikfunktion mit programmierbarer UND-Matrix ist eine programmierbare Ausgabeinvertierung. Sie besteht aus einem EXOR-Gatter mit einer Programmierstelle am zweiten Eingang (Abb. 4.81). Bei einer programmierten »0« liefert die EXOR-Verknüpfung den Wert selbst und bei einer »1« den invertierten Wert der programmierten Logikfunktion. Auf diese Weise kann die nachzubildende Funktion wahlweise nach den Nullen oder Einsen entwickelt werden (vgl. Abschnitt 2.2.4).

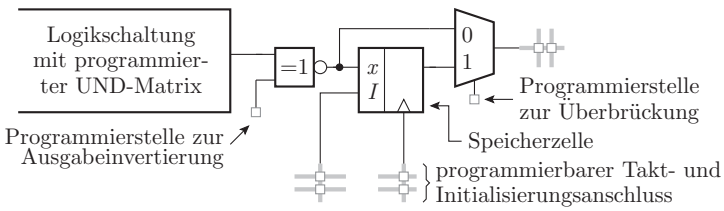


Abb. 4.81. Erweiterte programmierbare Funktionseinheit

Digitale Schaltungen benötigen immer eine größere Anzahl von Speicherzellen für die Abtastung und Zwischenspeicherung von Daten. In der üblichen Schaltungsarchitektur ist hinter jeder programmierbaren Logikfunktion eine

überbrückbare taktflankengesteuerte Speicherzelle angeordnet, die bei Bedarf in den Signalfluss eingefügt und mit Steuersignalen verbunden werden kann. Konfigurierbare Blockspeicher haben eine bestimmte Speicherkapazität und eine bestimmte Portanzahl. Konfigurierbar sind z.B. die genutzte Datenbreite, die genutzte Speichertiefe und die Anzahl der genutzten Ports. Bei konfigurierbaren Taktversorgungsschaltungen mit Phasenregelkreisen sind die Teilerverhältnisse und die Phasenverschiebungen der Ausgabetakte einstellbar (vgl. Abschnitt 4.3.6). Bei konfigurierbaren Rechenwerken und Prozessorkernen gibt es Bitbreitenoptionen und die Möglichkeiten, nur Teile davon zu nutzen.

4.6.4 Zusammenfassung

Programmierbare Logikschaltkreise bestehen aus programmierbaren Logikblöcken, programmierbaren Verbindungsnetzwerken, programmierbaren Eingabe-Ausgabe-Schaltungen und weiteren programmierbaren Elementen wie überbrückbaren Speicherzellen, programmierbaren Taktversorgungsschaltungen und programmierbaren Blockspeichern. Für sie wird ein Schaltungsentwurf nicht in Fertigungs-, sondern in Konfigurationsdaten übersetzt, mit denen die einzelnen Konfigurationsbits – Tabellenwerte für Logikfunktionen, Multiplexerauswahlwerte für die Überbrückung oder Einfügung von Speicherzellen etc. – programmiert werden. Auf diese Weise lässt sich die Funktion von Hardware genauso schnell und unkompliziert ändern wie die von Software. Weiterführende und ergänzende Literatur siehe [14, 18, 31, 43].

5

Komplexe Beispielentwürfe

In den vergangenen Kapiteln wurde gezeigt, wie die unterschiedlichen Bestandteile einer digitalen Schaltung realisiert, modelliert und simuliert werden. Dieses Kapitel führt anhand von drei Beispielen in den Entwurf komplexer Systeme ein. Zuvor werden zwei strukturbestimmende Aspekte für die Nachbildung rechenzeitintensiver Algorithmen in Hardware behandelt.

5.1 Pipeline-Verarbeitung und Speicherengpass

Eine wesentliche Zielgröße einer digitalen Schaltung ist ihre Verarbeitungsleistung. Sie wird durch die Anzahl der Operationen, die pro Zeit fertiggestellt werden, beschrieben. Um die Verarbeitungsleistung über die der bisher betrachteten sequenziellen Verarbeitung zu erhöhen, gibt es zwei Möglichkeiten:

- Parallelverarbeitung mit mehr Hardware und
- Pipeline-Verarbeitung.

Die Pipeline-Verarbeitung ist eine Art zeitversetzte Parallelverarbeitung, bei der statt zusätzlicher Hardware die vorhandene Hardware besser ausgelastet wird. Es genügt dabei jedoch nicht, nur die Verarbeitungswerke zu betrachten. Denn den Engpass bilden in der Regel die Blockspeicher, aus denen die zu verarbeitenden Eingabedaten bereitgestellt und in die die Ergebnisse geschrieben werden.

5.1.1 Prinzip der Pipeline-Verarbeitung

Pipeline-Verarbeitung heißt auf Deutsch Fließbandverarbeitung. Eine umfangreiche Aufgabe wird in eine Folge von nacheinander auszuführenden Arbeitsschritten mit vergleichbar großem Arbeitsaufwand aufgeteilt. Das zu bearbeitende Objekt fährt auf dem Fließband von einer Arbeitsstation zur nächsten. An jeder Station wird ein Arbeitsschritt ausgeführt. Die Fertigstellung

G. Kemnitz, *Technische Informatik*, eXamen.press,
DOI 10.1007/978-3-642-17447-6_5, © Springer-Verlag Berlin Heidelberg 2011

eines Einzelproduktes geht dadurch nicht schneller. Aber es wird gleichzeitig an N_P Objekten gearbeitet. In jedem Fließbandtakt rücken die zu bearbeitenden Objekte eine Station weiter. Am Anfang kommt ein neues unbearbeitetes Objekt auf das Fließband und am Ende verlässt ein fertiges Produkt das Fließband. Bei einer Aufteilung in N_P Teilaufgaben mit derselben Bearbeitungsdauer erhöht sich die Verarbeitungsleistung auf das N_P-fache (Abb. 5.1).

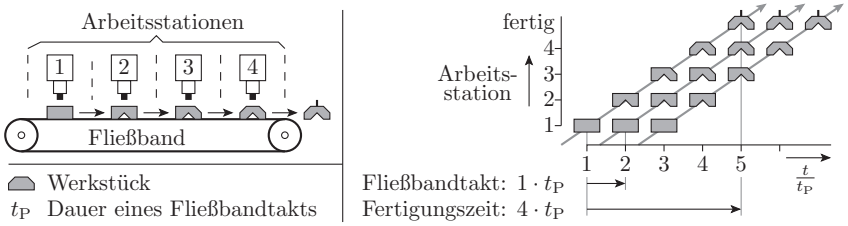


Abb. 5.1. Prinzip der Pipeline-Verarbeitung

In einer digitalen Schaltung ist das zu fertigende Produkt das Ergebnis einer Berechnung. Die Arbeitsstationen sind die nacheinander auszuführenden Operationen. Nach jeder Operation wird das Zwischenergebnis von einem Register abgetastet und an die nächste Operation weitergereicht. Die Teilung der Gesamtoperation in Schritte reduziert die Ausführungszeit der einzelnen Register-Transfer-Operationen. Die Gesamtschaltung kann idealerweise mit der mehrfachen Taktfrequenz arbeiten. Genau wie bei einem richtigen Fließband bleibt die Ausführungszeit insgesamt mindestens so groß wie bei einer sequenziellen Verarbeitung. Aber es werden mehr Ergebnisse pro Zeit fertig (Abb. 5.2).

