19 Strukturierter Entwurf

Dieses Kapitel gibt durch Beispiele eine Einführung in die Methoden des strukturierten Entwurf (structured VLSI design [4][15][19]). Ähnlich wie komplexe Programme, die durch die Entwicklung separater Unterprogramme entstehen, lassen sich auch einfache Grundelemente wie Gatter und Register zu immer höher entwickelten Schaltungen kombinieren. Dabei können auf jeder Stufe des Entwurfs Probleme entstehen, deren Lösung ein "Durchbrechen" der Abstraktionsbarrieren nach oben oder unten erforderlich macht, so daß ein VLSI-Entwurf eigentlich kein klassischer *Top-down* Entwurf ist, sondern eher ein Yoyo-Spiel, ein wiederholtes auf und ab zwischen den Abstraktions-Ebenen.

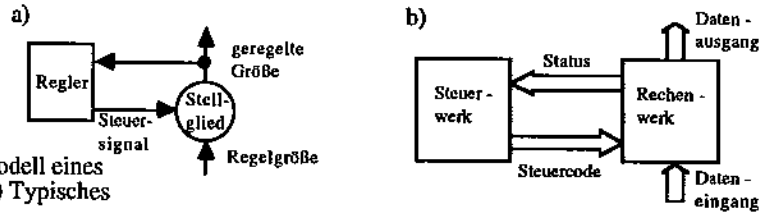


Bild 19.1: a) Modell eines Regelkreises, b) Typisches Digitalsystem

Am Beginn eines Entwurfs steht die rein operationale Sicht einer Komponente, d.h. im wesentlichen wird das Ein-/Ausgabe- und Zeitverhalten der geplanten Schaltung spezifiziert. Dazu nützt eine Hardwarebeschreibungssprache wie *KARL* und ihr grafisches Pendant *ABL*, mit deren Hilfe das Verhalten von Hardwaremodulen - durch programmiersprachenähnlichen Syntax - als Funktion der Eingaben beschrieben wird. Dabei betrachtet der Designer primär den Datenfluß, dessen Kontrolle er durch die Programmstruktur erreicht. Eine solche nur durch ihre nach außen sichtbaren Eigenschaften beschriebene Komponente ähnelt einem *Black Box* Modell, wie es beispielsweise die Elektrotechnik für Regelkreisbeschreibungen verwendet. Reicht hier jedoch oft ein Ein- und Ausgang für die zu regelnde Größe und ein einziger "Draht" als Eingang

eines Meßsignales, so gehört zu der typischen Aufgabenstellung digitaler Systeme eine weitaus größere Anzahl einzubeziehender Daten. Bild 19.1 veranschaulicht, wie im Gegensatz zur Regeltechnik ein Digitalsystem Daten, Steuerbefehle und Statusinformationen in einer bestimmten Wortbreite verarbeitet bzw. ausgibt.

Für Rechnerarchitekten ist Ausgangspunkt für den weiteren Entwurfsverlauf des oft ein rein strukturelles Modell, in dem er globale Zusammenhänge der Datenbewegungen überblickt. Obwohl diese Modell bezüglich der tatsächlichen physischen Implementation keine wirkliche Beschreibung der Struktur darstellt, hilft es ihm doch, die oftmals komplexe Aufgabenstellung optimal zu bewältigen. Um Verwirrungen zwischen diesem pseudo-strukturellen Modell und der Strukturbeschreibung des physischen Entwurfs zu vermeiden, nennen wir dieses Modell *Datenpfadmodell*. Dieser Begriff verdeutlicht, worauf es in dieser Sichtweise ankommt: Den gesamten Datenfluß überblicken und in seinem zeitlichen Ablauf beherrschen.

Bild 19.2 zeigt in ABL-Darstellung, wie einfach sich bei dieser Sichtweise beispielsweise die Beschreibung eines einzelnen Schieberegisters als rückgekoppelter Datenpfad darstellt. Das im Register gespeicherte Wort erreicht über einen Datenpfad den Schiebe-Operator, an dessen

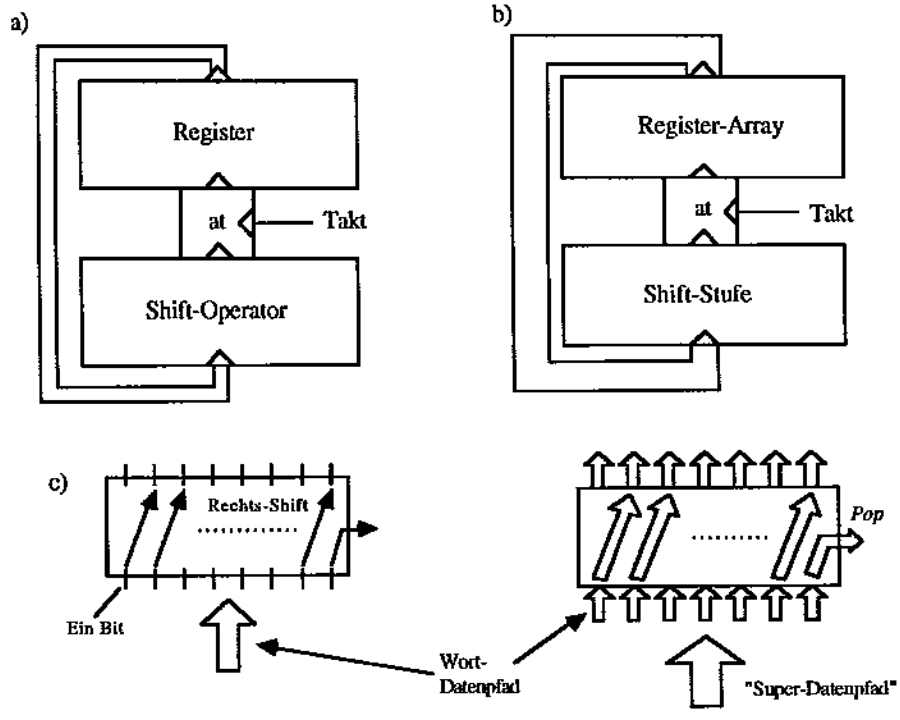


Bild 19.2: a) ABL-Darstellung eines Schieberegisters für ein Wort, b) -ein Datenfeld c) Shift-Operator bzw. -Stufe am Beispiel eines Rechts-Shift

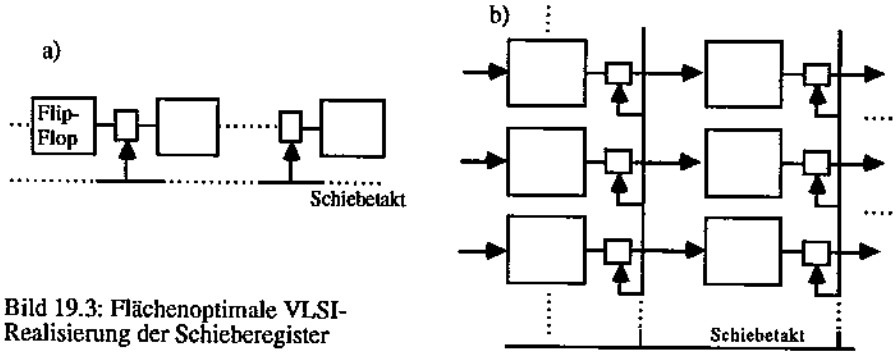


Bild 19.3: Flächenoptimale VLSI-Realisierung der Schieberegister

Ausgang nun das verschobenen Wort liegt. Mit dem nächsten Takt wird das Register dann mit diesem Wert geladen. Dieses Datenpfadmodell ist ebenso auf beliebig große Datenfelder (*Register-Arrays*) anwendbar, indem ein "Super-Datenpfad" gebildet wird, der sich aus Einzelpfaden gleicher Wortbreite zusammensetzt. Aus dem einzelnen Schift-Operator wird dann eine Shift-Stufe (s. auch das nächste Kapitel), die statt nur eines Bits nun jeweils ein ganzes Wort verschiebt. Der eigentliche, im Rückfluß manipulierte Datenpfad ist nach wie vor genauso deutlich erkennbar. Bild 19.2 c) zeigt zusätzlich, wie das rechts aus der Stufe geschobene Wort auf einen anderen Datenpfad gelangen und weiterverwendet werden könnte. Diese Ausgabe eines äußeren Feldelementes wird häufig auch als *Pop*-Operation bezeichnet.

