CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

# Ein Ansatz für die Nutzung teildefekter Field Programmable Gate Arrays (FPGAs) in der Serienproduktion

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

von

**Dipl.-Inform. Sven Frimont**

Gutachter:

**Prof. Dr. Wolfgang Nebel**
**Prof. Dr. Achim Rettberg**

Tag der Disputation: 13. Oktober 2009

Für Mareike

# Danksagung

Meinem Referenten, Herrn Prof. Nebel, danke ich für die wissenschaftliche Betreuung der Arbeit und seine wertvollen Anregungen. Insbesondere danke ich ihm für die Gewährung von zeitlichen Freiräumen während der Erstellung der Dissertation. Herrn Prof. Rettberg danke ich für sein Interesse an der Arbeit und die Übernahme des Korreferates.

Bedanken möchte ich mich auch bei den stets an einer Zusammenarbeit interessierten Kollegen aus meiner Arbeitsgruppe an der Universität und am OFFIS, dem „Oldenburger Forschungs- und Entwicklungsinstitut für Informatikwerkzeuge und -systeme“. Insbesondere Andreas Schallenberg, Frank Oppenheimer und Henrik Lipskoch haben durch Anregungen in zahlreichen Diskussionen wesentlich zum Gelingen dieser Arbeit beigetragen.

# Inhaltsverzeichnis

# 1 Einleitung

Dieses Kapitel führt den Leser in die Thematik ein und gibt einen Überblick über die wesentlichen Ziele und Inhalte der vorliegenden Arbeit.

## 1.1 Motivation

Digitale logische Schaltungen findet man heutzutage in fast jedem elektronischen Produkt. Je nach produzierter Stückzahl werden diese Schaltungen unterschiedlich realisiert. Bei einer Herstellung des Produktes in sehr hohen Stückzahlen kann gegenüber einer Realisierung aus den bereits am Markt erhältlichen Standardbausteinen eine Kostensenkung erreicht werden, indem die Schaltungen auf einen Chip integriert werden. Da der so entstehende Chip durch den Chip-Hersteller speziell für den vorgesehenen Verwendungszweck erstellt wird, spricht man hier von einer anwendungsspezifischen integrierten Schaltung (Application Specific Integrated Circuit – ASIC). Eine alternative Möglichkeit für die Realisierung der in einem Produkt benötigten Schaltungen bieten sogenannte FPGAs (Field Programmable Gate Arrays) als frei programmierbare ICs. Wenn die zu fertigenden Stückzahlen in einer Serienproduktion die hohen initialen Kosten für eine ASIC-Produktion nicht rechtfertigen, ist die Verwendung eines FPGAs für die Serienproduktion eine oft gewählte Alternative.

Die Fertigung von FPGAs in immer kleineren Strukturen verursacht - vor allem dann, wenn FPGAs erstmalig in einer neuen Prozessgeneration hergestellt werden - eine zunehmende Defektdichte bei der Herstellung. Da eine sinkende Ausbeute direkte Auswirkungen auf den Stückpreis eines FPGAs und seine Verfügbarkeit am Markt hat, nutzen die führenden FPGA-Hersteller Verfahren, die eine Steigerung der Ausbeute durch die Nutzung von defekten FPGAs ermöglichen.

## 1.2 Stand der Technik

Eine spezielle Routing-Architektur, die lange Verbindungssegmente verwendet, ermöglicht dem FPGA-Hersteller Altera die Anwendung eines Verfahrens für die Nutzung

von defekten FPGAs, das auf redundant vorhandenen Spalten von Logikblöcken basiert. Defekte Logikblock-Spalten werden nach der FPGA-Herstellung durch redundant vorhandene Spalten ersetzt. Die Ansteuerung der Logikblöcke durch den Konfigurationsspeicher auf dem FPGA wird so verändert, dass die Programmierung des FPGAs wie gewohnt erfolgen kann. Altera konnte durch die Anwendung dieses Verfahrens bereits in einigen der angebotenen FPGA-Familien die Ausbeute an nutzbaren FPGAs steigern [alt00] und setzt auch in der aktuellen Herstellung der Stratix IV-FPGAs ein Verfahren ein, das mit redundanten Schaltungen arbeitet [alt08]. Dieses Verfahren kann bei FPGAs, deren Architektur viele kurze Verbindungssegmente enthalten (*island-style* und *row-based* FPGAs, siehe Abschnitt 2.1.2), nur mit einer aufwändigen Erweiterung der Verbindungsstrukturen eingesetzt werden.

Der FPGA-Hersteller Xilinx wendet für seine im *island-style* konstruierten FPGAs ein Verfahren an, das nicht auf eine Veränderung der bestehenden FPGA-Architektur angewiesen ist und zudem keine Lokalisierung der Defekte auf dem FPGA erfordert[1]. Leicht defekte FPGAs werden erneut getestet, wobei ein spezieller Test durchgeführt wird, bei dem nur die von dem Kunden tatsächlich genutzten Teile des FPGAs getestet werden. Wenn die FPGAs diesen Test erfolgreich durchlaufen, werden sie dem Kunden für eine Serienproduktion verkauft. Ein Nachteil dieser Vorgehensweise ist der Verlust der Rekonfigurierbarkeit der FPGAs. Die defekten FPGAs können nur mit FPGA-Konfigurationen verwendet werden, mit denen sie vorher bei dem FPGA-Hersteller getestet wurden. Wenn später während der Produktion eine Nachbesserung des Designs erforderlich ist, können die bereits mit der FPGA-Konfiguration getesteten und erworbenen FPGAs nicht mehr verwendet werden. Ein nachträgliches Update einer FPGA-Konfiguration nach der Auslieferung des Produktes an den Kunden ist ebenfalls nicht mehr möglich.

## 1.3 Wesentliche Beiträge der Arbeit

Der Einsatz der oben vorgestellten Verfahren bei den führenden FPGA-Herstellern zeigt, dass die Steigerung der Ausbeute durch die Verwendung von defekt hergestellten FPGAs von zunehmender Bedeutung ist. Das von dem FPGA-Hersteller Altera verwendete Verfahren kann jedoch nicht für alle FPGA-Architekturen problemlos verwendet werden. Auch das von Xilinx genutzte Verfahren ist nicht unproblematisch aufgrund der eingeschränkten Rekonfigurierbarkeit des FPGAs.

Alternativ zu der Anpassung eines FPGAs an die vorliegende FPGA-Konfiguration besteht eine naheliegende Möglichkeit für die Nutzung von teildefekten FPGAs darin, die bei der Programmierung vorhandene Flexibilität auszunutzen. Da FPGAs aus vielen

[1] Easy Path FPGA Series [eas08]

gleich aufgebauten Schaltungselementen bestehen, kann die Konfiguration eines FPGAs derart erfolgen, dass die defekten Schaltungselemente des FPGAs nicht verwendet werden. Die Eignung dieses Ansatzes für eine Serienproduktion hängt von der Zeit (und damit den Kosten) ab, die für die Durchführung der folgenden Schritte benötigt wird:

1. *Lokalisierung der defekten Schaltungsteile auf dem FPGA*
2. *Erstellung einer individuellen FPGA-Konfiguration unter Umgehung der Defekte*

Für den in der vorliegenden Arbeit entwickelten Ansatz ist eine Lokalisierung der defekten Logikblöcke, also eine Diagnose der defekt produzierten FPGAs, eine unabdingbare Voraussetzung. Da in dem Ansatz defekte Logikblöcke umgangen werden, müssen diese zuvor in einer Diagnose des FPGAs identifiziert werden. Forschungen zu alternativen Diagnose-Verfahren zielen darauf ab, die momentan sehr hohen Kosten für eine Diagnose eines FPGAs zu reduzieren. Aufgrund der technologischen Entwicklung wird allgemein vermutet, dass zukünftige FPGA-Generationen deutlich mehr Defekte enthalten, als die zur Zeit produzierten FPGAs. Sinkende Kosten auf der Seite der Diagnose-Verfahren und steigende Kosten aufgrund einer sinkenden Ausbeute an FPGAs könnten so in Zukunft zu wirtschaftlich einsetzbaren Diagnose-Verfahren führen. Das Problem einer effizienten Lokalisierung von defekten Schaltungsteilen auf einem FPGA wird durch den in dieser Arbeit entwickelten Ansatz nicht behandelt. Die Position der defekten Logikblöcke innerhalb des FPGAs wird als bekannt vorausgesetzt.

Da die Programmierung von teildefekten FPGAs innerhalb einer Serienproduktion ein zeitkritischer Faktor ist, kann die Platzierung und Verdrahtung der Schaltung nicht für jedes FPGA individuell erfolgen, bevor das FPGA in der Produktion programmiert wird. Existierende Forschungsansätze, die eine beschleunigte Programmierung von teildefekten FPGAs in der Produktion ermöglichen, können wie folgt grob kategorisiert werden:

- *Dynamische Platzierung und Verdrahtung auf dem FPGA*
  Die Platzierung und die Verdrahtung der Schaltung werden erst nach der Programmierung in der Produktion auf dem FPGA durchgeführt.
- *Anpassung einer bestehenden Konfiguration des FPGAs unter Umgehung der Defekte*
  Eine flexible FPGA-Konfiguration wird jeweils mit einem geringen Zeitaufwand direkt vor oder direkt nach der Programmierung in der Produktion an das jeweils vorliegende defekte FPGA angepasst.

Der in der vorliegenden Arbeit entwickelte Ansatz kann der letztgenannten Kategorie zugeordnet werden. Eine besondere Herausforderung für Forschungsansätze in dieser Kategorie ist die Vorhersage des kritischen Pfades für die auf dem FPGA realisierte Schaltung, da sich die Anpassung der FPGA-Konfiguration an ein defektes FPGA auf die Verzögerungszeiten innerhalb dieser Schaltung auswirken kann. Da vorher nicht bekannt ist, welche Defekte auf den zu programmierenden FPGAs vorliegen, kann für

die Schaltung nur der längste Pfad, der aufgrund einer Anpassung an ein defektes FPGA entstehen kann, als obere Schranke für die Verzögerungszeit garantiert werden. Diese pessimistische obere Schranke für die Verzögerungszeit muss daher als kritischer Pfad für die realisierte Schaltung angesehen werden.

In der vorliegenden Arbeit wurde erstmals ein Verfahren entwickelt, das IP-Cores in einer späten Phase des Design-Flows durch defekttolerante IP-Cores ersetzt. Abbildung 1.1 skizziert diesen Vorgang im Hard-Core basierten Verfahren. Die Schaltungsbeschreibung, die in Form einer Netzliste vorliegt, wird zunächst, wie im ursprünglichen Design vorgesehen, auf eine FPGA-Konfiguration abgebildet. Das über eine statische Timing-Analyse ermittelte Zeitverhalten der Schaltung innerhalb der FPGA-Konfiguration wird als Grundlage für das Re-Mapping der ursprünglichen IP-Cores herangezogen. Ein Re-Mapping, und somit die Einführung von defekttoleranten Schaltungsteilen, findet nur für IP-Cores statt, in denen für das Zeitverhalten dieser Schaltungsteile im ungünstigsten Fall ausreichend Slack vorhanden ist. Durch dieses Vorgehen wird die Timing-Closure[2] für den abschließenden Place&Route-Vorgang, der aus dem DT-Mapping eine DT-Konfiguration erzeugt, erleichtert.

Für die Anpassung der defekttoleranten Schaltungsteile an ein defektes FPGA wurde in der vorliegenden Arbeit ein Verfahren entwickelt, das auf einer Idee basiert, die erstmals von Lach et al. [LMSP98a] vorgestellt wurde. Für jedes IP-Core werden mehrere Ausführungen erstellt, die räumlich unterschiedlich verteilt realisiert sind. Bei einer defekten Teil-Schaltung in einem FPGA wird bei dessen Programmierung wenn möglich eine Ausführung des IP-Cores verwendet, die diese Teil-Schaltung für die korrekte Funktion nicht benötigt.

Das Verfahren kann defekte FPGAs nur dann nutzen, wenn die defekten Schaltungsteile sich ausschließlich auf die Logikblöcke dieser FPGAs auswirken. Da von den FPGA-Herstellern keine Informationen herausgegeben werden, aus denen Rückschlüsse auf den bei der Herstellung der FPGAs erzielten Yield möglich sind, konnte im Rahmen dieser Arbeit nur eine grobe Abschätzung der möglichen Steigerung des Yields, die entsteht, wenn FPGAs mit defekten Logikblöcken genutzt werden können, erfolgen. Diese Abschätzung wurde basierend auf einem gängigen Ausbeutemodell vorgenommen und führt zu dem Ergebnis, dass der *random yield* bei einem ursprünglichen Wert von $Y_{RANDOM} = 0,5$ auf $Y_{RANDOM} = 0,66$ verbessert wird, wenn 40% der auf dem Chip im Durchschnitt auftretenden Defekte auf einen Logikblock begrenzt sind und FPGAs mit defekten Logikblöcken nicht als Ausschuss, sondern als verwendbare Chips betrachtet werden. Wenn man von $Y_{RANDOM} = 0,8$ ausgeht, kann unter den gleichen Bedingungen eine Verbesserung auf $Y_{RANDOM} = 0,875$ erreicht werden. Um einen Anreiz für den Kunden des FPGA-Herstellers zu schaffen, defekte FPGAs in der Serienproduktion

[2] „Timing-Closure“ beschreibt die Back-End-Aufgabe, die sich damit auseinandersetzt, das vom Front-End-Designer erwartete Zeitverhalten (Timing) zu erreichen bzw. zu implementieren.

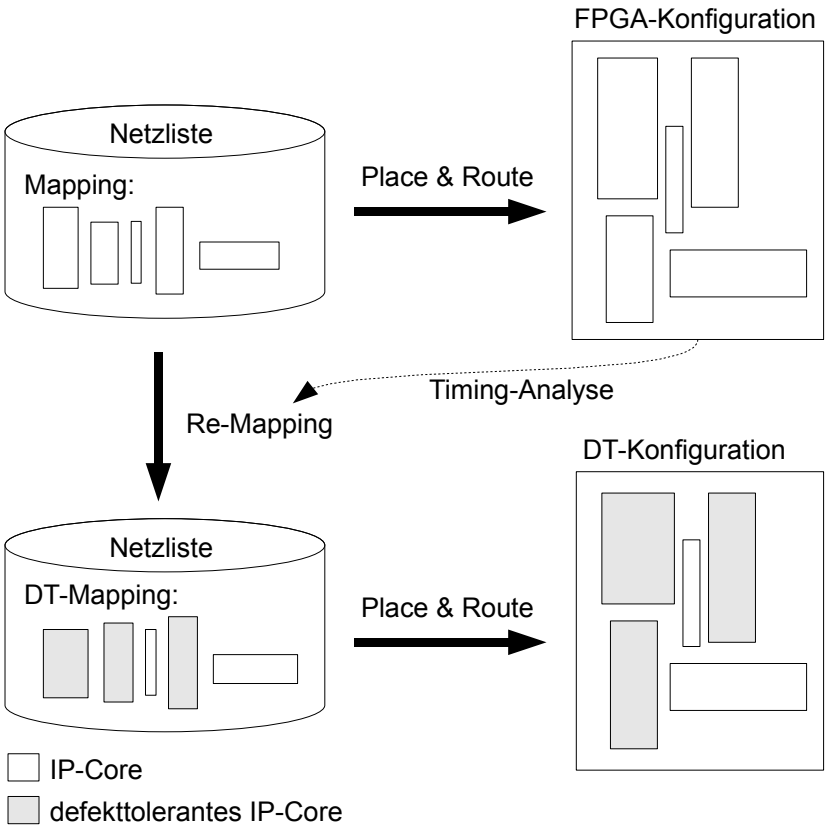



Abbildung 1.1: Das Re-Mapping im Hard-Core basierten Verfahren

einzusetzen, müssten defekte FPGAs jedoch günstiger als voll funktionsfähige FPGAs angeboten werden.

## 1.4 Inhaltsübersicht

Der weitere Aufbau der Arbeit gliedert sich wie folgt: **Kapitel 2** stellt einige Grundlagen vor, die für das Verständnis der folgenden Kapitel erforderlich sind. Da der in dieser Arbeit vorgeschlagene Ansatz auf einer vorherigen Diagnose der defekten FPGAs basiert, die mit den zur Zeit in der Herstellung von FPGAs üblichen Testmethoden nicht wirtschaftlich durchgeführt werden kann, stellt **Kapitel 3** Forschungen zu alternativen Diagnose-Verfahren vor, die eine Reduzierung der Kosten für die Diagnose eines FPGAs zum Ziel haben. **Kapitel 4** stellt im Anschluss daran andere Ansätze für die Nutzung von defekten FPGAs vor und vereinfacht durch eine Klassifizierung dieser Ansätze einen späteren Vergleich mit dem in dieser Arbeit entwickelten Verfahren. In **Kapitel 5** werden dann, nach einem ersten groben Überblick, alle Komponenten des in dieser Arbeit vorgeschlagenen Ansatzes ausführlich vorgestellt. In **Kapitel 6** erfolgt schließlich eine ausführliche Bewertung des Ansatzes und ein Vergleich mit den in Kapitel 4 dargestellten, vergleichbaren Arbeiten.

# 2 Grundlagen

Im diesem Kapitel werden Grundlagen dargestellt, die für ein Verständnis dieser Arbeit sinnvoll sind und es wird für die einzelnen Bereiche auf weiterführende Literatur verwiesen.

## 2.1 Aufbau eines FPGAs

Ein Field Programmable Gate Array (FPGA) ist ein programmierbarer Logikbaustein, der die Realisierung von digitalen logischen Schaltungen ermöglicht. Er besteht aus einer zweidimensionalen programmierbaren Schaltmatrix (engl. *Array*). Abbildung 2.1 zeigt die Hauptbestandteile dieser Matrix.

Die I/O Blöcke können als Eingabe-Block (engl. ***I**nput*) oder Ausgabe-Block (engl. ***O**utput*) konfiguriert werden und sind mit den Pins des FPGA-Bausteins verbunden, um eine Kommunikation mit der Außenwelt zu ermöglichen. Jeder Logikblock wird so konfiguriert, dass er einen Teil der zu realisierenden Schaltung implementiert. Die mit Hilfe der Logikblöcke implementierten Schaltungsteile werden durch ein programmierbares Netzwerk von Verbindungen (engl. *Routing*) untereinander und mit den I/O-Blöcken verbunden. Im Folgenden soll nach einer kurzen Beschreibung des Aufbaus eines Logikblocks auf Besonderheiten in der Anordnung der programmierbaren Verbindungen in unterschiedlichen FPGA-Architekturen eingegangen werden.

### 2.1.1 Aufbau eines Logikblocks

Um ein Schaltnetz in einem FPGA zu implementieren, werden die Schaltfunktionen nicht, wie aus dem VLSI-Entwurf bekannt, durch einzelne Gatter, sondern durch spezielle programmierbare Speicherbausteine, sogenannte *look-up table* (LUT) oder/und Multiplexer realisiert. Abbildung 2.2 zeigt die Schaltfunktion $out = i_0 i_3 \vee i_1 i_2 i_3 \vee i_0 i_1 i_2$ in verschiedenen Darstellungen.

Damit es möglich ist, neben digitalen Schaltnetzen in einem FPGA auch Schaltungen mit Speicherverhalten zu realisieren, enthält ein Logikblock neben den Komponenten für die Realisierung von logischen Schaltfunktionen zusätzlich Flipflops. Abbildung 2.3 skizziert den Aufbau eines Logikblocks.

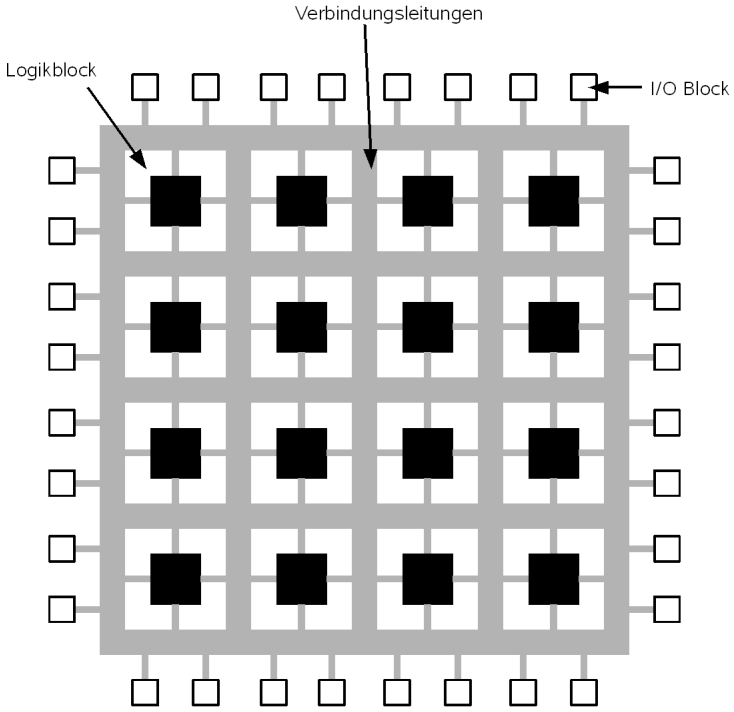
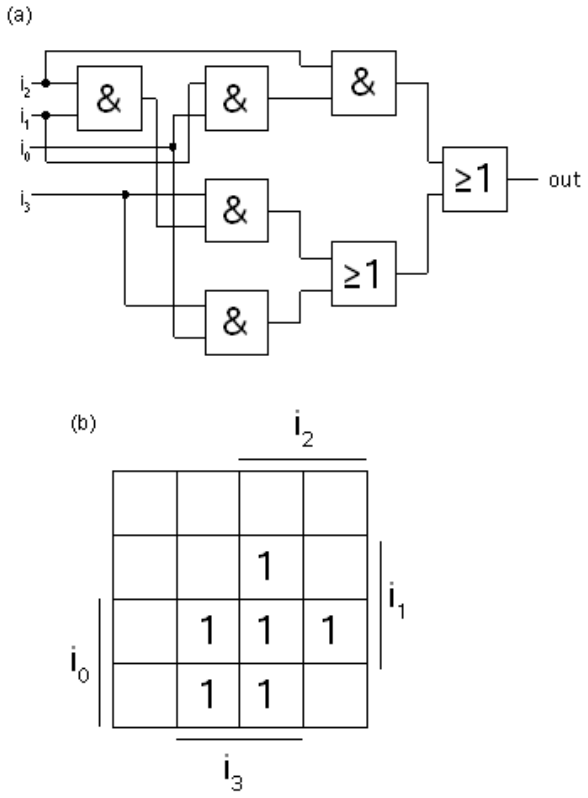



Abbildung 2.1: Hauptbestandteile eines FPGAs [REGSV93]

Der hier abgebildete Logikblock enthält N Logik-Elemente (engl. *basic logic elements* – BLE), die jeweils ein LUT mit *k* Eingängen enthalten. Der Ausgang des LUT wird über einen Multiplexer, entweder direkt oder über ein Flipflop, mit dem Ausgang des BLE verbunden. Die *I* Eingänge und *N* Ausgänge des Logikblocks sind durch eine Multiplexer-Schaltung an dem Eingang jedes LUT verfügbar. In der Abbildung wird die Programmierung zwischen den Logikblöcken des FPGAs und den Verbindungen zu anderen Logikblöcken (*Routing wire segments*) über SRAM-Zellen skizziert (*Programmable switch*). Abbildung 2.4 zeigt drei verschiedene Arten von Schaltern, die in SRAM-FPGAs eingesetzt werden, um diese Verbindungen zu realisieren. Andere Ansätze für die Programmierung der in einem FPGA enthaltenen programmierbaren Schalter, wie z.B. EPROM-Zellen oder Antifuse, werden innerhalb dieser kurzen Einführung in den Aufbau eines FPGAs nicht näher erläutert, da die für die Programmierung benutzte Technologie keine Auswirkungen auf den im Folgenden erläuterten Ansatz hat.

### 2.1.2 Programmierbare Verbindungsleitungen

Im Handel erhältliche FPGAs können bzgl. der Struktur des Verbindungsnetzwerkes zwischen den Logikblöcken und den I/O-Blöcken in drei Gruppen eingeteilt werden [BRM99]: *island-style*, *hierarchical* und *row-based* FPGAs.




(c)

| $i_3$ | $i_2$ | $i_1$ | $i_0$ | out |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Abbildung 2.2: Darstellung einer Schaltfunktion als (a) Gatternetzliste, (b) KV-Diagramm und (c) Inhalt eines LUT

### Verbindungsstruktur im island-style FPGA

Das zunächst betrachtete *island-style* FPGA-Modell ist in der Forschergemeinde weit verbreitet [CCP06]. Abbildung 2.5 skizziert die Verbindungsstruktur eines *island-style* FPGA. Die Logikblöcke sind jeweils von kurzen Anschlussleitungen umgeben (jeweils vier, in dem hier abgebildeten Modell), die über programmierbare Schalter mit den umliegenden Leitungen verbunden werden können (*programmable connection switch*). Diese Schalter werden zusammengefasst als *connection block* bezeichnet. An den Kreuzungspunkten der horizontal und vertikal angeordneten Verbindungsleitungen sind verschiedene Leitungslängen skizziert: Leitungen, die mehrere Kreuzungspunkte überspannen (*Long wire segment*) und Leitungssegmente, die durch Schalter, die unter der Bezeichnung *switch block* zusammengefasst werden, mit anderen Leitungssegmenten verbunden werden können, um eine längere Leitung zu bilden. Diese Leitungen in verschiedenen Längen sind ein wichtiger Bestandteil eines FPGAs, da jeder Schalter innerhalb einer längeren, d.h. unter Umständen aus mehreren kurzen Leitungssegmenten zusammengesetzten, Verbindung den Widerstand und somit das Delay der Verbindung

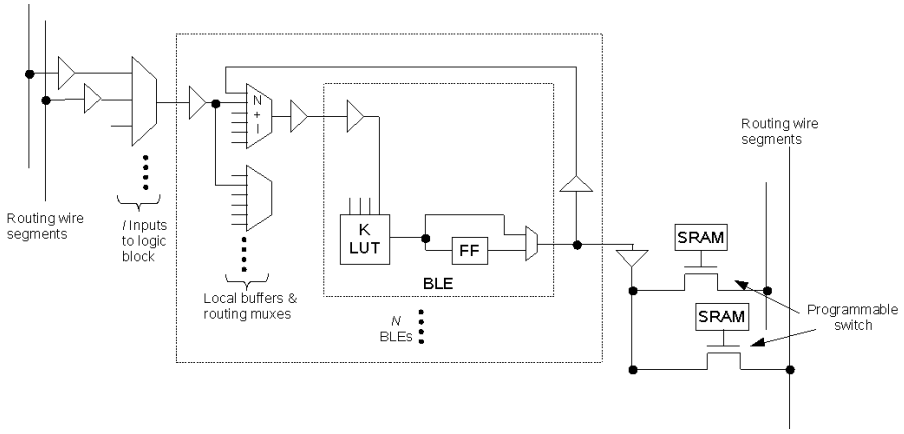



Abbildung 2.3: Ein detailliert dargestellter Logikblock [CCP06]

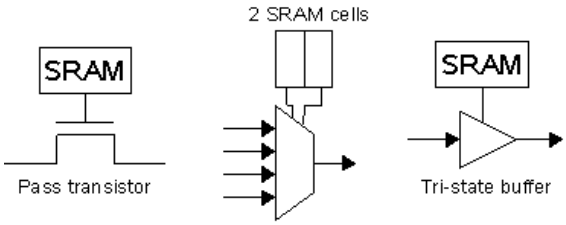



Abbildung 2.4: Verschiedene, in SRAM-FPGAs verwendete Schalter [BRM99]

sowie die insgesamt für das FPGA benötigte Fläche erhöht. Kurze Leitungssegmente sind allerdings von Vorteil, sobald ein Teil eines *long wire segments* nicht benötigt wird. In diesem Fall würde das *long wire segment* die zu treibende Kapazität und damit das Delay der Verbindung, sowie die für die Verbindung auf dem FPGA benötigte Fläche erhöhen [BR99].

Im Handel erhältliche FPGAs mit einer *island-style*-Architektur stellen daher eine Auswahl von verschiedenen Verbindungslängen mit einer unterschiedlichen Anzahl von Schaltern und Verbindungsmöglichkeiten zur Verfügung. Als Beispiel zeigt Abbildung 2.6 die Verbindungsmöglichkeiten in einem Xilinx Virtex-II FPGA.

Das Delay der Verbindungsleitungen in einem FPGA ist, aufgrund der Auswirkungen von kleinen Änderungen auf die Verzögerungszeiten und somit auf die Performance der durch das FPGA realisierten Schaltung, innerhalb der hier vorliegenden Arbeit besonders zu berücksichtigen:

> „Given the large resistances and capacitances involved, it is no surprise that FPGA routing delay is critical. [...] It is not the magnitude of the delay, but the character of the delay that causes routing problems. FPGA delay

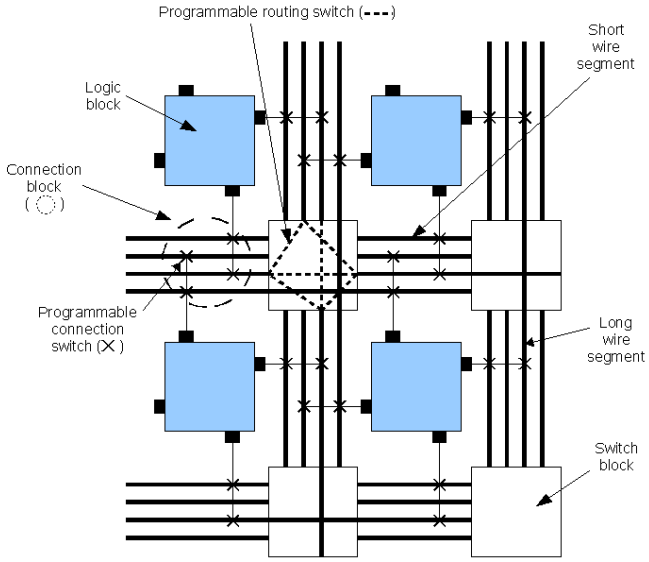



Abbildung 2.5: Ein *island-style* FPGA [BRM99]

comes in large increments, so a small change in path length can make a large difference in delay. [...]“ [Tri95]

### Verbindungsstruktur im *row-based* FPGA

Abbildung 2.7 zeigt die grundlegende Architektur von *row-based* FPGAs, wie sie z.B. von der Firma Actel angeboten werden [Act97]. In dieser Architektur sind die Verbindungsleitungen ausschließlich horizontal angeordnet. Auch in dieser Architektur gibt es Verbindungsleitungen mit unterschiedlichen Längen, die über Schalter miteinander verbunden werden (*Segmented Routing Tracks*). Die Logikmodule (*Logic Modules*) sind einfacher aufgebaut als die bisher betrachteten Logikblöcke, sollen aber im Rahmen dieser Arbeit nicht näher betrachtet werden.