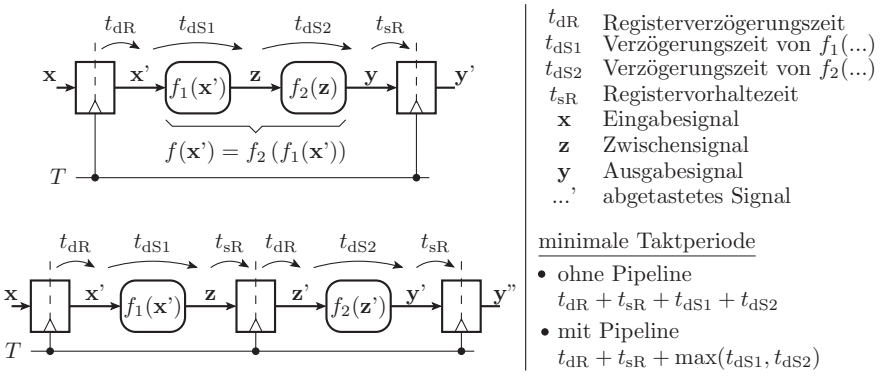


Abb. 5.2. Aufteilung einer Register-Transfer-Funktion in zwei Pipeline-Phasen

In VHDL wird eine Pipeline durch eine Folge von Signalzuweisungen in einem Abtastprozess beschrieben. Für die Schaltung in Abb. 5.2 unten lautet die Prozessbeschreibung z.B.

```
process(T)
 if RISING_EDGE(T) then
   x_del  <= x;
   z_del  <= f1(x_del);
   y_del2 <= f2(z_del);
 end if;
end process;
```

Die erste Signalzuweisung beschreibt die Abtastung des Eingabesignals. Der Funktionsaufruf in der zweiten Signalzuweisung beschreibt die erste kombinatorische Verarbeitungsschaltung und die Zuweisung selbst das nachfolgende Zwischenregister. Die dritte Signalzuweisung beschreibt die Verarbeitung des abgetasteten Zwischensignals durch die zweite kombinatorische Teilschaltung und die Abtastung mit dem Ausgaberegister.

5.1.2 Ausbalancieren einer Pipeline

Das Black-Box-Modell einer Pipeline ist eine Register-Transfer-Operation, deren Ergebnis erst nach N_P Verarbeitungsschritten für die Weiterverarbeitung oder die Ausgabe zur Verfügung steht (N_P – Pipeline-Tiefe). Die Mindestlänge der Taktperiode, mit der eine Pipeline betrieben werden kann, und damit ihre maximale Verarbeitungsleistung, wird von der längsten Register-Register-Verzögerung bestimmt. Zur Minimierung der maximalen Verzögerung müssen die Verzögerungszeiten der Operationen zwischen den Registern aneinander angeglichen werden. Dieser Angleich wird als Ausbalancieren der Pipeline bezeichnet. Zum Ausbalancieren werden alle Register, die ihre Daten nach derselben Anzahl von Abtastschritten in Bezug auf den Operationsstart übernehmen, mit einer Zeitschnittlinie verbunden. Dann werden die Register in dem Datenflussgraphen so verschoben, dass die maximale Verzögerung zwischen zwei aufeinanderfolgenden Zeitschnittlinien ein Minimum oder einen hinreichend kleinen Wert erreicht.

In Abb. 5.3 a bilden die Abtastregister ein Schieberegister am Ausgang der Verarbeitungsfunktion. Die Verzögerung vom Eingang bis zum ersten Register t_d1a ist sehr groß und die Verzögerungen zwischen den angehängten Abtastregistern sind fast null. Die zusätzlichen Register verzögern die Ergebnisfertigstellung, ohne die Verarbeitungsleistung zu erhöhen. Abbildung 5.3 b zeigt eine Schaltung mit demselben Anschlussverhalten und einer ausbalancierten Pipeline, in der die Verzögerungen zwischen den Registern nur etwa ein Drittel der ursprünglichen maximalen Verzögerung t_d1a in Abb. 5.3 a betragen. Die geänderte Schaltung kann etwa mit der dreifachen Frequenz getaktet werden und besitzt dann die dreifache Verarbeitungsleistung.

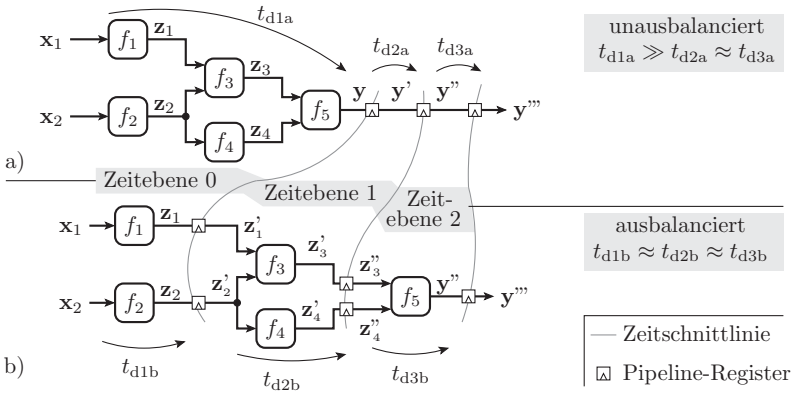


Abb. 5.3. Pipeline-Einbau a) Anhängen von Abtastregistern an den Ausgang b) Ausbalancieren der Pipeline

5.1.3 Engpass Speicher

Für eine schnelle Verarbeitung und Übertragung müssen die Eingabedaten mit der Verarbeitungsgeschwindigkeit bereitgestellt und die Ergebnisse nach ihrer Berechnung abgeholt werden. Die Datenquellen und die Empfänger für die Ergebnisse sind in der Regel Blockspeicher. Die Zugriffszeit auf einen größeren Blockspeicher übersteigt die Verzögerung von Logikgattern um Zehnerpotenzen und wächst mit der Datenkapazität (vgl. Abschnitt 4.4.1) [23]. Eine Vergrößerung der transferierten Datenmenge verlangt mehrere parallele Speicher, Mehrportspeicher, eine Erhöhung der Datenwortbreite oder andere schaltungsaufwändige Maßnahmen. Der Entwurf von schnellen Verarbeitungswerken beginnt deshalb in der Regel mit der Planung der Speicherstruktur, einer Ablaufplanung der Speicherzugriffe und einer Planung der Verarbeitungs-Pipelines. Die folgenden Abschnitte demonstrieren das an drei komplexen Beispielentwürfen.

5.2 FIR-Filter mit Blockspeichern

Ein FIR-Filter ist ein Filter mit endlicher Impulsantwort (FIR – **f**inite **i**mpulse **r**esponse) und ein wichtiger rechenzeitintensiver Algorithmus der digitalen Signalverarbeitung. Typische Anwendungen sind die Trennung der Spektralanteile in analogen Signalen, die Echo-Unterdrückung bei der Signalübertragung über lange Leitungen und die Bildverarbeitung. Trotz der vielfältigen Anwendungen ist die Funktion eines FIR-Filters einfach. Jeder Ausgabewert y_n ist eine gewichtete Summe von M aufeinanderfolgenden Eingabewerten:

$$y_n = \sum_{m=0}^{M-1} c_m \cdot x_k \quad \text{mit} \quad k = n - m \tag{5.1}$$

(x_k – Eingabewert; y_n – Ausgabewert; n – Nummer des Zeitschritts; c_m – Koeffizient; M – Länge der Impulsantwort, Größenordnung 100). Die Filterfunktion – Hochpass, Tiefpass, Glättungsfilter etc. – legt der Vektor der Koeffizienten c_m fest[1]. Es gibt Anwendungen in der Signalverarbeitung, in denen viele Millionen Ergebnisse pro Sekunde zu berechnen sind. Jede Ergebnisberechnung erfordert $2 \cdot M$ Speicherzugriffe mit den zugehörigen Adressrechnungen, M Multiplikationen und $M - 1$ Additionen. Für Anwendungen mit einem kontinuierlichen Rechenleistungsbedarf von 10^8 und mehr Operationen pro Sekunde ist eine Hardware-Lösung eine interessante Alternative zu einer Software-Lösung.

5.2.1 Planung der Speicherstruktur und der Pipeline-Abläufe

Abbildung 5.4 a zeigt eine Skizze für den Berechnungsfluss, den Gleichung 5.1 beschreibt. Für jeden Ausgabewert y_n werden M Eingabewerte mit M Koeffizienten multipliziert und die Produkte aufsummiert. Es fehlt der Datenspeicher, aus dem die Eingabedaten und Koeffizienten gelesen werden. Wie könnte der Speicher organisiert sein? Ein M-Port-Blockspeicher scheidet wegen der Größe von M hier aus. Eine nahe liegende Lösung wäre ein Schieberegister der Länge M, das die Eingabewerte abtastet und an seinen Ausgängen die um einen bis M Schritte verzögerten Eingabewerte bereitstellt (Abb. 5.4 b). An jedem Schieberegisterausgang ist ein Multiplizierer anzuschließen, der das Produkt $c_m \cdot x_k$ bildet. Der zweite Faktor ist eine Konstante, die fest codiert sein oder in einem Konfigurationsregister stehen kann. Die Teilprodukte werden mit einem Addiererbaum zusammengefasst. Die Umsetzung in eine VHDL-Beschreibung ist relativ einfach, aber die Schaltung wird sehr groß[2].