Ist das operationale Verhalten erst einmal festgelegt, geht es auf einer niedrigeren Abstraktionsebene des Entwurfs daran, Grundbausteine der digitalen Schaltungstechnik zu der gewünschten Funktion zu vereinen. Den Entwickler interessiert hier eher die direkte Kommunikation benachbarter Elemente als der globale Zusammenhang des Datenflußes. Ist für den Rechnerarchitekten beispielsweise der oben beschriebene Shift lediglich eine "Aktion" in der Zeit (operationale Denkweise), so beinhaltet diese Operation für den Lieferanten der Hardware auch räumliche Aspekte, denn ihn interessiert, wie und wo Signale die Schaltung durchlaufen und wo sie zu manipulieren sind (funktionale Denkweise). Er hat eine topologische Sicht auf die Schaltung, d.h. er versucht die einzelnen Elemente möglichst flächenoptimal zu plazieren. Das Datenpfadmodell hat für ihn keine Bedeutung mehr; in seiner Abstraktionsebene ist es nicht anwendbar.

In Bild 19.3 sieht man, wie sich das Schieberegister aus Bild 19.2 mit zunehmender Wortbreite und Datenfeldgröße zweidimensional "aufbläht" und schnell unüberschaubare Dimensionen annimmt. Hier ist es dann entscheidend, die kleinsten Einheiten und ihre Schnittstellen so zu entwerfen, daß sie in beliebiger Anzahl als regelmäßige Struktur einen Verbund bilden, dessen Funktion bereits aus dieser kleinsten Einheit vorhersagbar ist.

19.1 Entwurf eines Stapelregisters (Stack)

Als Beispiel eines strukturierten Entwurfes für die VLSI-Technologie soll ein Stapelregister (*Stack*) dienen [3][19], das aus einfachen Zellen mit den nötigen Grundfunktionen und einer

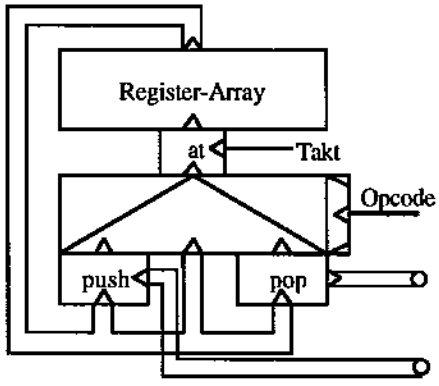


Bild 19.4: Datenpfad-Modell eines Stack.

Steuerschaltung aufgebaut ist. Dieses so entstandene Modul *Stack* kann über die definierte Schnittstelle dann als funktionales Objekt in einer höheren Abstraktionsebene verwendet werden, ohne daß man den Realisierungsdetails Beachtung schenken müßte.

Stapelregister arbeiten nach dem *last-in, first-out* (LIFO) Prinzip, d.h. zuletzt abgespeicherte Daten werden zuerst wieder ausgegeben. Entgegen dem Modell einer solchen Datenstruktur, die mit jedem hinzugefügten Datum nach oben wächst, liest unser Modul die Daten von rechts ein und stellt quasi einen um 90° nach rechts gedrehten Stack dar. Die Steuerungsanweisung *Push* soll ein Datum aufnehmen, wodurch die bereits gespeicherten Daten um jeweils eine Stelle verschoben werden. Eine Operation *Pop* gibt das zuletzt gespeicherte Datum aus und läßt alle darauffolgenden Daten aufrücken. Eine dritte Operation schließlich lädt lediglich die vorhandenen Register neu mit ihren momentanen Werten, ohne etwas zu verändern und führt dadurch den für dynamische Register nötigen *Refresh* durch. Das entsprechende Datenpfad-Modell zeigt Bild 19.4.

19.1.1 Quasi-statisches Flipflop

In Bild 19.6 wird ein quasi-statisches Flipflop gezeigt (mehr über Halbleiterspeicher s. in [2], [20], [21]). Es besteht aus zwei Invertern (Inv1, Inv2), einem Lade-Transistor T_L, einem Halte-Transistor T_H und einem Update-Transistor T_U. Die Funktionsweise dieser Schaltung ist sehr einfach. Sie entspricht der eines Master/Slave-Flipflops: während Φ_1 wird der Eingang **input** ausgewertet, während Φ_2 wird ein "update" des Ausgangs **output** durchgeführt.

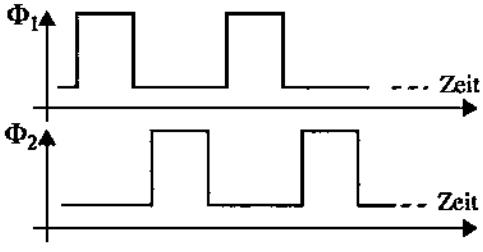


Bild 19.5: Nicht-überlappender 2-Phasen-Takt.

Über den Transistor T_L wird, wenn WRITE&PHI1 = 1 gilt, ein Wert in das Flipflop geschrieben. Der nach dem ersten Inverter (Inv1) folgende Transfer-Transistor T_U ("update-Transistor") hat die Aufgabe, einen neuen Wert erst dann an den Ausgang (output) durchzuschalten, wenn der alte Wert bereits verarbeitet wurde, d.h. z.B. in ein nachfolgendes Flipflop eingelesen wurde.

Wie man leicht erkennen kann, erhält man durch einfaches Hintereinanderschalten mehrerer solcher Zellen (im weiteren *Slices* genannt) ein Schieberegister. Der Transistor T_H, angesteuert durch das Signal HOLD&PHI1, hat die Aufgabe, solange kein neuer Wert eingelesen wird, den Wert des Flipflops aufzufrischen (1.Phase). Damit das Flipflop wie oben beschrieben arbeitet,

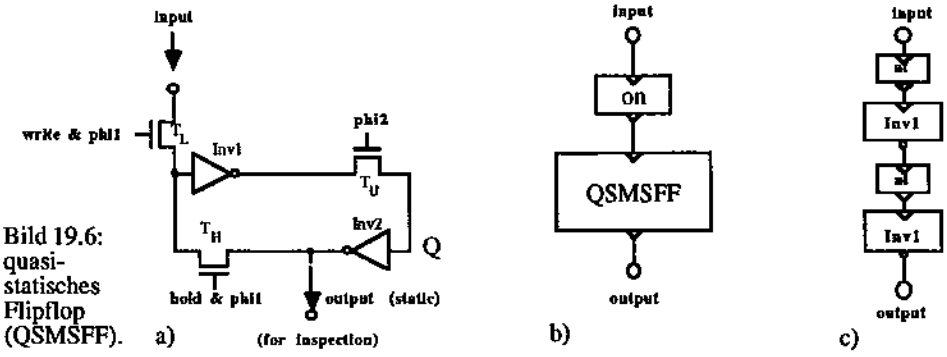


Bild 19.6: quasi-statisches Flipflop (QSMSFF).

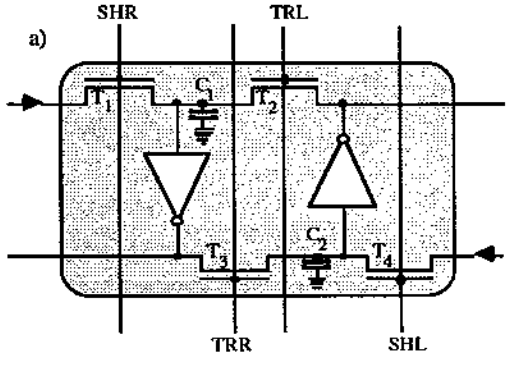
ist es notwendig, daß Φ_1 und Φ_2 ein nicht-überlappender Zwei-Phasen-Takt ist (Bild 19.5). Dies verhindert, daß ein Wert, der über T_L eingelesen wird, sofort über T_U und Inv2 am Ausgang anliegt, und somit bei einem Schieberegister direkt vom Eingang auf den Ausgang durchgeschaltet würde.

Zusätzlich müssen die Signale WRITE und HOLD zur gleichen Zeit immer einen unterschiedlichen Wert besitzen, damit am Eingang des Inverters Inv1 kein Widerspruch auftritt. Dies erreicht man beispielsweise, indem man 'HOLD' als 'not (WRITE)' definiert. Bild 19.6 b bzw. c zeigen die ungefähre Darstellung in der RT-Ebene für die Spezifikation (b) bzw. Realisation (c) des Typen QSMSFF.

19.1.2 Realisation des Stack

Das oben erläuterte Flipflop Prinzip machen wir uns nun zunutze, um zunächst ein elementares Modul des Stacks zu verwirklichen. Bild 19.7 zeigt, wie man durch geschicktes Plazieren der einzelnen Elemente eine *Slice* erhält, deren Ein- und Ausgänge ein direktes Aneinanderfügen erlauben.