### Verbindungsstruktur im *hierarchical* FPGA

Der FPGA-Hersteller Altera bietet FPGAs an, deren Verbindungsstruktur hierarchisch (*hierarchical*) aufgebaut ist. Abbildung 2.8 (übernommen aus [BR96]) zeigt den Aufbau eines Flex 8000 FPGAs von Altera. Ein Logikblock (*logic array block* – LAB) besteht in dieser Architektur aus acht Logikelementen und einer lokalen Verbindungsstruktur zwischen diesen Logikelementen. Diese lokale Verbindungsstruktur wird ausschließlich für die Verbindung zwischen den Logikelementen verwendet. Die Zeilen

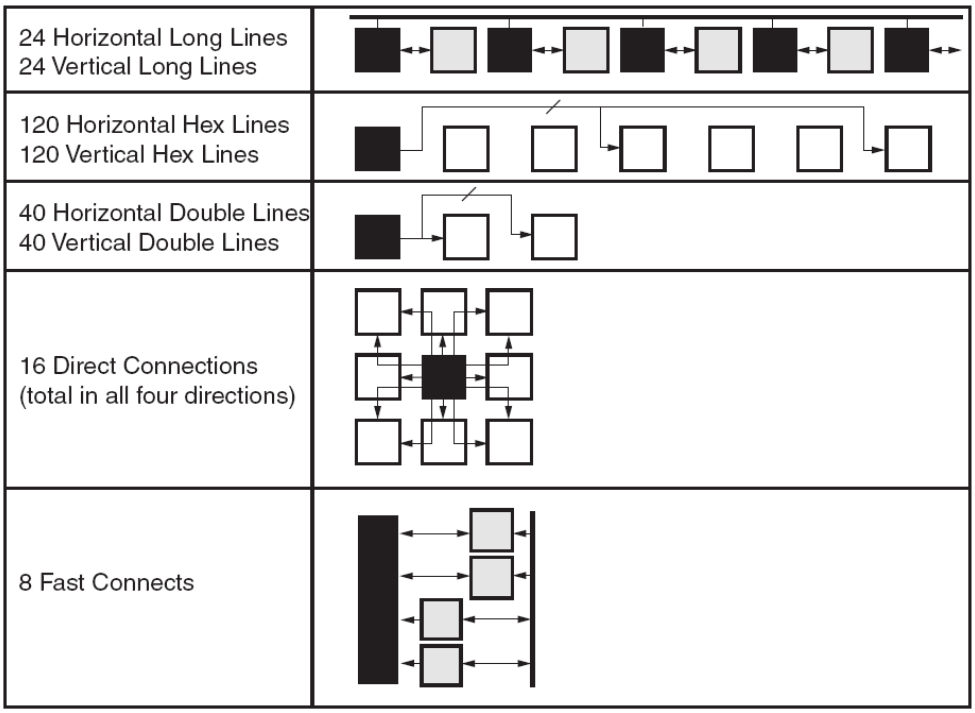



Abbildung 2.6: Verbindungsmöglichkeiten in einem Xilinx Virtex-II FPGA [Xil07b]

und Spalten des FPGA beinhalten ausschließlich Verbindungsleitungen, die die gesamte Breite, bzw. Höhe des FPGAs abdecken. Die Zeilen des FPGA werden für die Verbindung der LAB untereinander verwendet und die Spalten verbinden die Zeilen untereinander. Durch die Verwendung der Verbindungsleitungen in den Spalten, Zeilen und an den Logikblöcken für jeweils nur eine Aufgabe konnte Altera diese Verbindungen für ihre jeweilige Aufgabe optimieren. Ein weiterer Vorteil dieser hierarchisch aufgebauten Verbindungsarchitektur ist eine genauere Vorhersage der Verbindungszeiten zur Unterstützung des *Place&Route*-Vorgangs, verglichen mit der Vorhersage der Verbindungszeiten in *island-style* FPGA [WL94]. FPGA-Architekturen, auf denen die Verbindungsleitungen die gesamte Breite, bzw. Höhe des FPGAs abdecken, werden auch als Bus-basierte FPGA-Architekturen (*bus based*) bezeichnet.

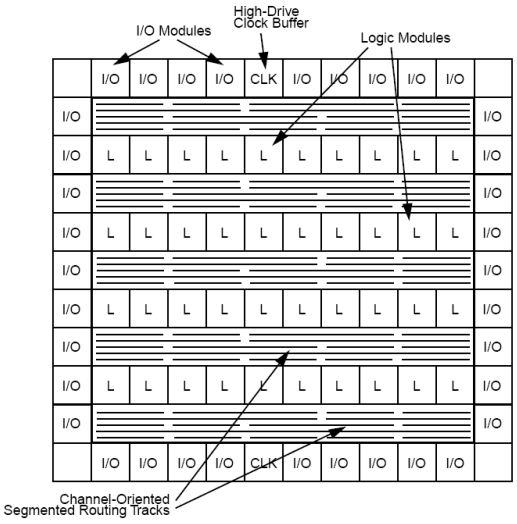



Abbildung 2.7: Die grundlegende Architektur der FPGAs der Firma Actel

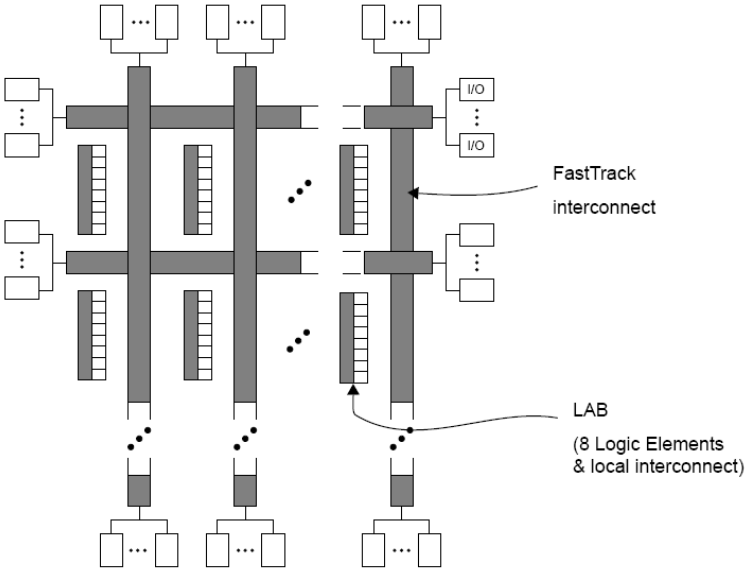



Abbildung 2.8: Die Architektur eines Altera Flex 8000 FPGAs [BR96]

## 2.2 Rechnergestützter Schaltungsentwurf für FPGAs

Die Realisierung einer Schaltung auf einem SRAM-FPGA geschieht, wie bereits weiter oben erwähnt, durch die Belegung von Speicherzellen, die dann für eine entsprechende Konfiguration des FPGA sorgen. Im Folgenden wird diese Belegung der Speicherzellen als *FPGA-Konfiguration* bezeichnet. Abbildung 2.9 stellt ein FPGA in zwei Ebe-

nen dar: Die Realisierung der Schaltung erfolgt durch Elemente im *Logic Layer*, die durch die Speicherzellen im *Configuration Memory Layer* konfiguriert werden. Nachdem im nächsten Abschnitt der allgemeine Entwurfsablauf (engl. *design flow*) beschrieben worden ist, dessen Ziel eine FPGA-Konfiguration ist, geht Abschnitt 2.2.2 auf den zunehmend IP-basierten FPGA-Entwurf ein. Eine wichtige Unterstützung für die im Entwurfsablauf genutzten Entwurfswerkzeuge stellt die Timing-Analyse dar. Da die Timing-Analyse im Hard-Core basierten Ansatz von besonderer Bedeutung ist, wird der Ablauf dieser Analyse kurz in Abschnitt 2.2.3 erläutert.

### 2.2.1 Allgemeiner Entwurfsablauf

Um die zu implementierende Schaltung für die Entwurfswerkzeuge verständlich zu beschreiben, nutzen FPGA-Schaltungsdesigner eine graphische Schaltplaneingabe (engl. *schematic entry*) oder eine Hardwarebeschreibungssprache, in der Regel VHDL oder Verilog. Die durch den FPGA-Hersteller bereitgestellten Entwurfswerkzeuge ermöglichen die automatische Generierung einer FPGA-Konfiguration aus dieser Schaltungsbeschreibung. Abbildung 2.10 zeigt den Weg von der Schaltungsbeschreibung zu einer FPGA-Konfiguration.

Innerhalb der in der Abbildung beschriebenen Synthese (*Synthesize to logic blocks*) besteht der erste Schritt darin, die Beschreibung der Schaltung in eine Gatternetzliste umzuwandeln. Nach einer technologieunabhängigen Optimierung, wird diese Gatternetzliste in eine Logikblock-Netzliste konvertiert. Ziel dieser in Abbildung 2.11 skizzierten Konvertierung ist die Minimierung der Anzahl der verwendeten Logikblöcke und/oder die Maximierung der Geschwindigkeit der Schaltung.

Bei der anschließenden Platzierung und Verdrahtung der Logikblöcke auf dem FPGA (engl. *Place & Route*, siehe auch Abbildung 2.10) wird jeder Logikblock aus der Netzliste derart auf dem FPGA platziert, dass verbundene Logikblöcke möglichst dicht zusammen liegen (*wire-length-driven placement*) und/oder die Geschwindigkeit der

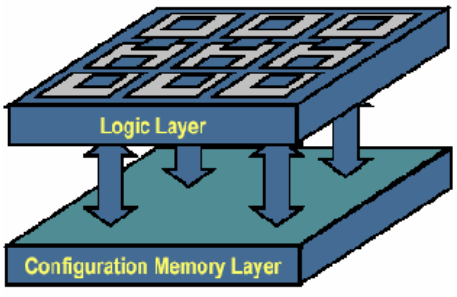



Abbildung 2.9: Die Konfiguration eines FPGA

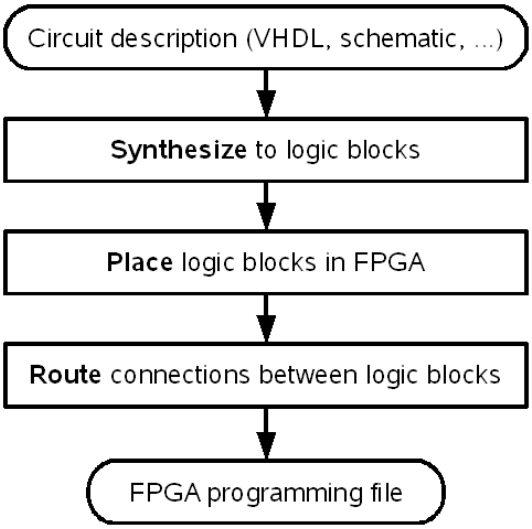



Abbildung 2.10: FPGA-Schaltungsdesign: Der automatisierte Weg zur FPGA-Konfiguration [BRM99]

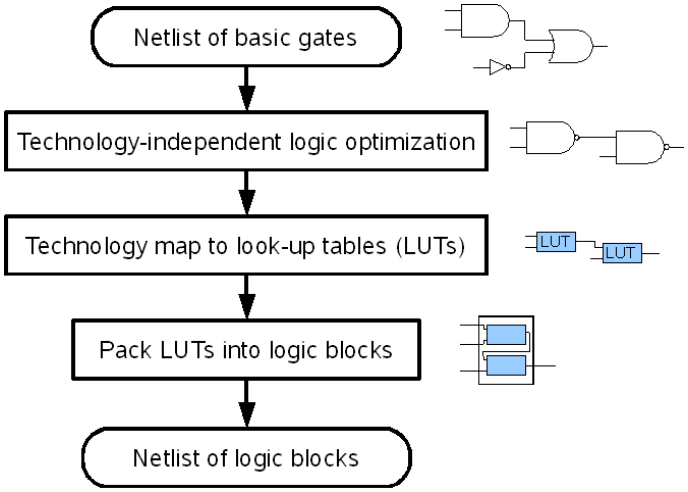



Abbildung 2.11: Details aus der Synthese [BRM99]

Schaltung bzgl. vorgegebener Timing-Constraints optimiert wird (*timing-driven placement*). Erst nach dem vollständigen Place & Route kann die Schaltung zeitgenau analysiert und simuliert werden, da die verwendeten Verbindungsleitungen und damit die Verzögerungszeiten (engl. *Delay*) der Verbindungen erst zu diesem Zeitpunkt feststehen. Die Anzahl der auf einem Chip möglichen Transistoren steigt mit jeder Prozessgeneration. Gründe dafür sind verbesserte Auflösungen für die bei der Chip-Herstellung verwendeten Lithografie, aber auch andere Verbesserungen der Produktionsbedingungen, die es erlauben, immer größere Chips zu produzieren. Während in den ersten FPGAs nur

wenige Gatter nachgebildet werden konnten, enthalten die mit den neuesten Prozesstechnologien gefertigten FPGAs zehntausende von Logikblöcken [1] sowie zusätzliche festverdrahtete und in das FPGA eingebettete Komponenten (Multiplizierer / Speicher / Prozessoren). Die ständige Weiterentwicklung der von den Herstellern bereitgestellten Entwurfswerkzeuge unterstützt den Schaltungsdesigner bei der Ausnutzung dieser festverdrahteten eingebetteten Komponenten.

### 2.2.2 IP-basierter Entwurf

Bereits 1998 wurde durch den damaligen Vizepräsidenten des FPGA-Herstellers Quicklogic, John Birkner, ein Vergleich der damals noch schwach ausgeprägten Wiederverwendung von bereits erstellten Teilschaltungen, bezeichnet als IP-cores, in FPGA-Designs mit der Verwendung von Standard-Logik-Chips in der digitalen Elektronik, die in den 70er Jahren durch das Erscheinen der 74xx-Serie von der Firma Texas Instruments begann, vorgeschlagen:

> „Back then, when a design project required an arithmetic logic unit, ALU, the best solution, was an off-the-shelf, standard product, 74181 series TTL device. These were available from a variety of sources in a number of speed and power options. [...] Designers found they could get to market faster with lower cost and higher performance by using standard product 7400-series multiplexors, decoders, registers, counters, adders, ALUs, and register files, compared to proprietary hand crafted functions constructed from gates and flip flops. We didn't call those functions intellectual property or IP cores in those days – we called them standard products.“ [Bir98]

Seit 1998 hat sich die Anzahl der angebotenen IP-cores als Standard-Komponenten für den Einsatz auf FPGAs deutlich vergrößert. Für die Hersteller bot sich bei dieser Entwicklung die Möglichkeit, durch kostenlos erhältliche IP-cores komplexe Funktionen auf die Architektur ihrer FPGAs zu optimieren und, neben einer Vereinfachung des Designs für den FPGA-Schaltungsdesigner, durch diese optimierten IP-Cores auch den Wechsel eines Kunden auf die FPGAs eines anderen Herstellers zu erschweren. Mit zunehmender Größe der auf dem Markt verfügbaren FPGAs und der zunehmenden Verbreitung von FPGAs, wächst aber auch der Anteil an IP-Cores, für deren Einsatz eine einmalige Lizenzgebühr an den FPGA-Hersteller gezahlt wird. In den letzten Jahren sind zusätzlich Drittanbieter von IP-Core für FPGA hinzugekommen[2].

[1] Der zur Zeit größte verfügbare FPGA der Firma Xilinx, der Virtex-5 XC5VFX200T enthält 30720 sogenannte „slices“, die jeweils aus vier BLEs bestehen. Jeweils zwei Slices bilden einen Logikblock [Xil08].

[2] Einige dieser Anbieter haben sich der „SignOnce IP License“ angeschlossen, die von der Firma Xilinx im September 2001 geschaffen wurde, um die Lizensierung von IP-Cores verschie-

Den oft hohen Kosten für die Entwicklung eines IP-Core für ein ASIC steht eine Lizenzgebühr gegenüber, die von dem Nutzer eines IP-Cores einmalig an den IP-Core Anbieter gezahlt wird. Diese hohen Kosten erschweren den Einsatz von IP-Core für FPGAs, die oft für Produktionen mit einer kleinen oder mittleren Stückzahl eingesetzt werden. Die IP-Core Anbieter für FPGAs sind daher gefordert neue Modelle für die Lizenzierung von IP-Cores zu entwickeln und für deren technische Realisierbarkeit zu sorgen. Denkbar sind z.B. Lizenzgebühren, die von der produzierten Stückzahl des Produktes abhängen [Kea02] oder nach einer festgelegten Zeit erneuert werden müssen [CK06].

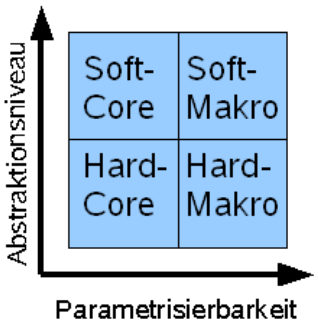



Abbildung 2.12: Soft vs. Hard / Makro vs. Core

Trotz des großen Angebotes von kostenlosen und günstigen IP-Cores für FPGAs durch die FPGA-Hersteller und trotz den in der Praxis oft noch nicht vorhandenen technischen Voraussetzungen für die Nutzung eines angepassten Lizenz-Modells, hat sich der Markt für IP-Cores vor allem durch die Möglichkeit, ganze System on Chip auf einem FPGA zu platzieren, in den letzten Jahren vergrößert. Die steigende Komplexität der FPGA-Designs und die immer wiederkehrende Nutzung von gleichen Standard-Komponenten, wie z.B. Bus-Interfaces oder Signalverarbeitungsblöcken, in einem System on Chip, sorgen dafür, dass der Aufwand für die Erstellung eines neuen FPGA-Designs durch die Verwendung von IP-Cores erheblich verringert werden kann.

Allgemein können die für das Design einer Schaltung auf einem FPGA wiederverwendbaren Schaltungsteile in verschiedenen Formen zur Verfügung gestellt werden. Je nach Form werden sie auch als Hard-Makro, Hard-Core, Soft-Makro und Soft-Core bezeichnet (siehe Abbildung 2.12).

dener Anbieter für die Nutzung in einem Design für Xilinx-FPGAs zu vereinfachen. Unter http://www.xilinx.com/ipcenter/signonce.htm findet man eine Auflistung dieser IP-Core Anbieter.

Die Bezeichnungen „Soft-“ und „Hard-“ bezeichnen in der Regel den Grad der Abstraktion, in der dieser wiederverwendbare Schaltungsteil angegeben wird. „Soft-“ deutet auf eine höhere Abstraktionsebene (z.B. eine Hardwarebeschreibungssprache) hin als „Hard-“ (z.B. eine Logikblock-Netzliste). Die Bezeichnungen „-Makro“ und “-Core“ spielen wie in Abbildung 2.12 angegeben auf die Parametrisierbarkeit des wiederverwendbaren Schaltungsteils an. Oft können noch Parameter verändert werden (z.B. die Bitbreite der Ein- und Ausgänge), bevor aus dem „-Makro“, automatisiert durch die Entwurfswerkzeuge, ein „-Core“ erstellt wird.

In der vorliegenden Arbeit bezeichnet ein Hard-Core einen wiederverwendbaren Schaltungsteil, der bereits in eine Logikblock-Netzliste synthetisiert wurde. Die Logikblöcke in dieser Netzliste wurden bereits relativ zueinander platziert und untereinander verbunden. Somit kann ein Hard-Core als ein fertiger Teil der Schaltung aufgefasst werden, der auf dem FPGA frei platziert werden kann. Eine Verdrahtung der Logikblöcke untereinander ist nach der Platzierung eines Hard-Cores auf dem FPGA nicht mehr notwendig.

Die hier als Hard-Core bezeichnete Form eines IP-Cores wird in anderen Veröffentlichungen auch als *firm IP* bezeichnet, da der Begriff Hard-Core in vielen Veröffentlichungen bereits für die Bezeichnung der auf modernen FPGA enthaltenen festverdrahteten Komponenten, wie z.B. Multiplizierer, Block-RAM oder Prozessoren verwendet wird. Diese festverdrahteten Komponenten werden in der vorliegenden Arbeit *nicht* als Hard-Core bezeichnet[3].

Ein Hard-Core kann durch das Entwurfswerkzeug als zusätzliches Element für die Zielmenge der Synthese einer Verhaltensbeschreibung angesehen werden (*inferencing*) oder direkt in der Hardwarebeschreibungssprache als Komponente genutzt werden (*instantiation*). Das Ergebnis der Synthese ist dann eine Netzliste, die aus Logikblöcken, IP-Cores und evtl. festverdrahteten FPGA-Elementen, wie z.B. Multiplizierern oder Block-RAMs besteht. Abbildung 2.13 zeigt die Synthese allgemeiner als in Abbildung 2.10 dargestellt.

### 2.2.3 Timing-Analyse

In einer synchronen Schaltung müssen die in einem Flipflop zu speichernden Daten eine bestimmte Zeit vor der das Flipflop steuernden Taktflanke anliegen (Set-Up-Zeit) und nach der Taktflanke noch eine bestimmte Zeit stabil bleiben (Hold-Zeit), um die Übernahme der Daten in das Flipflop zu ermöglichen. Die korrekte Funktion einer FPGA-Konfiguration ist nur dann gewährleistet, wenn diese Zeiten eingehalten werden. Neben

[3]Ein Hard-Core kann allerdings auch festverdrahtete Komponenten enthalten. In diesem Fall ist die Platzierbarkeit des Hard-Cores stark eingeschränkt, da von festverdrahteten Komponenten gleicher Art in der Regel nur eine geringe Anzahl auf dem FPGA existiert.

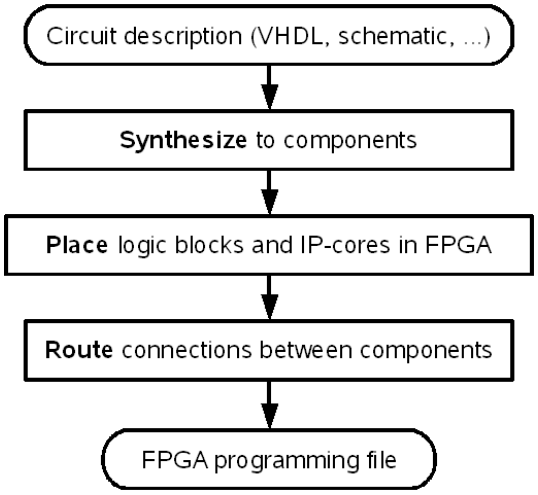



Abbildung 2.13: Eine allgemeinere Darstellung der Erstellung einer FPGA-Konfiguration aus einer Schaltungsbeschreibung

der Länge der Pfade durch ein Schaltnetz ist dabei auch das Verhältnis der Taktflanken zueinander entscheidend.

In Abbildung 2.14 wird für ein einfaches Beispiel mit zwei Flipflops skizziert, welche Zeiträume für eine korrekte Ansteuerung von FF2 berücksichtigt werden müssen.

Eine Verletzung der Set-Up-Zeit ist möglich, wenn die Daten an FF2 nicht rechtzeitig vor der steigenden Flanke an FF2 anliegen. Um dies auszuschließen, muss neben der Set-Up-Zeit des FFs und dem Pfad mit dem längsten Delay auch der kleinstmögliche Taktversatz zwischen den FFs betrachtet werden. Wenn der Taktversatz vernachlässigt wird, kann durch die Ermittlung des Pfades mit dem größten Delay die Taktfrequenz bestimmt werden, mit der die Schaltung maximal betrieben werden kann.

Zu einer Verletzung der Hold-Zeit kann es in der Schaltung kommen, wenn die Daten an FF2 nicht lange genug nach der steigenden Taktflanke anliegen. Für FPGAs werden in der Regel Flipflops eingesetzt, die eine Hold-Zeit von Null haben (*zero-hold-time flip-flops*). Wenn, wie in Abbildung 2.14 verdeutlicht, durch einen Taktversatz (*clock skew*) das zweite Flipflop später als das erste Flipflop schaltet, kann es allerdings dazu kommen, dass die Daten an dem Flipflop anliegen, bevor die vorherigen Daten übernommen worden sind, so dass auch hier die Hold-Zeit verletzt wurde. Neben dem Pfad mit dem kürzesten Delay, muss also auch der größtmögliche Taktversatz in Betracht gezogen werden, um eine Verletzung der Hold-Zeit an FF2 auszuschließen. Um den Taktversatz möglichst gering zu halten, werden spezielle Verteilungen des Taktsignals (*Clock Trees*) verwendet. Wenn der maximale Taktversatz kleiner ist als die Summe der Verzögerung des Signals durch ein Flipflop und durch eine zeitlich minimale Verbindung, dann kann innerhalb des FPGAs die Hold-Zeit nicht verletzt werden.

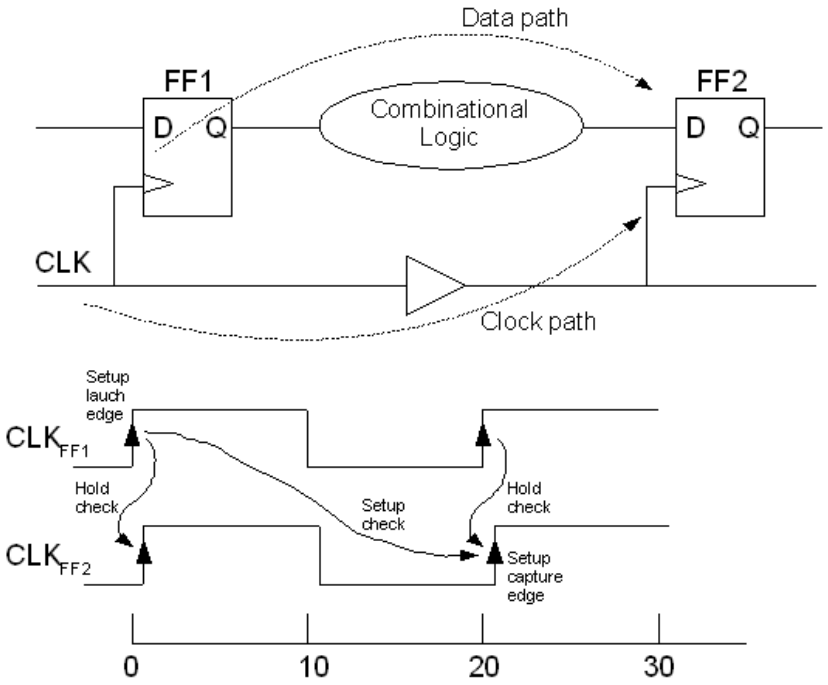



Abbildung 2.14: Die Überprüfung der Setup- (Setup check) und der Hold-Zeit an einem Beispiel mit zwei Flipflops [Syn04]

Die Timing-Analyse [RBHSC82] wird von Entwurfswerkzeugen unter anderem verwendet, um die Taktfrequenz zu ermitteln, mit der eine FPGA-Konfiguration maximal betrieben werden kann. Für diese Analyse wird jedes der in der FPGA-Konfiguration enthaltenen Schaltnetze als gerichteter Graph dargestellt. Eine mögliche Darstellung zeigt Abbildung 2.15. Die Knoten repräsentieren die Ein-und Ausgangs-Pins der in der Schaltung verwendeten Elemente, wie z.B. LUTs und Register. Die Kanten werden zwischen den Eingängen der kombinatorischen Elemente, in einem FPGA also vor allem LUTs, und deren Ausgängen eingezeichnet. Für alle Verbindungen der Elemente der Schaltung untereinander werden ebenfalls Kanten eingezeichnet. An jeder Kante wird nun ein Delay annotiert, das dem Delay für den maximalen Pfad innerhalb eines kombinatorischen Elements bzw. dem Delay für die Verbindung zwischen den Elementen entspricht. Alle Eingänge eines Schaltnetzes, d.h. externe Eingänge, wie in Abbildung 2.15 $In_A$ und $In_B$, oder die Ausgänge der Register haben keine eingehenden Kanten im Graphen. Alle Ausgänge haben keine ausgehenden Kanten.

Wenn die Anfangsknoten des Graphen mit der Ankunftszeit annotiert werden, lässt sich die minimal für den korrekten Betrieb der Schaltung benötigte Länge einer Taktperiode rekursiv ermitteln mit

$$T_{arrival}(i) = max_{\forall j \in fanin(i)}\{T_{arrival}(j) + delay(j,i)\}\ ,$$

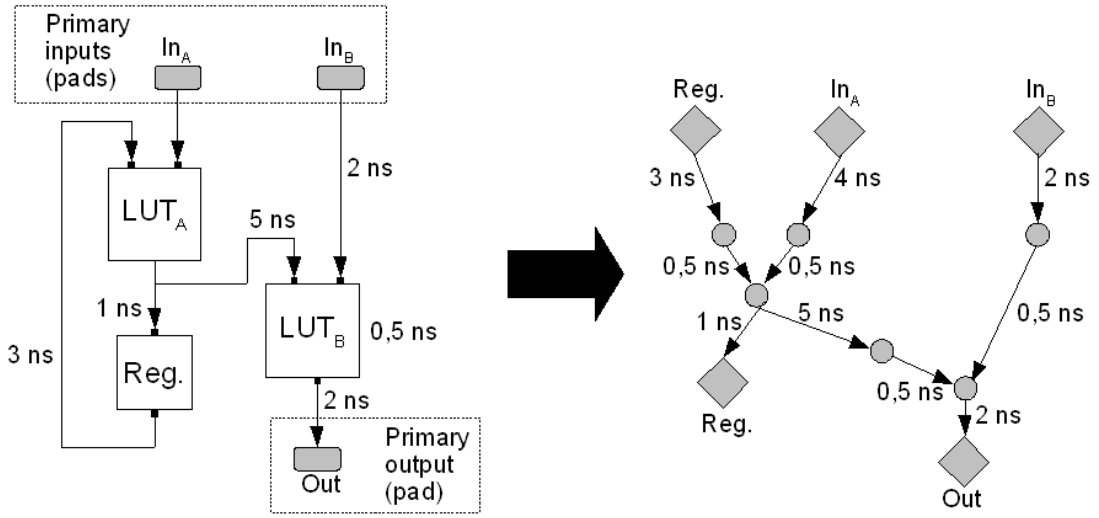



Abbildung 2.15: Eine einfache Schaltung als gerichteter Graph dargestellt [BRM99]

wobei $i$ und $j$ Knoten sind, $delay(i, j)$ der Delay-Wert der Kante zwischen den Knoten $i$ und $j$ ist und $fanin(i)$ die Menge aller Vorgängerknoten des Knoten i bezeichnet. Wenn für alle Knoten $T_{arrival}$ berechnet wurde, definiert der Knoten mit dem größten Wert für $T_{arrival}$ die minimale Taktperiode für die Schaltung. In dem Beispiel in Abbildung 2.15 ist die minimale Länge der Taktperiode durch den Knoten *Out* auf 12 ns festgelegt.

Neben der Ermittlung der maximalen Taktfrequenz erlaubt die Timing-Analyse [RBHSC82] die Beurteilung von vorliegenden Timing-Problemen. Wenn eine vorgegebene maximale Taktfrequenz und somit in jedem kombinatorischen Schaltnetz innerhalb einer Schaltung eine maximale Verzögerungszeit eingehalten werden soll, kann für jeden Knoten die Zeit $T_{required}$ berechnet werden, zu der die Daten an diesem Knoten benötigt werden, damit das Schaltnetz insgesamt die gewünschte maximale Verzögerungszeit einhält. Bezogen auf das oben angegebene Beispiel soll jetzt an den beiden Ausgängen des abgebildeten Schaltnetzes (Reg. und Out) die Ankunftszeit 12 ns annotiert werden. Diese Annotierung beschreibt die Zeit , zu der die Ankunft der Daten erforderlich ist, um die ermittelte maximale Verzögerungszeit einzuhalten. Die erforderliche Ankunftszeit für die restlichen Knoten kann dann wiederum rekursiv berechnet werden mit

$$T_{required(i)} = min_{\forall j \in fanout(i)}\{T_{required}(j) - delay(i, j)\} .$$

Eine Metrik dafür, wie wichtig jede der in dem Graphen betrachteten Verbindungen in Bezug auf die Einhaltung der maximalen Verzögerungszeit des Schaltnetzes ist, bietet der Slack, der in dem hier skizzierten Beispiel als

$$slack(i, j) = T_{required}(j) - T_{arrival}(i) - delay(i, j)$$

angegeben werden kann.

**Definition 2.1.** Der Slack ist der Umfang des Delays, das an einem Element, oder der Verbindung zwischen zwei Elementen, zusätzlich genutzt werden kann, ohne dass die maximale Verzögerungszeit des Schaltnetzes erhöht wird[4]. Für alle Elemente und Verbindungen, die auf dem kritischen Pfad liegen, ist der Slack null.

In Abbildung 2.15 führt der kritische Pfad von dem Eingang $In_A$ zu dem Ausgang *Out*. Für die Verbindung zwischen $In_B$ und $LUT_B$ ist in diesem Beispiel ein Slack von 7,5 ns vorhanden. Diese Verbindung könnte also beispielsweise über eine langsamere Verbindungsleitung laufen, ohne die maximale Verzögerungszeit des Schaltnetzes zu erhöhen. Ein negativer Slack hingegen würde eine Verletzung der maximalen Verzögerungszeit für das Schaltnetz bedeuten. Wenn eine Schaltung nicht mit der gewünschten Verzögerungszeit realisiert werden kann, gibt ein negativer Slack dem Schaltungsdesigner Hinweise darauf, an welchen Stellen Verbindungsleitungen verkürzt bzw. Elemente mit einer geringeren Verzögerungszeit eingesetzt oder andere Maßnahmen zur Senkung der Verzögerungszeit ergriffen werden müssen.