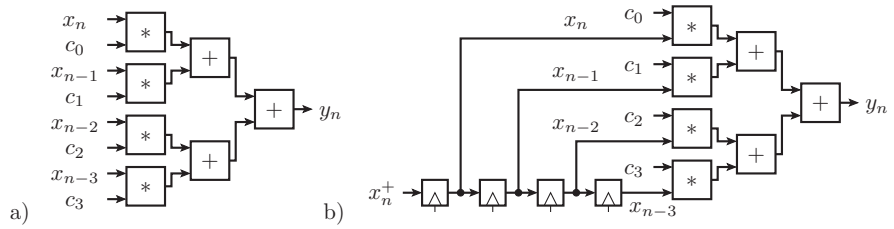


Abb. 5.4. Datenfluss für eine FIR-Operation mit $M = 4$ a) ohne Operandenspeicher b) mit einem Schieberegister als Operandenspeicher

[1] Der Koeffizientenvektor wird auch als Impulsantwort bezeichnet. Denn er ist gleich mit der Ausgabefolge für einen Impuls der Form $x_0 = 1{,}0$ und $x_{n\neq 0} = 0$ als Eingabefolge.

[2] Man würde zumindest versuchen, die Multipliziereranzahl durch Einbau von Pipeline-Stufen in die Multiplizierer drastisch zu verringern. Das erfordert eine vollkommen andere Schaltungsstruktur.

Wir wollen hier statt eines Schieberegisters, das für einen größeren Filtergrad M recht schaltungsaufwändig ist, einen wesentlich kompakteren Blockspeicher verwenden. Aus einem (1-Port-) Blockspeicher kann in jedem Zeitschritt nur ein Datenwort gelesen werden. Das ist der Flaschenhals für die Berechnung. Jetzt folgt ein Puzzle, in dem die Operationen auf Rechenwerke und Berechnungsschritte aufgeteilt werden. Die Berechnung für jeden Ausgabewert besteht aus M Teilberechnungen. Fast jede der Teilberechnungen beginnt mit der Erhöhung der Koeffizientenadresse um eins und der Erhöhung der Datenadresse um eins. Danach werden ein Koeffizient und ein Datenwert aus dem Speicher gelesen. Anschließend werden beide Werte miteinander multipliziert und das Ergebnis zur bisherigen Teilsumme addiert. Mit den Koeffizienten und Daten im selben Speicher müsste der Speicher entweder zwei Ports haben oder die beiden Werte wären nacheinander zu lesen. Die eine Lösung ist aufwändig und die andere halbiert die Verarbeitungsleistung. Die nahe liegende Alternative ist der Einsatz von zwei Blockspeichern, einem Schreib-Lese-Speicher für die Daten und einem Nur-Lese-Speicher für die Koeffizienten. Das Zwischenergebnis der Überlegungen bis hierher ist ein Berechnungsfluss, der mit zwei parallelen Adressrechnungen beginnt. Danach folgen die zwei parallelen Speicherzugriffe. Die gelesenen Werte werden multipliziert und das Produkt zum bisher akkumulierten Wert addiert. Das ist die Kernoperation, die für jeden zu berechnenden Ausgabewert mit geringfügigen Modifikationen M-mal hintereinander ausgeführt werden muss.

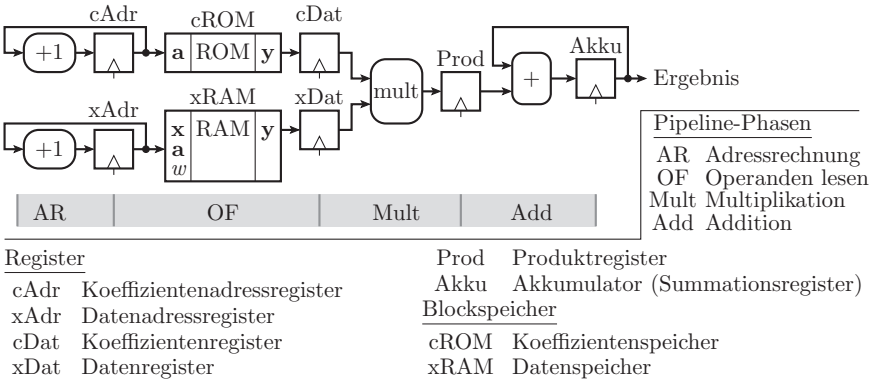


Abb. 5.5. Blockspeicher-Pipeline-Struktur der Kernfunktion des FIR-Filters

Als Nächstes soll die Kernoperation in Pipeline-Phasen unterteilt werden. Die Blockspeicherzugriffe, die den zeitlichen Engpass bilden, bekommen eine eigene Pipeline-Phase. Dasselbe gilt für die Multiplikation, die auch relativ viel Zeit benötigt[3]. Die verbleibenden Adressrechnungen zu Beginn des Be-

[3] Wenn die Multiplikation länger als der Speicherzugriff dauert, kann sie auch auf mehrere Pipeline-Phasen aufgeteilt werden.

rechnungsflusses und die Akkumulation der Produkte nach der Multiplikation erhalten gleichfalls je eine eigene Pipeline-Phase. Das Zwischenergebnis des Entwurfsprozesses ist die 4-stufige Verarbeitungs-Pipeline in Abb. 5.5. In der ersten Pipeline-Phase »AR« (**A**dress**r**echnung) werden die Adressen weitergeschaltet. In der zweiten Phase »OF« (**o**perand **f**etch) werden der Datenwert und der Koeffizient gelesen. In der nächsten Phase »Mult« (**Mult**iplikation) wird multipliziert und in der letzten Phase »Add« (**Add**ition) werden die Produkte aufsummiert.

5.2.2 Der gesamte Operationsablauf

Nach der Skizzierung der Kernoperation ist der Operationsablauf zu präzisieren. Nach der Initialisierung der Adressregister sind die ersten $M-1$ Eingabewerte x_0 bis x_{M-2} in den Datenspeicher zu schreiben. Mit dem Schreiben des Eingabewertes x_{M-1} beginnt die Berechnung des ersten Ausgabewertes. Dieser Wert wird gleichzeitig in das Datenregister »xDat« und der zugehörige Koeffizient c_0 in das Koeffizientenregister »cDat« übernommen. Im nächsten Schritt wird das Produkt des ersten Koeffizient-Wert-Paares gebildet und gleichzeitig das zweite Koeffizient-Wert-Paar aus den Blockspeichern gelesen. Danach wird das erste Produkt in das Akkumulatorregister übernommen, das zweite Produkt gebildet und das nächste Koeffizient-Wert-Paar aus dem Speicher gelesen etc. (Abb. 5.6). Die Koeffizientenadresse zählt immer zyklisch weiter. Die Daten stehen in absteigender Reihenfolge im Speicher und die Datenadresse zählt außer im letzten Schritt eines jeden Zyklus vorwärts. Nach dem letzten Schritt bleibt die Datenadresse unverändert und der älteste Wert wird mit dem neusten Wert überschrieben. Nach dem Startzyklus steht immer, während der Koeffizient »c2« gelesen wird (»cAdr=2«), ein abholbereites Ergebnis im Akkumulator.

cAdr	1	2	3	0	1	2	3	0	1	2	3	0	
xAdr	3	2	1	0	1	2	3	3	0	1	2	2	
xRAM	W(x_0)	W(x_1)	W(x_2)	W(x_3)	R(x_2)	R(x_1)	R(x_0)	W(x_4)	R(x_3)	R(x_2)	R(x_1)	W(x_5)	
cDat					c_0	c_1	c_2	c_3	c_0	c_1	c_2	c_3	
xDat					x_3	x_2	x_1	x_0	x_4	x_3	x_2	x_1	
Prod						$c_0 \cdot x_3$	$c_1 \cdot x_2$	$c_2 \cdot x_1$	$c_3 \cdot x_0$	$c_0 \cdot x_4$	$c_1 \cdot x_3$	$c_2 \cdot x_2$	
Akku							$c_0 \cdot x_3$	$\sum_{\ldots}^{1}$	$\sum_{\ldots}^{2}$	y_0	$c_0 \cdot x_4$	$\sum_{\ldots}^{1}$	
Zustand	Init			Startzyklus				Normalzyklus				Normal...	