Auch hier findet man das häufig in integrierten Schaltungen angewandte Prinzip, den Kontrollfluß senkrecht durch den Datenfluß zu legen. Vier Steuersignale kontrollieren das Lesen und



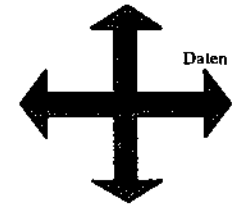


Bild 19.7: a) 1-Bit Speicherzelle, b) Daten- und Kontrollfluß

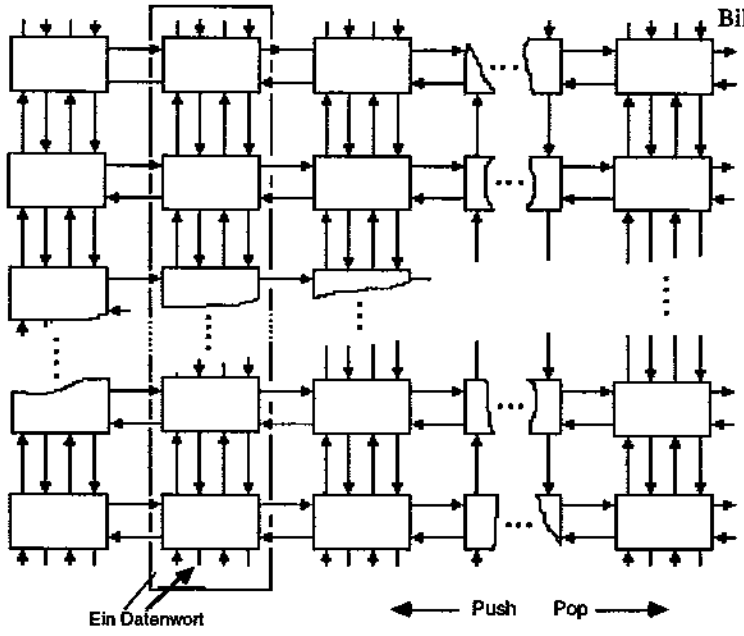


Bild 19.8 : Speicherzellenfeld für das Stack-Schieberegister

Schreiben von Daten von der linken bzw. rechten Seite oder führen das Auffrischen des gespeicherten Bits innerhalb der Speicherzelle durch:

- Daten vom linken Eingang einlesen (SHR: Shift right)
- Daten vom rechten Eingang einlesen (SHL: Shift left)
- Refresh von der linken zur rechten parasitären Kapazität (TRR: Transfer Right)
- Refresh von der rechten zur linken parasitären Kapazität (TRL: Transfer left)

Von den beiden Invertern werden lediglich die parasitären Kapazitäten C_1 und C_2 genutzt, die ein Signal etwa einen Taktzyklus lang speichern. Durch wechselseitiges Umladen beider Inverterkapazitäten über den TRL- bzw. TRR-Transfer wird ein dauerhaftes Halten des gespeicherten Datums erreicht.

Aus diesen Zellen entsteht nun ein Feld von Registern, dessen Höhe die Wortbreite der Daten, die der Stack verarbeiten kann, vorgibt. Die Anzahl der speicherbaren Datenworte ist somit durch die Breite des Zellenfeldes begrenzt. Die Anordnung der Zellen in einem solchen Feld gibt Bild 19.8 wieder. Auf der rechten Seite ist der Ein- und Ausgang, in den die Daten herein- bzw. herausgeschoben werden.

Erfassungs-zeit	Ausführungs-zeit	Operationskode	
		0	1
Φ_1	Φ_2	hold	pop
Φ_2	Φ_1	hold	push

Bild 19.9: Zeitabhängigkeit des Operationskode.

Die zeitliche Kontrolle der Abläufe in dem Schieberegister übernehmen zwei

sich nicht überlappende Taktphasen Φ_1 und Φ_2. Ein Operationscodesignal *OP* bestimmt die durch- zuführenden Schiebevorgänge, also einen Rechts-Shift für eine Pop-Operation und einen Links-Shift für einen Push-Befehl. Wird keine dieser Operationen angefordert, so erfolgt mit jedem Taktzyklus ein interner Refresh. Der Operationscode besteht lediglich aus einem Bit, so daß sich die auszulösende Operation aus dem Zeitpunkt ergibt, zu dem OP von *Low* auf *High* geht. Kommt das Operationssignal während Φ_1, so wird im nächsten Takt (Φ_2) ein *Pop* durchgeführt, geht OP während Φ_2 auf *High*, so erfolgt ein *Push* mit Φ_1. Der gesamte Befehlszyklus gliedert sich also in die Erfassungsphase (*Scan-Phase*), in welcher der Operationscode und der Zeitpunkt (Φ_1 oder Φ_2) ermittelt wird, und in die Ausführungsphase (*Execution-Phase*), in der dann die angeforderte Operation durchgeführt wird. Der Zusammenhang zwischen den Taktphasen und dem Operationscode ist nocheinmal in der Tabelle inBild 19.7 zusammengefaßt.

Bild 19.10 erläutert das genaue Timing der Schaltung. Wird kein *Push* oder *Pop* gewünscht, sondern ein *Hold* (OP := Low), so erfolgt jeweils ein TRR mit Φ_1 und ein TRL mit Φ_2, d.h. mit Φ_1 öffnet der Pass-Transistor T_3 (siehe Bild 19.7) und lädt die linke Inverterladung auf die rechte Kapazität um. Während Φ_2 wird dieser Vorgang über den Pass-Transistor T_2 umgekehrt und so ein ständiges Refreshing gewährleistet. Kommt nun ein OP-Signal zusammen mit Φ_1,

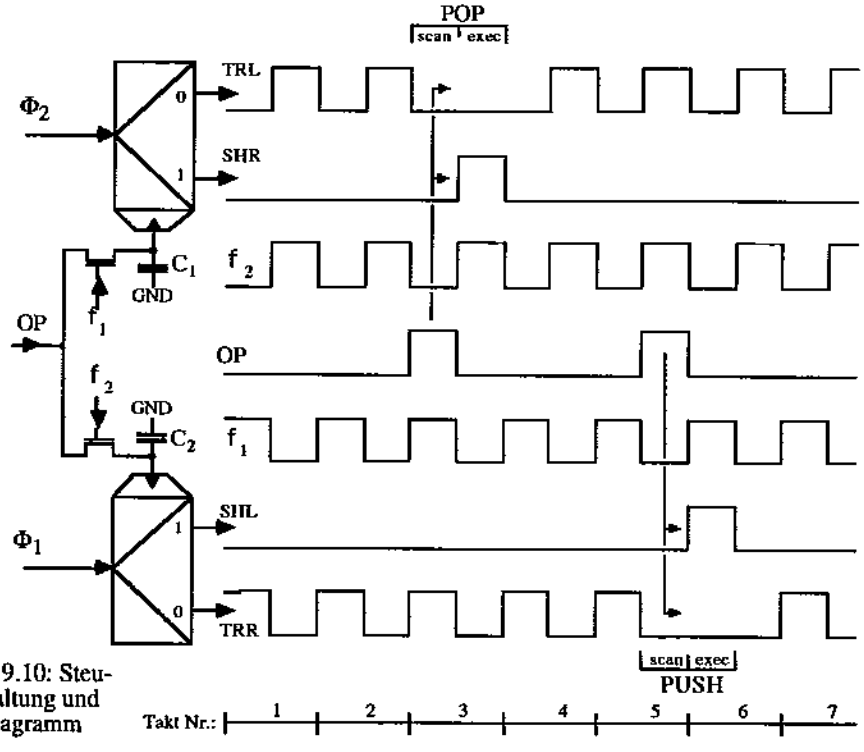


Bild 19.10: Steuerschaltung und Zeitdiagramm

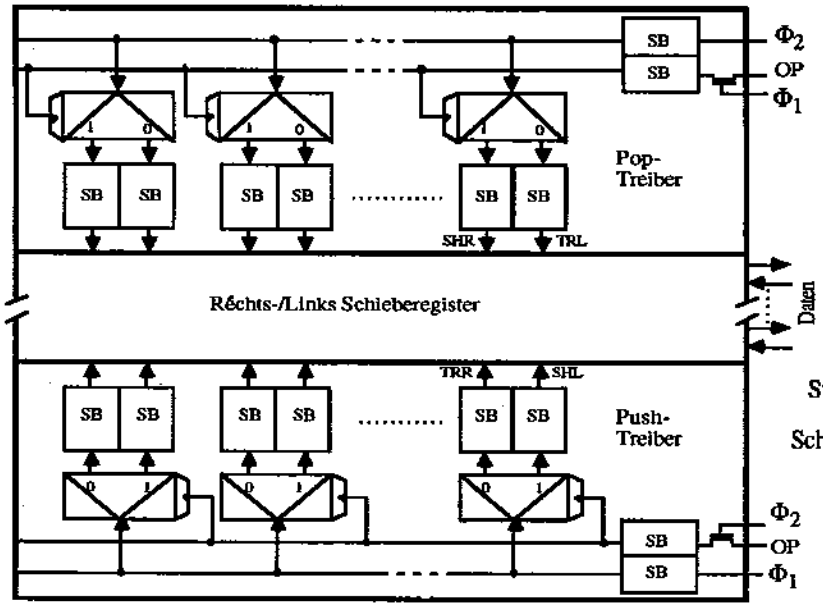


Bild 19.11: Stapelregister modul aus Schieberegister und Steuerschaltungen

so wird das TRL-Signal für eine Taktphase aus durch SHR ersetzt. Dadurch wird dann die linke Inverterkapazität nicht durch die rechte neu geladen, sondern über den Eingangs-Pass-Transistor T_1 das Datum der links daneben liegenden Zelle eingelesen. Insgesamt ergibt sich dann also eine Linksverschiebung aller Daten und das ganz rechte Wort kann am Ein-/Ausgang gelesen werden. Dies entspricht also der Pop-Operation. Analog setzt bei OP := High während Φ_2 das TRR-Signal einen Zyklus lang aus und es erfolgt statt dessen ein SHL, so daß der rechte Inverter nicht das Datum des intern linken übernimmt, sondern über den rechten Eingangs-Transistor T_4 das Signal der rechten Zelle erhält. Im ersten Wort wird dann der Eingang des Stapelregisters eingelesen, so daß insgesamt ein Push-Befehl ausgeführt wird.