Wenn innerhalb einer Schaltung positiver Slack an allen Pfaden vorhanden ist, die über ein Element dieser Schaltung verlaufen, kann dieses Element aufgrund dieses zusätzlichen Slacks in ein funktional gleiches Element ausgetauscht werden, das z.B. weniger Fläche verbraucht und/oder eine geringere Verlustleistung zur Folge hat. Das *gate sizing*-Problem besteht darin, eine Menge von Gattern in einer Schaltung zu finden, deren Fläche und Verlustleistung durch die Verwendung von kleineren Zellen aus der für den Entwurf einer integrierten Schaltung verwendeten Zellenbibliothek reduziert werden kann, ohne eine vorgegebene maximale Verzögerungszeit zu verletzen [GLPS97]. Auch eine Veränderung der Treshold-Spannung zur Reduzierung der Verlustleistung an diesen Gattern, bzw. eine Kombination aus beiden Techniken [Sri03] wurde als Möglichkeit für die Ausnutzung des Slacks in einer Schaltung erforscht.

Auch bei einem FPGA kann die Veränderung der Threshold-Spannung an Logikblöcken, an denen Slack vorhanden ist, für eine Reduzierung der Verlustleistung eingesetzt werden. Die Stratix III FPGAs der Firma Altera sind in Bereiche eingeteilt, die mit zwei verschiedenen Threshold-Spannungen betrieben werden können [Alt07]. Abbildung 2.16 skizziert die Verwendung dieser Bereiche. Die „Low-Power“-Logik verursacht weniger Verlustleistung als die „High Speed“-Logik, führt dafür aber zu einem erhöhten Delay des Bereiches. Wieviele und welche der Bereiche als „Low-Power“-Logik betrieben werden können, hängt von dem Slack auf den nicht-kritischen Pfaden und der Aufteilung der Logik auf die Bereiche ab. Die in Abbildung 2.17 gezeigte - für diesen Ansatz günstige - Slack-Verteilung ist, nach Angabe der Firma Altera, eine typische Verteilung des Slacks in einem FPGA-Design [Alt07].

[4]Dies gilt nur, wenn das Delay der anderen Elemente und Verbindungen konstant bleibt.

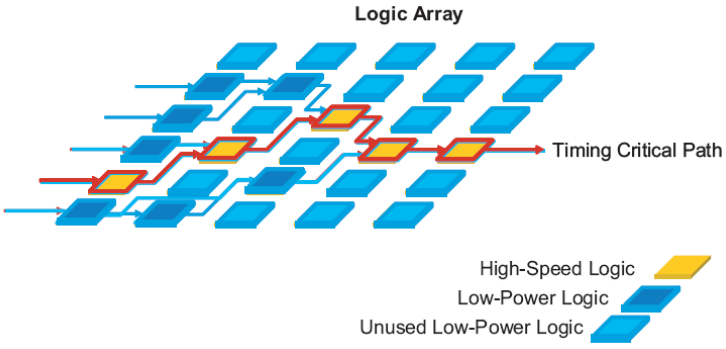



Abbildung 2.16: Stratix III Programmable Power Technology [Alt07]

In dem in der vorliegenden Arbeit vorgestellten Ansatz wird der an dem Hard-Core vorhandene Slack ausgenutzt, um die Defekttoleranz innerhalb der Hard-Core und dadurch auch die Defekttoleranz der gesamten Schaltung zu erhöhen.

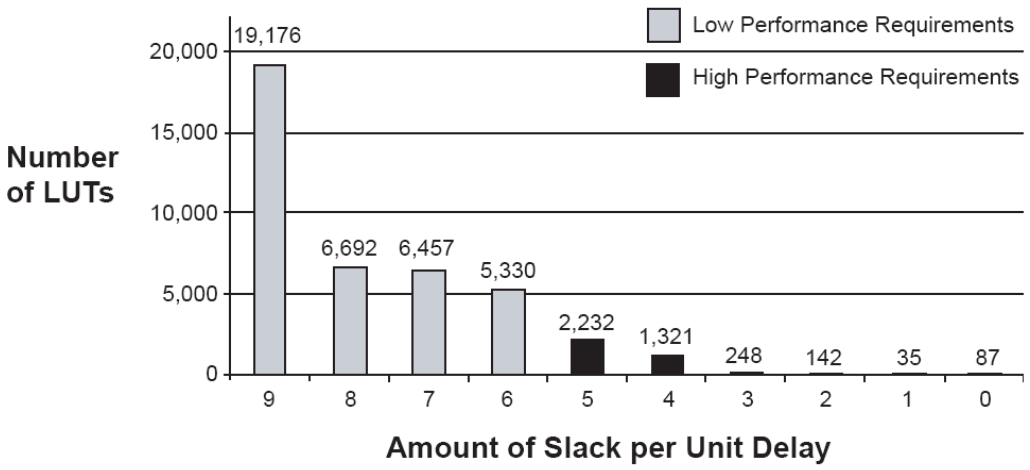



Abbildung 2.17: Ein Slack-Histogramm, das nach Angabe der Firma Altera eine typische Slack-Verteilung in einem FPGA-Design darstellt, und die Einteilung der LAB nach dem benötigten Delay [Alt07]

## 2.3 Die Ausbeute bei der Herstellung von FPGAs

Die Herstellung eines FPGAs erfolgt als integrierte Schaltung (IC). Auf einer dünnen Scheibe aus Halbleitersubstrat, dem sogenannten *Wafer*, werden jeweils mehrere ICs gleichzeitig hergestellt. Unter Reinstraum-Bedingungen wird der Wafer im Rahmen dieses Herstellungsprozesses unter Sauerstoffzufuhr erwärmt, wodurch sich eine Siliziumoxidschicht bildet. Im Anschluss daran wird der Wafer mit Photolack beschichtet und über eine Maske belichtet. Durch das Belichten ist der Photolack an einigen Stellen nicht mehr resistent und kann leicht entfernt werden. Die Stellen des Siliziumoxides, die nicht mehr durch Fotolack geschützt sind, können nun weggeätzt werden. Anschließend wird auch der restliche Fotolack entfernt. Die nicht mehr durch Siliziumoxid geschützten Stellen können jetzt in einem Diffusionsofen oder durch Ionenbeschuss dotiert werden. Der beschriebene Ablauf wiederholt sich bei der IC Herstellung mehrfach. Durch voneinander isolierte Metallisierungsschichten und Kontaktierungen durch die Schichten hindurch (Vias) werden anschließend die gewünschten Verbindungen hergestellt. Nachdem der Wafer, unter Aussparung der für die Kommunikation mit der Außenwelt vorgesehenen Stellen, mit einer isolierenden Schicht überzogen worden ist, werden die einzelnen ICs getestet. Jeder Chip, der diesen Test nicht besteht, wird markiert. Alle nicht markierten ICs werden anschließend in ein Gehäuse montiert und nach weiteren Tests für den Verkauf freigegeben.

Im Herstellungsprozess treten Fehler auf, die teilweise zu nicht verwendbaren ICs führen. Diese Fehler können in drei Kategorien eingeteilt werden [FP92]:

**Gross defects** Dieser Bezeichnung werden Fehler zugeordnet, die durch mechanische Beschädigungen des Wafers, z.B. Kratzer oder großflächig nicht vollständig abgetragenes Material, entstehen. Ein *gross defect* betrifft in der Regel einen großen Bereich des Wafers und führt dadurch oft zu einem Verlust in der Ausbeute an funktionierenden ICs.

**Parametric defects** Alle Fehler, die zu einer Veränderung der elektrischen Parameter führen, werden als *parametric defects* bezeichnet. Ein Beispiel für einen *parametric defect* ist eine ungleichmäßige Verteilung der Dotiersubstanz, die einen unterschiedlichen Widerstand der MOS-Transistoren verursachen kann. *Parametric defects* wirken sich auf die Performance und die Verlässlichkeit der ICs aus und können in extremen Fällen auch zu einem Verlust in der Ausbeute führen.

**Random defects** Alle Fehler, die nicht unter die bisher aufgeführten Kategorien fallen, werden als *random defects* bezeichnet. *Random defects* entstehen oft durch Partikel, die während der Produktion auf dem Wafer liegen und können sich in Form von fehlendem oder zusätzlichem leitenden Material auf den IC auswirken.

Da *gross defects* bei der Herstellung in der Regel auf ein Minimum reduziert werden können und *parametric defects* sich vor allem auf die Verlässlichkeit der ICs und nicht auf die Ausbeute auswirken, werden im Folgenden ausschließlich *random defects* betrachtet. Die Anzahl der auf dem Wafer auftretenden *random defects* kann mit Hilfe der Wahrscheinlichkeitsrechnung vorhergesagt werden.

*Random defects* können in verschiedene Typen unterteilt werden. Fehlendes leitendes Material auf dem Wafer kann eine Leitungsunterbrechung auf dem Chip hervorrufen, zusätzliches leitendes Material einen Kurzschluss. Ob ein *random defect* tatsächlich zu einer defekten Schaltung führt, hängt also von dem Ort, an dem der Fehler auftritt, der Größe des Fehlers und dem Fehlertyp ab. Durch die idealisierte Annahme eines kreisrunden Fehlers, kann aufgrund des Layouts eines vorliegenden ICs für jede Fehlergröße und jeden Fehlertyp die kritische Fläche $A_i^{(c)}(x)$ für einen Fehler vom Typ $i$ mit dem Durchmesser $x$ berechnet werden. Nur wenn der Mittelpunkt des betrachteten Fehlers innerhalb dieser kritischen Fläche liegt, führt der Fehler zu einer fehlerhaften Schaltung, und nur dann wird er in der hier vorliegenden Arbeit als ein Defekt bezeichnet[5]. Abbildung 2.18 skizziert die kritische Fläche, für einen Fehler, durch den Material mit dem Durchmesser $x$ auf dem Chip fehlt [KK98]. Nur der untere der drei abgebildeten Fehler führt zu einer offenen Verbindung, also einem Defekt (*open circuit fault*).

Die Auftrittswahrscheinlichkeit eines Defektes mit einem Durchmesser $x$ kann durch die folgende Wahrscheinlichkeitsdichtefunktion angegeben werden [KK07]:

[5] In der Ermittlung des Yields wird in den meisten Veröffentlichungen zwischen „defect“ und „fault“ unterschieden. Ein im Herstellungsprozess entstandener Fehler wird als „defect“ bezeichnet. Wenn dieser Fehler zu einer fehlerhaften Schaltung auf dem Chip führt, wird er zu einem „fault“. Erst ein „fault“ entspricht somit in der hier vorliegenden Arbeit einem Defekt.

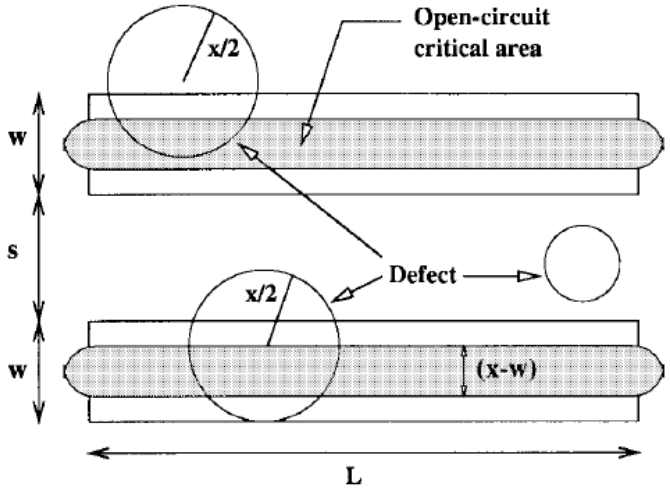



Abbildung 2.18: Die kritische Fläche für einen Fehler mit dem Durchmesser x, wenn das Resultat dieses Fehlers fehlendes leitendes Material auf dem Chip ist

$$f_d(x) = \begin{cases} kx^{-p} & wenn\, x_0 \leq x \\ 0 & sonst \end{cases}$$

wobei $k = \frac{(p-1)x_0^{p-1}x_M^{p-1}}{x_M^{p-1}-x_0^{p-1}}$ eine normalisierende Konstante, $x_0$ die Grenze für die bei der Lithographie verwendeten Auflösung und $x_M$ der maximale Durchmesser eines Defektes ist. Der Parameter $p$ kann empirisch ermittelt werden und liegt typischerweise zwischen 2 und 3,5 [KK07].

Die Wahrscheinlichkeit $\Theta_i(x)$, dass ein Fehler vom Typ $i$ mit dem Durchmesser $x$ einen Defekt verursacht, kann bei Annahme einer Gleichverteilung der Fehler über die Fläche durch

$$\Theta_i(x) = \frac{A_i^{(c)}(x)}{A_{chip}} \tag{2.1}$$

beschrieben werden, wobei $A_{chip}$die Gesamtfläche des ICs bezeichnet.

Die durchschnittliche Wahrscheinlichkeit für die Verursachung eines Defektes durch einen Fehler vom Typ $i$ kann dann als ein durch $f_d(x)$ gewichteter Mittelwert über alle auftretenden Fehlergrößen angegeben werden [KK07]:

$$\Theta_i = \int_{x_0}^{x_M} \Theta_i(x) f_d(x) dx \tag{2.2}$$

Wenn die durchschnittliche Anzahl der Fehler $d_i$ für jeden Fehler-Typ $i$ pro Flächeneinheit bekannt ist, kann jetzt die durchschnittliche Anzahl von Defekten auf dem Chip, im Folgenden bezeichnet mit $\lambda$, errechnet werden:

$$\lambda = \sum_i \Theta_i A_{chip} d_i \tag{2.3}$$

Die Wahrscheinlichkeit, dass $k$ Defekte auf dem Chip vorhanden sind, kann bei einer Einteilung der Chipfläche in statistisch voneinander unabhängigen Flächen gleicher Größe durch die Binomial-verteilung angegeben werden. Mit der diskreten Zufallsvariablen

X="Anzahl der Fehler auf dem IC"

und der durchschnittlichen Anzahl von Defekten auf dem IC, also dem Erwartungswert von X

$E(X) = \lambda$

gilt dann für $n$ statisch voneinander unabhängige Flächen für die Wahrscheinlichkeit von $k$ Defekten auf dem IC:

$$P(X = k) = \left( \begin{array}{c} n \\ k \end{array} \right) \left(\frac{\lambda}{n}\right)^k \left(1 - \frac{\lambda}{n}\right)^{n-k} \tag{2.4}$$

Wenn wir für Formel (2.4) $n \to \infty$ annehmen, kann die Binomialverteilung durch eine Poisson-Verteilung ersetzt werden:

$$lim_{n\to\infty} P(X = k) = lim_{n\to\infty} \left( \begin{array}{c} n \\ k \end{array} \right) \left(\frac{\lambda}{n}\right)^k \left(1 - \frac{\lambda}{n}\right)^{n-k} = \frac{e^{-\lambda}\lambda^k}{k!} \tag{2.5}$$

Für die Wahrscheinlichkeit für einen Chip ohne random defects ($k = 0$) ergibt sich aus Formel (2.5) dann:

$$Y_{RANDOM} = P(X = 0) = e^{-\lambda} \tag{2.6}$$

Die Wahrscheinlichkeit dafür, dass kein Defekt auf dem FPGA auftritt, entspricht dem Verhältnis von funktionsfähigen FPGAs zu gefertigten FPGAs und wird allgemein als Ausbeute (engl. *chip yield*) $Y_{chip}$ bezeichnet. Wenn mit $Y_{GLOBAL}$ die Wahrscheinlichkeit bezeichnet wird, dass kein *gross defect* oder *parametric defect* auf dem FPGA auftritt, gilt

$$Y_{chip} = Y_{RANDOM} * Y_{GLOBAL} = \frac{funktionsfähige\,FPGAs}{gefertigte\,FPGAs} \tag{2.7}$$

Der Hard-Core basierte Ansatz verbessert die Ausbeute durch eine Steigerung der Wahrscheinlichkeit, dass auch ein FPGA mit defekten Logikblöcken genutzt und somit als „funktionsfähiges FPGA“ eingestuft werden kann.

# 3 Test und Diagnose von FPGAs

Die Ermittlung von Defekten ist unbedingt erforderlich, um die Qualität der ausgelieferten FPGAs zu gewährleisten. Eine Diagnose, d.h. die Lokalisierung des Defektes, ist jedoch nur notwendig, wenn der Defekt aktiv umgangen werden soll. Wenn die defekten FPGAs nach der Herstellung weggeworfen werden, ist eine Lokalisierung des Defektes sinnlos, es sei denn, diese Lokalisierung findet aus statistischen Gründen statt. In [Mey03] wird die Unterscheidung zwischen Test und Diagnose einer Schaltung wie folgt beschrieben:

> „Da der Test Teil der Produktion ist und auf jede produzierte Schaltung angewendet werden muss, sind die Anforderungen an einen optimalen Test, die maximal mögliche Fehlerabdeckung bei minimaler Testdauer und minimaler Größe des Testwertsatzes zu erreichen. Die aus dem Test erhaltenen Informationen können lediglich zur Bestimmung der korrekten Funktion der Schaltung dienen. Die zwangsläufig erwünschte hohe Fehlerabdeckung jedes einzelnen Testmusters geht damit zu Lasten der Aussagekraft, welcher der modellierten Fehler bei einer fehlerhaften Testantwort in der Schaltung vorliegt. Die Diagnose soll, im Unterschied zum Test, die Bestimmung des Fehlerortes innerhalb der Schaltung mit dem Ziel ermöglichen, den dem Fehler zugrunde liegenden Defekt aufzuspüren. Die Anforderungen an die zu verwendenden Testmuster sind – im Gegensatz zum Test – eine möglichst eindeutige Zuordnung zwischen einem modellierten Fehler und einem Testmuster, um für ein ausfallendes Testmuster über eine möglichst geringe Anzahl an von ihm abgedeckten Fehlern die Zahl der in Frage kommenden Fehlerorte gering zu halten. Durch die Verwendung mehrerer Testmuster, die eine unterschiedliche Kombination an Fehlern abdecken, kann die Menge der möglicherweise in der Schaltung vorliegenden Fehler über eine Schnittmengenbestimmung der von den ausfallenden Vektoren detektierbaren Fehler verringert werden.“

Wegen der kostspieligen Testzeit wird ein Test bei heutigen Verfahren sofort abgebrochen, wenn ein Fehler detektiert wurde. Die so eingesparte Testzeit ist signifikant, da sehr viele Defekte bereits mit den ersten der angelegten Testmuster detektiert werden können. Eine Diagnose, durch die zusätzliche Kosten für defekte FPGAs entstehen, wird daher generell als nicht wirtschaftlich durchführbar angesehen [Tri08].

Für den in dieser Arbeit vorgestellten Hard-Core basierten Ansatz ist eine Lokalisierung der defekten Logikblöcke, also eine Diagnose der defekt produzierten FPGAs, eine unabdingbare Voraussetzung. Da defekte Logikblöcke umgangen werden, müssen sie zuvor durch eine Diagnose des FPGAs identifiziert werden. Forschungen zu alternativen Diagnose-Verfahren zielen darauf ab, die Kosten für eine Diagnose des FPGAs zu reduzieren. Aufgrund der technologischen Entwicklung wird allgemein vermutet, dass zukünftige FPGA-Generationen deutlich mehr Defekte enthalten, als die zur Zeit produzierten FPGAs. Sinkende Kosten auf der Seite der Diagnose-Verfahren und steigende Kosten aufgrund einer sinkenden Ausbeute an FPGAs könnten so in Zukunft zu wirtschaftlich einsetzbaren Diagnose-Verfahren führen.

Im Gegensatz zu Testverfahren, die mit extern angelegten Testmustern arbeiten, wurden auch Verfahren vorgeschlagen, die auf einem Built-In-Self-Test (BIST) basieren [SMSP98, ASE04, VDS04, SD06]. Diese Ansätze programmieren in der Regel die auf dem FPGA bereits vorhandenen Strukturen für die Durchführung des Tests, d.h. es werden Schaltungsteile des FPGAs für den Test programmiert, die im Anschluss an die Durchführung des BIST wie gewohnt für den laufenden Betrieb verwendet werden können. Es sind somit keine Schaltungen erforderlich, die nur für den Test in das FPGA integriert und nach dem Test nicht mehr verwendet werden. Durch einen BIST können Kosten eingespart werden, die bei einem externen Test für die oft sehr teuren Geräte (meist zusammengefasst unter der Bezeichnung ATE – Automatic Test Equipment) anfallen. Bei einem BIST reduziert sich der Aufwand für das ATE auf die Bereitstellung der FPGA-Konfigurationen, die für die BIST-Logik verwendet werden, die Einleitung der Test-Prozedur und das Auslesen der Test-Resultate und der Diagnose-Daten aus dem FPGA.

Nachdem im nächsten Abschnitt auf die für die Diagnose von defekten FPGAs in akademischen Ansätzen verwendeten Fehlermodelle eingegangen worden ist, erläutert Abschnitt (3.2) zwei grundlegende Verfahren für einen FPGA-BIST, die von Stroud, Abromovici et al. vorgeschlagen [SKCA96, SLA97, AS01, ASH+99, ASE04] und später von anderen Forschergruppen aufgegriffen und verbessert wurden [VDS04, SD06]. Die vorgestellten BIST-basierten Diagnose-Verfahren eignen sich insbesondere auch für die Erstellung einer Defect Map, wie sie für den Hard-Core basierten Ansatz benötigt wird. Im darauffolgenden Abschnitt werden Möglichkeiten für die Bereitstellung einer Defect Map für jedes defekte FPGA diskutiert.

## 3.1 Fehlermodelle

Die Modellierung von möglichen Fehlern in den Verbindungsleitungen von FPGAs kann in der Regel an einem ausführlichen Modell erfolgen, da die Realisierung der

Verbindungen auf den FPGAs in den Handbüchern der FPGA-Hersteller dafür ausreichend beschrieben wird. Folgende Fehlermodelle finden üblicherweise Verwendung bei der Modellierung von Fehlern in den für die Verbindungen benötigten Komponenten [SWHA98]:

- Fehlermodelle für die programmierbaren Schalter (*Programmable connection switch* / *Programmable routing switch*, vgl. Abbildung 2.5 auf Seite 11) zwischen den Verbindungsleitungen
  - Stuck-open / stuck-closed
- Fehlermodelle für die Verbindungsleitungen
  - Stuck-at-0 / stuck-at-1
  - Leitungsunterbrechung
  - Brückenfehlermodell (*bridging fault model*)

Der Aufbau der Logikblöcke eines FPGAs wird von den führenden FPGA-Herstellern nur teilweise auf Gatter-Ebene beschrieben. Von den nicht auf Gatterebene beschriebenen Blöcken (zu denen z.B. die look-up table, Multiplexer, FlipFlops und weitere Bestandteile eines Logikblocks gehören) ist das Verhalten auf funktionaler Ebene jedoch in der Regel genau dokumentiert. Da jedes Fehlermodell eine Beschreibung der betrachteten Schaltung als Grundlage für die Beschreibung möglicher Defektauswirkungen verwendet, spiegelt sich das nicht vorhandene Gattermodell in den in der Literatur verwendeten Fehlermodellen für den Test und die Diagnose von FPGAs wider. Arbeiten, die speziell die Diagnose von Logikblöcken zum Ziel haben, verwenden daher oft ein funktionales Fehlermodell, das den Einfluss der Fehler in den Logikblöcken auf die Schaltung realisierungsunabhängig beschreibt. Die in Abbildung 2.5 als *connection block* zusammengefassten programmierbaren Schalter werden in vielen der angegebenen Ansätzen als zu dem Logikblock dazugehörig angesehen. Defekte in diesen Schaltern werden in diesem Fall als funktionale Fehler in den betroffenen Logikblöcken interpretiert.

Eine weitere Möglichkeit mit der nicht bekannten internen Struktur der Logikblöcke umzugehen, ist ein pseudo-vollständiger BIST (*pseudo exhaustive test*). Bei diesem Test wird jeder Ausgang eines Logikblocks getestet, indem alle möglichen $2^n$ Kombinationen an die $n$ Eingänge des Logikblocks angelegt werden, von denen der getestete Ausgang abhängt. Auch wenn durch das Testen mit allen möglichen Eingangskombinationen in der Regel mehr Kombinationen getestet werden als bei der Ermittlung einer optimalen Testmusterabdeckung, bietet diese Vorgehensweise den Vorteil, dass die erforderlichen Testmuster (=Eingangskombinationen) durch einfache Schaltungen auf dem FPGA generiert werden können und keine Modellierung des Logikblocks für die Ermittlung der Testmuster erforderlich ist. Durch den Vergleich der Ausgänge des getesteten Logikblocks mit den Ausgängen eines fehlerfreien Logikblocks können, bei

einem Durchlauf derselben Testmuster an den Eingängen der Logikblöcke, defekte Logikblöcke erkannt werden.

## 3.2 Diagnose über FPGA-BIST

Stroud et. al schlagen in [SKCA96] ein BIST-Verfahren für den Test der Logikblöcke in FPGAs vor. Die schlechte Implementierbarkeit dieses Verfahrens aufgrund eines massiven Bedarfes an globalen Verbindungsleitungen auf dem FPGA wurde in [SLKA96] verbessert. Da die Test-Zeit durch das in [SLKA96] dargestellte Verfahren jedoch 33% mehr Zeit für die Durchführung des Tests benötigt, wird in [SLA97] eine Kombination der in [SKCA96] und [SLKA96] vorgeschlagenen Verfahren vorgestellt, die schließlich in [HGWS99] und zuletzt in [AS01] nochmals verbessert worden ist. Durch dieses Verfahren ist neben dem Test der Logikblöcke auch eine Lokalisierung von defekten Logikblöcken möglich. Im Folgenden soll der Ablauf dieser Diagnose kurz erläutert werden.

3.1 zeigt den Aufbau der verwendeten BIST-Architektur. Aus den Logikblöcken in der ersten Zeile des FPGAs werden Testmustergeneratoren (*test pattern generators* - TPGs) konfiguriert, die über globale Verbindungsleitungen Testmuster an die Eingänge der zu testenden Logikblöcke (*blocks under test* – BUTs) anlegen. Ab der dritten FPGA-Zeile werden in jeder zweiten Zeile Logikblöcke für den Vergleich der Antworten der BUTs auf die angelegten Testmuster konfiguriert (*output response analyzers* – ORAs). Durch Rekonfigurationen des FPGAs durch das ATE werden die BUTs in alle möglichen Betriebs-Modi gebracht und die Testmustergeneratoren jeweils so programmiert, dass alle für den jeweiligen Modus erforderlichen Testmuster – im folgenden als Testsequenz bezeichnet - an die BUTs angelegt werden. Für jeden Logikblock wird auf diese Weise ein vollständiger Test durchgeführt.

Nach jeder Rekonfiguration werden die folgenden Schritte ausgeführt:

1. Start der Test-Sequenz: Das ATE initialisiert TPGs, BUTs und ORAs und startet den Test (über BIST Start/Reset, siehe Abbildung 3.1)
2. Generierung der Testmuster durch die TPGs
3. Vergleich der Testantworten durch die ORAs.
4. Auslesen der Test-Ergebnisse aus dem FPGA (über Pass/Fail in Abbildung 3.1) durch das ATE.

Schritt 2 und 3 werden auf dem FPGA ohne Einwirkung von außen parallel für alle BUTs durchgeführt.

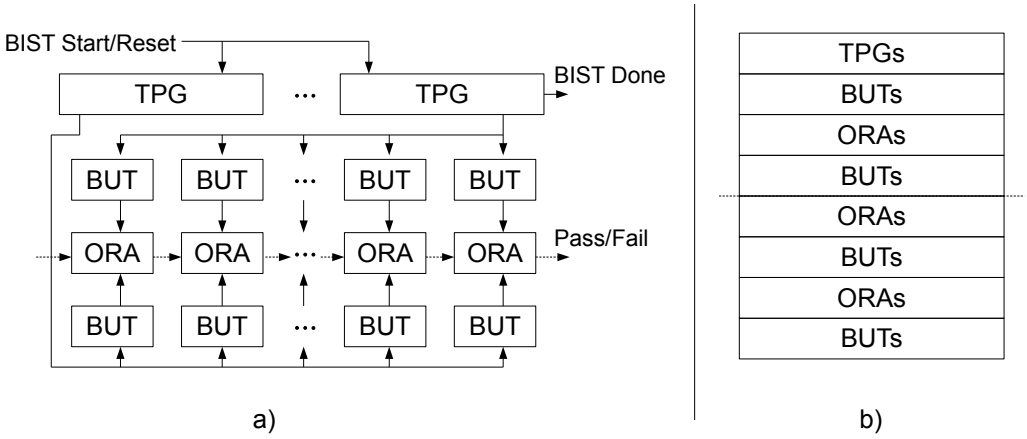



Abbildung 3.1: (a) Die BIST-Architektur aus [AS01] (b) Eine mögliche Zuordnung der Zeilen des zu testenden FPGAs

Die Bewertung der Testantworten erfolgt über einen Vergleich der BUTs, die jeweils oberhalb und unterhalb der ORAs liegen. Die ORAs können zum Auslesen der Test-Ergebnisse zu einem Shift-Register zusammengefasst werden.

Nachdem alle Betriebs-Modi der BUTs getestet wurden, werden die Zeilen des FPGAs entsprechend Abbildung 3.2 (b) umkonfiguriert, so dass in den darauffolgenden Tests die BUTs zu TPGs und ORAs werden und umgekehrt. Der Test ist beendet, wenn alle Logikblöcke des FPGAs als BUT in allen möglichen Betriebs-Modi getestet wurden. Alle Konfigurationen mit der Aufteilung entsprechend Abbildung 3.2 (a) und alle Konfigurationen mit der Aufteilung entsprechend Abbildung 3.2 (b) werden in [AS01] zusammengefasst jeweils als eine *test session* bezeichnet. Orientiert an der Richtung, in der die Testmuster zu den BUTs geleitet werden, wird *test session* (a) als NS und *test session* (b) als SN bezeichnet.

Eine Diagnose, also die Lokalisierung von defekten Logikblöcken, besteht aus der Abbildung einer nicht erwarteten Testantwort eines BUTs auf Fehler, die diese Testantwort erklären können. Durch die Anzahl der Shift-Operationen, die erforderlich sind bis das Ergebnis des fehlgeschlagenen Vergleiches am Pass/Fail-Ausgang anliegt, kann die Spalte festgestellt werden, in der ein Defekt aufgetreten ist. Auch die Bestimmung der Zeile ist möglich, da aus den Meldungen der ORAs Rückschlüsse auf die verursachende Zeile möglich sind. Abbildung 3.3 zeigt die möglichen Rückmeldungen für einen Fehler in der jeweils in der Tabelle in der ersten Spalte angegebenen Zeile für ein FPGA mit 8x8 Logikblöcken. Ein fehlerhafter Logikblock innerhalb eines ORAs oder eines TPGs muss nicht notwendigerweise dessen Funktion beeinträchtigen. Wenn wir jeweils die ORAs und TPGs als fehlerfrei annehmen (siehe Spalte „only BUT failures“), ist die Zuordnung des defekten Logikblocks zu einer Zeile einfach, da entweder in den inneren

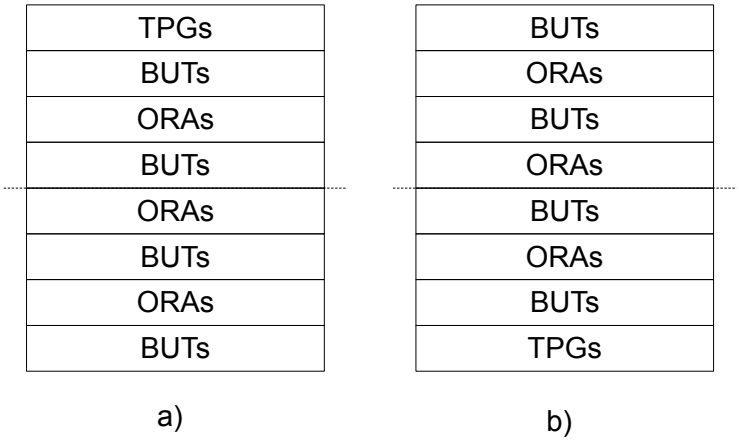



Abbildung 3.2: Durch zwei verschiedene Zuordnungen der Zeilen wird jede Zeile des FPGA als BUT getestet [AS01]

Zeilen jeweils der ORA über und unter dem defekten Logikblock einen Defekt signalisiert, oder, in den äußeren Zeilen, durch die „Fail“-Meldung von nur einem ORA die Zeile bestimmt werden kann.