Register		Blockspeicher	Speicheroperationen
cAdr Koeffizientenadressreg.	xDat Datenregister	cROM Koeffizientenspeicher	W(...) Speichern
xAdr Datenadressregister	Prod Produktreg.	xRAM Datenspeicher	R(...) Lesen
cDat Koeffizientenregister	Akku Akku.-Reg.		

Abb. 5.6. Beispielablauf der FIR-Operation für $M = 4$

Bearbeitungsstand: Halbformale Beschreibung der Blockspeicher-Pipeline-Struktur und des Operationsablaufs, aus denen in den weiteren Entwurfsschritten das Simulationsmodell zu entwickeln ist.

5.2.3 Datentypen und Bearbeitungsmethoden

Eine komplexe Funktionsbeschreibung sollte immer in überschaubaren Schritten, nach denen jeweils eine Zwischenkontrolle erfolgt, entwickelt werden. In Anlehnung an die objektorientierte Programmierung empfiehlt es sich, mit der Definition der Typen der zu verarbeitenden Datenobjekte und den Unterprogrammen zu ihrer Verarbeitung zu beginnen (vgl. Abschnitt 3.4.1). Der VHDL-Beschreibungsrahmen für Datentypen und Unterprogramme ist ein Package und der Beschreibungsrahmen für Testabläufe ein Testrahmen.

Das erste Simulationsmodell einer zu entwerfenden Funktionseinheit dient in der Regel zur Kontrolle des Algorithmus (vgl. Abschnitt 3.4.5 und Abschnitt 3.4.6). Für die Darstellung der zu verarbeitenden Daten genügen in dieser Entwurfsphase wertebereichsbeschränkte reelle Zahlen und für die Adressen wertebereichsbeschränkte natürliche Zahlen:

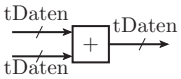
```
-- Typvereinbarungen zur Beschreibung eines FIR-Filters
constant M: POSITIVE := 4;                -- Filtergrad
subtype tAdr is NATURAL range 0 to M-1;
constant max_abs: REAL := 1000.0;         -- Wertebereichsgrenze
type tDaten is real range -(max_abs) to max_abs;
type tMem is array (NATURAL range <>) of tDaten;
```

Die zwei Adressregister und die vier Datenregister der Pipeline in Abb. 5.5 sollen zur gemeinsamen Übergabe an die Bearbeitungs-Prozeduren zu einem Verbund zusammengefasst werden:

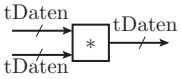
```
-- Datentyp für den Pipeline-Zustand
type tPipeline is record
 cAdr, xAdr: tAdr;
 cDat, xDat, Prod, Akku: tDaten;
end record;
```

Der Gesamtzustand des FIR-Filters besteht aus dem Datenspeicherzustand und dem Pipeline-Zustand.

Eine Aufgabe des Simulationsmodells ist die Kontrolle, dass bei den arithmetischen Operationen keine Wertebereichsüberläufe oder andere Situationen, die die Ergebnisse invalidieren, auftreten. Im Beispiel wird für die Einprogrammierung dieser Kontrollen der kleinste Wert des Zahlenbereichs als Pseudo-Wert für »ungültig« reserviert und als Initialwert, als Operationsergebnis für ungültige Operanden sowie bei einem Über- und Unterlauf des Zahlenbereichs zugewiesen. Die Operatoren für die Addition und die Multiplikation werden entsprechend überladen. Sie bilden zuerst aus den nach REAL konvertierten Operanden das Ergebnis und entscheiden dann anhand der Gültigkeit der Operanden und des Ergebniswertes, ob der Pseudo-Wert für »ungültig« oder das nach »tDaten« zurückkonvertierte Ergebnis zurückgegeben wird:




```
function "+"(a, b: tDaten) return tDaten is
  constant sum: REAL := REAL(a) + REAL(b);
 begin
  if a=tDaten'LOW or b=tDaten'LOW or (abs sum)>max_abs then
  return tDaten'LOW;   -- Pseudo-Wert für ungültig
  else return tDaten(sum);
  end if;
 end function;
```




```
function "*"(a, b: tDaten) return tDaten is
 constant prod: REAL := REAL(a) * REAL(b);
begin
 if a=tDaten'LOW or b=tDaten'LOW or (abs prod)>max_abs then
  return tDaten'LOW;   -- Pseudo-Wert für ungültig
 else return tDaten(prod);
 end if;
end function;
```

⇒WEB-Projekt: P5.2/FIR1_pack.vhdl

Als Nächstes sind alle Register-Transfer-Funktionen aus Abb. 5.6 auf Bearbeitungsmethoden aufzuteilen. Eine in Hardware nachzubildende Bearbeitungsmethode sollte nur Operationen zusammenfassen, die im selben Zeitschritt ausgeführt werden. In der Initialisierungsphase wird in jedem Zeitschritt die Koeffizientenadresse um eins erhöht, die Datenadresse um eins verringert und ein Eingabewert in den Datenspeicher geschrieben. Dieses Verhalten lässt sich mit einer Prozedur mit dem zu speichernden Datenwert als Eingabeparameter sowie mit dem Datenspeicher und dem Pipeline-Objekt als les- und veränderbare Signale beschreiben:

```
procedure InitStep(x: tDaten; signal xRAM: inout tMem;
                              signal pp: inout tPipeline) is
begin
 pp.cAdr <= (pp.cAdr + 1) mod M;
 pp.xAdr <= pp.xAdr - 1;
 xRAM(pp.xAdr) <= x;
end procedure;
```

cAdr	1	2	...	$M-1$
xAdr	$M-1$	$M-2$	...	1
xRAM	W(x_0)	W(x_1)	...	W(x_{M-2})

Die Register-Transfer-Operationen im Start- und im Normalzyklus sind einander sehr ähnlich (Abb. 5.7). Die Koeffizientenadresse wird modulo-M hochgezählt, das Koeffizientenregister übernimmt den adressierten Inhalt des Koeffizientenspeichers und das Produktregister das berechnete Produkt. Für die anderen Datenobjekte weicht die auszuführende Operation jeweils für eine Koeffizientenadresse von der Standardoperation ab. Der Datenspeicher wird bei Zählstand null beschrieben statt gelesen. Bei Zählstand zwei wird dem Akkumulator das Produkt statt der Summe zugewiesen und beim höchstwertigen Zählstand wird die Datenadresse nicht weitergezählt. Die in Abb. 5.7 mit einem grauen Dreieck gekennzeichneten Zuweisungen sind im Startzyklus

cAdr	0	1	2	...	$M-2$	$M-1$
xAdr	$+1 \bmod M$	$+1 \bmod M$	$+1 \bmod M$	...	$+1 \bmod M$	nicht zählen
xRAM	$\mathrm{W}(x_n)$	$\mathrm{R}(x_{n-1})$	$\mathrm{R}(x_{n-2})$	...	$\mathrm{R}(x_{n-M+2})$	$\mathrm{R}(x_{n-M+1})$
cDat	c_{M-1}	c_0	c_1	c_2	...	c_{M-2}
xDat	x_{n-M+1}	x_n	x_{n-1}	x_{n-2}	...	x_{n-M+2}
Prod	$c_{M-2} \cdot x_{n-M+2}$	$c_{M-1} \cdot x_{n-M+1}$	$c_0 \cdot x_n$	$c_1 \cdot x_{n-1}$	$c_2 \cdot x_{n-2}$	...
Akku	$\sum_{...}^{M-3}$	$\sum_{...}^{M-2}$	$y_n = \sum_{...}^{M-1}$	$c_0 \cdot x_n$	$\sum_{...}^{2}$	...

a)

cAdr	Koeffizientenadressregister	xDat	Datenregister	$x_{...}$	Datenwert
cROM	Koeffizientenspeicher	Prod	Produktregister	$c_{...}$	Koeffizient
cDat	Koeffizientenregister	Akku	Akkumulationsregister	W(...)	Schreiben
xAdr	Datenadressregister		im Startzyklus optional	R(...)	Lesen
xRAM	Datenspeicher		geänderter Datenfluss	M	Filtergrad

b)

```
procedure NormalStep(x: tDaten; cROM: tMem; signal xRAM:
  inout tMem; signal pp: inout tPipeline) is
begin
 pp.cAdr <= (pp.cAdr + 1) mod M;
 if pp.cAdr=0 then xRAM(pp.xAdr) <= x; pp.xDat <= x;
 else pp.xDat <= xRAM(pp.xAdr);
 end if;
 pp.cDat <= cROM(pp.cAdr); pp.Prod <= pp.cDat * pp.xDat;
 if pp.cAdr=2 then pp.Akku <= pp.Prod;
 else pp.Akku <= pp.Akku + pp.Prod;
 end if;
 if pp.cAdr/=M-1 then pp.xAdr <= (pp.xAdr + 1) mod M;
 end if;
end procedure;
```

⇒Web-Projekt:P5.2/FIR1_pack.vhdl

Abb. 5.7. Speicher- und Registerzuweisungen in allen Takten nach der Initialisierung a) tabellarisch b) als Bearbeitungsprozedur

nicht erforderlich. Aber sie stören auch nicht, so dass auf Fallunterscheidungen verzichtet werden kann. Die Unterbindung der Ausgabe im Startzyklus erfolgt in der Ausgabefunktion.