Links neben dem Zeitdiagramm in Bild 19.10 ist das Mischdiagramm der Steuerschaltung abgebildet. Liegt kein OP-High-Signal vor, so werden durch die beiden Multiplexer MUX1 und MUX2 die Phasen Φ_1 bzw. Φ_2 auf die TRR- bzw. TRL-Leitungen durchgeschaltet. Kommt nun ein OP-Impuls während Φ_1, so wird MUX1 über den oberen Pass-Transistor umgeschaltet und der nächste Φ_2-Takt erscheint nicht auf der TRL-Leitung, sondern als SHR-Signal. Dadurch wird dann der oben beschriebene Pop-Befehl eingeleitet. Wird OP während Φ_2 gegeben, so schaltet MUX2 über den unteren Pass-Transistor um und gibt den nächsten Φ_1-Takt auf die SHL- statt TRR-Leitung und erzwingt dadurch die Push-Operation.

Die beiden Kondensatoren C_1 und C_2 sind wiederum parasitäre Kapazitäten von Invertern und sollen den bereits über den Pass-Transistor gelangten Operationscode für die Dauer eines Taktes speichern, um ein sicheres Reagieren des Stapelregisters sicherzustellen.

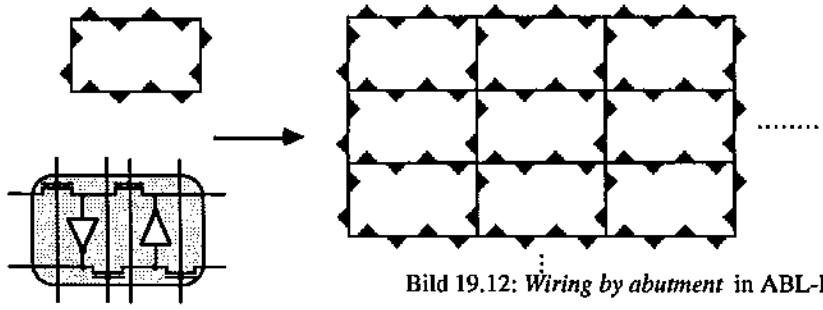


Bild 19.12: *Wiring by abutment* in ABL-Darstellung

Das fertige Modul und wird in Bild 19.11 gezeigt. Da jede einzelne Signalleitung abhängig von der Größe und Wortbreite des Stapelregisters sehr viele Elemente der Schaltung anzusteuern hat, würde aufgrund der hohen kapazitiven Last die Steilheit der Flanken eines Steuerimpulses extrem beeinträchtigt - ein typischer Nachteil der hier verwendeten Verhältnislogik. Um die steigende und fallende Flanke symmetrisch steil zu halten, wird daher in jede Signalleitung ein Treiber integriert, der die Kapazitäten rasch aufladen bzw. Ladungen hoher Kapazitäten schnell ableiten kann, so daß das ursprünglich angelegte Signal nicht verzerrt wird. Diese in der Verhältnislogik universell einsetzbaren Bausteine heißen *Super-Buffer (SB).*

19.2 Topologische Aspekte

Das etwas aufwendigere 2-Phasen-Timing der Schaltung ist der Preis für die Verwendung von dynamischen Registern, die dafür aber solch einfache Datenpfadstrukturen ermöglichen. Ein einfacherer, einphasiger Takt hätte aufwendigere, statische Flipflops erfordert, so daß die vielfache Verwendung davon in solch einem Registerfeld eine Menge Platz vergeudet hätte. Allerdings lassen sich dynamische Flipflops im Gegensatz zu statischen nur schwer in der Register-Transfer (*RT*) Ebene modellieren. Auch Pass-Transistoren sind dort keine Grundelemente, da sie keine definierte Datenpfadrichtung haben.

Bild 19.13 zeigt die einfache Struktur der Flipflops. Die Zelle aus Bild 19.7 besteht ihrerseits aus zwei Hälften (Bild 19.13 a), die im Prinzip identisch sind und nur um 180° gegeneinander verdreht und aneinandergefügt wurden. Der Entwurf des Schieberegisters auf Schaltkreis-, Topologie- und Geometrieebene beschränkt sich also auf den Inverter und die beiden Pass-Transistoren. Bild 19.13 b zeigt das Stickdiagramm der halben Zelle und Bild 19.13 c) das nach Kopieren und Drehen entstehende Stickdiagramm des kompletten Flipflops. Das daraus sich ergebende flächenoptimale Layout für eine Zelle gibt Bild 19.14 wieder. Die fertige Zelle kann dann beliebig oft kopiert und direkt aneinandergefügt werden (*wiring by abutment*), so daß sich eine hochregelmäßige und kompakte Struktur des Schieberegisters ergibt. Bild 19.12 veranschaulicht dieses Verbinden durch Anstoßen anhand einer ABL-Darstellung der Zellen.

Auch die beiden Treiberstufen sind strukturell identisch und entstehen aus wenigen einfachen Grundelementen wie Multiplexern und Super-Buffer-Bausteinen. Ist der Pop-Treiber - durch Iteration der Ansteuerung einer Datenwortspalte - entworfen, so ergibt sich die Push-Steue-

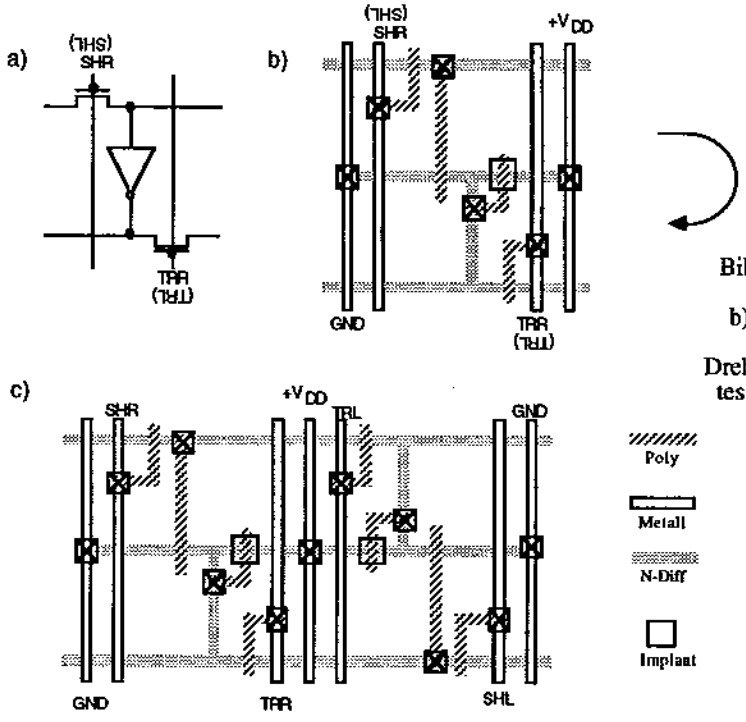


Bild 19.13: a) Flipflop Grundelement und b) Stickdiagramm (n-MOS). c) durch Drehung eines Duplikates entstandenes Stickdiagramm für ein komplettes Flipflop

rung aus einer Drehung um 180°. Lediglich die Einspeisung von Φ_1&OP bzw. Φ_2&OP muß noch zusätzlich über je einen Pass-Transistor für beide Treiberstufen erfolgen.