Erläuterung am Beispiel aus Abbildung 3.3: Ein defekter BUT in Zeile vier verursacht „Fail“-Meldungen von O3 und O5 in *test session* NS. Ein defekter BUT in Zeile eins verursacht eine „Fail“-Meldung in O2 in *test session* SN.

Auch dann, wenn man davon ausgeht, dass ORAs und TPGs von den defekten Logikblöcken betroffen sind, kann aus den Rückmeldungen der ORAs eindeutig auf die fehlerhafte Zeile geschlossen werden (siehe Spalte „with potiential ORA/TPG failures“).

Erläuterung am Beispiel aus Abbildung 3.3: Wenn ein ORA fehlerhaft arbeitet, kann er einen Fehler melden, obwohl die verglichenen Ausgänge der BUTs übereinstimmen. Wenn in der *test session* NS in der zweiten Zeile ein Fehler vorliegt, kann dieser Fehler sich auch auf den in der *test session* SN an gleicher Stelle platzierten O2 auswirken, er wurde daher als potentieller Defekt in Klammern dort aufgeführt. Da wie in Abbildung 3.1 ersichtlich zwei TPGs in jeder test session vorhanden sind, würde sich ein Fehler in einem der TPGs so auswirken, dass alle ORAs eine „Fail“-Meldung liefern. Auch diese Fälle wurden für beide test sessions in Klammern in die Tabelle eingetragen. Auch mit der Berücksichtigung von Fehlern in den TPGs und ORAs ist die „Signatur“ jeder Zeile der Tabelle eindeutig, d.h. es kann eindeutig auf die Zeile geschlossen werden, in der sich der defekte BUT befindet.

Auch die Diagnose von mehreren defekten Logikblöcken (*multiple faulty PLBs*) ist mit dem Ansatz möglich und wird ausführlich in [AS01] erläutert. Es gibt allerdings einige wenige Kombinationen, in denen die defekten Logikblöcke nicht lokalisiert werden

| Faulty Row | Function Session NS | Function Session SN | only BUT failures | | | | | | with potential ORA/TPG failures | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Session NS | | | Session SN | | | Session NS | | | Session SN | | |
| | | | $O_3$ | $O_5$ | $O_7$ | $O_2$ | $O_4$ | $O_6$ | $O_3$ | $O_5$ | $O_7$ | $O_2$ | $O_4$ | $O_6$ |
| 1 | TPG | BUT | | | | X | | | (X | X | X) | X | | |
| 2 | BUT | ORA | X | | | | | | X | | | (X) | | |
| 3 | ORA | BUT | | | | X | X | | (X) | | | X | X | |
| 4 | BUT | ORA | X | X | | | | | X | X | | | (X) | |
| 5 | ORA | BUT | | | | | X | X | | (X) | | | X | X |
| 6 | BUT | ORA | | X | X | | | | | X | X | | | (X) |
| 7 | ORA | BUT | | | | | | X | | | (X) | | | X |
| 8 | BUT | TPG | | | X | | | | | | X | (X | X | X) |

Abbildung 3.3: Die Rückmeldungen der ORAs bei einem fehlerhaften Logikblock [AS01]

können. In [AS01] heißt es dazu in der Zusammenfassung am Ende der Veröffentlichung:

> „The multiple faulty PLBs that cannot be diagnosed appear to be very restrictive situations unlikely to occur in practice.“

Eine Diagnose der Defekte in Routing-Ressourcen durch ein ähnliches Verfahren wurde in [SNLA02] beschrieben. Dieses Verfahren soll an dieser Stelle nicht betrachtet werden, da für den Hard-Core basierten Ansatz in der vorliegenden Form keine Diagnose der Routing-Ressourcen benötigt wird.

Neben der Diagnose durch ein Offline-BIST, d.h. der Diagnose eines FPGAs, bevor eine Schaltung auf dem FPGA implementiert wird, wurden auch Verfahren für ein Online-BIST vorgeschlagen. Ein Online-BIST geht davon aus, dass sich bereits eine Schaltung auf dem FPGA befindet und führt einen Test/eine Diagnose durch, ohne die Funktion

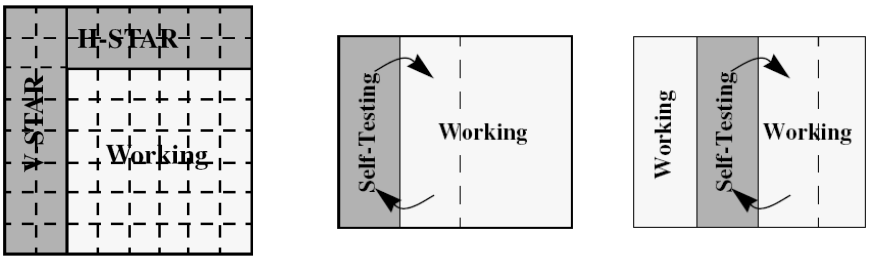



Abbildung 3.4: Zwei „self-testing areas“ und deren Arbeitsweise [ASH+99]

der Schaltung zu beeinträchtigen. Dieses Vorgehen kann sinnvoll sein, wenn Defekte erst zur Laufzeit auf dem FPGA entstehen. Dies können z.B. Defekte auf dem FPGA sein, die bereits während der Herstellung des FPGAs entstanden sind und direkt nach der Herstellung noch kein Fehlverhalten des FPGAs verursacht haben, aber später zur Laufzeit durch Alterungseffekte auf dem FPGA als Defekt in Erscheinung treten. Vor allem in sicherheitskritischen Systemen ist daher ein BIST sinnvoll, um diese Defekte zur Laufzeit zu entdecken [SS99]. Das erste Verfahren für ein Online-BIST für FPGAs wurde 1998 von Shnidman et al. [SMSP98] vorgestellt. Dieses Verfahren erlaubt die Diagnose von defekten Logikblöcken für Bus-basierte FPGAs, also FPGAs, in denen Verbindungsleitungen vorhanden sind, welche die gesamte Höhe und die gesamte Breite des FPGAs umfassen. Das erste Online-BIST Verfahren, das auch für FPGAs mit segmentierten Verbindungsleitungen eingesetzt werden kann und auch eine Diagnose der Verbindungsleitungen ermöglicht, wurde 1999 von Stroud, Abromovici et al. [ASH+99, ASSE00] vorgestellt. Dieses Verfahren basiert auf der Idee von „self testing areas“ (STARs), die über das FPGA bewegt werden, während die übrigen auf dem FPGA vorhandenen Logikblöcke weiterhin die Schaltung auf dem FPGA implementieren. Abbildung 3.4 skizziert eine vertikale und eine horizontale STAR, die horizontal, bzw. vertikal über das FPGA bewegt werden. Innerhalb der STARs findet ein Test statt, der vergleichbar mit dem bereits für den Offline-BIST vorgestellten Test verläuft.

Die in der STAR enthaltenen Logikblöcke werden entsprechend Abbildung 3.5 in Bereiche unterteilt, die jeweils einen TPG (für die Implementierung eines TPG sind 3 Logikblöcke erforderlich), zwei BUT und einen ORA enthalten. Durch die abgebildeten sechs Konfigurationen für diesen Bereich ist gewährleistet, dass jeder der sechs Logikblöcke mindestens einmal die Rolle eines BUTs einnimmt. Nachdem in jeder der Konfigurationen alle Testmuster durch den TPG erzeugt wurden, wird der STAR einen Logikblock weiter geschoben. Wenn durch den ORA ein Fehler gemeldet wird, können entsprechend Abbildung 3.6 neue Aufteilungen für die Test-Konfigurationen gewählt werden, die eine eindeutige Identifizierung des defekten Logikblocks ermöglichen. Eine Weiterentwicklung dieses Ansatzes wurde von Verma et al. [VDS04] veröffentlicht.

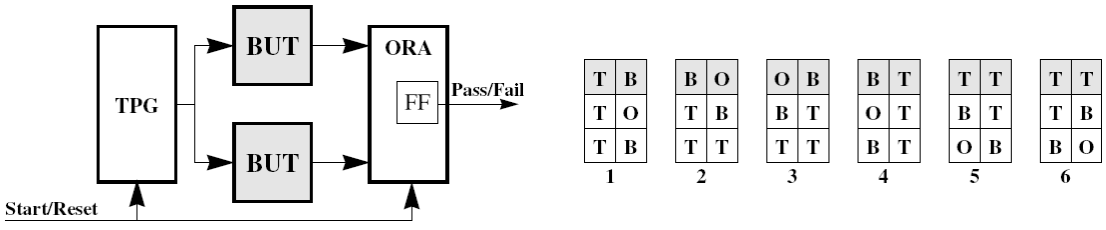



Abbildung 3.5: Rotierende Konfigurationen für die Logikblöcke in einer STAR [ASH+99]

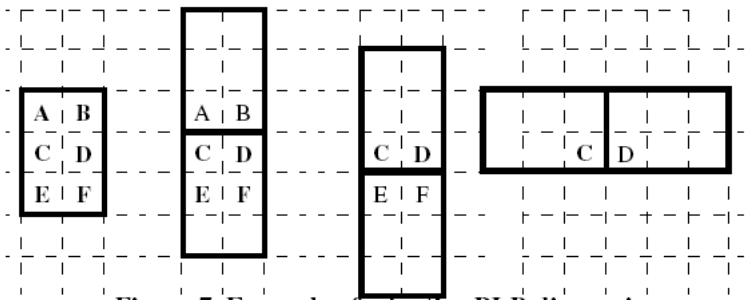



Abbildung 3.6: Neue Aufteilungen der Testbereiche zur Diagnose durch STARs [ASH+99]

Der Bereich, der den Tester enthält, wird hier als ROTE (ROving TEster) bezeichnet und es ist ein ROTE ausreichend, um einen defekten Logikblock zu lokalisieren. Dieses Verfahren für ein Online-BIST kann auch für ein Offline-BIST eingesetzt werden, indem der ROTE einmalig über das gesamte FPGA bewegt wird [VDS04].

## 3.3 Erstellung und Bereitstellung einer Defect Map

Viele der im Abschnitt „Vergleichbare Arbeiten“ besprochenen Arbeiten basieren auf der Diagnose der defekten Komponenten in einem defekten FPGA. Mishra und Goldstein skizzieren die Idee einer Defect Map, die auch eine Voraussetzung für das hier vorgestellte Hard-Core basierte Verfahren ist [MG03]:

> „The key idea behind defect tolerance in FPGAs is that reconfigurability allows one to find the defects and then to avoid them. We expect that reconfigurable fabrics made from next generation technologies will go through a post-fabrication testing phase at which point they will be configured for self-diagnosis. Unlike the dedicated BIST structures often incorporated in current digital designs, the test circuits placed on the fabric during this self-diagnosis phase will utilize ressources that will be available later for normal fabric operation, and so testing will not incur either an area or a delay penalty. The result of the test phase will be a defect map which contains locations of all the defects. This map can be used by place-and-route tools to layout circuits on the fabric which avoid the defects. The ability to tolerate defects in the final product eases the requirements on the manufacturing process. In some sense, this introduces a new manufacturing paradigm: one which trades-off post-fabrication programming for cost and complexity at manufacturing time.“

Die Bereitstellung einer Defect Map für ein defektes FPGA ist auf verschiedenen Wegen denkbar:

- *Jedes defekte FPGA wird mit einer Kennzeichnung versehen*, die eine eindeutige Identifizierung des FPGAs erlaubt. Die *Defect Maps* werden dann entweder in einer zentralen Datenbank gespeichert oder zusammen mit den FPGAs an den Kunden ausgeliefert. Da über die *Defect Maps* Rückschlüsse auf die Ausbeute bei der Herstellung der FPGAs möglich sind, müsste eine zentrale Datenbank so geschützt werden, dass nur für die bei dem Kunden vorhandenen FPGAs ein Zugriff möglich ist. Die Speicherung der Daten für alle defekt produzierten FPGAs in einer internen Datenbank bei dem Hersteller und die Auslieferung der Daten zusammen mit den FPGAs vermeidet die Sicherheitsprobleme einer externen Datenbank.
- *In jedes FPGA wird ein nichtflüchtiger Speicher integriert*, in dem die *Defect Map* des FPGAs gespeichert ist.

Bevor ein Verfahren für die Bereitstellung einer Defect Map eingesetzt werden kann, sind weitere Forschungen notwendig, um das kostengünstigste Verfahren zu ermitteln. Die oben aufgeführten Ansätze sollen lediglich erste Ideen skizzieren.

## 3.4 Zusammenfassung

Mit den bisher für den Test von FPGAs nach der Herstellung eingesetzten Verfahren ist eine Diagnose der defekten Logikblöcke sehr zeitaufwändig und daher nicht realisierbar. In diesem Kapitel wurden Forschungsansätze zu BIST-Verfahren für FPGAs vorgestellt, die eine Diagnose defekter Logikblöcken ermöglichen. Zum Abschluss des Kapitels wurden erste Ideen für die Bereitstellung einer defect map für defekte FPGAs skizziert.

# 4 Vergleichbare Arbeiten

Abbildung 4.1 zeigt eine mögliche Klassifizierung der im Folgenden dargestellten Ansätze, die an die Gliederung des Papers „Yield Enhancing Defect Tolerance Techniques for FPGAs“ von Asbjoern Djupdal und Pauline C. Haddow aus dem Jahr 2006 [DH06] angelehnt ist. Weitere Informationen zu den in [DH06] beschriebenen Ansätzen sowie Referenzen und Kurzbeschreibungen zu weiteren Arbeiten, die für die Tolerierung von Defekten auf FPGAs verwendet werden können, findet man in einer Übersicht zu fehlertoleranten Methoden für FPGAs aus dem Jahr 2006 [CEB06] und in einer weiteren Übersicht zu diesem Thema aus dem Jahr 2003 [DI03].

**Ein Vergleich des Hard-Core basierten Ansatzes mit den in diesem Kapitel vorgestellten Arbeiten folgt, nachdem der Hard-Core basierte Ansatz in Kapitel 5 ausführlich vorgestellt wurde, in Kapitel 6.**

Neben dem Einsatz für die Verbesserung der Ausbeute eignen sich einige der im Folgenden vorgestellten Arbeiten auch für die Tolerierung von Defekten, die erst zur Laufzeit, also nachdem die Herstellung des FPGAs abgeschlossen ist, auf dem FPGA auftreten. Die Tolerierung von Defekten zur Laufzeit zielt darauf ab, die Verlässlichkeit eines FPGAs zu verbessern. Ein Laufzeit-Defekt kann z.B. durch Alterungseffekte auf dem FPGA entstehen. Eine weitere Ursache für Laufzeit-Defekte sind sogenannte *Single Event Upsets* (SEUs). Ein SEU ist eine durch Strahlung hervorgerufene Änderung des Zustands einer Speicherzelle auf dem FPGA. Da neben den Änderungen der Speicherinhalte im *Logic Layer*, z.B. der Änderung des Inhaltes eines Flipflops, wie in Abbildung 4.2 (übernommen aus [AT05]) dargestellt, auch SEUs im Konfigurationsspeicher (*Configuration Memory Layer*, vgl. Abbildung 2.9 auf Seite 14) eines FPGAs auftreten können, ist durch einen SEU auch eine Veränderung der durch das FPGA implementierten Schaltung möglich [GCZJ03].

Bei der Klassifizierung in Abbildung 4.1 wird zwischen passiven Methoden, die keine Diagnose des FPGAs erfordern, und aktiven Methoden unterschieden. Bei aktiven Methoden erfolgt eine Anpassung der auf dem FPGA abgebildeten Schaltung an Defekte, die zuvor durch eine Diagnose ermittelt wurden.

Nachdem im folgenden Abschnitt einige passive Verfahren für die Tolerierung von Defekten betrachtet worden sind, werden in Abschnitt 4.2 aktive Verfahren vorgestellt, die mit zusätzlich in die FPGA-Architektur eingeführten redundanten Komponenten arbeiten. Die FPGA-Konfiguration, die für die Programmierung der so veränderten FPGAs

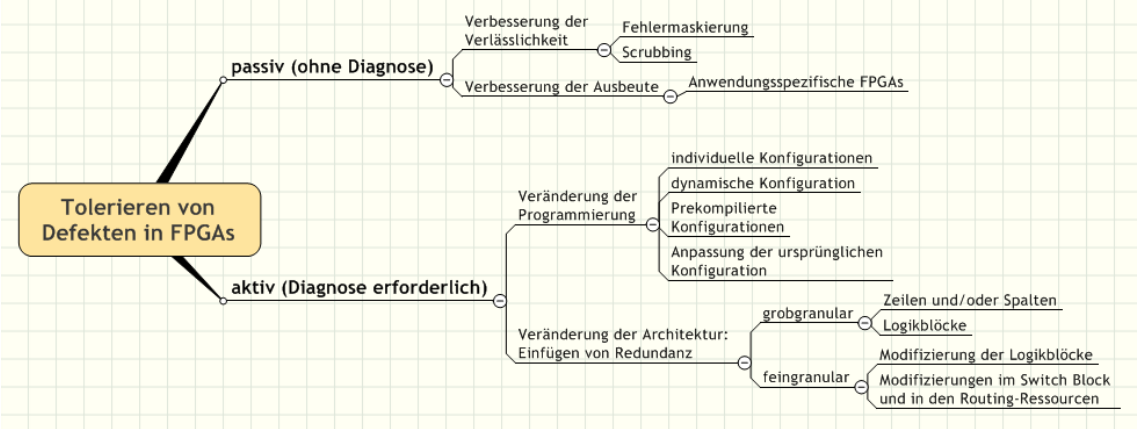



Abbildung 4.1: Klassifizierung der Arbeiten mit dem Ziel der Tolerierung von Defekten in FPGAs

benötigt wird, entspricht hier der FPGA-Konfiguration für FPGAs ohne Defekte. Die danach in Abschnitt 4.3 vorgestellten aktiven Verfahren arbeiten im Gegensatz dazu mit einer an das defekte FPGA angepassten FPGA-Konfiguration.

Bei dem Hard-Core basierten Verfahren handelt es sich um ein aktives Verfahren, d.h. es ist eine vorherige Diagnose der defekten FPGAs erforderlich. Das Verfahren eignet sich insbesondere für SRAM-basierte *island-style* FPGAs, wie sie im zweiten Kapitel dieser Arbeit vorgestellt wurden. Nachdem der Hard-Core basierte Ansatz in Kapitel 5 ausführlich vorgestellt wurde, ist der Darstellung der Vorteile des Hard-Core basierten Verfahrens gegenüber den in diesem Kapitel vorgestellten Verfahren ein eigener Abschnitt in Kapitel 6 gewidmet.

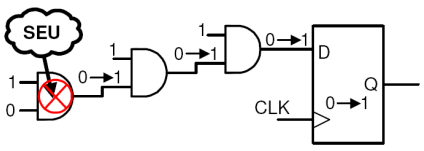



Abbildung 4.2: Ein SEU im Logic Layer kann, wie hier gezeigt, einen falschen Wert in einem Flipflop erzeugen [AT05]

## 4.1 Passive Verfahren

Verfahren für den Ausgleich von Defekten zur Laufzeit beruhen oft auf einem unter der Bezeichnung *Triple Module Redundancy* (TMR) bekannten Ansatz. Die Schaltung wird verdreifacht und das Ergebnis über einen Mehrheitsentscheid bestimmt. Da für die Realisierung einer Schaltung als TMR ungefähr die dreifache Fläche benötigt wird und sich dadurch auch die Verlustleistung entsprechend erhöht, wird das Verfahren nur für Schaltungen auf FPGAs in sicherheitskritischen oder schwer zugänglichen Systemen, z.B. im Weltraum [Car06], eingesetzt. Oft wird TMR in Verbindung mit einem periodischen Überschreiben der Konfiguration des FPGAs mit der Originalkonfiguration verwendet. Durch diese periodische „Erneuerung“ der Konfigurationsdaten, die als *Scrubbing* bezeichnet wird, werden durch SEUs fehlerhaft gewordenen Konfigurationen korrigiert.

Da die Ausbeute an FPGAs mit zunehmender Fläche sinkt (vgl. Abschnitt 2.3), ist der Einsatz von TMR für eine Erhöhung der Ausbeute nicht einsetzbar. Auch das Scrubbing ist für eine Steigerung der Ausbeute nicht einsetzbar, da dauerhafte Defekte auf einem FPGA mit einer „Erneuerung“ der Konfigurationsdaten nicht behoben werden.

Durch die Firma Xilinx wird ein Verfahren für die Nutzung von defekten FPGA eingesetzt, in dem teildefekte FPGAs ohne vorherige Diagnose eingesetzt werden[1]. Dazu wird ein erneuter Test von leicht defekten FPGAs durchgeführt, der nur die Teile des FPGAs testet, die von der durch den Kunden gewünschten FPGA-Konfiguration auf dem FPGA tatsächlich genutzt werden [Tri08]. Wenn die teildefekten FPGAs diesen Test erfolgreich durchlaufen, werden sie dem Kunden für eine Serienproduktion verkauft. Ein großer Vorteil dieses Verfahrens ist die Einsparung einer zeitaufwendigen Diagnose, also der genauen Lokalisierung der Defekte. Der Nachteil dieser Vorgehensweise ist der Verlust der Rekonfigurierbarkeit der FPGAs. Die Funktion der FPGAs ist nur sichergestellt für FPGA-Konfigurationen, die für den Test verwendet wurden.

## 4.2 Einfügen von redundanten Komponenten in die FPGA-Architektur

Ein Ausgleich einer defekten Komponente in einem defekten FPGA kann durch ein logisches Auswechseln der defekten Komponente mit einer zusätzlich auf dem Chip vorhandenen identisch aufgebauten Komponente erreicht werden. Da redundant vorhandene Komponenten zusätzliche Fläche auf dem Chip belegen und die Ausbeute bei der Herstellung eines Chips mit steigender Fläche abnimmt, lohnt sich dieses Vorgehen nur dann, wenn die zusätzlich auf den Chip gebrachte Komponente viele andere

[1] Easy Path FPGA Series [eas08]

der Komponenten des Chips im Defektfall ersetzen kann. Da alle Logikblöcke in einem FPGA identisch aufgebaut sind, ist die Idee naheliegend, Reserve-Logikblöcke in die FPGA-Architektur zu integrieren. Wenn bei der Herstellung eines FPGAs defekte Logikblöcke diagnostiziert werden, kann das FPGA in einem späteren Produktionsschritt mit den Reserve-Logikblöcken, z.B. durch ein gezieltes Durchbrennen von ebenfalls in die Architektur eingebrachten Sicherungen (*fuses*), so „verdrahtet“ werden, dass es wie gewohnt verwendet werden kann. Die wichtigsten der in Abbildung 4.1 als grobgranular eingeordneten Ansätze, die mit dieser Idee arbeiten, sollen im Folgenden vorgestellt werden.

Ergänzend zu grobgranularen Redundanz-Techniken gibt es Ansätze, die durch Redundanz *innerhalb* der Logikblöcke, z.B. durch die Verwendung von LUTs mit fehlerkorrigierenden Codes [KL04] oder durch Redundanz innerhalb anderer Komponenten eines FPGAs Defekte tolerieren können. Diese Ansätze werden in Abbildung 4.1 als feingranular klassifiziert. Doumar und Ito [DI00b] fügen in jeden *switch block* eine zusätzliche Verbindungsleitung ein, wodurch eine alternative Verbindung zwischen den an diesem *switch block* angeschlossenen Verbindungsleitungen ermöglicht wird. Mit dieser alternativen Verbindung kann ein Defekt in einem switch block umgangen werden. Yu und Lemieux [RWL05] fügen der FPGA-Architektur zusätzliche Reserve-Verbindungsleitungen und zusätzliche Multiplexer hinzu und schaffen dadurch die in Abbildung 4.3 skizzierte Möglichkeit, eine defekte Verbindungsleitung zu umgehen.

### 4.2.1 Redundante Zeilen und/oder Spalten

Eine Möglichkeit, Reserve-Logikblöcke in die FPGA-Architektur einzufügen, ist die Verwendung von redundanten Spalten (und / oder Zeilen) mit Reserve-Logikblöcken. Wenn eine defekte Spalte diagnostiziert wird, kann in einem späteren Produktionsschritt anstelle der defekten Spalte die Spalte mit den Reserve-Logikblöcken verwendet werden. Dieses Verfahren wurde von Hatori et al. [HSN+93] vorgeschlagen und später von Howard et al. [HTA94] und Durand und Piguet [DP94] weiterentwickelt. Abbildung

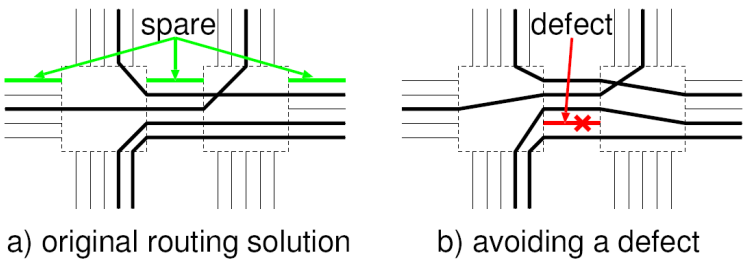



Abbildung 4.3: Die Umgehung einer defekten Verbindungsleitung [RWL05]

4.4 skizziert das Verfahren, so wie es von Hatori et al. [HSN+93] erstmals vorgeschlagen wurde. Im Rahmen dieses Verfahrens werden zusätzliche redundante Zeilen in die FPGA-Architektur eingefügt. Eine ebenfalls zusätzlich in die FPGA-Architektur eingebrachte Auswahllogik (*Selector*) sorgt für eine neue Zuordnung der ursprünglichen von der FPGA-Konfiguration angesprochenen Zeile (*Row Decoder*), wenn bei der Diagnose des FPGAs eine Zeile als Defekt ermittelt wurde. Abbildung 4.4 (a) zeigt das FPGA mit einer Reserve-Zeile und Abbildung 4.4 (b) die neue Zuordnung der Zeilen als Reaktion auf eine defekte Zeile. Während die horizontalen Verbindungsleitungen unverändert bleiben, werden die vertikalen Verbindungsleitungen in der neuen FPGA-Architektur um eine Zeile verlängert.

Howard et al. [HTA94] haben eine Variation dieses Verfahrens vorgeschlagen, in dem die Logikblöcke zu größeren Blöcken zusammengefasst werden. Die in die FPGA-Architektur eingefügten Reserve-Zeilen (oder Reserve-Spalten) bestehen hier aus diesen größeren Blöcken und nicht aus einzelnen Logikblöcken. Durch einen eigenen Konfigurationsspeicher für jeden der Blöcke kann die Auswirkung von Defekten, die sich auf den gesamten Konfigurationsspeicher auswirken, auf den Konfigurationsspeicher eines Blockes begrenzt werden. Um die zusätzlichen Verzögerungszeiten für Verbindungen zwischen den Blöcken bei der Überquerung eines defekten Blocks möglichst gering zu halten, werden in die Architektur zusätzliche lange Verbindungsleitungen eingeführt, die eine Direktverbindung zwischen den Blöcken ermöglichen.

Durand und Piguet [DP94] benutzen eine FPGA-Architektur, die sich an einem binären Entscheidungsdiagramm orientiert. Da in dieser Architektur ausschließlich direkt benachbarte Zellen verbunden sind, muss keine Verlängerung der Verbindungsleitungen erfolgen. Die Rekonfiguration im Fall eines Defektes erfolgt in diesem Ansatz durch eine Ersetzung der defekten Spalte mit einer Reserve-Spalte. Vergleichbar mit dem Ansatz von Hatori et al. (vgl. Abbildung 4.4) werden die Konfigurationsdaten aller Logikblöcke, die auf der Spalte und rechts von der Spalte mit dem Defekt liegen, um eine Spalte nach rechts verschoben. Damit nicht die gesamte Spalte verschoben werden muss, schlagen die Autoren eine horizontale Aufteilung des FPGAs in mehrere Bereiche vor, so dass die Verschiebung der Spalten innerhalb eines Bereiches bei einem Defekt in dem jeweiligen Bereich ausreichend ist und die Verzögerungszeit in den anderen Bereichen nicht durch die „Überbrückung“ der defekten Spalte erhöht wird.

Ein Nachteil der in diesem Abschnitt vorgestellten Verfahren liegt in der hohen Anzahl von Logikblöcken, die im Fall eines Defektes „geopfert“ werden, da jeweils komplette Zeilen oder Spalten (bzw. komplette Spalten eines Bereiches im Verfahren von Durand und Piguet [DP94]) ausgetauscht werden. Ein weiterer Nachteil ist der zusätzliche Aufwand für die Umgehung der defekten Zeilen oder Spalten, in Form von zusätzlichen programmierbaren Schaltern und verlängerten Verbindungsleitungen, sofern nicht wie in einem Bus-basierten FPGA Verbindungsleitungen verwendet werden, die die gesamte Höhe oder Breite des FPGAs umfassen.

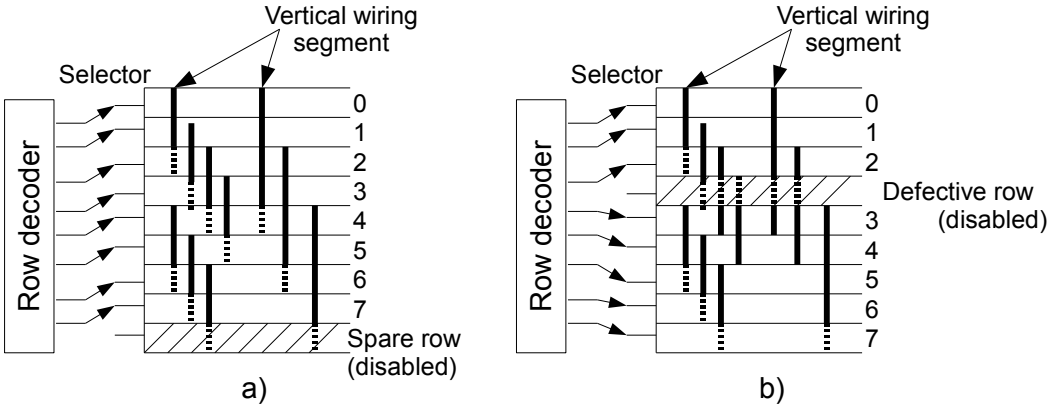



Abbildung 4.4: Anpassung einer FPGA-Architektur nach [HSN+93] an einen Defekt; a) ohne defekte Zeile; b) mit defekter Zeile

Da die von Altera verwendete hierarchische FPGA-Architektur Bus-basiert ist, sind hier die Voraussetzungen für die Nutzung von redundanten Zeilen günstig. Altera konnte durch die Anwendung dieses Verfahrens bereits in einigen der angebotenen FPGA-Familien den Yield steigern [alt00] und setzt auch in der aktuellen Herstellung der Stratix IV-FPGAs ein Verfahren ein, das mit redundanten Zeilen arbeitet [alt08].

### 4.2.2 Redundante Logikblöcke

Anstelle der auf Zeilen und Spalten basierenden Redundanz können auch *einzelne* Reserve-Logikblöcke eingesetzt werden, um jeweils die Funktion eines defekten Logikblocks zu übernehmen.