Bearbeitungsstand: Für die einzelnen Bestandteile und für die gesamte Blockspeicher-Pipeline-Struktur sind Datentypen definiert und für die Datentypen sind die benötigten Bearbeitungsmethoden programmiert.

5.2.4 Das erste komplette Simulationsmodell

Die erste simulierbare Beschreibung ist im Beispielprojekt ein Prozess im Testrahmen. Innerhalb des Testrahmens werden vereinbart:

- eine initialisierte Konstante für den Koeffizientenspeicher:
  ```
  constant cROM: tMem(tAdr) := (0.5, 1.0, -1.0, -0.5);
  ```
- ein Signal für den Datenspeicher

```
signal xRAM: tMem(tAdr);
```

- ein Signal für das Pipeline-Objekt
  ```
  signal pp: tPipeline;
  ```
- ein Dateiobjekt zum Einlesen der Eingabewerte x_n
  ```
  file Eingabedatei: TEXT open READ_MODE is "Eingabe.txt";
  ```
- und eine Konstante für die Taktperiode
  ```
  constant Tp: DELAY_LENGTH := 10 ns;
  ```

Der Filtergrad M steckt in der Typvereinbarung von »tAdr« und ist vier (vgl. Abschnitt 5.2.3).

Der Ablauf der Filteroperationen beginnt mit einer Schleife mit Initialisierungsschritten gefolgt von einer Schleife mit Verarbeitungsschritten. Die grau unterlegten Programmzeilen sind Textverarbeitungsanweisungen für die Ein- und Ausgabe:

```
-- Vereinbarungen im Testprozess
variable x: tDaten;
variable InitRdy: BOOLEAN;
variable s: tString;
-- Anweisungsfolge im Testprozess
-- Initialisierung und Ausgabe Tabellenkopf
pp.cAdr <= 1; pp.xAdr <= tAdr'HIGH; wait for tp;
write("Eingabe:" & StatusKopfText & "Ausgabe:");
-- Initialisierungszyklus
while pp.cAdr/=0 loop
 read(Eingabedatei, x); write(str(x) & str(pp, xRAM));
 InitStep(x, xRAM, pp); wait for tp;
end loop;
-- Endlosschleife für Verarbeitungsschritte
loop
 if pp.cAdr=0 then
  if ENDFILE(Eingabedatei) then wait; end if;
  read(Eingabedatei, x); assign(s, str(x));
 else assign(s, "        ");
 end if;
 append(s, str(pp, xRAM));
 if pp.cAdr=2 then
  if not InitRdy then InitRdy := TRUE;
  else -- Ausgabe
   append(s, str(pp.Akku));
  end if;
 end if;
 write(s);  -- Ausgabe der Ergebniszeichenkette
 NormalStep(x, cROM, xRAM, pp); wait for tp;
end loop;
```

⇒WEB-Projekt: P5.2/Test_FIR1.vhdl

In der Initialisierungsschleife wird in jedem Schritt ein Eingabewert aus der Datei gelesen, die Prozedur »InitStep(...)« aufgerufen und ein Takt gewartet. Der Start- und der Normalzyklus sind zusammengefasst. Die Unterscheidung im Ablauf erfolgt mit der prozessinternen Variablen »InitRdy«, die im ersten Zyklus nach der Initialisierung die Ausgabe unterdrückt. Die Hauptarbeit leistet die Prozedur »NormalStep(...)«. Bei dem Koeffizientenzählstand »0« (Übernahme eines neuen Eingabewertes) wird zusätzlich vor dem Verarbeitungsschritt ein Wert aus der Datei gelesen und bei dem Koeffizientenzählstand »2« erfolgt zusätzlich nach der Warteanweisung ab dem zweiten Zyklus eine Ausgabe.

Die Textausgaben werden mit den für Textobjekte vom Typ »tString« im Package »Tuc.Ausgabe« definierten Prozeduren

- assign(*Textobjekt*, *Text*): alten Text löschen und neuen Text zuweisen,
- append(*Textobjekt*, *Text*): neuen Text anhängen und
- write(*Textobjekt*): Text auf dem Bildschirm ausgeben

erzeugt (siehe Abschnitt 3.3). Die Str-Funktionen für die Umwandlung der projektspezifischen Datentypen »tDaten« etc. in Textdarstellungen sind mit im Package »P5.2/FIR1_pack.vhdl« definiert. Sie sind ähnlich wie die bereits behandelten Str-Funktionen aufgebaut und repräsentieren einen erheblichen Anteil des Gesamtprogrammieraufwands.

Die Darstellung der Testausgaben hat entscheidenden Einfluss darauf, ob fehlerhafte Zustände und Ausgaben bei der Simulation als solche erkannt werden. Für die Kontrolle von Pipeline-Operationen ist die in Abb. 5.8 dargestellte Tabellenform zu empfehlen. Die durch einen einfachen Strich getrennten Spalten gehören zur selben Pipeline-Phase. Nach einem Doppelstrich beginnt die nächste Pipeline-Phase. Ungültige Werte werden durch »XX« dargestellt. Zum Simulationsbeginn sind nur die Koeffizienten- und die Datenadresse gültig. Die Werte im Datenspeicher sind alle ungültig und werden von der höchsten Adresse absteigend mit den ersten Eingabewerten beschrieben. Der Datenwert für den niederwertigsten Speicherplatz wird – wie in Abb. 5.6 festge-

Eingabe:	cAdr	xAdr		cDat	xDat		Prod	Akku		r(0)	r(1)	r(2)	r(3)	Ausgabe:
100.0	1	3		XX	XX		XX	XX		XX	XX	XX	XX	
200.0	2	2		XX	XX		XX	XX		XX	XX	XX	100.0	
300.0	3	1		XX	XX		XX	XX		XX	XX	200.0	100.0	
400.0	0	0		XX	XX		XX	XX		XX	300.0	200.0	100.0	
	1	1		0.2000	400.0		XX	XX		400.0	300.0	200.0	100.0	
	2	2		0.5000	300.0		80.0	XX		400.0	300.0	200.0	100.0	
	3	3		-0.5000	200.0		150.0	80.0		400.0	300.0	200.0	100.0	
600.0	0	3		-0.2000	100.0		-100.0	230.0		400.0	300.0	200.0	100.0	
	1	0		0.2000	600.0		-20.0	130.0		400.0	300.0	200.0	600.0	
	2	1		0.5000	400.0		120.0	110.0		400.0	300.0	200.0	600.0	110.0
	3	2		-0.5000	300.0		200.0	120.0		400.0	300.0	200.0	600.0	
-600.0	0	2		-0.2000	200.0		-150.0	320.0		400.0	300.0	200.0	600.0	
	1	3		0.2000	-600.0		-40.0	170.0		400.0	300.0	-600.0	600.0	
	2	0		0.5000	600.0		-120.0	130.0		400.0	300.0	-600.0	600.0	130.0

Abb. 5.8. Simulationsausgaben des FIR-Filtermodells (r(i) – Abkürzung für xRAM(i))

legt – gleichzeitig in das Datenregister »xDat« kopiert. Die Datenweitergabe über eine Pipeline-Phase ist immer mit einer Verzögerung um einen Taktschritt verbunden. Anhand der Testausgaben sind alle spezifizierten Abläufe und Operationen in einer anschaulichen Weise kontrollierbar. Der zusätzliche Programmieraufwand für übersichtliche, gut auswertbare Ausgaben – Richtwert 50% des Gesamtprogrammieraufwands – gilt als gute Investition. Denn der Test, die Fehlersuche und nicht erkannte Entwurfsfehler sind in der Regel erheblich teurer.