Durch Aneinanderfügen von Kopien lediglich zweier einfacher Zellen ist ohne aufwendige zusätzliche Verdrahtung somit eine komplexe und doch kompakte Schaltung entstanden, die auf

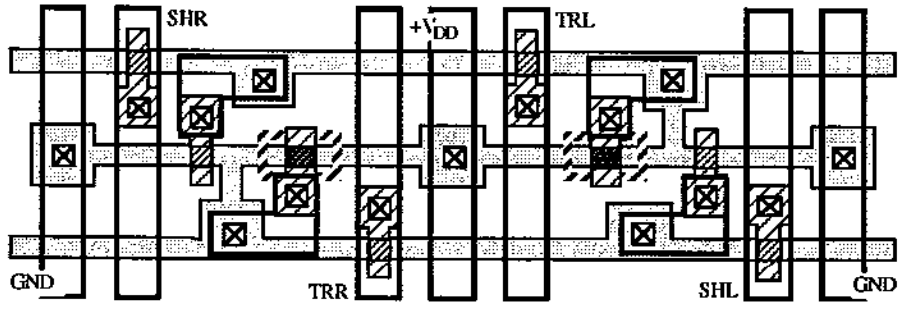


Bild 19.14: n-MOS-Layout einer Flipflop-Zelle

Hardware-Ebene die komplette Datenstruktur *Stack* implementiert. Durch geschickte Wahl der Funktionsgruppen und ihrer Schnittstellen konnte der topologische Aufwand extrem gering gehalten werden. Ein Minimum an Entwurfsarbeit ist dank der regelmäßigen Struktur ausreichend, um das Stapelregister in beliebiger Größe zu entwickeln.

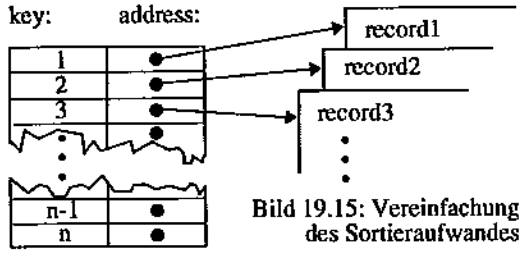


Bild 19.15: Vereinfachung des Sortieraufwandes

19.3 Der Shuffle Sort: Beispiel eines "Smart Memory"

Sortieralgorithmen lassen sich mit enormem Leistungsgewinn in integrieren [14][15][16][9]. Nun wollen wir, ebenfalls mit Hilfe des in Abschnitt 19.1.1 entwickelten Flipflops, eine Sortierschaltung in Hardware realisieren [14]. Zugrunde gelegt sei der *Bubble Sort* -Algorithmus. Der Bubble Sort vergleicht jeweils zwei Nachbarwerte ACCWORD[j-1] und ACCWORD[j] (eine Art Abtastfenster). Nehmen wir an es seien N Objekte nach einem bestimmten Schlüssel zu sortieren (Bild 19.15). Das "Fenster" wandert im Inkrementierer von +1 durch das ganze Objektfeld (innere Schleife: $2 \le j \le N$). Im ungünstigsten Fall muß die innere Schleife N-1 mal wiederholt werden (äußere Schleife: $2 \le i \le N$).

Eine Software-Lösung (*Bubble Sort*) hat dann in etwa folgendes Aussehen, wobei mit *swap* die Operation bezeichnet sei, die im Fall key[j-1] > key[j] beide Objekte miteinander vertauscht:

```
loop i = 2..N
   loop j = 2..N
```

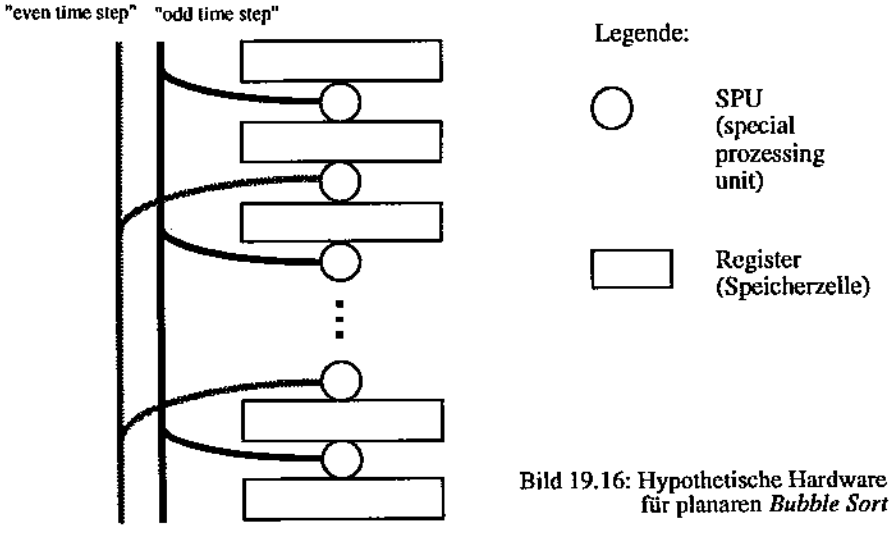
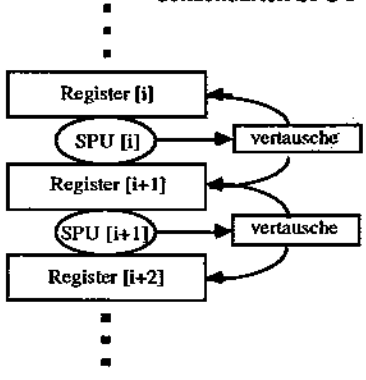


Bild 19.16: Hypothetische Hardware für planaren *Bubble Sort*

```
        if key[j-1] > key[j] then swap( key[j-1], key[j] )
        endif;
      endloop j;
    endloop i;
```

Bild 19.17: Zugriffskonflikt bei gleichzeitiger Aktivierung zweier benachbarten SPU's



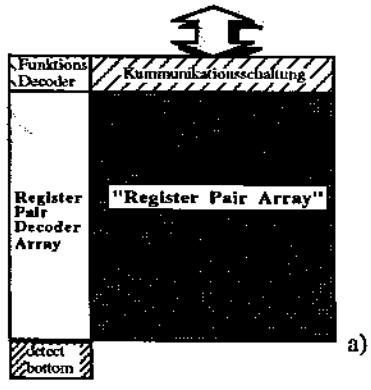
Im folgenden wird nun eine Hardware-Lösung aus der Software-Lösung durch *Parallelisierung* der inneren Schleife erarbeitet. Dadurch ergibt sich eine *smart memory*-Lösung, d.h. ein Speicher, der etwas mehr kann als nur lesen und schreiben (z.B. auch vergleichen und vertauschen).

19.3.1 Ein erster Entwurf

Als ersten Entwurf bekommen wir die hypothetische Hardware nach Bild 19.16. Hier werden abwechselnd eine Speicherzelle und eine SPU (special processing unit) nacheinander plaziert und miteinander verschaltet (durch *wiring by abutment*).

Die SPU ist für den Vergleich zweier benachbarter Speicherinhalte und gegebenenfalls für ihre Vertauschung zuständig. Bei diesem Entwurf ergibt sich ein Zugriffskonflikt wenn zwei benachbarte SPU's gleichzeitig mit "swap" aktiviert werden (vgl. Bild 19.17), der durch zwei Steuersignale bzw. Arbeitszyklen behoben werden kann (Bild 19.16). Die SPU's werden so angesteuert, daß immer nur entweder die geradzahligen oder die ungeradzahligen SPU's tätig sind. Auf diese Weise wird verhindert, daß zwei benachbarte SPU's gleichzeitig auf dieselbe Speicherzelle zugreifen.

Die Lösung hat einen Nachteil. Sie ist nicht flächenoptimal, da doppelt soviel SPU's vorhanden sind, als benutzt werden. Da nur jeweils die Hälfte der SPU's arbeitet, können wir deren Zahl



a)

b)

Bild 19.18: Zweite hypothetische Hardware für den planaren *Bubble Sort*.