Diese Methode wurde von Hanchek und Dutt [HD98] beschrieben und als *node covering* bezeichnet. Im Defektfall werden hier zusätzlich in die Architektur des FPGAs integrierte Reserve-Verbindungssegmente genutzt, damit die Verbindungen zwischen den verschobenen Logikblöcken und der restlichen Schaltung beibehalten werden können. Bei einer horizontalen Verschiebung um einen Logikblock braucht jeder Logikblock in einer Zeile die gleichen Anschlussmöglichkeiten wie der benachbarte Logikblock. Abbildung 4.5 zeigt den Einsatz der zusätzlichen Verbindungssegmente bei zwei defekten Logikblöcken in unterschiedlichen Zeilen, die jeweils durch einen Reserve-Logikblock in derselben Zeile ersetzt werden. Dieser Ansatz wurde später zu einer dynamischen Methode erweitert [DST99] . Als dynamisch wird bei dieser Erweiterung die Zuordnung der Reserve-Verbindungssegmente zu den Verbindungen, die durch sie ersetzt werden, bezeichnet. Durch die dynamische Nutzung der Reserve-Verbindungssegmente wird die Anzahl der insgesamt auf dem FPGA benötigten Verbindungssegmente reduziert.

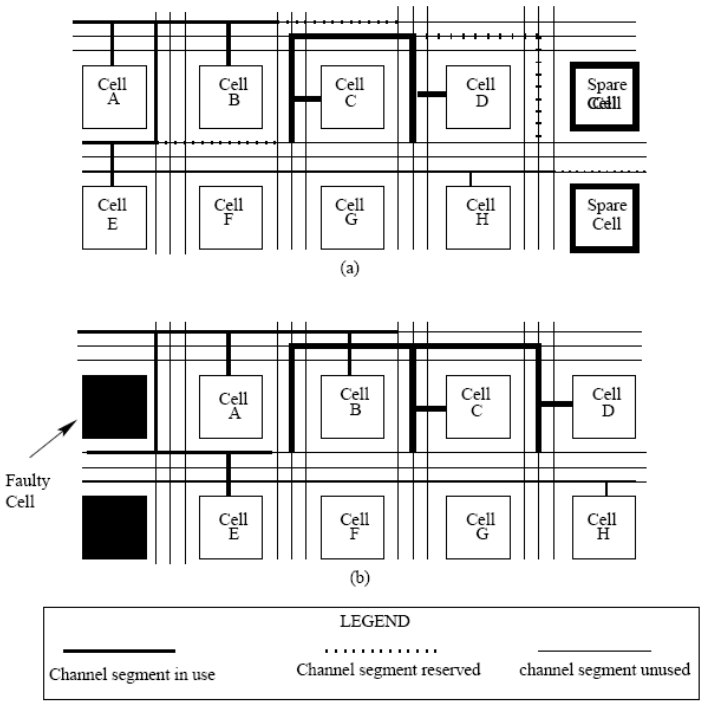



Abbildung 4.5: Der statische node covering Ansatz [HD98]: (a) Reserve-Verbindungssegmente und (b) deren Nutzung im Fall von Defekten in den Logikblöcken A und E

Auch das von Kelly und Ivey [KI94] vorgestellte Verfahren reserviert, vergleichbar mit dem Verfahren von Hanchek und Dutt [HD98], jeweils einen Logikblock, hier allerdings in Kombination mit einem *switch block*, als Reserve am Ende einer Zeile. Wenn die FPGA-Konfiguration auf das FPGA übertragen wird, erfolgt eine automatisierte Anpassung des FPGAs an die Konfiguration, so dass die defekten Elemente des FPGA nicht genutzt werden. Diese Anpassung ist für die Entwurfswerkzeuge nicht sichtbar.

Verglichen mit Verfahren, in denen Zeilen oder Spalten mit Reserve-Logikblöcken eingesetzt werden, ist die Nutzung von einzelnen Reserve-Logikblöcken potentiell dafür geeignet, mehr defekte Logikblöcke auszugleichen, da keine ganze Reserve-Zeile oder -Spalte eingesetzt werden muss, um einen Defekt auszugleichen. Ein Nachteil ist auch hier das zusätzliche Delay in einer FPGA-Architektur mit segmentierten Verbindungsleitungen, das durch längere Verbindungssegmente und zusätzliche programmierbare Schalter entsteht.

## 4.3 Veränderung der Programmierung des FPGAs

Die im Folgenden beschriebenen Arbeiten tolerieren Defekte, indem Veränderungen im Design-Flow und/oder Veränderungen in der Konfiguration des FPGA vorgenommen werden. Die grundlegende Idee für diese Verfahren ist die Erstellung einer FPGA-Konfiguration, die defekte Ressourcen auf dem FPGA nicht verwendet.

### 4.3.1 Individuelle Konfiguration für jedes FPGA

Die durch den FPGA-Hersteller bereitgestellten Entwurfswerkzeuge für den Place&Route-Vorgang können Ressourcen bei der Platzierung der Komponenten der Netzliste und der Verbindung dieser Komponenten als gesperrt betrachten. Diese Möglichkeit kann für die Tolerierung von Defekten auf dem FPGA ausgenutzt werden. Wenn für die zu konfigurierenden FPGAs als Ergebnis einer Diagnose jeweils eine Defect-Map zur Verfügung steht, kann so für jedes FPGA eine Konfiguration erzeugt werden, in der die durch die Defect-Map angegebenen fehlerhafte Bereiche des FPGAs nicht verwendet werden.

Ein Beispiel für die erfolgreiche Verwendung von defekten FPGAs ist ein bei Hewlett-Packard entwickelter Rechner [CAC+97] [HKSW98], der zur Erforschung von parallelen Rechnerarchitekturen eingesetzt wurde. Durch die Verwendung von defekten FPGAs konnten Kosten eingespart werden, wie in [HKSW98] berichtet wurde:

> "Only 217 of the FPGAs used in Teramac were free of defects; the rest (75% of the total used) were free of charge, because the commercial foundry that made them would normally have discarded them."

Die Nutzbarkeit der defekten FPGAs wurde erreicht, indem auf den verwendeten FPGAs durch Testschaltungen Fehler gesucht, die gefundenen Fehler in einer Datenbank gespeichert und im Anschluss daran der für den Place&Route-Vorgang zuständigen Software nur defektfreie Ressourcen zur Verfügung gestellt wurden. Für die Erstellung der Defect-Maps für die FPGAs wurde eine zeitaufwendige externe Diagnose der FPGAs durchgeführt. Nähere Informationen zu der durchgeführten Diagnose wurden in [CAC+97] veröffentlicht. Abbildung 4.6 skizziert die Nutzung von defekten FPGAs im Teramac Custom Computer.

Neben der aufwendigen externen Diagnose für die defekten FPGAs, gibt es einen weiteren Grund, der gegen den Einsatz des in Abbildung 4.6 dargestellten Verfahrens in einer Serienproduktion spricht. Abbildung 4.7 zeigt die unterschiedlichen Beschreibungsformen eines FPGA Design-Flows, in diesem Fall dem Design Flow für die FPGAs der Firma Xilinx, und eine ungefähre Abschätzung des Zeitaufwands, der benötigt wird, um jeweils die nächste Beschreibungsform zu erreichen. Auch für den Fall, dass die Synthese und das Mapping bereits erfolgt sind, und das Design als Netzliste vorliegt, wäre

der für jedes defekte FPGA durchzuführende Place&Route-Vorgang zu zeitaufwändig, um ihn innerhalb einer Serienproduktion durchzuführen.

### 4.3.2 Dynamisches Place & Route

Der Zeitaufwand für die Diagnose und die Durchführung des Place&Route-Vorgangs für jedes FPGA kann innerhalb der Produktion vermieden werden, wenn beides erst nach dem Einschalten des FPGAs intern in dem FPGA durchgeführt wird. Macias und Durbeck beschreiben in [MD04], wie auf einer speziell dafür ausgelegten Architektur, der Cell Matrix [DM01], die Diagnose des Chips und die Platzierung und Verdrahtung der Schaltung auf dem Chip durchgeführt werden kann. Abbildung 4.8 zeigt, wie aus einer initial vorliegenden Matrix mit defekten Zellen sogenannte Superzellen (*supercells*) konstruiert werden, die aus n x n Zellen der Matrix bestehen. Jede Superzelle realisiert einen Teil der durch eine Netzliste vorgegebenen Schaltung.

Im Unterschied zu den Logikblöcken der FPGA-Architekturen können die Zellen der Cell Matrix ihre benachbarten Logikblöcke konfigurieren. Sobald eine Superzelle auf der Cell Matrix eingerichtet wurde, testet diese Superzelle mit Hilfe von extern an die Cell Matrix angelegten Testdaten einen benachbarten Bereich von n x n Zellen. Wenn dieser Bereich keine Defekte enthält, programmiert die Superzelle diesen Bereich ebenfalls als Superzelle. Falls im Test Defekte festgestellt wurden, wird der getestete Bereich komplett als defekt betrachtet und gesperrt. Nachdem die Superzellen auf der Cell Matrix eingerichtet worden sind, findet in einer zweiten Phase eine eindeutige Zuweisung von Teilen der zu realisierenden Schaltung auf die Superzellen statt, und die Verbindungen zwischen den Superzellen werden hergestellt. Diese Platzierung und Verdrahtung findet komplett innerhalb der Cell Matrix statt. Die Superzellen auf der Cell Matrix nummerieren sich dazu in einem ersten Schritt fortlaufend, orientiert an ihrer Position

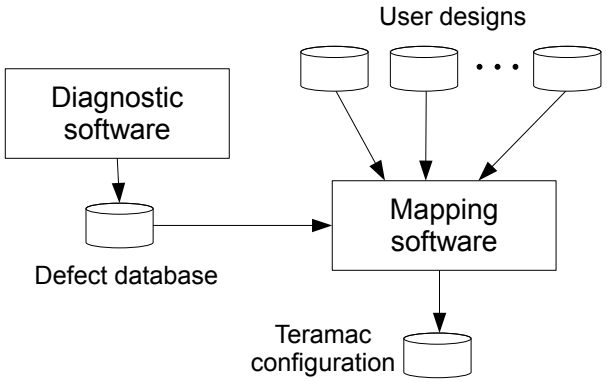



Abbildung 4.6: Toleranz von Defekten im Teramac Custom Computer [CAC+97]

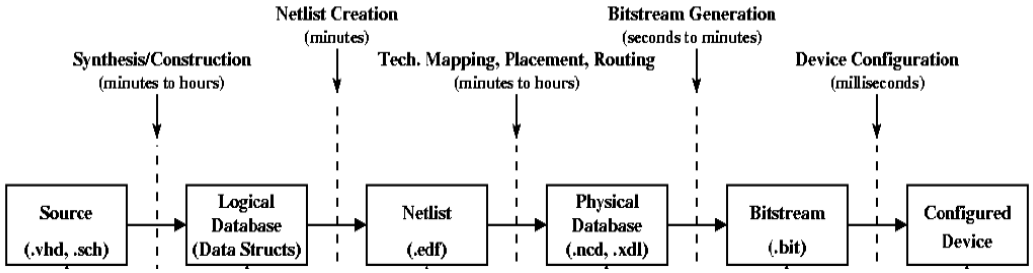



Abbildung 4.7: Die Zwischenformate im FPGA-Design-Flow der Firma Xilinx [CGJW02]

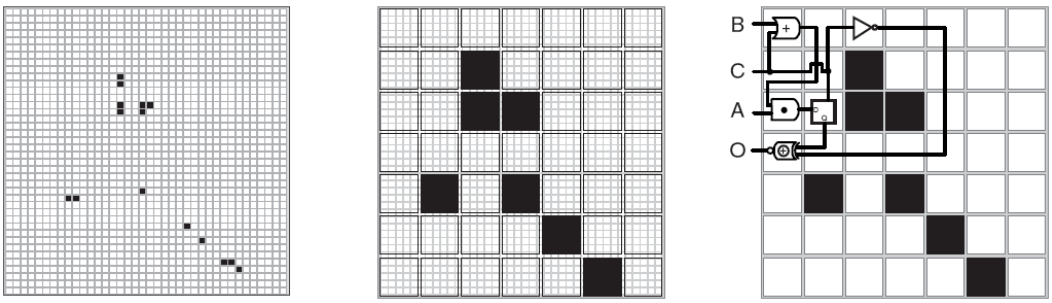



Abbildung 4.8: Links: Eine Cell Matrix [MD04] mit defekten (=schwarz dargestellten) Bereichen; Mittel: Nicht defekte Zellen werden zu Superzellen zusammengefasst; Rechts: Die Superzellen implementieren die durch eine Netzliste vorgegebene Schaltung

innerhalb der Cell Matrix. Da die in der vorgegebenen Netzliste enthaltenen Knoten ebenfalls fortlaufend nummeriert sind, kann jede Superzelle jetzt die Funktion des Knotens übernehmen, der in der Netzliste mit der gleichen Nummer bezeichnet wurde. Jede Superzelle konfiguriert im Anschluss daran die Verbindungen zu ihren Vorgängerknoten innerhalb der Netzliste, nachdem über ein zwischen den Superzellen bestehendes Netzwerk ein Kontakt zu den jeweiligen Vorgängerknoten aufgenommen wurde [MD02]. Von der Cell-Matrix existiert eine Hardware-Implementierung mit 8 x 8 Zellen [MD05].

### 4.3.3 Anpassung der Konfiguration

Um den Zeitaufwand für die Programmierung eines defekten FPGAs möglichst gering zu halten, kann anstelle der erneuten Erstellung einer FPGA-Konfiguration eine bestehende FPGA-Konfiguration angepasst werden. Diese Anpassung kann erfolgen, wenn die Konfiguration in Form eines Bitstreams für die Programmierung des FPGAs vorliegt ,oder wenn das FPGA bereits konfiguriert wurde (vgl. Abbildung 4.7).

Die Anpassung der Konfiguration auf einem bereits konfigurierten FPGA wird in einem Verfahren durchgeführt, das von Doumar und Ito [DI00a] vorgeschlagen wird. Bei diesem Ansatz werden Logikblöcke auf dem FPGA platziert, die nicht für die Implementierung des FPGA-Designs benötigt werden. Direkt nach dem Einschalten des FPGAs wird eine Diagnose durchgeführt. Durch eine Verschiebung (*shifting*) des gesamten FPGA-Designs kann in Anschluss an die Diagnose jeder defekte Logikblock ausgeglichen werden, indem das Design so verschoben wird, dass ein im Design nicht benötigter Logikblock auf den defekten Logikblock verschoben wird. Der defekte Logikblock wird somit durch den nicht benötigten Logikblock maskiert. Für die Verteilung der nicht benötigten Logikblöcke werden von den Autoren die in Abbildung 4.9 gezeigten Verteilungen vorgeschlagen. Die linke Verteilung erlaubt die Maskierung eines Defektes, indem das Design höchstens um einen Logikblock vertikal und / oder horizontal verschoben wird. Die rechte der abgebildeten Verteilungen erlaubt die Maskierung eines Defektes mit bereits einer Verschiebung, entweder horizontal oder vertikal. Ein Vorteil dieses Verfahrens ist die einfache Anpassung der FPGA-Konfiguration an das defekte FPGA. Diese Methode kann jedoch in der Regel nur einen Defekt auf dem gesamten FPGA tolerieren. Durch das Einfügen der nicht genutzten Logikblöcke in die FPGA-Konfiguration wird das Delay an einigen Verbindungsleitungen innerhalb des FPGA gegenüber einer FPGA-Konfiguration, die ohne Reserve-Logikblöcke erstellt wurde, erhöht [DI00a].

### 4.3.4 Prekompilierte Schaltungsteile

Wenn wir davon ausgehen, dass maximal ein defekter Logikblock je FPGA vorhanden ist, können für ein FPGA mit x Logikblöcken x verschiedene Konfigurationen erstellt werden, so dass jede dieser Konfigurationen einen anderen der auf dem FPGA vorhandenen Logikblöcke nicht nutzt. Während der Serienproduktion kann anschließend aus den im Voraus erstellten Konfigurationen mit geringem Zeitaufwand eine Konfiguration ausgewählt werden, die das FPGA so konfiguriert, dass der defekte Logikblock nicht verwendet wird.

Dieses Verfahren ist nur für kleine FPGAs praktikabel, da die Anzahl der zu berechnenden FPGA-Konfigurationen der Anzahl der auf dem FPGA vorhandenen Logikblöcke entspricht. Spätestens, wenn Konfigurationen für den Ausgleich von zwei defekten Logikblöcken an beliebigen Stellen auf dem FPGA in der Datenbank bereitgehalten werden sollen, ist dieser „naive“ Ansatz nicht mehr durchführbar. Bereits für ein FPGA mit 16000 Logikblöcken[2] wäre eine Berechnung und Speicherung von $\binom{16000}{2} = 127992000$ Konfigurationen notwendig.

[2]Es sind bereits FPGAs mit über 25000 Logikblöcken erhältlich

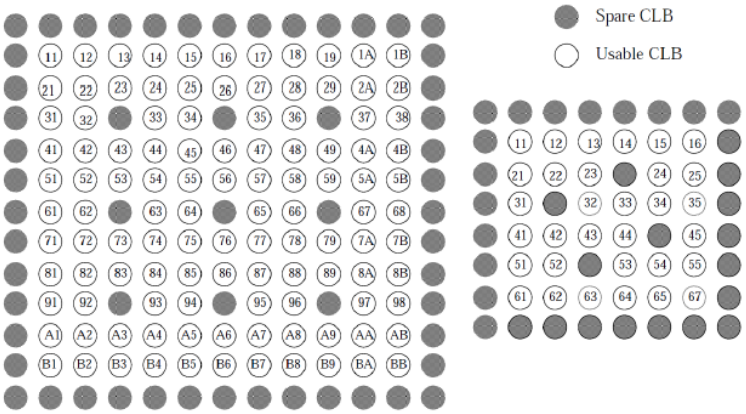



Abbildung 4.9: Zwei mögliche Verteilungen von Reserve-Logikblöcken, die für den Ansatz von Doumar et al. [DI00a] eingesetzt werden können.

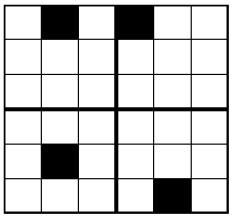


Abbildung 4.10: 6x6 Logikblöcke in Bereiche mit je 3x3 Logikblöcken partitioniert. Nicht verwendete Logikblöcke sind schwarz dargestellt [LMSP98b]

Ein von Lach et al. [LMSP98b] veröffentlichter Ansatz enthält erstmals die Idee, nicht die gesamte Konfiguration für jeden möglichen Defekt auf dem FPGA, sondern nur Teilregionen im Voraus zu berechnen, in denen jeweils ein Logikblock nicht verwendet wird. Abbildung 4.10 zeigt die Partitionierung eines aus 6x6 Logikblöcken bestehenden Arrays in mehrere, als *tiles* bezeichnete, Bereiche aus 3x3 Logikblöcken. Diese Partitionierung bietet gegenüber der Berechnung von Konfigurationen für das nicht partitionierte 6x6 Array folgende Vorteile:

- Der Zeitaufwand für die Erstellung der Konfigurationen wird reduziert und die Anzahl der maximal tolerierbaren Defekte wird erhöht
  - Wenn auf dem 6x6 Array maximal ein defekter Logikblock toleriert werden soll, werden 36 FPGA-Konfigurationen benötigt. Bei den gleichen Voraussetzungen für ein *tile* werden ebenfalls $3 * 3 * 4 = 36$ FPGA-Konfigurationen benötigt, die jedoch jeweils weniger umfangreich sind und daher den Place&Route-Vorgang mit weniger Zeitaufwand durchlaufen. Ein so partitioniertes FPGA kann in jedem *tile* einen defekten Logikblock und insgesamt bis zu vier defekte Logikblöcke tolerieren.

- Der benötigte Speicher für die FPGA-Konfigurationen wird reduziert
  - Wenn man davon ausgeht, dass für die Speicherung der Konfigurationen des unpartitionierten Arrays $36 * X$ Bytes benötigt werden, und die Speicherung eines tile nach der Partitionierung nur $4 * 9 * X/4 = 9 * X$ Bytes benötigt, ist 75% weniger Speicher für die Bereithaltung der Konfigurationen ausreichend

Neben dem Begriff des *tile* führt [LMSP98b] noch den Begriff *atomic fault tolerant block* (AFTB) als Bezeichnung für eine mögliche Konfiguration eines *tile* ein. Abbildung 4.11 verdeutlicht diese Zusammenhänge für ein *tile*, das auf einer Fläche von 2x2 Logikblöcken die Funktion $Y = (A \vee B) \vee (C \wedge D)$ realisiert. In der Abbildung sind die drei Bestandteile eines *tile* sichtbar:

- Logikblöcke, auf denen die Funktion des *tile* realisiert wird
- Eine Spezifikation der Verbindungen zu den umliegenden *tiles*, im Folgenden als Interface bezeichnet
- Eine Netzliste

Ein AFTB wird unter Vorgaben für eine konkrete Platzierung mit den Place&Route-Tools des FPGA-Herstellers aus der zu dem *tile* gehörenden Netzliste erstellt. Abbildung 4.11 zeigt vier AFTBs (I bis IV). In jedem AFTB wird genau ein Logikblock nicht verwendet.

Das Interface des *tiles* muss fest auf dem FPGA platziert sein, damit ein beliebiger der dazugehörigen AFTB ohne Auswirkungen auf die Umgebung in die Schaltung eingesetzt werden kann. Eine Auswirkung bleibt jedoch bestehen: Je nach gewähltem AFTB besitzt das *tile* unterschiedliche Timing-Eigenschaften, da die Verbindungen zwischen den Logikblöcken und zu dem Interface über unterschiedliche Pfade innerhalb des *tile* erfolgen.

Der Algorithmus für die Erstellung einer an defekte FPGAs anpassbaren FPGA-Konfiguration ist in [LMSP98b] mit dem in Algorithmus 1 auf Seite 53 dargestellten Pseudocode angegeben und soll im Folgenden kurz erläutert werden.

In den Zeilen drei und vier werden die benutzte Fläche und das Zeitverhalten für die initial erstellte FPGA-Konfiguration aus der Schaltung extrahiert, sowie eine Berechnung der Wahrscheinlichkeit, dass diese Konfiguration auf einem defekten FPGA funktioniert, vorgenommen. Der berechnete Wert entspricht dabei der Wahrscheinlichkeit dafür, dass ein FPGA keine Defekte oder nur Defekte in Logikblöcken enthält, die von der in Zeile 2 erstellten FPGA-Konfiguration nicht verwendet werden.

In Zeile fünf beginnt eine While-Schleife, die terminiert, wenn alle durch den Benutzer gewünschten Eigenschaften für die bisher erzeugten Konfigurationen erfüllt sind. Zu diesen Eigenschaften zählen

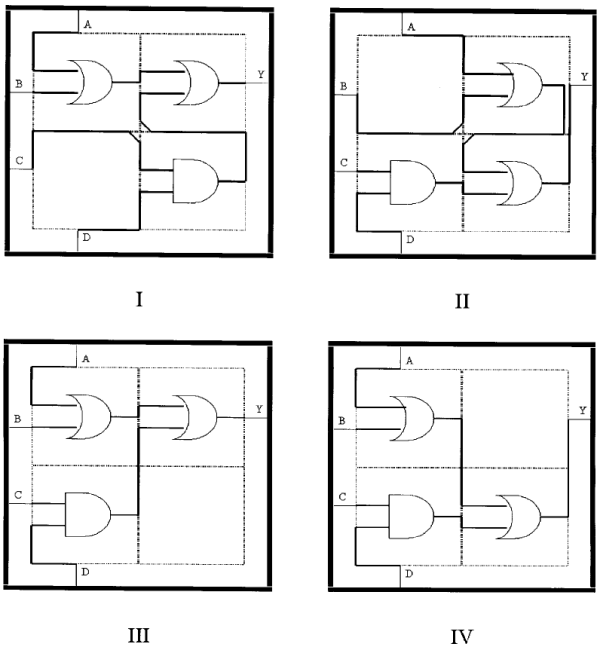



Abbildung 4.11: Vier AFTBs für ein *tile* das 2x2 Logikblöcke umfasst und die Funktion $Y = (A \vee B) \vee (C \wedge D)$ realisiert

- die benötigte Fläche auf dem FPGA (*area criteria*),
- die Einhaltung aller Timing-Vorgaben (*timing criteria*),
- die gewünschte Wahrscheinlichkeit für eine Funktion der mit Hilfe der bereits erzeugten AFTBs erreichbaren FPGA-Konfigurationen (*reliability criteria*) und
- der für die bereits erzeugten AFTBs benötigte Speicher.

Die While-Schleife terminiert ebenfalls, wenn die gewünschten Eigenschaften nach dem Ausführen aller möglichen Partitionierungen nicht erfüllt werden konnten (*tiling possibilities exhausted*).

Zeile sechs partitioniert die in Zeile zwei erzeugte FPGA-Konfiguration in *tiles*. Diese Partitionierung erfolgt nach den folgenden Kriterien:

- Anzahl der Verbindungen, die über das Interface mit der Außenwelt hergestellt werden müssten
- Die Dichte der Logik
- Makros
- Größe

Genauere Angaben zu dem Partitionierungs-Algorithmus wurden nicht veröffentlicht.

**Algorithmus 4.1** : Ein Algorithmus für die Erstellung einer an defekte FPGAs anpassbaren FPGA-Konfiguration

```
1  while !(complete or design possibilities exhausted) do
2      create initial non_ft_design;
3      extract timing and area information;
4      calculate design reliability;
5      while !(complete or tiling possibilities exhausted) do
6          partition design into tiles;
7          if !meet area criteria then
8              break;
9          while !(complete or AFTB possibilities exhausted) do
10             partition tiles into AFTBs;
11             calculate AFTB reliability;
12             if !meet reliability criteria then
13                 break;
14             order tiles by ft realization difficulty;
15             order AFTBs by ft realization difficulty;
16             for (j = 1; j <= # of tiles; decreasing difficulty) do
17                 for (i = 1; i <= # of AFTBs; decreasing difficulty) do
18                     create ft_design(i,j);
19                     if !(success and meet timing criteria) then
20                         break;
```

Wenn durch die Partitionierung die auf dem FPGA benötigte Fläche die durch den Benutzer vorgegebene maximal zulässige Fläche übersteigt, muss die Partitionierung erneut erfolgen, und in Zeile acht wird die aktuelle Iteration der While-Schleife beendet.

Die Erstellung der AFTBs beginnt ab Zeile neun mit einer Schleife, die terminiert, wenn alle durch den Benutzer gewünschten Eigenschaften (siehe oben) erfüllt sind, oder die Eigenschaften mit der in Zeile sechs vorgenommenen Partitionierung nicht erfüllt werden können. Auch in diesem Fall muss eine erneute Partitionierung erfolgen.

In Zeile zehn wird jedem der in Zeile sechs erstellten *tiles* eine Menge von AFTBs zugewiesen[3]. Da es vorkommt, dass Logikblöcke innerhalb der Netzliste des *tile*, für das die AFTBs erzeugt werden sollen, direkt nebeneinander auf dem FPGA platziert

[3]Dem in Abbildung 4.11 dargestellten tile würden an dieser Stelle z.B. die Platzierungen I bis IV zugewiesen. An dieser Stelle erfolgt jedoch noch keine Realisierung der AFTBs durch die Place&Route-Tools des Herstellers.

werden müssen, können unter Umständen nicht alle theoretisch möglichen AFTBs für das betreffende *tile* erzeugt werden, die für die Tolerierung eines Defektes an jeder Stelle des *tile* notwendig wären. Aus diesem Grund erfolgt in Zeile 11 eine Berechnung der Wahrscheinlichkeit für eine Funktion der FPGA-Konfiguration (*reliability*) aufgrund der tatsächlich für die *tiles* erzeugbaren AFTBs.

Wenn die durch den Benutzer gewünschte Wahrscheinlichkeit für eine Funktion (*reliability criteria*) nicht erreicht wurde, verursacht Zeile 13 eine erneute Durchführung der AFTB-Zuweisungen durch die Fortsetzung des Algorithmus in Zeile zehn.

Genauere Angaben zu dem Algorithmus für die Zuweisung der AFTBs an die Partitionen, die in Zeile sechs vorgenommen wird, wurden nicht veröffentlicht.

Nachdem in Zeile 14 und 15 die tiles und die AFTBs so sortiert wurden, dass schwierig zu realisierende *tiles* bzw. AFTBs in den nachfolgenden For-Schleifen zuerst den Place&Route-Vorgang durchlaufen, findet die eigentliche Realisierung der in Zeile 9 den *tiles* zugeordneten AFTBs in den Zeilen 16 bis 20 statt. Wenn die Realisierung eines geplanten AFTBs nicht möglich ist, wird die Realisierung der AFTBs - durch die Vorsortierung in Zeile 14 und 15 begünstigt - zu einem früheren Zeitpunkt abgebrochen (*!success*).

Erst nachdem alle AFTBs realisiert wurden, kann der kritische Pfad bestimmt werden, da erst jetzt die maximalen Verzögerungszeiten durch die tiles ermittelt werden können. Wenn die Zeitvorgaben nicht eingehalten werden können, wird daher erst nach der kompletten Erstellung aller AFTBs der Algorithmus in Zeile 20 erfolglos abgebrochen (*!meet timing criteria*).

Falls noch nicht alle möglichen AFTBs realisiert wurden, kann eine Fortsetzung in Zeile neun erfolgen. Wenn alle Möglichkeiten für die Konstruktion der AFTB ausgeschöpft sind, erfolgt mit einer Fortsetzung des Algorithmus in Zeile sechs eine alternative Partitionierung der Schaltung in *tiles*. Wenn auch an dieser Stelle die Möglichkeiten ausgeschöpft sind, wird in Zeile zwei eine alternative FPGA-Konfiguration erstellt, sofern noch alternative Realisierungs-Möglichkeiten vorhanden sind.

Für eine vollständige Automatisierung des von Lach et al. angegebenen Algorithmus sind Angaben erforderlich, die in keiner der Veröffentlichungen zu diesem Ansatz [LMSP98a, LMSP98b, LMSP00] zu finden sind. Es ist nicht erkennbar, wie der Ansatz in den rechnergestützten Schaltungsentwurf für ein FPGA eingebaut wird, ohne dass ein erheblicher Eingriff „von Hand“ notwendig ist. Ein weiterer Kritikpunkt an dem hier beschriebenen Algorithmus ist die Verteilung von nicht verwendeten Logikblöcken im Design, durch die das Delay an einigen Verbindungsleitungen erhöht wird.

## 4.4 Zusammenfassung

In diesem Kapitel wurden die wichtigsten Verfahren diskutiert, die eine Nutzung von defekten FPGAs ermöglichen.

Einige der vorgestellten Ansätze fügen redundante Logikblöcke in die FPGA-Architektur ein. Dieses Vorgehen ist bei FPGAs mit segmentierten Verbindungsleitungen problematisch, da in diesem Fall neben zusätzlichen programmierbaren Schaltern, die das „Verbergen“ der defekten FPGA-Komponenten ermöglichen, zusätzliche bzw. längere Verbindungssegmente in der FPGA-Architektur benötigt werden. Diejenigen der vorgestellten Ansätze, die mit einer veränderten Programmierung für ein defektes FPGA arbeiten, sind daher für den Einsatz auf einem FPGA mit segmentierten Verbindungsleitungen potentiell besser geeignet.

Ein individuell für jedes defekte FPGA durchgeführter Place&Route-Vorgang ist innerhalb einer Serienproduktion sehr zeitaufwändig und daher unwirtschaftlich. Der hohe Zeitaufwand für die Erstellung einer individuellen FPGA-Konfiguration lässt sich durch Ansätze reduzieren, die eine schnelle Anpassung einer vorhandenen FPGA-Konfiguration ermöglichen. Das Verfahren von Doumar und Ito [DI00a] verändert die Konfiguration auf einem bereits konfigurierten FPGA. Die vorgeschlagene Methode kann in der Regel nur einen Defekt auf dem gesamten FPGA tolerieren und verändert das Zeitverhalten eines FPGA-Designs durch zusätzlich in die FPGA-Konfiguration eingefügte Reserve-Logikblöcke. Der von Lach et al. [LMSP98b] veröffentlichte Ansatz arbeitet erstmals mit prekompilierten Schaltungsteilen, die innerhalb der Serienproduktion mit einem geringen Zeitaufwand derart ausgetauscht werden können, dass die Nutzung von FPGAs mit defekten Logikblöcken möglich ist. Auch in diesem Ansatz werden zusätzliche Reserve-Logikblöcke in die FPGA-Konfiguration eingefügt und dadurch das Zeitverhalten des FPGA-Designs verändert. Die Verteilung der Reserve-Logikblöcke in der FPGA-Konfiguration wird bei den vorgestellten Verfahren ohne Rücksicht auf die für das FPGA-Design vorliegenden Timing-Constraints vorgenommen. Da vorher nicht bekannt ist, welche Defekte auf den zu programmierenden FPGAs vorliegen, kann für die Schaltung nur der längste Pfad, der aufgrund einer Anpassung an ein defektes FPGA entstehen kann, als obere Schranke für die Verzögerungszeit garantiert werden. Diese pessimistische obere Schranke für die Verzögerungszeit muss daher als kritischer Pfad für die realisierte Schaltung angesehen werden.