Bearbeitungsstand: Die Zielfunktion ist in einer simulierbaren Form beschrieben und wurde mit Beispieleingaben getestet.

5.2.5 Ersatz der Zahlentypen durch Bitvektortypen

In Hardware werden Zahlen durch Bitvektoren dargestellt. In einem strukturierten Entwurf mit Bearbeitungsmethoden und Unterprogrammen lassen sich die Zahlentypen gegen die entsprechenden Bitvektoren austauschen, ohne dass das gesamte Modell neu geschrieben werden muss. Die Darstellung der Adresswerte »0« bis »3« erfordert einen 2-Bit-Vektor für vorzeichenfreie Zahlen:

```
subtype tAdr is tUnsigned(1 downto 0);
```

Mit einem Bitvektor als Adresse ändern sich die Operationen für die Adressrechnung und die Speicherzugriffe. Aus den Additionen bzw. Subtraktionen mod-4 werden einfache Additionen bzw. Subtraktionen, weil arithmetische Ergebnisse immer mod-(2^n) (n – Ergebnisbitanzahl) gebildet werden. Die Speicherzugriffe mit einem Bitvektor als Adresse müssen auch die Fälle für ungültige und unzulässige Adressen mit erfassen. Im Beispiel werden deshalb die Speicherzugriffe durch Aufrufe der Lesefunktion

```
function read(Mem: tMem; adr: tAdr) return tDaten;
```

und der Schreibprozedur

```
procedure write(signal Mem: inout tMem; adr: tAdr; x: tDaten);
```

aus Abschnitt 3.4.3 ersetzt. Die Daten sollen durch 16-Bit vorzeichenbehaftete Festkommazahlen dargestellt werden:

```
subtype tDaten: tSigned(15 downto 0);
```

Die Koeffizienten haben einen Betrag kleiner eins. Die gedachte Kommaposition sei hinter dem führenden Bit. Der kleinste mit einer Vorkommastelle und fünfzehn Nachkommastellen darstellbare Koeffizient ist $1{,}00\ldots0_2$ und hat den Wert -1. Der größte darstellbare Koeffizient ist 0,11...1 mit dem Wert $1-2^{-15}$. Der Wertebereich für Daten soll wie im ersten Modell mindestens von -1000 bis $+1000$ reichen. Das erfordert einschließlich Vorzeichenbit elf Stellen vor

dem Komma. Die restlichen fünf Bit sind Nachkommastellen. Bei einer Multiplikation addieren sich die Anzahlen der Vor- und der Nachkommastellen. Die Produkte aus Koeffizienten- und Datenwerten haben zwölf Vorkomma- und 20 Nachkommastellen.

Die akkumulierten Werte sollen, um Wertebereichsüberläufe auszuschließen, dreizehn Vorkommastellen erhalten (siehe hierzu auch Aufgabe 5.1). Die Anzahl der Nachkommastellen muss nicht größer als die der Daten sein. Für den akkumulierten Wert genügt entsprechend ein 18-Bit-Vektor:

```
subtype tAkku is tSigned(17 downto 0);
```

Damit die Produkte ohne Konvertierung an den Akkumulator zugewiesen werden können, sollen sie denselben Typ mit derselben gedachten Kommaposition haben. Die Multiplikation ist entsprechend so umzudefinieren, dass zu dem normal berechneten Produkt vorzeichenerweitert eine Vorkommastelle hinzugefügt und die fünfzehn niederwertigsten Nachkommastellen abgeschnitten werden[4]:

```
function mult(a, b: tDaten) return tAkku is
 constant prod: tSigned(2*tDaten'LENGTH-1 downto 0):=a*b;
begin
 return prod(prod'HIGH) & prod(prod'HIGH downto tDaten'HIGH);
end function;
```

Für die Kontrolle, dass die geänderten Datendarstellungen und Operationen das Eingabe-Ausgabe-Verhalten des Gesamtmodells nicht beeinträchtigen, sollen sich die Wertedarstellungen der Adressen, Daten und Koeffizienten nicht von denen im ersten Modell unterscheiden. Das erfordert eine Anpassung der Str-Funktionen zur Konvertierung in die Ausgabetexte. Für die Datendarstellung wird der Wert der Daten durch 2^5 dividiert und mit sieben Zeichen und einer Nachkommastelle dargestellt:

```
function str_dat(x: tSigned) return STRING is
begin
 if is_x(x) then return "    XX";
 else return rechts(str(REAL(int(x))/(2.0**5), 1),7);
 end if;
end function;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

Die ganzzahlige Wertepräsentation der Koeffizienten wird durch 2^{15} geteilt und mit vier Nachkommastellen dargestellt:

[4] Überladen des »*«-Operators scheidet hier aus, weil »tDaten« ein Untertyp und kein eigener Typ ist. Der Wert von »tDaten'HIGH« ist fünfzehn und der Wert von »tProd'HIGH« ist 31.

```
function str_coeff(c: tSigned) return STRING is
begin
 if is_x(c) then return "    XX";
 else return rechts(str(REAL(int(c))/(2.0**15), 4),7);
 end if;
end function;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

Als Nächstes müssten die geänderten Typvereinbarungen und Unterprogramme so in die beiden Bearbeitungsprozeduren und in den Testrahmen der vergangenen Abschnitte eingearbeitet werden, dass die Simulationsausgaben im fehlerfreien Fall genau wie in Abb. 5.8 aussehen. Letzteres ist nur eine Empfehlung, aber eine nützliche. Wir wollen diese Schritte überspringen. Nach dem Ersatz aller Zahlentypen durch Bitvektortypen bildet die Simulation bereits die Registerstruktur und überwiegend auch die Register-Transfer-Funktionen der zu entwerfenden Schaltung bit-genau nach. Damit sind wir wieder einen Schritt näher am Ziel.

5.2.6 Ein- und Ausgabe über Signale

In den bisherigen Simulationsmodellen für den FIR-Filter wurden die benötigten Eingabedaten aus einer Datei gelesen und die Ausgaben in einer Zeichenkettenvariablen zusammengesetzt und ausgegeben. Eine Schaltung kommuniziert mit ihrer Umgebung über Signale statt über Ein- und Ausgabeanweisungen. In der nächsten Version des Simulationsmodells wird der Testrahmen deshalb in drei über Signale kommunizierende Prozesse geteilt, und zwar in

- einen Prozess für die Bereitstellung der Eingabesignale,
- einen Prozess zur Modellierung des Testobjekts und
- einen Ausgabeprozess, der möglichst dieselben Textausgaben wie die bisherigen Simulationsmodelle erzeugen soll (Abb. 5.9 a).

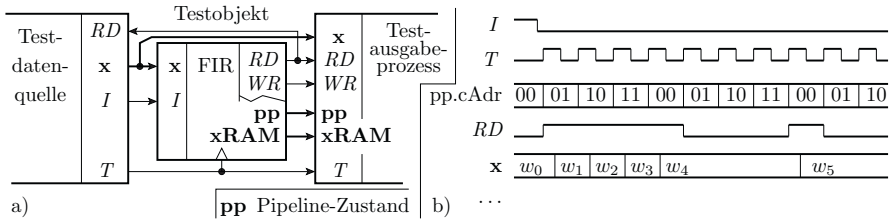
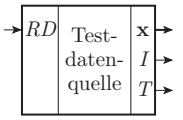


Abb. 5.9. Testrahmen für den FIR-Filter mit Ein- und Ausgabesignalen a) Prozessstruktur b) Signalverläufe

Das Testobjekt – die Funktionsbeschreibung des FIR-Filters – erhält außer dem Dateneingabesignal **x** und dem Datenausgabesignal **y** einen Takteingang