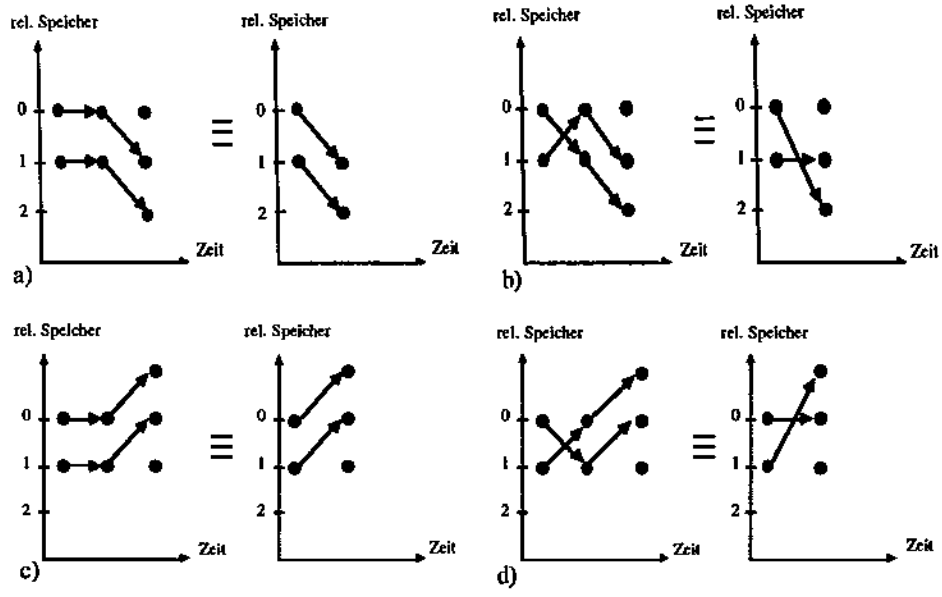


Bild 19.19: Zur Vereinfachung einer SPU: a) Verschieben nach unten ohne Vertauschen (shift down), b) Verschieben nach unten mit Vertauschen (swap down) c) Verschieben nach oben ohne Vertauschen (shift up) d) Verschieben nach oben mit Vertauschen (swap up).

auf die Hälfte minimieren. Dies hat zur Folge, daß man nun die Speicherinhalte abwechselnd nach oben bzw. unten "schiften" muß (daher der Begriff *shuffle sort*, entlehnt von "shuffle board", deutsch: Waschbrett), damit abwechselnd geradzahlige Wertepaare und ungeradzahlige Wertepaare miteinander verglichen werden können (vgl. Bild 19.16).

19.3.2 Verbesserte Version des Sortierers

Somit kommen wir zu einem zweiten, besseren Entwurf (siehe Bild 19.18). Die Kommunikationsschaltung ist für die Beschaltung der Eingänge der Flipflops und das Herausschreiben der sortierten Werte zuständig. Ein Funktionsdekodierer ist dafür verantwortlich, ob sortiert, eingelesen oder die Speicherwerte herausgeschrieben werden. Das "detect bottom" Signal gibt an, ob (1) der Speicher bereits voll ist, d.h. es können keine Werte mehr eingelesen werden, bzw. um der Sortierschaltung zu sagen, daß keine Werte mehr vorhanden sind, die noch sortiert werden sollen und (2) wann die Daten im Speicher unten angekommen sind, damit sie dann anstatt durch *push*-Transport mit *pop*-Transport im Speicher sortiert werden. Es müssen also keine "dummies" in den Speicher eingelesen werden, wenn die Zahl der sortierenden Elemente kleiner ist, als die Kapazität des Speichers.

Auf das Aussehen dieser Teile der Sortierschaltung soll hier nicht näher eingegangen werden. Es sei nur gesagt, daß sie relativ einfach in Funktion und Aufbau sind und wenig Chipfläche

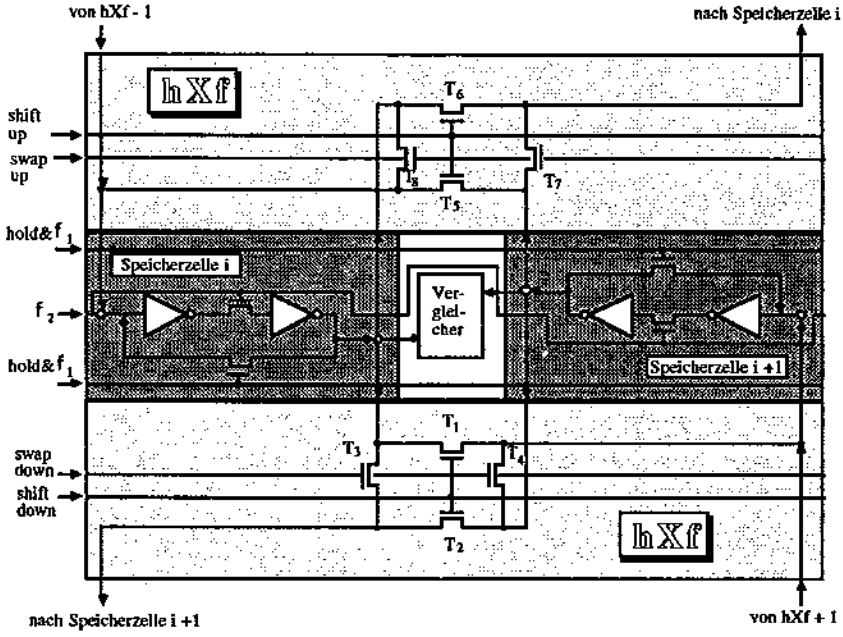


Bild 19.20: Mischdiagramm der Register-Pair-Array-Zelle: Grundriß und Schaltplan

im Entwurf einnehmen. Die Hardware-Lösung ergibt sich aus der Software-Lösung durch eine time-to-space-Konversion. Die Parallelisierung der inneren Schleife (sequence in time) hat einen "smart memory array" ergeben (sequence in space).

Für die SPU ergibt sich nun folgende Aufgabenstellung. Sie soll (1) das Schieben (*schiften*) zwischen den Zellen realisieren und (2) eine Vertauschung zweier Zellen ermöglichen, sofern dies nach einem Vergleichsergebnis notwendig ist. Diese Funktionen kann man noch zusätzlich zusammenfassen und somit ergeben sich folgende Möglichkeiten:

1.) die Werte werden unvertauscht nach unten geschoben (Bild 19.19 a)

2.) die Werte werden vertauscht nach unten geschoben (Bild 19.19 b)

3.) die Werte werden unvertauscht nach oben geschoben (Bild 19.19 c)

4.) die Werte werden vertauscht nach oben geschoben (Bild 19.19 d)

Man erkennt, daß sich für das Schieben nach unten derselbe Aufbau wie für das Schieben nach oben ergibt, und erhält daher folgenden Aufbau einer SPU (siehe Bild 19.20):

Die Transfer-Transistoren T_1 bis T_8 erfüllen die Schiebe- und Austausch-Funktionen. T_1 und T_2 sind für das Weiterschieben nach unten ohne vertauschen, T_3 und T_4 für das Weiterschieben nach unten mit vertauschen, T_5 und T_6 für das Weiterschieben nach oben ohne vertauschen und T_7 und T_8 für das Weiterschieben nach oben mit vertauschen zuständig (vgl. Bild 19.19). Der

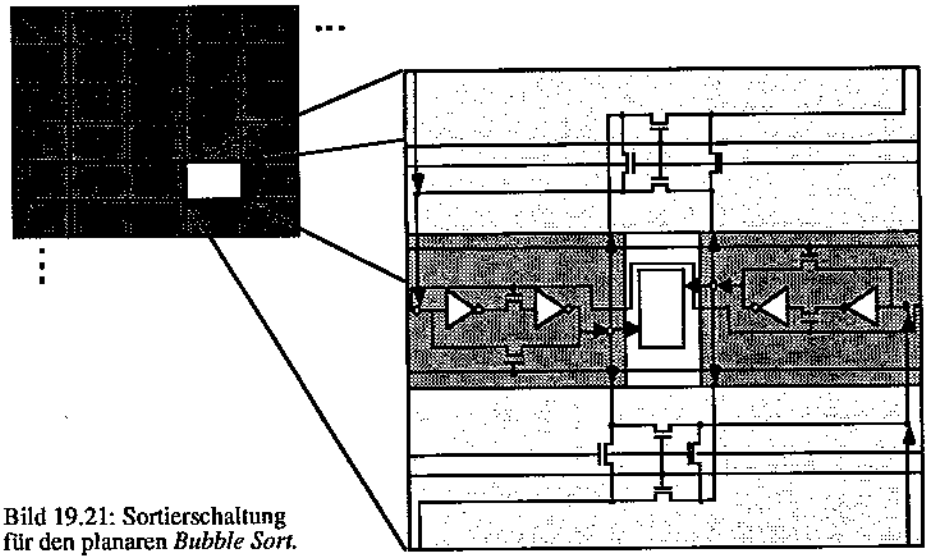


Bild 19.21: Sortierschaltung für den planaren *Bubble Sort*.

Vergleicher ist ebenfalls über Bit-Slices iterativ realisiert. Als Eingänge existieren die Inhalte der benachbarten Flipflops (vgl. Bild 9.7.3), deren Werte miteinander verglichen werden. Entsprechend dem Ergebnis des Vergleichs und einem zusätzlichen Übertrag-Bit der nächstniederwertigen Registerpaarscheibe wird der Wert des Übertrag-Bits bestimmt und zur nächsthöherwertigen Registerpaarscheibe weitergeleitet. Aus diesem Übertragbit (beim höchstwertigen Registerpaar) wird dann ermittelt ob vertauscht wird oder nicht. Es müssen noch die Steuerleitungen verdrahtet werden und dann kann man die Sortierschaltung nach Bild 19.21 aufbauen, wobei die Verdrahtung außerhalb der Speicher-SPU-Slices nur angedeutet ist.