Ein ausführlicher Vergleich des innerhalb dieser Arbeit entwickelten Ansatzes mit den vergleichbaren Ansätzen erfolgt in Kapitel **6.**

# 5 Ein Hard-Core basierter Ansatz

Nachdem im vorhergehenden Kapitel ein Überblick über bestehende Ansätze zur Nutzung von teildefekten FPGAs gegeben wurde, soll in diesem Kapitel ein neuartiger Hard-Core basierter Ansatz erläutert werden. Dieser Ansatz ist nach bestem Wissen des Autors das erste Verfahren zur Erhöhung der Ausbeute an nutzbaren FPGAs, das mit prekompilierten Schaltungsteilen in Form von Hard-Cores arbeitet.

Um das Verfahren anwenden zu können, müssen folgende Vorbedingungen erfüllt sein:

- Durch eine entsprechende Diagnose wurde nach der Produktion des FPGAs durch den Hersteller eine *defect map* erzeugt, in der die defekten Logikblöcke verzeichnet sind.
- Das Design basiert vorwiegend auf IP-Cores, die in Form von Hard-Cores vorgegeben sind.

Die Hard-Cores können in einer ersten Annäherung, vergleichbar mit dem Verfahren von Lach et al. [LMSP98a], als *tile*, also einem Ausschnitt der Schaltung mit einem festen Interface, betrachtet werden. Die grundlegende Idee des Hard Core basierten Verfahrens ist die Bereitstellung von defekttoleranten Versionen der Hard-Cores durch den IP-Core-Hersteller. Diese defekttoleranten Versionen werden dann im Rahmen der Erstellung einer defekttoleranten FPGA-Konfiguration durch ein Re-Mapping in die Schaltung eingesetzt.

Abbildung 5.1 zeigt verschiedene Versionen eines Hard-Cores, die funktional gleich sind. Version 1 stellt den ursprünglich durch den IP-Anbieter entworfenen IP-Core dar, innerhalb dem jeder der in der Version enthaltenen 32 Logikblöcke verwendet wird. Für Version 2 liegen 33 verschiedene Varianten vor. Jede dieser Varianten nutzt einen Logikblock nicht. Um den Speicheraufwand für die Varianten zu reduzieren, können auch, vergleichbar mit der aus dem Ansatz von Lach et al. bekannten Partitionierung, innerhalb einer Version Partitionen erzeugt werden. Version 3 in Abbildung 5.1 wurde analog zu Abbildung 4.10 auf Seite 50 konstruiert, d.h. die Version besteht aus vier Partitionen, die über feste Interfaces miteinander kommunizieren. Wenn der Hersteller des Hard-Core verschiedene Versionen anbietet, kann im FPGA-Design-Flow zunächst mit der initialen Version, in Abbildung 5.1 ist dies Version 1, gearbeitet werden. Nachdem die Entscheidung gefallen ist, dass defekte FPGAs in der Serienproduktion eines Produktes eingesetzt werden sollen, kann das Hard-Core spät im Design-Flow automatisiert ausgetauscht, d.h. Version 2 oder Version 3 anstelle von Version 1 in das Design eingesetzt

werden. Da die Abbildung der zu realisierenden Funktion auf die Realisierung dieser Funktion durch funktionale Einheiten auf dem FPGA im allgemeinen als Mapping bezeichnet wird, kann in diesem Fall von einem Re-Mapping gesprochen werden. Jedes Mapping, das mindestens auf einen defekttoleranten Hard-Core abbildet, wird im Folgenden als DT-Mapping bezeichnet. Abbildung 5.2 skizziert den Re-Mapping Vorgang, in dem das ursprüngliche Mapping zu einem DT-Mapping umgeformt wird. Im Gegensatz zu der aus dem ursprünglichen Mapping erstellten FPGA-Konfiguration entsteht durch den Place&Route-Vorgang nach dem Re-Mapping eine defekttolerante FPGA-Konfiguration, die in Abbildung 5.2 und im Folgenden als DT-Konfiguration bezeichnet wird.

**Definition 5.1.** Eine **DT-Konfiguration** bezeichnet eine FPGA-Konfiguration, die mindestens eine defekttolerante Hard-Core-Version enthält.

Der nächste Abschnitt erläutert, wie der in dieser Arbeit entwickelte Ansatz in den Design-Flow eingebettet wird. In dem darauffolgenden Abschnitt wird die Bereitstellung von DT-Versionen der IP-Cores inklusive der für die Durchführung des Re-Mappings benötigten Informationen erläutert. Für die Durchführung des Re-Mappings im Design-Flow wird anschließend in Abschnitt 5.3 eine innerhalb der vorliegenden Arbeit entwickelte Heuristik vorgestellt. Bei der Programmierung von defekten FPGAs in einer Serienproduktion ist die Anpassung einer DT-Konfiguration an FPGAs mit defekten Logikblöcken an verschiedenen Stellen erforderlich. Diese Anpassung kann mit einem sehr geringen Aufwand erfolgen und wird in Abschnitt 5.4 erläutert.

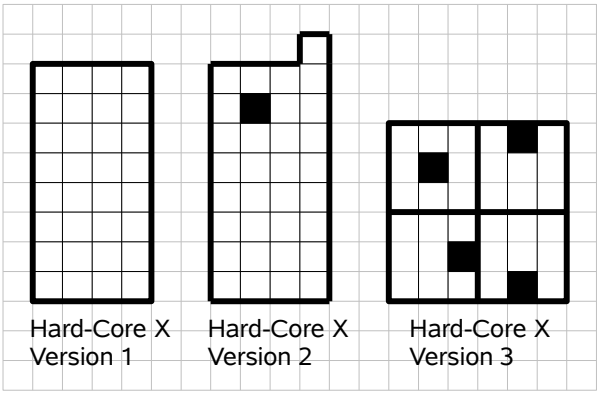



Abbildung 5.1: Drei Versionen eines Hard-Core

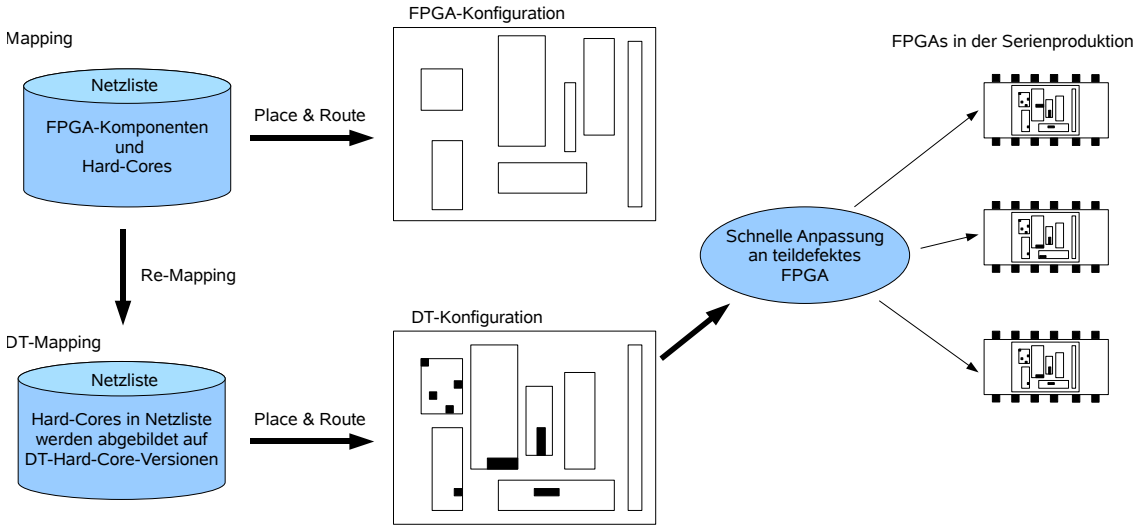



Abbildung 5.2: Re-Mapping und schnelle Anpassung der DT-Konfiguration in einer Serienproduktion

## 5.1 Design-Flow

Der bereits in Kapitel 2 dargestellte Schaltungsentwurf für FPGAs kann in verschiedene Phasen unterteilt werden (siehe Abbildung 5.3)

1. *Design Entry/Design Synthesis*: In der Synthese (*Design Synthesis*) wird aus der Beschreibung der Schaltung in einer Hardware-Beschreibungssprache oder aus einer grafischen Beschreibung der Schaltung (*Schematic*) eine Netzliste erzeugt. Diese Netzliste kann anschließend mit dem Ziel einer funktionalen Überprüfung simuliert werden. Da die Abbildung der Netzliste auf die FPGA-Technologie noch nicht erfolgt ist, stehen noch keine Informationen über das Zeitverhalten der Schaltung für die Simulation zur Verfügung.
2. *Design Implementation*: Nachdem der FPGA-Schaltungsdesigner das funktionale Verhalten der Schaltung überprüft hat, sind in dieser Phase nach dem Mapping erste Zeitinformationen vorhanden, da die Netzliste nun auf die im FPGA vorhandenen Logikblöcke und die verwendeten IP-cores abgebildet wurde. Zeitinformationen über die Verbindungen der Logikblöcke und der IP-cores untereinander und über die Verbindungen mit der Außenwelt können allerdings nach dem Mapping in einer Simulation oder in einer Timing-Analyse nur geschätzt werden, für genauere Zeitangaben ist die Platzierung und die Verdrahtung (*Place & Route*) unumgänglich.
3. *Bitstream Generation*: Wenn das funktionale und das zeitliche Verhalten der Schaltung der Spezifikation entspricht, kann aus der vollständigen Beschreibung

der Schaltung, die an dieser Stelle noch in einem Zwischenformat vorliegt, der Bitstream für die Programmierung des FPGAs generiert und an das FPGA übermittelt werden.

Zur Unterstützung des Schaltungsentwurfs ist die Überprüfung (*Design Verification*) somit durch

- eine Funktionale Simulation (*Functional Simulation*),
- eine Analyse des Zeitverhaltens der Schaltung (*Static Timing Analysis*),
- eine Simulation mit genauen Signallaufzeiten, die in der Design Implementation ermittelt wurden (*Timing Simulation*) und
- eine Überprüfung des korrekten Verhaltens auf dem FPGA (*In-Circuit-Verification*)

möglich. Da die Implementierung des Designs, also die 2. Phase, zeitaufwendig ist, erfolgt eine Simulation mit dem Ziel der funktionalen Überprüfung des Designs bereits innerhalb der ersten Phase. Erst wenn das funktionale Verhalten der Spezifikation entspricht und mit der Überprüfung des Zeitverhaltens fortgefahren werden soll, wird die 2. Phase durchlaufen.

Die im Folgenden beschriebene Methode zur Erstellung eines DT-Mappings geschieht im Anschluss an die Phase *Design Implementation*, also erst nach der Erstellung und dem Test der ursprünglichen FPGA-Konfiguration, und kann als eine Erweiterung dieser Phase eingeordnet werden. Auf der Grundlage von dem in einer Timing-Analyse ermittelten Slack wird durch einen Austausch von Hard-Core-Versionen ein Re-Mapping durchgeführt. Durch eine erneute Durchführung des Place&Route-Vorgangs erfolgt dann die Erstellung der DT-Konfiguration. Dieser Ablauf ist in Abbildung 5.4 skizziert.

Die Programmierung der defekten FPGAs in der Serienproduktion kann in Abbildung 5.3 der 3. Phase zugeordnet werden.

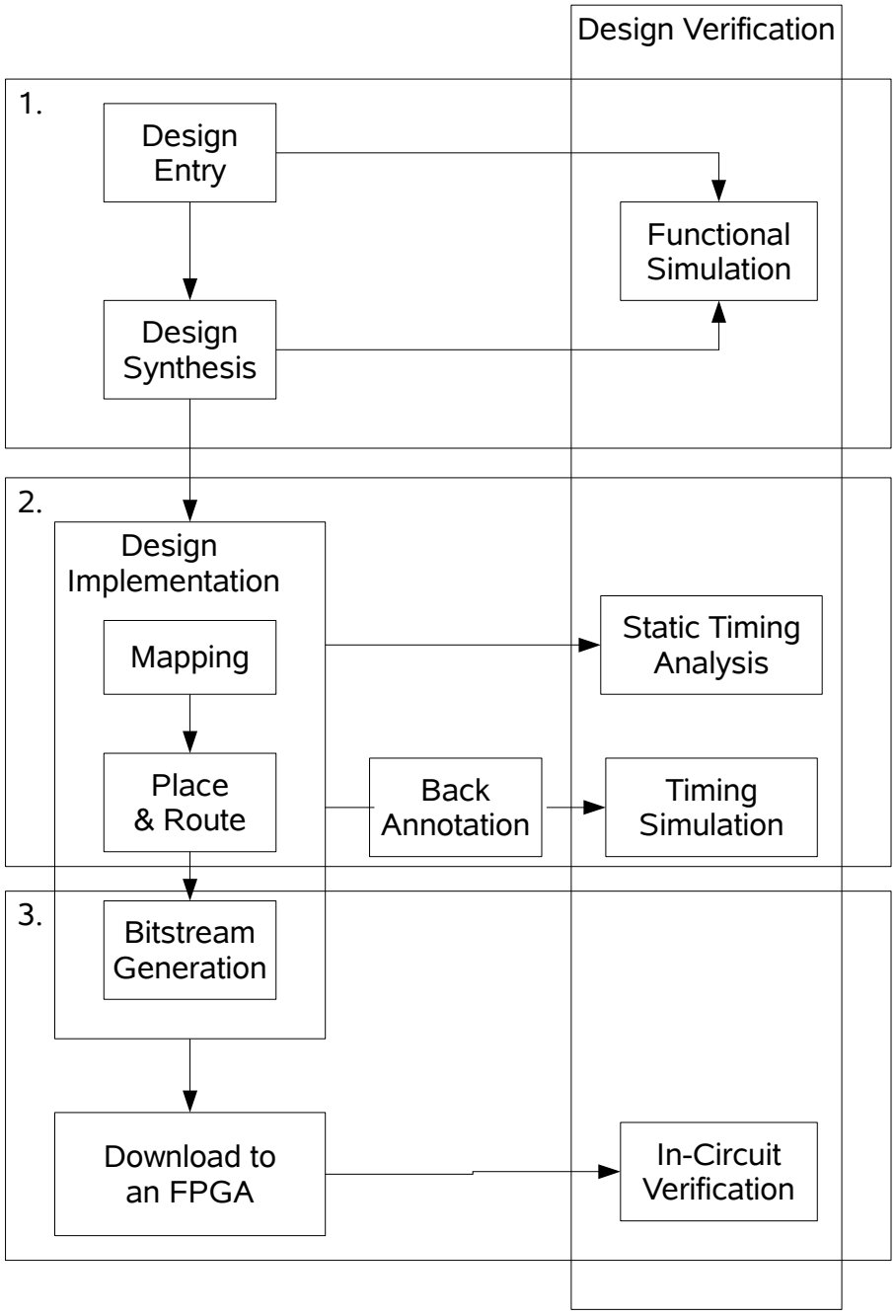



Abbildung 5.3: Der Schaltungsentwurf für eine FPGA-Konfiguration lässt sich in drei Phasen unterteilen

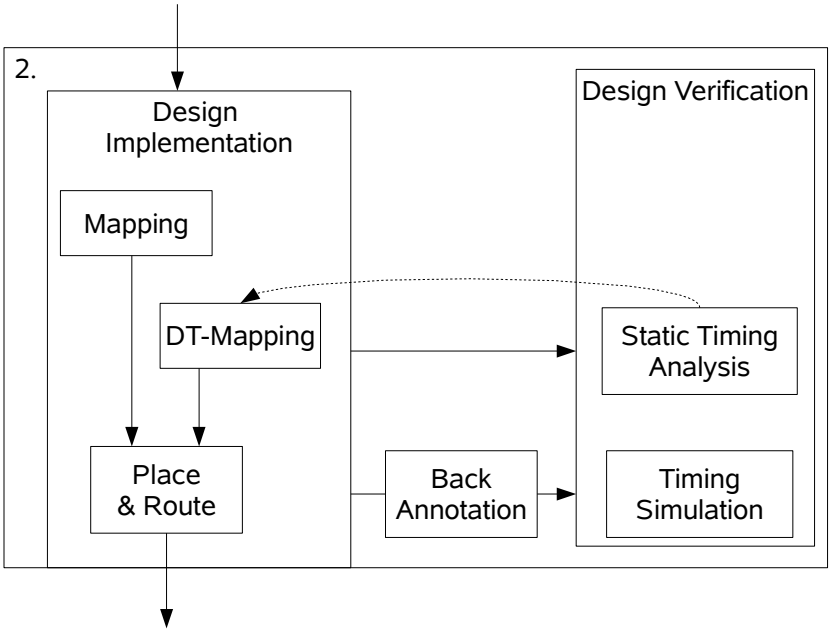



Abbildung 5.4: Das Re-Mapping im Design-Flow

## 5.2 Bereitstellung einer Hard-Core-Version

Bevor es um die Bereitstellung der erforderlichen DT-Versionen der Hard-Cores geht, soll zunächst die Erstellung eines Hard-Cores im Allgemeinen betrachtet werden.

Durch die FPGA-Hersteller wird in der Regel bereits eine große Anzahl von IP-Cores für die Verwendung in FPGA-Designs angeboten. Die marktführenden FPGA-Hersteller Xilinx und Altera bieten über Entwurfswerkzeuge die Möglichkeit, das gewünschte Hard-Core aus vorgegebenen IP-Core-Templates zu generieren und in das FPGA-Design einzubinden. Neben IP-Cores für die Realisierung von Basisfunktionen, wie einzelne Gatter, Zähler und Komperatoren, sind auch komplexe IP-Cores, z.B. für den Automobilbereich und für DSP-Anwendungen als Template verfügbar.

Abbildung 5.5 definiert die Beschreibungsformen eines IP-Cores so, wie sie innerhalb der vorliegenden Arbeit verwendet werden. Wie bereits in Abbildung 2.12 auf Seite 17 angedeutet, besitzt ein Hard-Core ein niedriges Abstraktionsniveau. Es ist bereits auf ein konkretes FPGA, oder zumindest auf eine FPGA-Familie eines bestimmten FPGA-Herstellers, festgelegt. Weiterhin kann ein Hard-Core nicht mehr über Parameter spezialisiert werden. Ein Hard-Core entspricht innerhalb des Hard-Core basierten Ansatzes einer Logikblock-Netzliste mit zusätzlichen Informationen, die die relative Platzierung der in der Netzliste enthaltenen Logikblöcke zueinander und ihrer Verbindungen untereinander beschreiben (Place&Route-Informationen). Ein Hard-Core kann dadurch als geschlossene Einheit auf dem FPGA platziert werden, d.h. es ist keine weitere Platzierung und Verdrahtung innerhalb des Hard-Cores notwendig.

Die für die Erstellung eines DT-Mappings erforderlichen DT-Versionen des ursprünglichen Hard-Cores werden entweder durch den IP-Core-Anbieter bereitgestellt oder durch ein Entwurfswerkzeug aus einem Template generiert. Dabei müssen die folgenden Punkte entschieden werden:

- Wie viele Defekte sollen auf der Fläche des Hard-Cores toleriert werden?
- Soll durch eine Partitionierung des Hard-Cores der Speicheraufwand reduziert und gleichzeitig die Anzahl der maximal tolerierbaren Defekte erhöht werden?

Abbildung 5.6 zeigt verschiedene Versionen für einen Hard-Core, der das Beispiel aus Abbildung 4.11 auf Seite 52 implementiert. Neben einer Version des Hard-Core, die einen Fehler tolerieren kann – nachfolgend bezeichnet als 1-DT-Hard-Core-Version -, wird auch eine mögliche Implementierung für eine 2-DT-Hard-Core-Version dargestellt. Allgemein wird eine Version die n Defekte auf der Fläche des Hard-Core toleriert als n-DT-Hard-Core-Version, im Folgenden oft abgekürzt als n-DT-Version, bezeichnet. In dem Beispiel aus Abbildung 5.6 sind mit der 1-DT-Version 4 Varianten verknüpft und mit der 2-DT-Version 10 Varianten. Da bei den Versionen in Abbildung 5.6 keine Partitionierung innerhalb des Hard-Cores erfolgte, handelt es sich hier in beiden Fällen

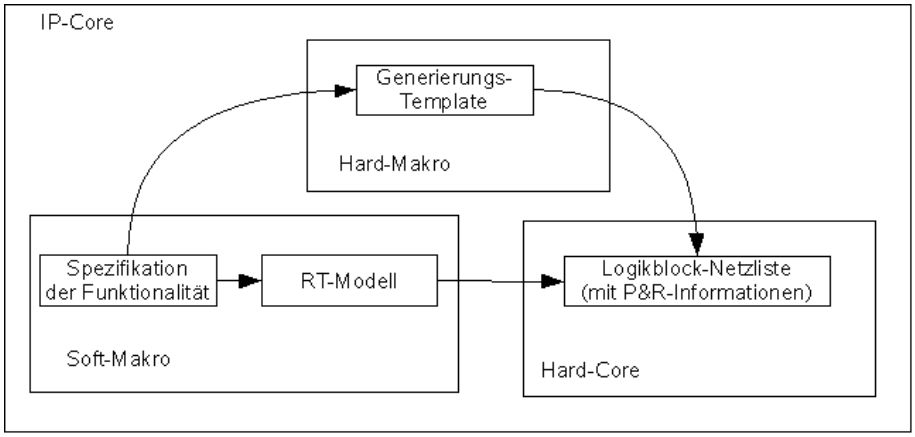



Abbildung 5.5: Die Beschreibungsformen eines IP-Core

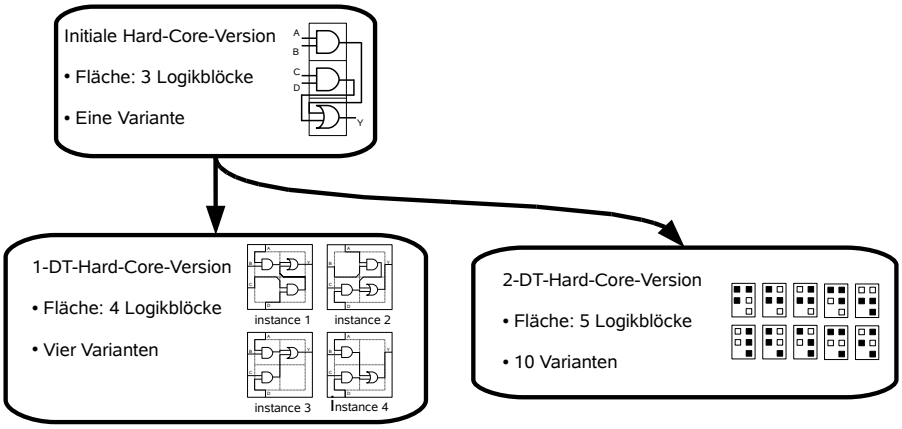



Abbildung 5.6: Ein Hard-Core mit defekttoleranten Versionen

um „atomare“ Versionen. Als „atomar“ werden im Folgenden Versionen bezeichnet, die nicht partitioniert wurden.

Abbildung 5.7 zeigt einige der hier und in den folgenden Abschnitten verwendeten Begriffe und deren Zusammenhang. Von dem initial vorhandenen Hard-Core werden durch den IP-Anbieter verschiedene Versionen erstellt. Zu einer Version gehören die für die Anzahl der tolerierbaren Defekte notwendigen Varianten. Partitionierte Versionen werden aus kleineren prekompilierten Schaltungsteilen zusammengesetzt. Da bei partitionierten Versionen die Anzahl der korrigierbaren Defekte von der Position dieser Defekte abhängt, wird die Bezeichnung n-DT-Version im Folgenden nur für atomare Versionen verwendet.

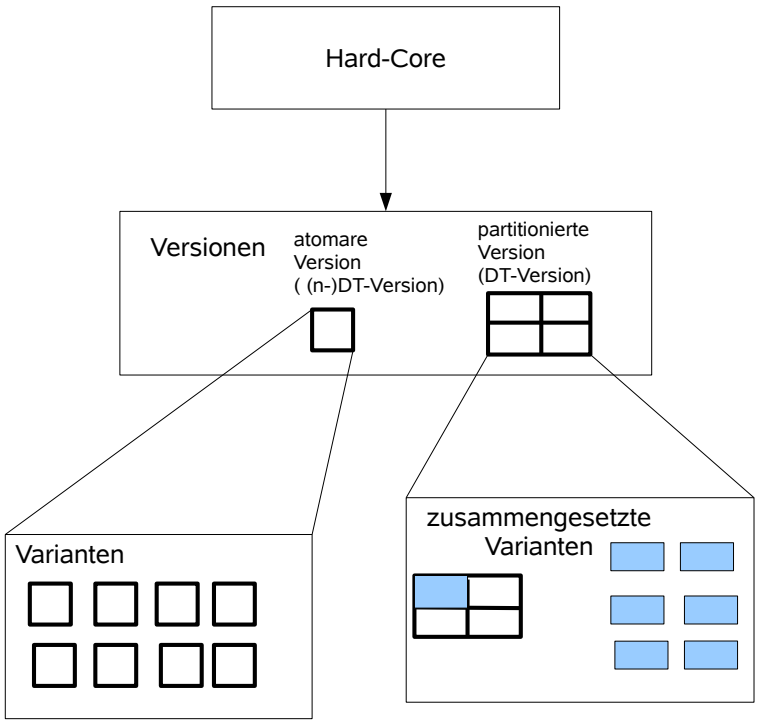



Abbildung 5.7: Zusammenhang der Begriffe „Hard-Core“ / „Versionen“ (atomar / partitioniert) / „Varianten“ / „zusammengesetzte Varianten“

### 5.2.1 Generierung der Varianten

Um den durch die Entwurfswerkzeuge automatisierten Place&Route-Vorgang zu steuern, können eine Reihe von Vorgaben für die Platzierung (*Placement-Constraints*) erfolgen. In der Regel sind unter Anderem folgende Vorgaben möglich (siehe z.B. auch [Xil07a]):

- Logikblöcke können einer vorgegebene Position auf dem FPGA fest zugeordnet werden.
- Die relative Platzierung von Logikblöcken zueinander kann festgelegt werden. Werden z.B. sechs Logikblöcke übereinander in einer Spalte platziert, kann festgelegt werden, dass diese Form beibehalten werden soll, egal, an welcher Stelle auf dem FPGA die Platzierung erfolgt.

Weiterhin kann für die Platzierung eine Begrenzung der Fläche erfolgen, auf der ein vorgegebener Schaltungsteil platziert werden soll (*Area-Constraints*).

Da alle Varianten einer Version auf derselben Hard-Core-Netzliste basieren, kann in einem Place&Route-Vorgang durch die zusätzliche Vorgabe von gesperrten Logikblöcken

gezielt eine Variante erzeugt werden, in der die gesperrten Logikblöcke frei bleiben. Abbildung 5.8 zeigt die Erstellung einer Variante für die Version A eines Hard-Cores X.

**Beispiel:**

Im 1-DT-Hard-Core aus Abbildung 5.6 muss für die automatisierte Erstellung der benötigten Varianten jeweils ein Logikblock als „gesperrt“ angegeben werden. Für die 2-DT-Version bestehen die Sperrlisten jeweils aus zwei Einträgen.

Innerhalb eines Hard-Cores kann es Logikblöcke geben, die aufgrund der Struktur des FPGAs direkt nebeneinander oder untereinander platziert werden müssen, da eine schnelle Direktverbindung zwischen den Logikblöcken benötigt wird. Entsprechend dieser Einschränkungen in der Beweglichkeit kann es erforderlich sein, eine größere Fläche für eine Version eines Hard-Cores einzuplanen, als im vereinfachten Beispiel in Abbildung 5.6 dargestellt. Abbildung 5.9 (a) zeigt ein Hard-Core, bei dem die rund markierten Blöcke direkt untereinander jeweils über eine spezielle Direkt-Verbindung kommunizieren und daher relativ zueinander in der bestehenden Anordnung verbleiben müssen. Mit der in Abbildung 5.9 (b) gezeigten Form des Hard-Core, für die drei zusätzliche Logikblöcke benötigt werden, ist in diesem Fall die Erstellung von nur vier Varianten für eine 1-DT-Version ausreichend, da es so für jeden Logikblock eine Variante der Version gibt, in der dieser Logikblock nicht verwendet wird.

Im Folgenden soll ein Algorithmus in Pseudocode (Algorithmus 5.1) erläutert werden, der für die Erstellung einer n-DT-Version verwendet werden kann.

**Algorithmus 5.1** : Erstellung einer n-DT-Version

```
1 while !complete do
2     Lege die Form und die Größe (Anzahl der Logikblöcke = x) für die Version fest;
3     Erstelle Sperrliste für die (x n) Kombinationsmöglichkeiten der n zu sperrenden
      Logikblöcke;
4     for alle Sperrlisten do
5         if !(Logikblöcke in Sperrliste sind in einer bereits erstellten Variante ungenutzt)
          then
6             Erstelle die Variante (P & R);
7             if !success then
8                 break;
```

In Zeile zwei des Algorithmus werden Form und Größe des Hard-Cores festgelegt. Für

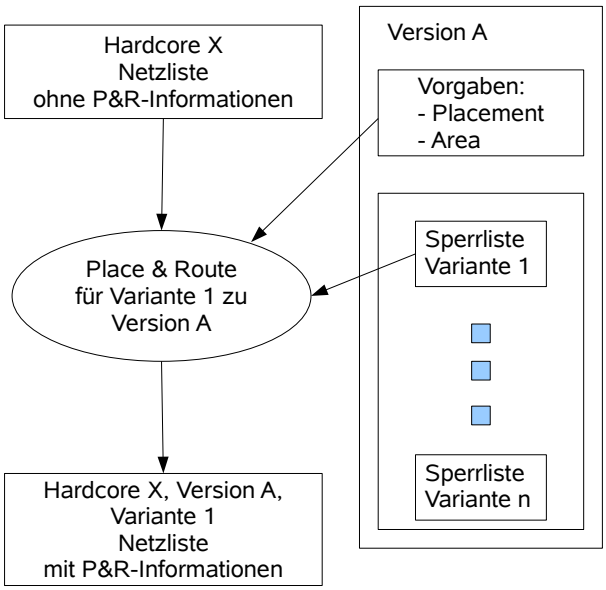



Abbildung 5.8: Die Erstellung einer Variante für eine Hard-Core-Version

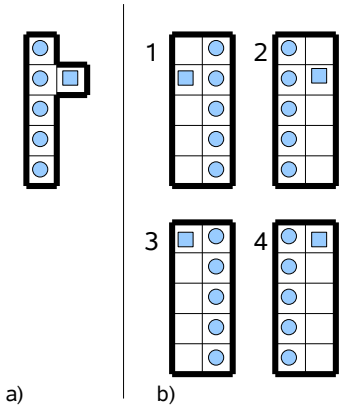



Abbildung 5.9: (a) Relativ zueinander festgelegte Logikblöcke (rund markiert) im Hard-Core; (b) Eine 1-DT-Version aus vier Varianten

alle möglichen Kombinationen der $n$ zu sperrenden Logikblöcke auf der Fläche des Hard-Cores der Größe x wird in Zeile drei jeweils eine Sperrliste erzeugt.