T, einen Initialisierungseingang I, ein Bestätigungssignal für Leseoperationen RD und ein Bestätigungssignal für Schreiboperationen WR. Das Signal für die Lesebestätigung wird während der Lesetakte aktiviert und signalisiert der Datenquelle, dass sie ab dem nächsten Takt den Folgewert ausgeben soll. Das Schreibbestätigungssignal WR wird in den Ausgabetakten aktiviert und signalisiert dem Testausgabeprozess, dass im Akkumulatorregister ein gültiger Ausgabewert steht, der an die Ausgabezeile anzuhängen ist. Der Testeingabeprozess soll durch eine nebenläufige Prozedur beschrieben werden:

```
procedure Testdatenquelle(Dateiname: STRING; signal RD: STD_LOGIC;
      signal T, I: out STD_LOGIC; signal x: out tDaten;
      tp: DELAY_LENGTH := 10 ns) is
 variable pstr: tPString; variable ZNr: NATURAL;
 variable r: REAL;
 file f: TEXTIO.TEXT open READ_MODE is Dateiname;
begin
  I <= '1'; T <= '0'; wait for tp; I <= '0' after tp/10;
  loop
   T <= '1', '0' after tp/2;
   if RD='1' then
    if TEXTIO.ENDFILE(f) then wait; end if;
    read(f, pstr); get(pstr, r, -1000.0, 1000.0);
    if pstr.Status/=ok then
     write("Zeile " & str(znr) & ":" & str(pstr.err_msg)); wait;
    else x <= to_tSigned(INTEGER(r * (2.0**5)), x'LENGTH) after tp/3;
         ZNr := ZNr + 1;
    end if;
   end if;
   wait for tp;
 end loop;
end procedure;
```

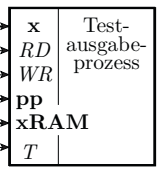


⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

Die Eingabedatei enthält die Filtereingabedaten. Das Initialisierungssignal I, das Taktsignal T und der Eingabevektor $\mathbf{x}$ werden innerhalb der Prozedur generiert. Nebenläufig aufgerufen generiert die Prozedur zu Beginn einen Initialisierungsimpuls und erzeugt danach mit der Taktperiode T_P Taktimpulse. Nach der ersten aktiven Taktflanke wird der erste gültige Eingabewert an das Signal $\mathbf{x}$ zugewiesen. Danach wird nach jeder aktiven Taktflanke, bei der das Lesebestätigungssignal aktiv ist – d.h. bei den nachfolgenden drei und dann in jedem vierten simulierten Takt –, der nächste Eingabewert aus der Datei gelesen und verzögert an $\mathbf{x}$ zugewiesen (Abb. 5.9 b). Wenn das Dateiende erreicht ist, terminiert die Prozedur mit einer Warteanweisung ohne Weckbedingung. Durch das Ausbleiben der Eingabeänderungen terminieren auch die anderen Prozesse des Testrahmens und die Simulation stoppt.

Der Testausgabeprozess soll etwa dieselben Ausgaben wie in Abb. 5.8 erzeugen. Dazu müssen der Prozedur, die nebenläufig aufgerufen den Protokollprozess nachbildet, das Taktsignal, das Schreib- und das Lesebestätigungssi-

gnal, das Eingabesignal, der Pipeline-Zustand und der Zustand des Datenspeichers übergeben werden. Nach Aufruf schreibt die Prozedur die Kopfzeile der Ausgabetabelle und in einer Endlosschleife nach jeder aktiven Taktflanke die Textzeilen mit dem Bearbeitungszustand und optional dem Eingabe- oder Ausgabewert. Zur Kontrolle der Zeitverläufe wird jeder Ausgabezeile zusätzlich die aktuelle Simulationszeit vorangestellt (siehe später Abb. 5.11):




```
procedure Testausgabe(signal T, RD, WR: STD_LOGIC;
   signal pp: tPipeline; signal x: tDat;
   signal xRAM: tMem) is
 variable s: tString;
begin
  write("  Zeit  |Eingabe|Z| ...  r(3)|Ausgabe");(1)
  loop                              -- Endlosschleife
   wait until RISING_EDGE(T);       -- warte auf aktive Taktflanke
   assign(s, rechts(str(NOW, 1),8));
  -- Anhängen eines Zeilentextes wie in Abb.5.8,
  -- der den Zustand der Schaltung beschreibt
   write(s);                        -- Textzeile ausgeben
  end loop;
 end procedure;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

((1) – Ausgabe der Kopfzeile).

Bearbeitungsstand: Ein komplettes Verhaltensmodell der Soll-Funktion mit Anschlusssignalen und einer internen Bitvektordarstellung. Der dazu entwickelte Testrahmen kann auch für alle weiteren schnittstellengleichen Versionen des Simulationsmodells genutzt werden.

5.2.7 Umformung in eine Automatenbeschreibung

In dem bis hierher entwickelten Ablaufmodell wird in den Schleifenkörpern nach jeder Register-Transfer-Operation eine Taktperiode lang gewartet (Abb. 5.10 a). Der funktionsgleiche synthesefähige Abtastprozess darf nur eine Warteanweisung enthalten, und zwar am Ende der Anweisungsfolge im Prozess (Abb. 5.10 b, vgl. Abschnitt 1.4.2). Nach der Warteanweisung, d.h. zu Beginn der Anweisungsfolge des Prozesses, verzweigt der Kontrollfluss mit Hilfe von Fallunterscheidungen. Jeder Wartezustand in der Ablaufbeschreibung benötigt hierfür einen eigenen Kontrollflusszustand.

Der Kontrollflusszustand für drei nacheinander abzuarbeitende Schleifen besteht aus dem Schleifenzähler und einem Aufzählungstyp zur Unterscheidung, welche der drei Schleifen gerade abgearbeitet wird:

```
type tZustand is XX, Init, Startzyklus, Normalzyklus;
```

Der vierte Zustandswert »XX« bedeutet »ungültig« und dient als Initialwert für den Simulationsstart. Der Schleifenzähler ist der Zähler für die Koeffizientenadresse, die im Register »pp.cAdr«, gespeichert wird, so dass hierfür keine

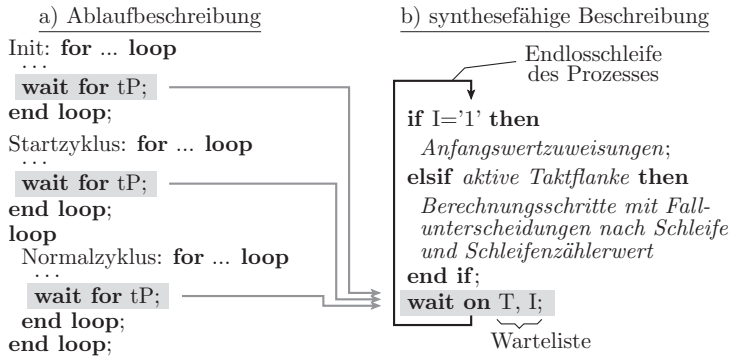


Abb. 5.10. Kontrollflussumformung in einen synthesefähigen Abtastprozess

zusätzliche Hardware erforderlich ist. Der Datentyp für den Pipeline-Zustand erweitert sich zu

```
-- erweiterter Datentyp für den Pipeline-Zustand
type tPipeline is record
  cAdr, xAdr: tAdr;
  cDat, xDat: tDaten;
  Prod, Akku: tAkku;(1)
  Zustand: tZustand;
 end record;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

((1) – breiterer Bitvektortyp als »tDaten«, um Wertebereichsüberläufe zu vermeiden, vgl. Abschnitt 5.2.5).

Die Übergangsfunktion sei durch eine Prozedur beschrieben, die bei jeder aktiven Taktflanke aufgerufen wird:

```
-- Übergangsfunktion des gesamten FIR-Filters
procedure PipeStep(x: tDat; cROM: tMem; signal xRAM: inout tMem;
  signal pp: inout tPipeline) is
begin
 case pp.Zustand is
  when XX => null;
  when Init =>
    pp.cAdr <= pp.cAdr + "1";
    pp.xAdr <= pp.xAdr - "1";
    write(xRAM, pp.xAdr, x);
    if pp.cAdr=to_tUnsigned(M-1, tAdr'LENGTH) then
     pp.Zustand <= Startzyklus; end if;
  when others =>
    -- Operationen im Start- und Nomalzyklus
    ...
  end case;
 end procedure;
```

cAdr	1	2	...	$M-1$
xAdr	$M-1$	$M-2$	...	1
xRAM	W(x_0)	W(x_1)	...	W(x_{M-2})

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

Der Eingabevektor **x** und der Koeffizientenspeicher »cROM« sind Eingabewerte, der Datenspeicher »xRAM« und das Pipeline-Objekt les- und veränderbare Signale. In den Initialisierungsschritten wird die Koeffizientenadresse, die als Zustandszähler dient, hochgezählt. Die Datenadresse wird abwärts gezählt und der Eingabewert auf den adressierten Datenspeicherplatz geschrieben. Die Speicherzugriffe erfolgen hier mit der Read-Funktion und der Write-Prozedur aus Abschnitt 3.4.3, die im Gegensatz zu einem indizierten Feldzugriff auch den Fall, dass die Zugriffsadresse ungültig oder unzulässig ist, mit berücksichtigen. Für die Synthese sind diese Unterprogramme entsprechend zu vereinfachen oder wieder durch indizierte Zugriffe zu ersetzen. Im letzten Initialisierungsschritt wird der Zustand von »Init« auf »Startzyklus« geändert.