Ein Nachteil der bisher entwickelten Schaltung ist, daß die Verarbeitungszeit etwa genauso lang ist wie die Kommunikationszeit, d.h. daß das Sortieren ebenso viel Zeit in Anspruch nimmt, wie das Ein- bzw. Auslesen der zu sortierenden Daten. Dies ist ein weiterer Optimierungspunkt. Die Lösung ist,

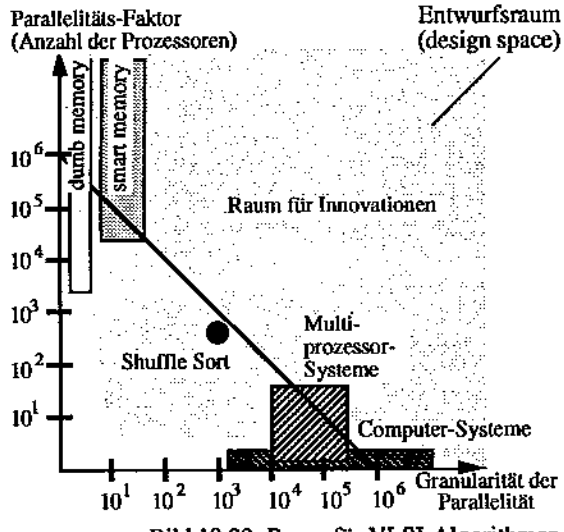


Bild 19.22: Raum für VLSI-Algorithmen.

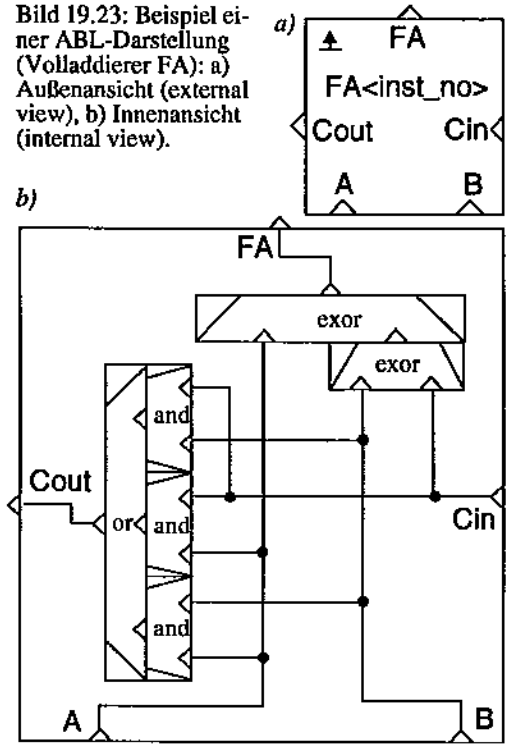


Bild 19.23: Beispiel einer ABL-Darstellung (Volladdierer FA): a) Außenansicht (external view), b) Innenansicht (internal view).

daß schon während des Einlesens sortiert wird. Dies ist durch die globale Ansteuerung ohne wesentliche Schaltungsänderung möglich.

19.3.3 Zeitverhalten: Analyse der Effizienz

Ausgehend von der Zahl der Vergleiche bei *Bubble Sort* ($C = (n^2-n)/2$) erhält man beim *Shuffle Sort* dadurch, daß n/2 Vergleiche parallel ausgeführt werden, einen Wert von $C = ((n^2-n)/2)/(n/2) = \underline{n-1}$. Durch das gleichzeitige Sortieren und Einlesen erhält man damit bei N=1000 Elementen eine gesamte Verbesserung (durch die Parallelität) etwa um die Größenordnung $O \approx 1000$. Die Einsparung durch die Elimination des "von-Neumann-Engpaßes" (der relativ langsame Kommunikationskanal zwischen CPU und Speicher) bewegt sich um den Faktor 100. Desweiteren ist eine Taktdauer bei Programmen üblicherweise um 100ns, während sie auf dem Chip etwa bei 10ns liegt.

Aus allen diesen Optimierungen folgt ein Verbesserungsfaktor für die Hardware-Lösung bei 1000 Elementen etwa von der Größenordnung $\underline{O \approx 10^6}$ gegenüber der Software-Lösung. Um die Vielfalt von VLSI-Strukturen (z.B. Speicher, Multiprozessorsysteme usw.) zu verdeutlichen, ist im folgenden Bild 19.22 der Zusammenhang zwischen Parallelitätsfaktor (Zahl der verarbeitenden Einheiten) und Granularität der Parallelität (Zahl der Transistoren im Verhältnis zu den Prozessoren) dargestellt.

Weitere Beispiele von strukturierten Entwürfen und deren Durchführung werden in Kapitel 20 vorgestellt.

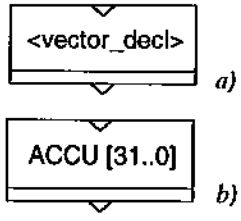


Bild 19.24: ABLED-Dialog: a) Frage durch ABLED, b) die Antwort des Anwenders.

19.4 Eine graphische Sprache für strukturierten Entwurf

Im Rahmen des CVT-Projekts[1] wurde die graphische Notation ABL als interaktiver graphischer Editor ABLED [7] [8] [12] implementiert, gemeinsam durch CSELT[2] und die Univer-

1) 1983 - 1986 gefördert durch die Kommission der Europäischen Gemeinschaften

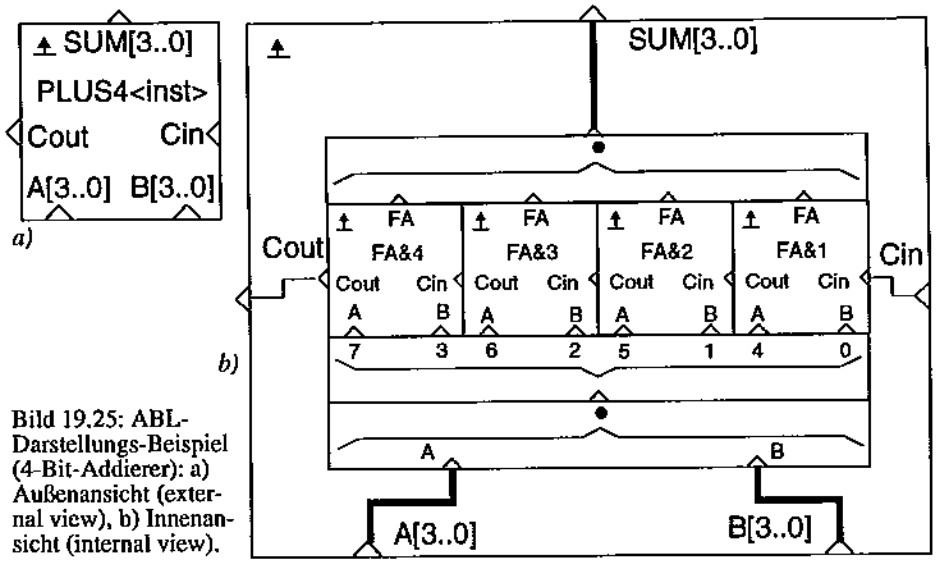


Bild 19.25: ABL-Darstellungs-Beispiel (4-Bit-Addierer): a) Außenansicht (external view), b) Innenansicht (internal view).

sität Kaiserslautern. ABLED lief auf Apollo Domain unter AEGIS und GKS (Für diese Hardware-Plattform hatte man sich zu einer Zeit entschieden, als SUN frisch gegründet war und in Europa noch keinen Vertrieb hatte.). Für die Beschreibung von Entwurfs-Methodologien wurde die Version GENMON von ABLED abgeleitet [1].

Bild 19.23 bis Bild 19.26 veranschaulichen den Gebrauch von ABLED seiner Benutzer-Oberfläche. Bild 19.24 zeigt ein Beispiel für ABLED's schnellen Dialog: Das Systems plaziert seine Frage direkt in die Ansicht (view) des Objekts, das gerade editiert wird (<vector_decl> in Bild a), und der Benutzer überschreibt dies direkt mit seiner Antwort (ACCU[31..0] in Bild b). Diese Art des Dialoges hat den Vorteil, daß keine separaten Dialog-Boxen herumgeschoben werden müssen um das ABL-Diagramm aufzudecken. Frei für die Wahl durch den Benutzer stellt ABLED für Zellen alternativ zwei verschiedene Ansichten (views) zur Verfügung: eine Außenansicht (external view, Beispiel in Bild 19.23 a), und eine innere Ansicht (internal view, Bild 19.23 b). ABLED unterstützt gleichzeitig vier verschiedene Abstraktions-Ebenen: Die Pseudo-Switching-Ebene, die Logik-Ebene (Beispiel in Bild 19.23 b), die nicht-prozedurale RT-Ebene (Beispiel in Bild 19.25 b), sowie eine abstrakte Ebene (z. B. durch den Anwender definierte Wahrheitstafeln, Übergangstabellen, etc.).