Die ab Zeile vier beginnende For-Schleife durchläuft für alle Sperrlisten den Place&Route-Vorgang. Wenn die in der Sperrliste angegebene Kombination von gesperrten Logikblöcken durch eine bereits vorhandene Variante abgedeckt wird, ist ein Place&Route für diese Kombination nicht erforderlich. In diesem Fall werden somit weniger Varianten als vorhandene Sperrlisten erzeugt. Nur dann wenn keine der bereits vorhandenen Varianten auf die in der Sperrliste angegebene Kombination passt, wird

der Place&Route-Vorgang durchgeführt.

Wenn der Place&Route-Vorgang für eine Variante nicht erfolgreich war, konnte die n-DT-Version nicht auf der in Zeile zwei vorgegebenen Fläche realisiert werden. In diesem Fall sorgt Zeile acht dafür, dass der Algorithmus in Zeile zwei mit einer vergrößerten Fläche fortgesetzt wird. Die Entscheidung über die Form und Größe für die Version wird manuell vorgenommen, der Rest des Algorithmus erfordert keinen manuellen Eingriff.

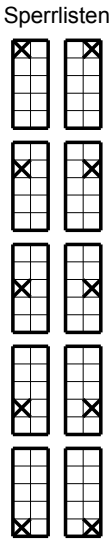
Für das Beispiel in Abbildung 5.9 kann für eine Fläche, die weniger als fünf Logikblöcke übereinander nicht zulässt, keine 1-DT-Version des Hard-Core erzeugt werden. Legt der Hersteller die in diesem Beispiel abgebildete Fläche mit der Größe $x = 10$ fest, werden $\begin{pmatrix} x \\ n \end{pmatrix} = 10$ Sperrlisten in Zeile 2 des Algorithmus erstellt. Nachdem die in Abbildung 5.9 gezeigten Varianten durch die oberen vier der in Abbildung 5.10 links gezeigten Sperrlisten erzeugt worden sind, ist für die weiteren Sperrlisten keine neue Variante erforderlich. Für jeden Logikblock auf der Fläche der Hard-Core-Version wird gespeichert, welche Variante(n) den defekten Logikblock nicht nutzt (nutzen). Diese Informationen stehen dann für die spätere Programmierung eines defekten FPGAs in der Serienproduktion zur Verfügung.

Auch für die Erstellung von Varianten, die zusammengesetzt eine Variante für eine partitionierte Version ergeben (vgl. Abbildung 5.7 auf Seite 65), kann Algorithmus 5.1 genutzt werden. Die vorhergehende Partitionierung der Version erfolgt manuell durch den Hersteller des IP-Core. Für jede Partition werden getrennt Sperrlisten erstellt. Für jeden Logikblock in einer Partition der Version wird gespeichert, welche Variante(n) der Partition den defekten Logikblock nicht nutzt (nutzen).

### 5.2.2 Zeitverhalten einer Hard-Core-Version

Neben den bisher kennengelernten und in Abbildung 5.6 auf Seite 64 dargestellten Merkmalen einer Version - der Fläche und der Anzahl der zu dieser Version gehörenden Varianten -, werden zusätzlich Delay-Informationen für jede erstellte Version gespeichert, um später eine Vorhersage des längsten Pfades in einem FPGA-Design zu ermöglichen, in dem die Hard-Core-Version verwendet wird.

Abbildung 5.11 (a) zeigt eine initial in eine Schaltung eingesetzte Hard-Core-Version ohne DT-Eigenschaften. In (b) ist eine 1-DT-Version und in (c) eine 2-DT-Version jeweils mit einem festen Interface, das durch den grauen Rahmen symbolisiert wird, skizziert. Die aus dem Hard-Core zu diesem Interface verlaufenden Pfade können entweder von einem Flipflop ausgehend nur einen Punkt am Interface durchlaufen (p1 und p2), zwei Punkte im Interface durchlaufen (p4), oder komplett innerhalb des Hard-Cores verlaufen (p3). Da unterschiedliche Platzierungen für die in Abbildung 5.11 (b) und (c) gezeigten Varianten erstellt werden, sind durch die Verwendung unterschiedlicher




Koordinaten

| | |
|---|---|
| X0Y0 | X1Y0 |
| X0Y1 | X1Y1 |
| X0Y2 | X1Y2 |
| X0Y3 | X1Y3 |
| X0Y4 | X4Y4 |

| Logikblock-Koordinate | Varianten, die den Logikblock nicht verwenden |
|---|---|
| X0Y0 | 1 |
| X1Y0 | 2 |
| X0Y1 | 3 |
| X1Y1 | 4 |
| X0Y2 | 1; 3 |
| X1Y2 | 2; 4 |
| X0Y3 | 1; 3 |
| X1Y3 | 2; 4 |
| X0Y4 | 1; 3 |
| X1Y4 | 2; 4 |

Abbildung 5.10: Die Sperrlisten zu dem Beispiel aus Abbildung 5.9 und die für jede Logikblock-Koordinate gespeicherten Varianten, die den Logikblock nicht verwenden

Verbindungs-Segmente auf der FPGA-Architektur auch unterschiedliche Verzögerungszeiten auf den skizzierten Pfaden möglich.

Für die Erstellung eines DT-Mappings wird für jede Hard-Core-Version die maximale Verzögerungszeit für jeden Pfad $p_i$ innerhalb einer Hard-Core-Version benötigt, um später eine Vorhersage des kritischen Pfades in der DT-Konfiguration zu ermöglichen. Daher werden zunächst die Verzögerungszeiten $delay_{p_i}(x_1), \ldots, delay_{p_i}(x_n)$ für einen Pfad $p_i$ in jeder der Varianten $x_1, \ldots, x_n$ ermittelt. Die maximale Verzögerungszeit (*Worst Case Delay Time*) für den Pfad $p_i$ ist dann:

$$WCDT_{p_i} := max(delay_{p_i}(x_1), \ldots, delay_{p_i}(x_n)) \tag{5.1}$$

Für alle Pfade, deren Anfangs- und Endpunkt innerhalb eines Hard-Core liegt (z.B. Pfad p3 in Abbildung 5.11), ist durch die Angabe des Pfades mit der höchsten WCDT die maximale Taktfrequenz, bis zu der diese Version in eine Schaltung eingesetzt werden kann, fest vorgegeben.

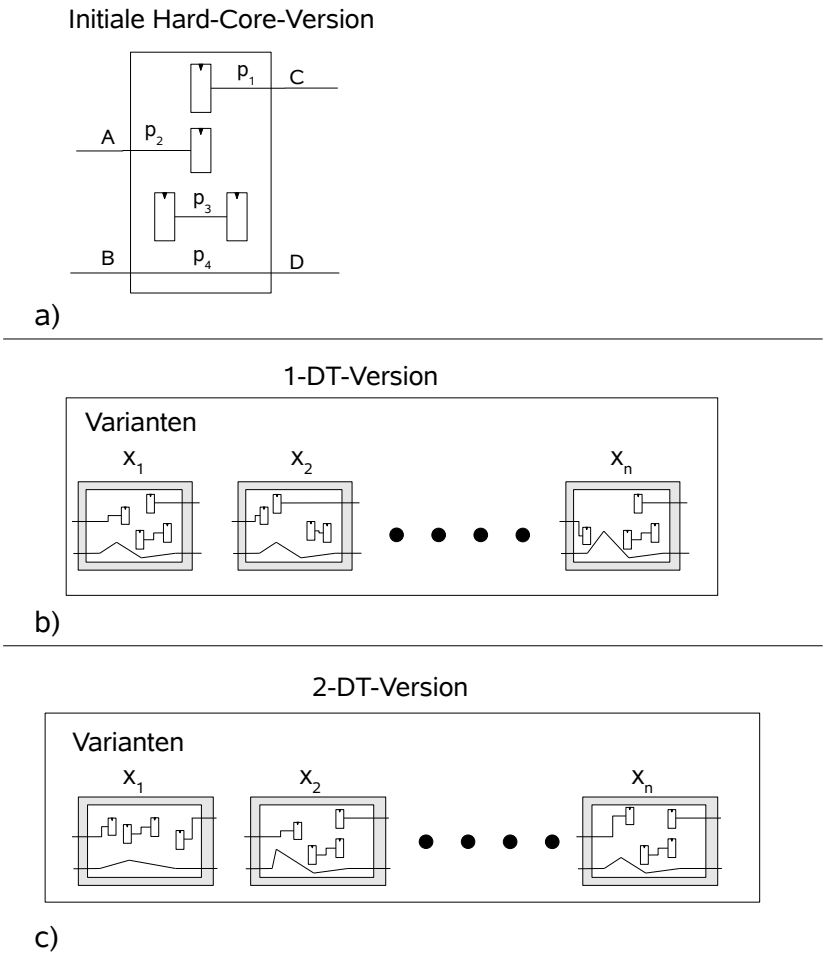



Abbildung 5.11: Mögliche Pfade in Hard-Core-Versionen

**Beispiel:**

Abbildung 5.13 zeigt alle Varianten der 1-DT-Version des Hard-Cores aus Abbildung 5.6 (Seite 64) auf der in Abbildung 2.5 (Seite 11) abstrakt dargestellten Architektur eines FPGAs. Für die Realisierung werden folgende Eigenschaften für diese Architektur festgelegt:

- In jedem Logikblock ist jeweils nur ein BLE vorhanden, welches ein LUT mit drei Eingängen und ein Flipflop enthält (siehe Abbildung 2.3 auf Seite 10).
- Die durch die *switch blocks* möglichen Verbindungen orientieren sich an der abstrakten Darstellung eines *switch blocks* in Abbildung 5.12. Zwischen allen Punkten, die identische Ziffern enthalten, ist eine Verbindung möglich, also z.B. zwischen den Punkten W1 und S1, aber nicht zwischen den Punkten W1 und S2.

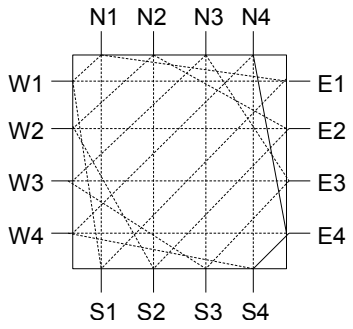



Abbildung 5.12: Die Verbindungsmöglichkeiten in einem *switch block*

Von den drei Eingängen des LUT werden für die Implementierung der Varianten aus Abbildung 5.6 jeweils nur zwei verwendet, um die benötigte Schaltfunktion zu realisieren. Die Flipflops werden hier nicht verwendet, da es sich bei dem Beispiel um eine rein kombinatorische Schaltung handelt.

Abbildung 5.13 zeigt alle Verbindungen, die innerhalb der Varianten zwischen den Logikblöcken bestehen[1].

Wenn wir die Verzögerungszeit auf den abgebildeten Verbindungen vergleichen, verlaufen beide Verbindungen in I und II über zwei *connection blocks*, einen *switch block* und zwei Verbindungssegemente. In III hingegen ist für eine der Verbindungen und in IV sogar für beide Verbindungen ein aufwendigerer Pfad über drei Verbindungssegmente und zwei *switch blocks* zu durchlaufen.

Zu den Verbindungen innerhalb der Varianten der Hard-Core-Version kommen noch Verbindungsleitungen für die Bereitstellung eines festen Interfaces hinzu. Das Interface wird in Abbildung 5.14 über zusätzliche Logikblöcke erstellt, die ihre Position nicht verändern und somit einen festen Anschluss an die Umgebung der Hard-Core-Version ermöglichen.

Die WCDTs der in Abbildung 5.14 gezeigten Pfade werden zusammen mit den Informationen über die Fläche & Form der Version gespeichert und stehen für die Ermittlung des kritischen Pfades in der Timing-Analyse für eine DT-Konfiguration, die diese Version verwendet, zur Verfügung.

Bei partitionierten Versionen wird für jeden Pfad die WCDT von allen an diesem Pfad beteiligten Partitionen summiert und die so errechnete WCDT für den Pfad zusammen mit der Version gespeichert.

[1]Da die vereinfachte Darstellung nicht alle Verbindungsmöglichkeiten eines *connection block* darstellt, erlauben wir für dieses Beispiel den Wechsel des Ausgangs auf die rechte oder auf die obere Seite des BLE, um eine Verbindung zwischen den BLEs zu ermöglichen. Weiterhin sind in dieser vereinfachten Darstellung nur kurze Verbindungssegmente enthalten. In einer realen FPGA-Architektur stehen in der Regel verschiedene Verbindungslängen zur Verfügung (vgl. Abbildung 2.5 und Abbildung 2.6).

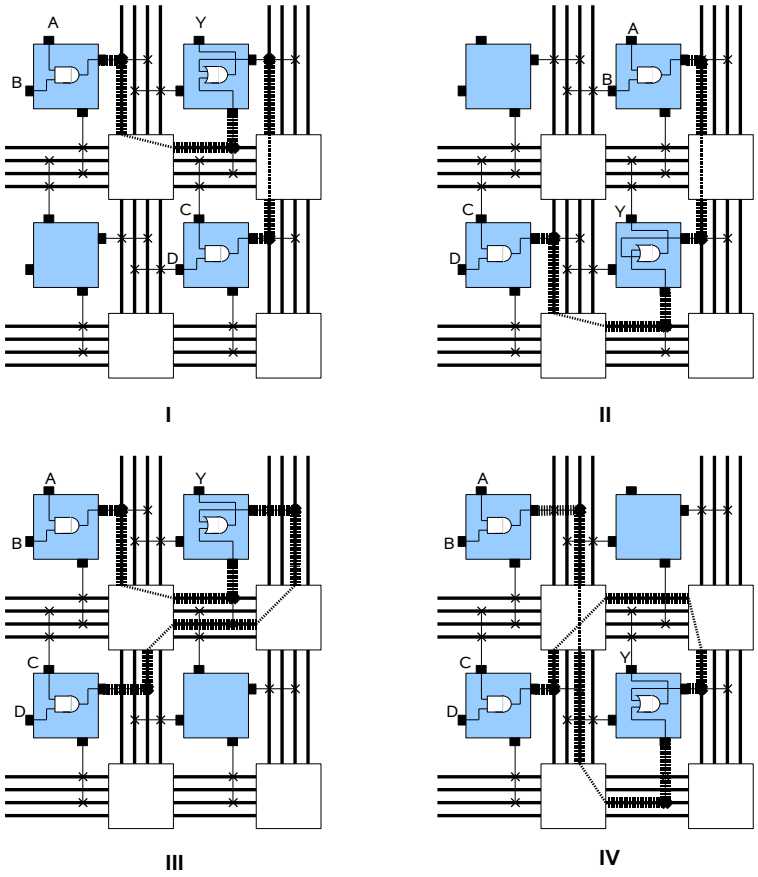



Abbildung 5.13: Beispiel für verschiedene Varianten einer Hard-Core-Version

### 5.2.3 Funktionswahrscheinlichkeit einer Hard-Core-Version

In diesem Abschnitt wird die Berechnung der Funktionswahrscheinlichkeit einer Hard-Core-Version innerhalb einer DT-Konfiguration erläutert.

Wenn die durchschnittliche Anzahl von *random defects*, die einen Logikblock treffen, als $\lambda_{LB}$ bekannt ist, kann daraus über das Poisson-Model

$$Y_{LB} = e^{-\lambda} \tag{5.2}$$

auf die Wahrscheinlichkeit für einen defektfreien Logikblock $Y_{LB}$geschlossen werden.

Wenn wie in Abschnitt 2.3 die durchschnittliche Anzahl der Defekte $d_i$ für jeden Defekt-Typ $i$ pro Flächeneinheit als bekannt vorausgesetzt wird, kann, nach einer Ermittlung der kritischen Fläche $A_i^{(c)LB}(x)$ innerhalb eines Logikblocks für einen Fehler vom Typ $i$ mit dem Durchmesser $x$ , über

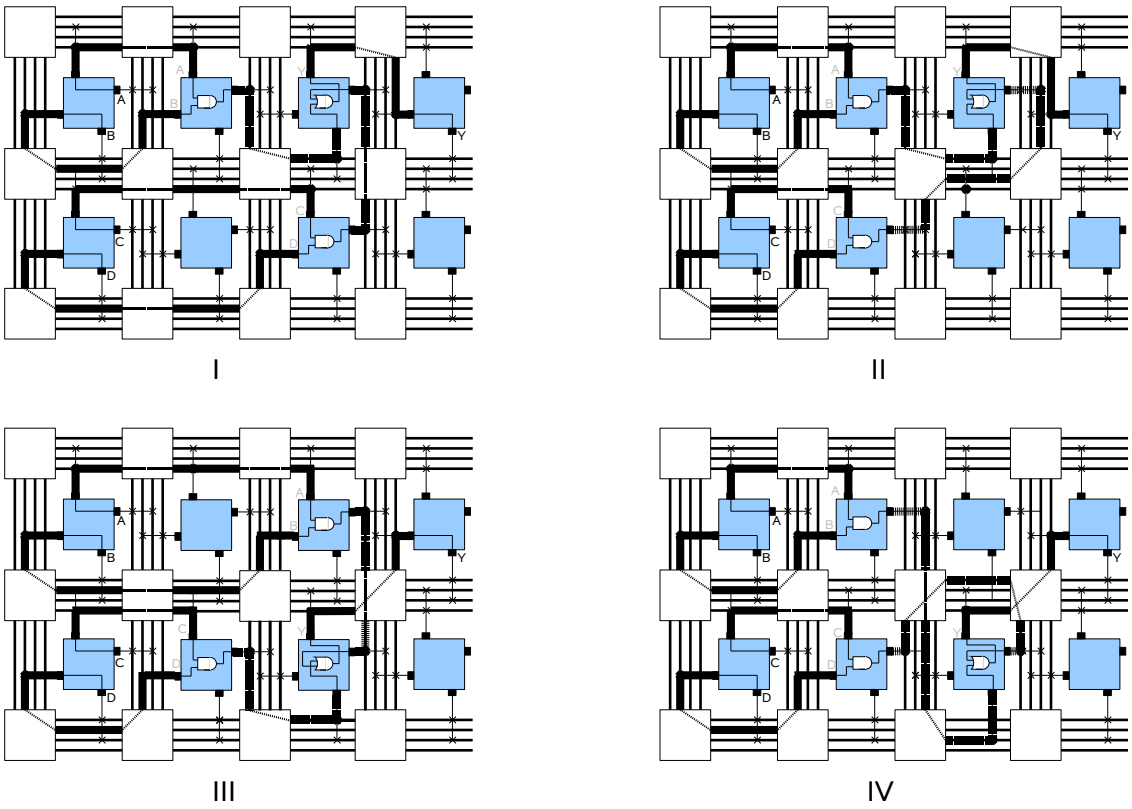



Abbildung 5.14: Varianten einer Hard-Core-Version mit festem Interface

$$\Theta_i(x) = \frac{A_i^{(c)LB}(x)}{A_{LB}} \tag{5.3}$$

mit der Gesamtfläche $A_{LB}$ des Logikblocks und

$$\lambda_{LB} = \sum_i \Theta_i A_{LB} d_i \tag{5.4}$$

die in Formel 5.2 benötigte Fehlerdichte für einen Logikblock, $\lambda_{LB}$, ermittelt werden. Die Berechnung der gewichtet gemittelten Wahrscheinlichkeit $\Theta_i$ für einen Fehler vom Typ $i$ erfolgt dazu wie in Abschnitt 2.3 erläutert.

Da von den FPGA-Herstellern keine Informationen herausgegeben werden, aus denen Rückschlüsse auf den Yield in der Herstellung der FPGAs möglich sind [Tri08], konnte im Rahmen dieser Arbeit die Berechnung von $Y_{LB}$ nicht durchgeführt werden.

Wenn $Y_{LB}$ bekannt ist, lässt sich durch die im Folgenden dargestellte Binomialverteilung die Wahrscheinlichkeit dafür berechnen, dass genau $k$ von $N$ Logikblöcken defektfrei sind:

$$f(k,N) = \left( \begin{array}{c} N \\ k \end{array} \right) (Y_{LB})^k (1 - Y_{LB})^{N-k} \tag{5.5}$$

Nun sei $N$ die Anzahl der für die Instanz $i$ einer Hard-Core-Version auf dem FPGA reservierten Logikblöcke. Da eine n-DT-Hard-Core-Version $n$ freie Logikblöcke enthält, werden nur $M = N - n$ nicht defekte Logikblöcke für die Realisierung der Instanz benötigt. Die Wahrscheinlichkeit dafür, dass in der Instanz $i$ nur Defekte liegen, die toleriert werden können, im Folgenden bezeichnet mit $p(i)$, errechnet sich dann durch Aufsummierung der Wahrscheinlichkeiten für alle Binomialverteilungen $f(k,N)$ unter denen die Instanz $i$ verwendbar ist, d.h.

$$p(i) := \sum_{k=M}^{N} f(k,N) = \sum_{k=M}^{N} \left( \begin{array}{c} N \\ k \end{array} \right) (Y_{LB})^k (1 - Y_{LB})^{N-k} \tag{5.6}$$

Diese Gleichung wird in Abbildung 5.15 für die Instanz einer 2-DT-Version verdeutlicht. Es gibt für die abgebildeten 2-DT-Version mit einer Fläche von 4 Logikblöcken vier verschiedene Möglichkeiten dafür, dass $M = 4 - 2 = 2$ Logikblöcke nicht defekt sind. Da $Y_{LB}$ der Wahrscheinlichkeit für einen nicht defekten und $1 - Y_{LB}$ der Wahrscheinlichkeit für einen defekten Logikblock entspricht, ist die Wahrscheinlichkeit für eine dieser vier Möglichkeiten jeweils $(Y_{LB})^2 * (1 - Y_{LB})^2$. Die Wahrscheinlichkeit dafür, dass genau 2 von 4 Logikblöcken nicht defekt sind, entspricht der Summe der Wahrscheinlichkeit für alle diese Möglichkeiten. Entsprechend Formel 5.5 gilt dann $4 * (Y_{LB})^2 * (1 - Y_{LB})^2 = \left( \begin{array}{c} 4 \\ 2 \end{array} \right) (Y_{LB})^2 (1 - Y_{LB})^2 = f(2,4)$ für den ersten Summanden in Abbildung 5.15. Nachdem auch die übrigen Summanden (es gibt vier Möglichkeiten für drei nicht defekte Logikblöcke und eine Möglichkeit dafür, das kein Logikblock defekt ist) ermittelt wurden, erhält man $p(i)$ durch die Aufsummierung der Wahrscheinlichkeiten für alle Möglichkeiten insgesamt.

Für partitionierte Versionen kann die Berechnung der Wahrscheinlichkeit $p(i)$ erfolgen, indem diese Wahrscheinlichkeit für jede Partition getrennt berechnet wird und anschließend eine Multiplikation der Ergebnisse erfolgt. Wenn $n$ die Anzahl der Partitionen einer partitionierten Version $i$ ist und $p(i,x)$ die Wahrscheinlichkeit dafür, dass in der Partition $x$ nur Defekte auftreten, die durch diese Version toleriert werden können, dann gilt $p(i) = \prod_{x=1}^{n} p(i,x)$.

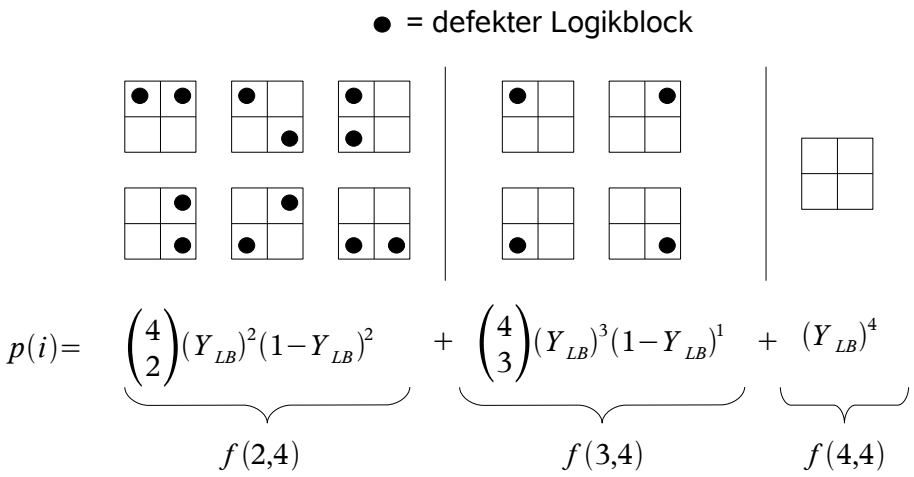



Abbildung 5.15: Ermittlung der Wahrscheinlichkeit $p(i)$ für die Instanz $i$ einer 2-DT-Version mit 4 Logikblöcken

## 5.3 Durchführung des Re-Mapping

Die in einer Netzliste enthaltenen Instanzen von Hard-Cores werden vor dem Re-Mapping auf nicht defekttolerante Hard-Core-Versionen abgebildet. Die Veränderung dieser Abbildung durch das Re-Mapping soll im Folgenden mit dem in Abbildung 5.16 skizzierten Modell verdeutlicht werden. Wir fassen innerhalb dieses Modells die in der betrachteten Netzliste enthaltenen Instanzen von Hard-Cores in der Menge $HC$ zusammen.

**Definition 5.2.** $HC$ bezeichnet die Menge der in einer Netzliste enthaltenen Instanzen von Hard-Core-Versionen.

Weiterhin betrachten wir das Mapping der Instanzen von Hard-Cores (im Folgenden kurz als Instanzen bezeichnet) auf Hard-Core-Versionen als einen veränderbaren Zustand. Für jede Instanz stehen verschiedene Hard-Core-Versionen zur Verfügung. Jedes mögliche Mapping einer Instanz $hc \in HC$ auf eine dieser Hard-Core-Versionen soll im Folgenden als Zustand $state(hc)$ dieser Instanz modelliert werden.

**Definition 5.3.** $state(hc)$ bezeichnet den "Zustand" und somit die Hard-Core-Version, die aktuell für das Mapping der Instanz $hc \in HC$ verwendet wird.

Die verschiedenen Hard-Core-Versionen, auf die das Mapping einer Instanz $hc \in HC$ aus der Netzliste erfolgen kann, werden ebenfalls zu einer Menge, bezeichnet mit $states(hc)$, zusammengefasst.

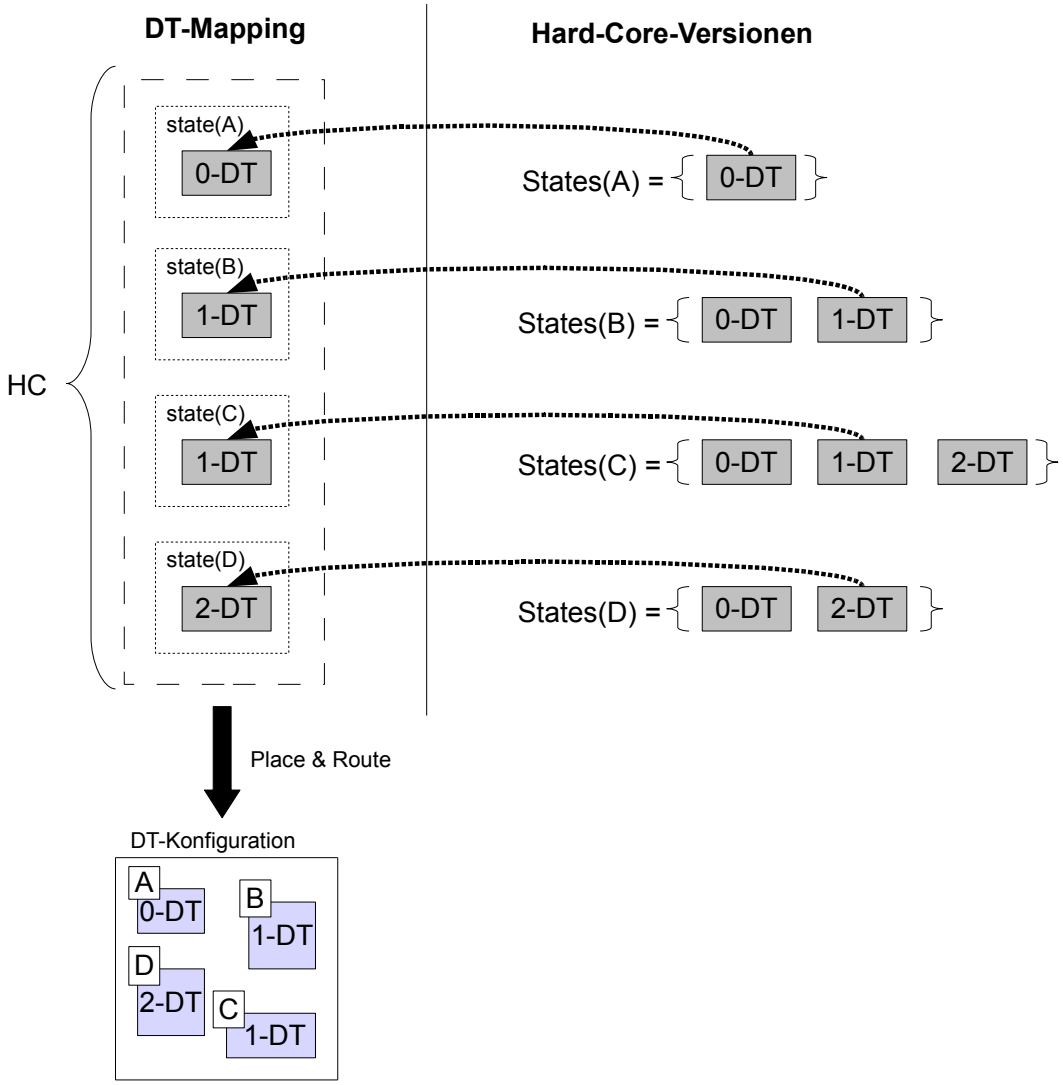



Abbildung 5.16: Das Modell für den Re-Mapping-Vorgang

**Definition 5.4.** *states*(*hc*) bezeichnet die Menge der Hard-Core-Versionen, die für das Mapping der Instanz $hc \in HC$ zur Verfügung stehen.

Für die zur Verfügung stehende Menge an Hard-Core-Versionen sind in Abbildung 5.16 auf der rechten Seite Beispielmengen für jede Instanz angegeben. Gezeigt wird hier der Zustand des Modells nach dem Re-Mapping, also nachdem das ursprüngliche Mapping so verändert wurde, dass einige der Instanzen in der Netzliste auf defekttolerante Hard-Core-Versionen abgebildet werden. Nach der Ausführung des Re-Mappings wird der Place&Route-Vorgang erneut durchgeführt (vgl. Abbildung 5.2 auf Seite 59).

### 5.3.1 Funktionswahrscheinlichkeit einer DT-Konfiguration

Um die Berechnung Funktionswahrscheinlichkeit für ein DT-Konfiguration zu beschreiben, definieren wir zunächst einen Platzhalter für die Funktionswahrscheinlichkeit einer Instanz $hc \in HC$ in dem neu eingeführten Modell.

**Definition 5.5.** $p(hc \in HC, x \in states(HC))$ gibt die Funktionswahrscheinlichkeit der Instanz $hc \in HC$ an, wenn für diese Instanz die Hard-Core-Version $x \in states(HC)$ eingesetzt wird.

Wenn $N(hc, x)$ die Anzahl der für die Instanz $hc$ bei einem Mapping dieser Instanz auf die mit $x$ bezeichnete Hard-Core-Version und $n(hc, x)$ die Anzahl der in jeder Variante dieser Hard-Core-Version freien Logikblöcke angibt, werden nur $N(hc, x) - n(hc, x)$ nicht defekte Logikblöcke für die Realisierung der Instanz $hc$ im Zustand $x$ benötigt und es gilt mit Formel 5.6 auf Seite 74

$$p(hc, x) := \sum_{k=N(hc,x)-n(hc,x)}^{N(hc,x)} \binom{N(hc, x)}{k} (Y_{LB})^k (1 - Y_{LB})^{N(hc,x)-k} \tag{5.7}$$

Da die Funktionswahrscheinlichkeit einer Instanz in einer DT-Konfiguration unabhängig von den Funktionswahrscheinlichkeiten der übrigen Instanzen ist, kann die Funktionswahrscheinlichkeit einer DT-Konfiguration als Produkt der Funktionswahrscheinlichkeiten der in ihr platzierten Instanzen von Hard-Core-Versionen angegeben werden.