In allen anderen Fällen, d.h. im Zustand »Startzyklus« oder »Normalzyklus«, sind die Fallunterscheidungen aus Abb. 5.7 zu übernehmen:

- Schritt 0: Datenwert in den Datenspeicher schreiben statt lesen (FU1),
- Schritt 2: Produkt in den Akkumulator kopieren statt zum Akkumulatorinhalt zu addieren und Ergebnis ausgeben (FU2) und
- letzter Schritt: Datenadresse nicht weiterzählen (FU3).

Im Startzyklus entfällt die Ergebnisausgabe. Statt dessen wird der Zustand weitergeschaltet:

```
-- Operationen im Start- und Nomalzyklus
pp.cAdr <= pp.cAdr + "1";                    (1)
if pp.cAdr="00" then                          -- Fallunterscheidung FU1
 write(xRAM, pp.xAdr, x);                    (2)
 pp.xDat <= x;
else
 pp.xDat <= read(xRAM, pp.xAdr);             (2)
end if;
pp.cDat <= read(cROM, pp.cAdr);              (2)
pp.Prod <= mult(pp.cDat, pp.xDat);           (3)
if pp.cAdr="10" then                          -- Fallunterscheidung FU2
 pp.Akku <= pp.Prod;
 if pp.Zustand=Startzyklus then pp.Zustand <= Normalzyklus; end if;
else
 pp.Akku <= pp.Akku + pp.Prod;
end if;
if pp.cAdr/="11" then                         -- Fallunterscheidung FU3
 pp.xAdr <= pp.xAdr + "1";
end if;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

((1) – für 2-Bit-Vektoren automatisch mod 4; (2) – Lesefunktion und Schreibprozedur aus Abschnitt 3.4.3; (3) – Spezialmultiplikation für Faktoren vom Typ »tDaten« und dem Ergebnistyp »tAkku« aus Abschnitt 5.2.5).

Die Schaltungsschnittstelle des FIR-Filters in Abb. 5.9 a besitzt drei Ausgabesignale: das Lesebestätigungssignal, das Schreibbestätigungssignal

und den Akkumulatorzustand. Die beiden Bit-Signale ergeben sich aus dem Pipeline-Zustand und sollen durch getrennte Funktionen berechnet werden. Das Lesebestätigungssignal ist im Initialisierungszustand immer und in den beiden anderen Zuständen nur, wenn die Koeffizientenadresse null ist, zu aktivieren:

```
function RdGnt(pp: tPipeline) return STD_LOGIC is
begin
 if pp.Zustand=Init or pp.cAdr=to_tUnsigned(0, tAdr'LENGTH)
  then return '1';
 else return '0';
 end if;
end function;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

Das Signal für die Schreibbestätigung ist nur im Normalzyklus, wenn die Koeffizientenadresse den Wert zwei hat, zu aktivieren:

```
function WrGnt(pp: tPipeline) return STD_LOGIC is
begin
 if pp.Zustand=Normalzyklus and pp.cAdr=to_tUnsigned(2, tAdr'LENGTH)
  then return '1';
 else return '0';
 end if;
end function;
```

⇒WEB-Projekt: P5.2/FIR2_pack.vhdl

Das komplette Verhaltensmodell soll als Prozess beschrieben werden. Dazu sind im Testrahmen folgende Konstanten und Signale zu vereinbaren:

```
constant cROM: tMem := (to_tKoeff( 0.2), to_tKoeff( 0.5),
                        to_tKoeff(-0.5), to_tKoeff(-0.2));
signal xRAM: tMem(0 to M-1);
signal pp: tPipeline;
signal T, I, rd, wr: STD_LOGIC;
signal x: tDat;
```

⇒WEB-Projekt: P5.2/Test_FIR2.vhdl

Die Funktion »to_tKoeff(...)« ist eine im Package definierte Hilfsfunktion zur Konvertierung einer Gleitkommazahl in die Bitvektordarstellung für Koeffizienten. Die Ein- und Ausgabe wird durch nebenläufige Aufrufe der Prozeduren aus dem Vorabschnitt beschrieben:

```
Testdatenquelle("Eingabe.txt", rd, T, I, x);
Testausgabe(T, rd, wr, pp, x, xRAM);
```

Das Verhaltensmodell der kompletten Übergangsfunktion des FIR-Filters ist ein Abtastprozess mit asynchroner Initialisierung. Zur Initialisierung werden den beiden Adressregistern und dem Zustandsregister ihre Anfangswerte zugewiesen und bei jeder aktiven Taktflanke wird die Prozedur, die die Übergangsfunktion beschreibt, aufgerufen:

```
process(T, I)
begin
  if I='1' then
    pp.cAdr <= to_tUnsigned(1, tAdr'LENGTH);
    pp.xAdr <= to_tUnsigned(M-1, tAdr'LENGTH);
    pp.Zustand <= Init;
 elsif RISING_EDGE(T) then
   PipeStep(x, cROM, xRAM, pp);
 end if;
end process;
```

⇒WEB-Projekt: P5.2/Test_FIR2.vhdl

Die beiden Ausgabesignale werden nebenläufig aus dem Pipeline-Zustand mit Hilfe der vordefinierten Funktionen gebildet:

```
rd <= RdGnt(pp);   wr <= WrGnt(pp);
```

Abbildung 5.11 zeigt die Simulationsausgaben. Die noch fehlenden Schritte bis zu einer synthesefähigen Schaltung beinhalten

- Beseitigung aller Abfragen und Zuweisungen mit dem Pseudo-Wert »ungültig«,
- den entsprechenden Ersatz bzw. die Vereinfachung der Read-Funktion und der Write-Prozedur und
- eine Kapselung der Testobjektbeschreibung in eine eigenständige Entwurfseinheit mit Schnittstellenbeschreibung[5].

Zeit	Eingabe	Z	cAdr	xAdr	cDat	xDat	Prod	Akku	r(0)	r(1)	r(2)	r(3)	Ausgabe
10.0 ns	XX	I	1	3	XX	XX	XX	XX	XX	XX	XX	XX	
20.0 ns	100.0	I	1	3	XX	XX	XX	XX	XX	XX	XX	XX	
30.0 ns	200.0	I	2	2	0.5000	XX	XX	XX	XX	XX	XX	100.0	
40.0 ns	300.0	I	3	1	-0.5000	XX	XX	XX	XX	XX	200.0	100.0	
50.0 ns	400.0	S	0	0	-0.2000	XX	XX	XX	XX	300.0	200.0	100.0	
60.0 ns		S	1	1	0.2000	400.0	XX	XX	400.0	300.0	200.0	100.0	
70.0 ns		S	2	2	0.5000	300.0	80.0	XX	400.0	300.0	200.0	100.0	
80.0 ns		S	3	3	-0.5000	200.0	150.0	80.0	400.0	300.0	200.0	100.0	
90.0 ns	600.0	N	0	3	-0.2000	100.0	-100.0	230.0	400.0	300.0	200.0	100.0	
100.0 ns		N	1	0	0.2000	600.0	-20.0	130.0	400.0	300.0	200.0	600.0	
110.0 ns		N	2	1	0.5000	400.0	120.0	110.0	400.0	300.0	200.0	600.0	110.0
120.0 ns		N	3	2	-0.5000	300.0	200.0	120.0	400.0	300.0	200.0	600.0	

Abb. 5.11. Simulationsausgaben des FIR-Filtermodells

[5] Für den Test und die Fehlersuche in den ersten Entwurfsphasen ist es günstiger, die Zielfunktion im Testrahmen in einem Prozess als sequenzielles Programm zu beschreiben. Denn so ist das Einfügen, Ändern und Löschen von Testausgaben einfacher. Die Kapselung in eine separate Entwurfseinheit ist erst später nützlich, wenn die betrachtete Funktionseinheit gründlich getestet ist, zuverlässig funktioniert und in eine übergeordnete Entwurfseinheit als Teilschaltung eingefügt wird.