Bild 19.25 b illustriert den extensiven Gebrauch der Domino-Notation[3] [10] bei der Synthese eines 4-Bit-Addierers aus 4 (1-Bit-)Volladdierern FA (full adder) und ABLED-Primitiven (der *dot* "•" steht für die *catenation*[4], und die Box mit den geschweiften Klammern ohne dot zeigt

2) Centro Studi et Laboratori Telecomunicazioni, Turin, Italien (Forschungszentrum der STET-Gruppe)

3) abstrakte Modellierung von *wiring by abutment* durch die ABL-Notation: durch Rechtecke dargestellte Funktionsblöcke können direkt aneinanderstoßen, wie Domino-Steine aneinandergelegt werden. Vorteil: das Liniengewirr für die Darstellung komplexer Verdrahtung kann drastisch reduziert werden.

4) Zusammenfassung mehrerer Worte zu einem Verbund-Wort.

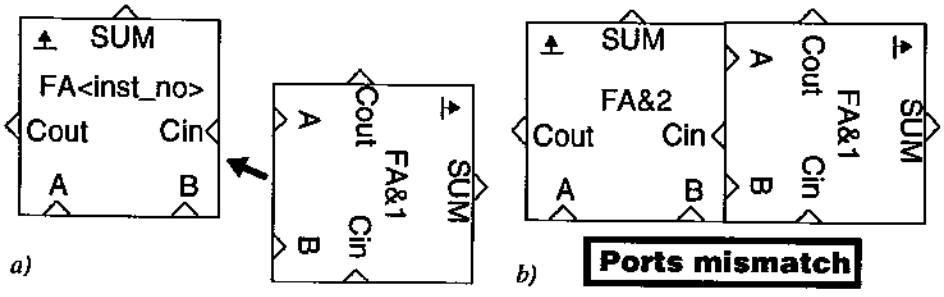


Bild 19.26: Veranschaulichung des graphischen *on-line syntax check* : a) vorher, b) nachher.

subscripted splitting [5] eines 8-Bit-Wortes in 8 einzelne 1-Bit-Worte). ABLED unterstützt die Domino notation mit *abutment* durch *snap-in* ("Einschnappen", siehe Bild 19.26 a und b) und durch automatische Generierung interner Verdrahtungs-Netzlisten sowie einen graphischen *on-line syntax check* (siehe Beispiel in Bild 19.26 b: im Falle nicht zusammenpassender Ports verweigert ABLED die Generierung des Interkonnekt unter Zeigen einer Warnung).

ABLED hat verschiedene Schnittstellen, wie beispielsweise einen PICT generator [5] zur Unterstützung der Dokumentation. Ein ABL2KARL-Übesetzer ermöglicht die Eingabe mit ABLED entworfener Designs in das KARL-Simulation-System [22]. Solche ABL-Datenstrukturen können nach KARL [10] [11] [13] weiterübersetzt werden in das Austauschformat RT-

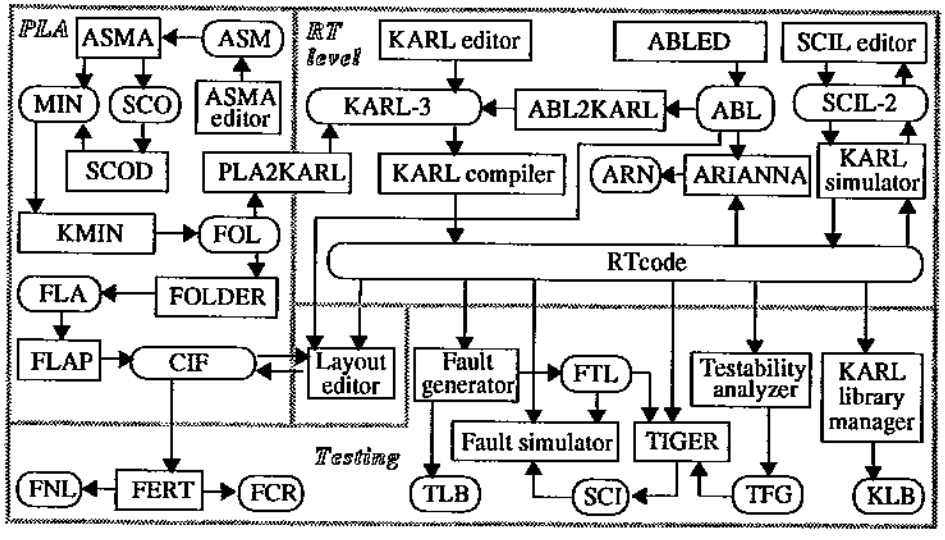


Bild 19.27: CVT-Software-Paket um das KARL / ABL-System.

5) Auflösung (Verzweigung) eines Bit-Vektors durch Indizierung in mehrere Teilvektoren oder in Einzel-Bits.

code [18], das durch eine Anzahl weiterer CAD-Werkzeuge im Rahmen des CVT-Framework akzeptiert wird (siehe Bild 19.27).

19.5 Literatur

[1] A. Bonomo, G. Girardi, A. Lecce, L. Maggiulli: GENMON: a specialized ABL editor for design methodology descriptions; 2nd ABAKUS workshop, Igls, Austria, 1988.

[2] R. Comerford, G. F. Watson: Memory catches up; IEEE Spectrum Oct. 1992

[3] L. Conway: Introduction to VLSI Design; Course Notes, MIT, Cambridge 1979

[4] D. Fairbairn; VLSI - A New Frontier for System Designers; Computer, January 1982

[5] D. Finkler: PICT-Generator für das MLED-System; Univ. Kaiserslautern 1988

[6] J. Di Giacomo: Digital Bus Handbook; McGraw-Hill, 1990

[7] G. Girardi, R. Hartenstein, U. Welters: ABLED: a RT level Schematic Editor and Simulator user Interface; Int'l EUROMICRO Symp.; Brussels, Belgium, 1985.

[8] G. Girardi, R. Hartenstein, U. Welters: KARL (textual) and ABL (graphic) : A User/Designer interface in microelectronics; in (Hrsg.: J. Encarnaçao): CAD-Schnittstellen und Datentransfer-Formate im Elektronik-Bereich; Springer-Verlag, 1986.

[9] A. Halaas: VLSI-Implemented Algorithms for Fundamental Searching and Sorting Problems; Report, Fachbereich Informatik, Universität Kaiserslautern 1981

[10] R. Hartenstein: Fundamentals of Structured Hardware Design; North Holland, Amsterdam / New York 1977

[11] R. Hartenstein, E. von Puttkamer: KARL - a Hardware Description Language as a part of a CAD tool for VLSI; CHDL'79, Int'l Symp. on Computer Hardware Description Languages and their Applications, Palo Alto, CA, USA, 1979; IEEE New York, 1979

[12] R. Hartenstein, U. Welters: VLSI Design and Simulation at Register Transfer Level; in (eds.: W. Fichtner, M. Morf): Proc. IFIP Summer School on VLSI Design, Beatenberg Switzerland, 1986; Kluwer Publishing Co., Boston 1986

[13] R. Hartenstein: KARL-3 Reference Manual; CVT report, Univ. Kaiserslautern, 1986

[14] R. W. Hartenstein, K. P. Bastian, W. Nebel: VLSI-Algorithmen: innovative Schaltungstechnik statt Software; Fachbereich Informatik; Uni Kaiserslautern, April 1985; GME-Tagung Baden-Baden, 1985, VDE-Verlag, Berlin 1985

[15] R. W. Hartenstein: VLSI-Bausteine in geringen Stückzahlen für Spezialanwendungen; Elektronische Rechenanlagen 22 (1980), Heft 4

[16] R. Hartenstein: Die "Neue Mikroelektronik" in der Informatik: Voraussetzungen und auswirkungen; GI-Jahrestagung, 1981 Kaiserslautern, Springer-Verlag, 1981

[17] R. Hartenstein: KARL and ABL; in (ed.: J. P. Mermet): Fundamentals and Standards in Hardware Description Languages; Kluwer Academic Publishers, Boston, 1993

[18] R. Hauck: KARL User guide; Universität Kaiserslautern, 1986

[19] C. Mead, L. Conway: Introduction to VLSI Systems; Addison-Wesley, 1980

[20] B. Prince: Semiconductor Memories; J. Wiley & Sons, Chichester, UK, 1991

[21] S. A. Ward, R. H. Halstead Jr.: Computation Structures; MaGraw-Hill, 1990

[22] U. Welters: ABL2KARL translator: Algorithm description; CVT report, Univ. Kaiserslautern 1984