**Definition 5.6.** $p(HC)$ bezeichnet die Wahrscheinlichkeit dafür, dass in allen Instanzen $hc \in HC$ nur Defekte liegen, die von den Varianten der eingesetzten Hard-Core-Versionen ausgeglichen werden können
Es gilt $p(HC) := \prod_{hc \in HC} p(hc, state(hc))$

Abbildung 5.17 zeigt ein DT-Mapping von einer Netzliste, die aus vier Instanzen von Hard-Cores $HC = \{A, B, C, D\}$ besteht und die Berechnung der Funktionswahrscheinlichkeit für dieses Mapping. Die Flächen $N(hc, state(hc))$ sind - ebenso wie die Anzahl der tolerierbaren Defekte $n(hc, state(hc))$ innerhalb dieser Hard-Core-Versionen -, bereits vor dem Place&Route-Vorgang bekannt. Da die Funktionswahrscheinlichkeit einer DT-Konfiguration unabhängig von der Platzierung der Hard-Core-Versionen auf dem FPGA ist, gilt die Funktionswahrscheinlichkeit für jede mögliche DT-Konfiguration, die aus dem DT-Mapping durch einen Place&Route-Vorgang erzeugt wird..

### 5.3.2 Zeitverhalten einer DT-Konfiguration

Das veränderte Zeitverhalten einer DT-Konfiguration gegenüber der ursprünglichen Konfiguration soll im Folgenden an dem zu Beginn dieses Kapitels dargestellten Modell

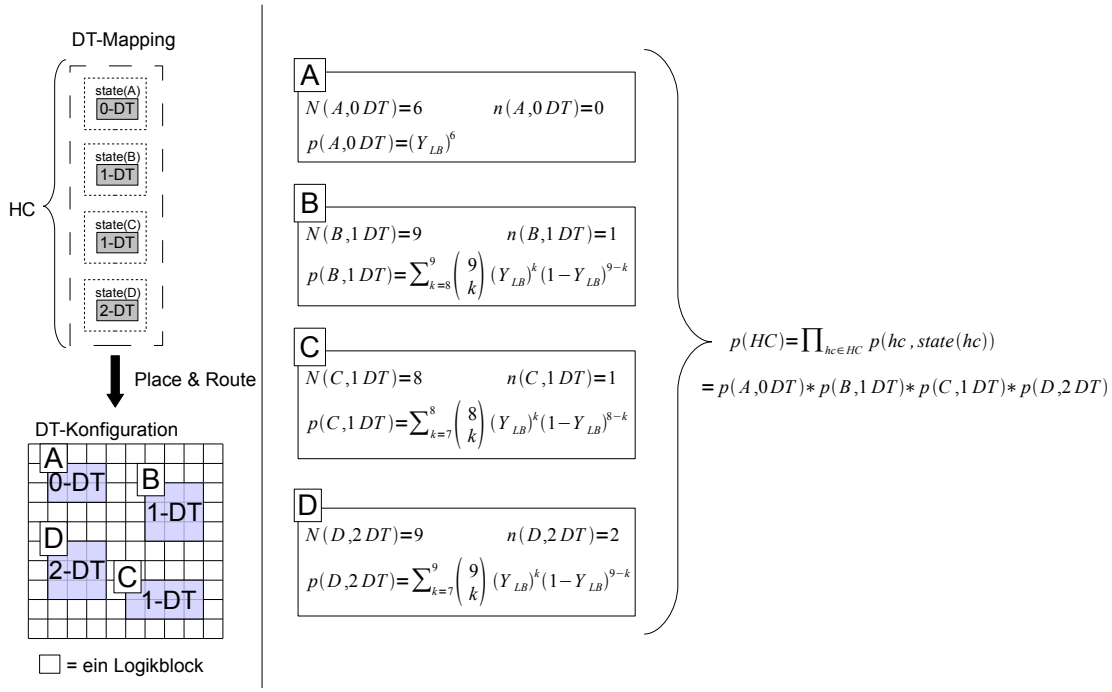



Abbildung 5.17: Beispiel für die Berechnung der Funktionswahrscheinlichkeit eines DT-Mappings

untersucht werden. In diesem Abschnitt wird das Modell erweitert und ein konkretes Beispiel aufgebaut, das dann weiterhin innerhalb des Kapitels als „laufendes Beispiel“ verwendet wird. Anhand des laufenden Beispiels wird zunächst das veränderte Zeitverhalten einer DT-Konfiguration untersucht.

Es sei $G = (V, E)$ ein gerichteter Graph, der die in einer Netzliste enthaltenen kombinatorischen Schaltnetze darstellt. Jeder Knoten $v \in V$ in diesem Graphen steht für einen Pfad durch eine Komponente der Netzliste. Die Komponenten der Netzliste repräsentieren Logikblöcke, IP-Cores und bei Bedarf auch festverdrahtete FPGA-Elemente, wie z.B. Multiplizierer oder Block-RAMs. Jede gerichtete Kante $(u, v) \in E$ steht für eine Verbindung von einem Ausgang der Komponente $u \in V$ zu einem Eingang der Komponente $v \in V$.

Abbildung 5.18 zeigt eine Netzliste und die Darstellung der in ihr enthaltenen kombinatorischen Schaltnetze als gerichteten Graphen $G = (V, E)$. Die Netzliste besteht zum einen aus Instanzen von Hard-Cores ($A$ bis $E$), die durch verschiedene Versionen auf dem FPGA realisiert werden können und zum anderen aus Instanzen von statischen, nicht austauschbaren Komponenten ($S_1$ bis $S_3$).

**Definition 5.7.** *nodes*$(hc \in HC)$ bezeichnet die Menge der Knoten, die zu der Instanz $hc \in HC$ gehören.

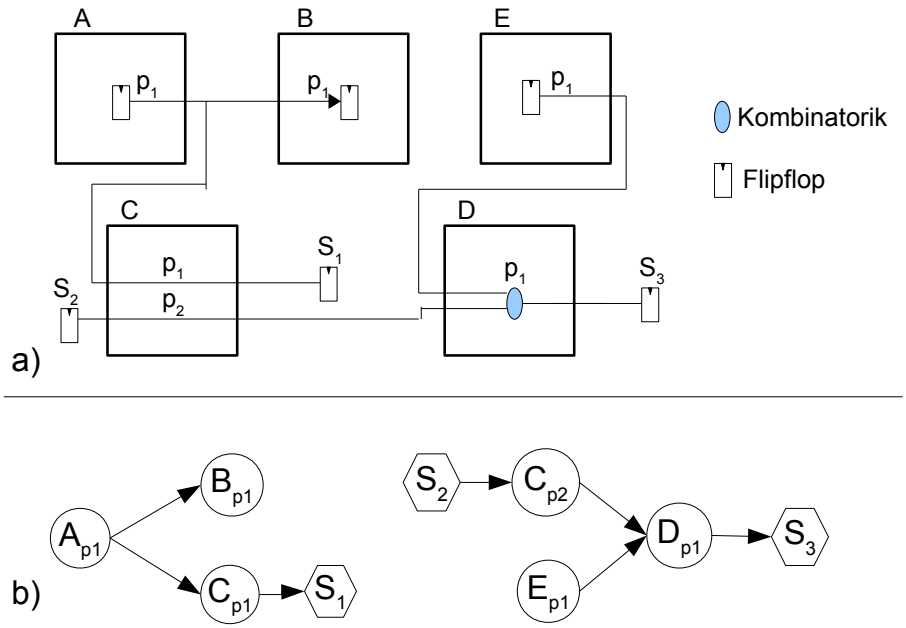



Abbildung 5.18: (a) Ein aus Hard-Cores und statischen Teilen bestehender Ausschnitt einer Schaltung; (b) Die Darstellung der darin enthaltenen Schaltnetze als gerichteter Graph

Für das laufende Beispiel gilt nach Definition 5.7 für die in der abgebildeten Netzliste enthaltenen Hard-Core-Versionen $HC = \{A, B, C, D, E\}$ :

- $nodes(A) = \{A_{p1}\}$
- $nodes(B) = \{B_{p1}\}$
- $nodes(C) = \{C_{p1}, C_{p2}\}$
- $nodes(D) = \{D_{p1}\}$
- $nodes(E) = \{E_{p1}\}$

Alle Knoten, die zu einem Hard-Core gehören, also die in Abbildung 5.18 als Kreis dargestellten Knoten, gehören zu der Menge der dynamischen Knoten $V_{dyn} \subset V$.

**Definition 5.8.** $hc_v(v)$ gibt für jeden Knoten $v \in V_{dyn}$ den Hard-Core an, der diesen Knoten enthält, d.h. $hc_v(v) := \{hc \in HC | v \in nodes(hc)\}$

Für jede Instanz eines Hard-Cores in der Netzliste $HC$ sollen im laufenden Beispiel zwei verschiedene Hard-Core-Versionen zur Verfügung stehen. Nach Definition 5.4 (Seite 76) beschreiben wir dies mit:

$states(A) = states(B) = states(C) = states(D) = states(E) = \{0DT, 1DT\}$

Der Zustand „0DT“ steht für die vorhandene 0-DT-Version des Hard-Cores und der Zustand „1DT“ für die 1-DT-Version des Hard-Cores, d.h. im laufenden Beispiel ist für alle verwendeten Hard-Cores neben der 0-DT-Version auch eine 1-DT-Version vorhanden.

Vor dem Re-Mapping erfolgt das Mapping ausschließlich auf 0-DT-Versionen. Nach Definition 5.3 auf Seite 75 gilt also

$state(A) = state(B) = state(C) = state(D) = state(E) = 0DT.$

Die Auswirkung des veränderten Zeitverhaltens der verwendeten Hard-Core-Versionen auf das Zeitverhalten nach einem Re-Mapping kann durch die Annotation der Verzögerungszeiten an dem gerichteten Graphen aus Abbildung 5.18 verdeutlicht werden. Für jeden Knoten bezeichnet im Folgenden $d(v)$ die Verzögerungszeit eines Signals von dem Eingang des Knotens bis zu den Eingängen der direkten Nachfolger des Knotens. Der Wert für $d(v)$ ergibt sich aus der Summe der Verzögerungszeit des durch den Knoten repräsentierten Pfades innerhalb einer Komponente $d_{self}(v)$ und der Verzögerungszeit $d_{out}(v)$, die benötigt wird, um das Verbindungsnetz am Ausgang des Knotens anzusteuern. Es gilt

$$d(v) = d_{self}(v) + d_{out}(v) \tag{5.8}$$

Der mit $d_{self}(v)$ bezeichnete Anteil an der Verzögerungszeit ist für jeden Knoten $v \in V$ bereits vor dem Place&Route-Vorgang durch die Angaben des Herstellers für die Logikblöcke und die festverdrahteten Komponenten und durch die Angaben der IP-Core-Hersteller für die Pfade durch die Hard-Cores bekannt. Die Verzögerungszeiten der in der Schaltung verwendeten Verbindungsnetze zwischen den Komponenten der Netzliste können erst nach dem Place&Route-Vorgang der FPGA-Konfiguration entnommen werden, d.h. $d_{out}(v)$ ist für alle Knoten $v \in V$ erst nach der Fertigstellung der FPGA-Konfiguration bekannt.

Abbildung 5.19 verdeutlicht diesen Zusammenhang für das laufende Beispiel. An dem gerichteten Graphen in 5.19 (b) wurde $d(v)$ für jeden Knoten $v \in V$ annotiert. Wir nehmen zur Vereinfachung des Beispiels für jeden Knoten $v \in V$ die Verzögerungszeit $d_{self}(v) = 1$ an. Nach dem Place&Route-Vorgang werden die Verzögerungszeiten der Verbindungsnetze $d_{out}(v)$ für alle $v \in V$ mit einer statischen Timing-Analyse ermittelt. Für das laufende Beispiel nehmen wir an, dass jedes der in Abbildung 5.19 (b) gezeigten Verbindungsnetze ebenfalls eine Verzögerungszeit mit dem Delay-Wert 1 hat. Für jeden Knoten mit einem Verbindungsnetz am Ausgang des Knotens setzten wir daher $d_{out}(v) = 1$. Für Knoten ohne Verbindungsnetz an ihrem Ausgang setzen wir $d_{out}(v) = 0$. Da $d_{self}(D_{p1a}) = d_{self}(D_{p1b})$ gilt, können wir in Abbildung 5.19 die in Abbildung 5.20 skizzierte Vereinfachung für den Graphen vornehmen.

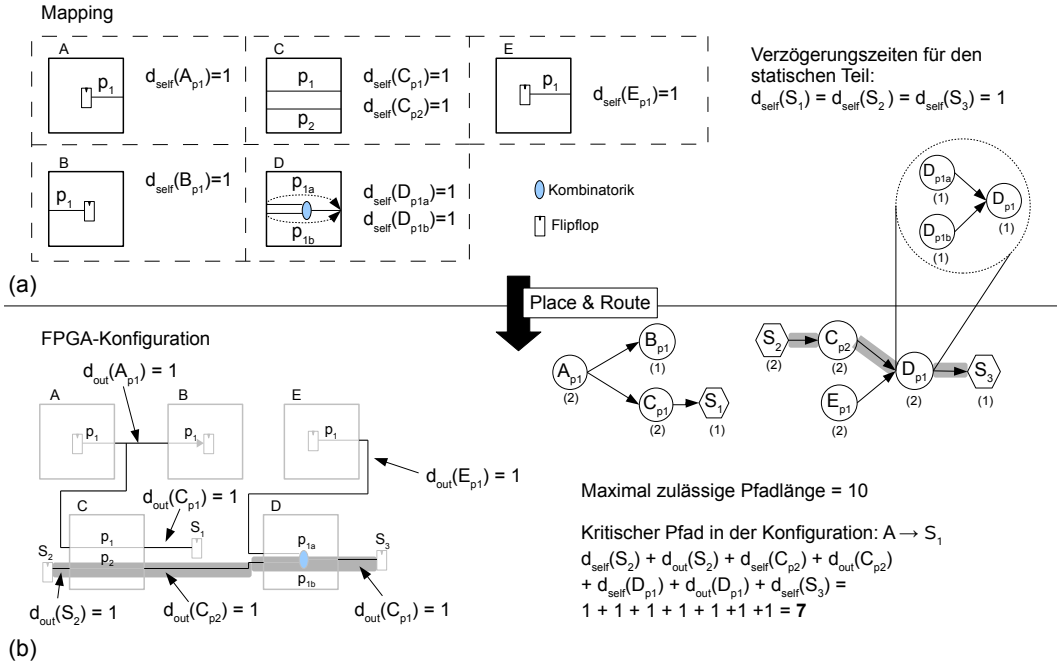



Abbildung 5.19: (a) Die Verzögerungszeiten der Komponenten einer Netzliste sind bereits nach dem Mapping bekannt; (b) Die Verzögerungszeiten der Verbindungsnetze - und somit auch der kritische Pfad - können hingegen erst nach dem Place&Route-Vorgang aus der FPGA-Konfiguration ermittelt werden.

Wir setzten für das laufende Beispiel die maximal zulässige Pfadlänge auf den Wert 10. Der durch Addition entlang der Pfade bestimmte kritische Pfad (in Abbildung 5.19 markiert) liegt mit dem Wert 7 innerhalb der maximal zulässigen Pfadlänge.

Das Re-Mapping der Instanzen hat zur Folge, dass die (mit Hilfe der Formel 5.1 auf Seite 69 berechnete) maximale Verzögerungszeit $d_{self}(v) = WCDT_v$ für den durch einen Knoten $v \in V_{dyn}$ repräsentierten Pfad anstelle der bisherigen Verzögerungszeit durch diesen Pfad eingeplant wird. Wir bezeichnen das dem Knoten zugeordnete Delay $d_{self}(v)$ im Zustand $x$ jetzt als $d(v, x)$ und geben die Verzögerungszeiten der Knoten $v \in V_{dyn}$ für das laufende Beispiel in Tabelle 5.1 an. Um auch weiterhin die in Abbildung 5.20 skizzierte Vereinfachung des Graphen vornehmen zu können, setzten wir $d(D_{p1a}, 0) = d(D_{p1b}, 0) = d(D_{p1}, 0)$ und $d(D_{p1a}, 1) = d(D_{p1b}, 1) = d(D_{p1}, 1)$.

Wir gehen nun, wie in Abbildung 5.21 skizziert, zunächst davon aus, dass für alle in der Netzliste enthaltenen Instanzen von Hard-Core-Versionen $hc \in HC$ ein Re-Mapping auf n-DT-Hard-Core-Versionen mit dem höchsten verfügbaren Wert für $n$ (im laufenden Beispiel sind das die 1-DT-Hard-Core-Versionen) vorgenommen wird. Weiterhin gehen wir davon aus, dass sich $d_{self}(v)$, wie in Abbildung 5.21 gezeigt, für jeden dynamischen

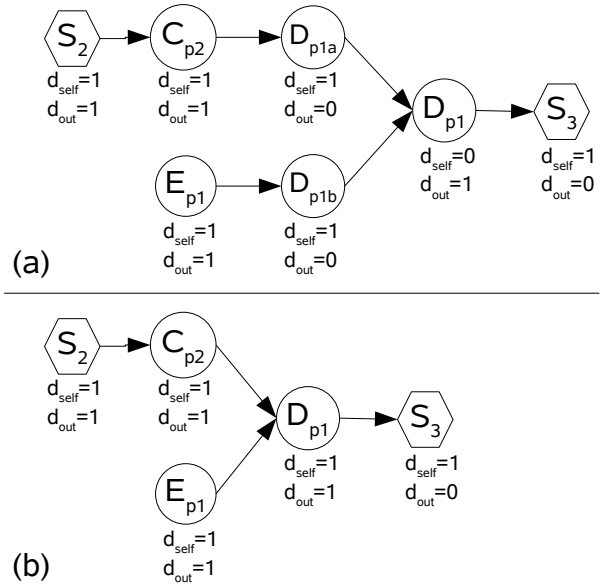



Abbildung 5.20: Die Darstellung des Graphen aus Abbildung 5.18 mit (a) zwei zusätzlichen Knoten und (b) einer vereinfachten Darstellung, wenn $d_{self}(D_{p1a}) = d_{self}(D_{p1b})$ gilt.

Knoten $v \in V_{dyn}$ durch dieses Re-Mapping entsprechend der in Tabelle 5.1 angegebenen Werte ändert. Unter diesen Voraussetzungen kann die maximal zulässige Pfadlänge nach dem Re-Mapping nicht mehr eingehalten werden, da bereits ohne die Verzögerungszeiten für die Verbindungsnetze für den Pfad $A \rightarrow S_1$ eine Verzögerungszeit von 10 entstanden ist. Im Place&Route-Vorgang kann daher das aus Abbildung 5.19 übernommene Timing-Constraint „Maximal zulässige Pfadlänge = 10“ nicht mehr erreicht werden.

Das Problem der Einhaltung aller vorgegeben Timing-Constraints wird allgemein als Timing-Closure-Problem bezeichnet. Wenn mit dem Place&Route-Vorgang die Erstellung der FPGA-Konfiguration unter Einhaltung aller vorgegebenen Timing-Constraints abgeschlossen wird, ist die Timing-Closure für ein FPGA-Design erreicht.

Im obigen Beispiel konnte die Timing-Closure durch den Place&Route-Vorgang nicht erreicht werden, da das Re-Mapping ohne „Verzögerungszeit-Reserve“ für die Verbindungsnetze zwischen den Komponenten der Netzliste durchgeführt wurde. Als Vorbereitung auf die Durchführung des Re-Mappings werden daher nun zunächst diejenigen Verzögerungszeiten, die für diese Verbindungsnetze vor dem Re-Mapping in der FPGA-Konfiguration benötigt wurden, durch eine statischen Timing-Analyse ermittelt. Der in dieser Arbeit entwickelte Re-Mapping-Algorithmus garantiert, dass nach dem Re-Mapping mindestens diese Verzögerungszeiten für die Platzierung der Komponenten

| Pfad<br>$v \in V_{dyn}$ | Delay in 0-DT-Version von v<br>$d(v,0)$ | Delay in 1-DT-Version von v<br>$d(v,1)$ | Zusätzlich erforderliches Delay für eine Änderung von $state(hc_v(v))$<br>$\Delta d(v,0,1) := d(v,1) - d(v,0)$ |
|---|---|---|---|
| $A_{p1}$ | 1 | 3 | 2 |
| $B_{p1}$ | 1 | 7 | 6 |
| $C_{p1}$ | 1 | 5 | 4 |
| $C_{p2}$ | 1 | 4 | 3 |
| $D_{p1}$ | 1 | 3 | 2 |
| $E_{p1}$ | 1 | 5 | 4 |

Tabelle 5.1: Zu dem laufenden Beispiel: Die Verzögerungszeiten der Knoten $v \in V_{dyn}$ und das zusätzlich erforderliche Delay für ein Re-Mapping

der Netzliste und die Erstellung der Verbindungsnetze zwischen diesen Komponenten durch das Place&Route-Tool zur Verfügung stehen.

**Definition 5.9.** Ein Re-Mapping wird genau dann als ***timing-closure friendly*** bezeichnet, wenn für die Platzierung und Verdrahtung des aus dem Re-Mapping hervorgegangenen DT-Mappings auf jedem Pfad mindestens die Verzögerungszeit eingeplant wird, die in der FPGA-Konfiguration vor dem Re-Mapping für die Platzierung und Verdrahtung benötigt wurde.

Zur Verdeutlichung von Definition 5.9 wird jetzt zunächst gezeigt, dass das oben für das laufende Beispiel durchgeführte Re-Mapping von allen Instanzen nicht *timing-closure friendly* ist. Im Anschluss daran werden zwei Re-Mappings für das laufende Beispiel vorgestellt, die *timing-closure friendly* sind. Um eine genauere Betrachtung des Zeitverhaltens vor und nach dem Re-Mapping zu ermöglichen, soll nun zunächst der Slack (siehe Definition 2.1 auf Seite 22) an den Knoten des Graphen aus Abbildung 5.19 annotiert werden.

Es sei $I \subset V$ die Menge der Eingangsknoten, d.h. Knoten ohne eingehende Kanten, und $O \subset V$ die Menge der Ausgangsknoten, d.h. Knoten ohne ausgehende Kanten. Für alle $v \in I$ sei weiterhin der Zeitpunkt $a(v)$ (*arrival time*) angegeben, zu dem das Signal an den Eingängen der direkten Nachfolger des Knotens eintrifft. Dann kann von den Knoten $v \in I$ ausgehend für jeden Knoten $v \in V \backslash I$ der späteste Zeitpunkt $a(v)$ berechnet werden, zu dem das Signal an den Eingängen der direkten Nachfolger des Knotens $v$ zur Verfügung steht. Es gilt

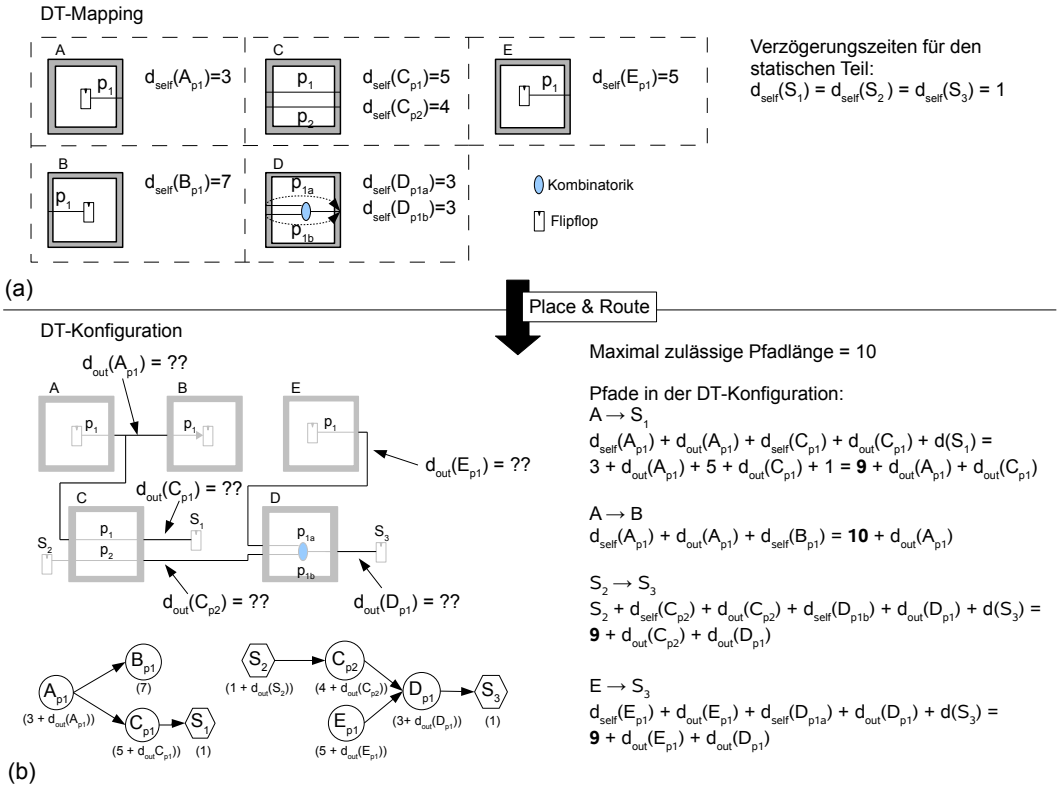



Abbildung 5.21: Veränderung der Verzögerungszeiten, wenn im laufenden Beispiel für alle Instanzen $hc \in HC$ ein Re-Mapping auf die jeweilige 1-DT-Version der Instanz erfolgt.

$$a(v) = max_{u \in FI(v)}(a(u) + d(v)) \tag{5.9}$$

, wenn $FI(v)$ alle direkten Vorgängerknoten von $v$ bezeichnet.

Wenn für alle Ausgangsknoten $v_{out} \in O$ der späteste Zeitpunkt, zu dem das Signal am Ausgang benötigt wird, als $r(v_{out})$ (*required arrival time*) angegeben ist, kann rekursiv für jeden Knoten $v \in V \backslash O$ der späteste Zeitpunkt $r(v)$ berechnet werden, zu dem das Signal an den Eingängen der direkten Nachfolger des Knotens spätestens benötigt wird. Es gilt

$$r(v) = min_{w \in FO(v)}(r(w) - d(w)) \tag{5.10}$$

, wenn $FO(v)$ alle direkten Nachfolger von $v \in V$ bezeichnet.

Wir setzten nun für alle Eingangsknoten $v_{in} \in I$ die *arrival time* auf den Zeitpunkt $a(v_{in}) = 0 + d(v_{in})$ und für alle Ausgangsknoten $v_{out} \in O$ die *required arrival ti-*

*me* $r(v_{out})$ auf die maximale Verzögerungszeit, die durch eine Timing-Analyse für die FPGA-Konfiguration nach dem Place&Route-Vorgang ermittelt wurde. Der Slack $s(v)$ für jeden Knoten $v \in V$ ist dann die Differenz zwischen $r(v)$ und $a(v)$

$$s(v) = r(v) - a(v) \tag{5.11}$$

In Abbildung 5.22 sind $d(v)$, $a(v)$, $r(v)$ und $s(v)$ für den Graphen aus Abbildung 5.19 (b) jeweils für jeden Knoten $v \in V$ angegeben.

Wenn wir von einer maximal zulässigen Verzögerungszeit von 10 Zeiteinheiten ausgehen, können wir die in Abbildung 5.22 angegebene *arrival time* $a(v)$ für jeden Knoten mit einem rekursiven Durchlauf mit Formel 5.9 wie folgt berechnen:

$$\begin{aligned}
I &= \{A_{p1}, S_2, E_{p1}\} \\
\forall v \in I :\ a(v) &= 0 + d(v) \\
a(B_{p1}) &= a(A_{p1}) + d(B_{p1}) = 3 \\
a(C_{p1}) &= a(A_{p1}) + d(C_{p1}) = 4 \\
a(S_1) &= a(C_{p1}) + d(S_1) = 5 \\
a(C_{p2}) &= a(S_2) + d(C_{p2}) = 4 \\
a(D_{p1}) &= max_{u \in FI(D_{p1})}(a(u) + d(D_{p1})) = a(C_{p2}) + d(D_{p1}) = 6 \\
a(S_3) &= a(D_{p1}) + d(S_3) = 7
\end{aligned}$$

Durch einen weiteren rekursiven Durchlauf mit Formel 5.10 kann die *required arrival time* (kurz: *arrival time*) $r(v)$ für jeden Knoten berechnet werden:

$$\begin{aligned}
O &= \{B_{p1}, S_1, S_3\} \\
\forall v \in O :\ r(v) &= 10 \\
r(C_{p1}) &= r(S_1) - d(S_1) = 9 \\
r(A_{p1}) &= min_{w \in FO(A_{p1})}(r(w) - d(w)) = r(C_{p1}) - d(C_{p1}) = 7 \\
r(D_{p1}) &= r(S_3) - d(S_3) = 9 \\
r(C_{p2}) &= r(D_{p1}) - d(D_{p1}) = 7 \\
r(E_{p1}) &= r(C_{p2}) = 7 \\
r(S_2) &= r(C_{p2}) - d(C_{p2}) = 5
\end{aligned}$$

Abbildung 5.23 stellt für das in Abbildung 5.21 durchgeführte Re-Mapping einen annotierten Graph $G = (V, E)$ dar, wenn für die Platzierung und Verdrahtung für jeden

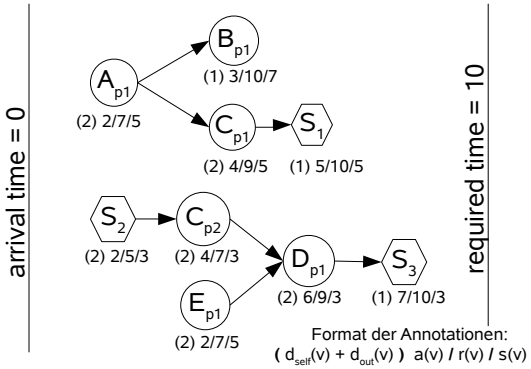



Abbildung 5.22: Die Berechnung des Slack an den Graphen aus Abbildung 5.19 (b)

Knoten $v \in V$ die vor dem Re-Mapping aus der FPGA-Konfiguration ermittelte Verzögerungszeit (im laufenden Beispiel gilt für jeden Knoten $d_{out}(v) = 1$) annotiert wird. Der negative Slack an den Knoten zeigt, dass bei der Erstellung dieses Re-Mapping offensichtlich nicht ausreichend Verzögerungszeit eingeplant wurde. Das Re-Mapping ist daher nach Definition 5.9 nicht *timing-closure friendly*.

Eine Möglichkeit für die Erstellung eines Re-Mappings, das *timing-closure friendly* ist, besteht in der schrittweisen Veränderung des ursprünglichen Mappings. Wenn wir den an den Knoten in Abbildung 5.22 annotierten Slack, also den Wert, um den das Delay der Knoten jeweils ohne eine Verlängerung des kritischen Pfades erhöht werden kann, und das in Tabelle 5.1 in der letzten Spalte angegebene zusätzlich erforderliche Delay für ein Re-Mapping der Instanzen vergleichen, stellen wir fest, dass für jeden dynamischen Knoten $v \in V_{dyn}$ des Graphen $\Delta d(v, 0, 1) < s(v)$ gilt. Aufgrund von Abhängikeiten der Knoten untereinander muss nach einer Veränderung von $d_{self}(v)$ an einem Knoten in einer Komponente des Graphen (siehe Definition 5.10) der an den Knoten verbleibende Slack jedoch erneut berechnet werden. Erst danach kann beurteilt werden, ob der verbleibende Slack an den Knoten $v \in V$ für die Erhöhung von $d_{self}(v)$ an weiteren Knoten ausreichend ist.

**Definition 5.10.** Es sei $G = (V, E)$ ein beliebiger Graph

- $G$ heißt zusammenhängend, wenn für alle $u, v \in V$ ein Pfad von $u$ nach $v$ oder ein Pfad von $v$ nach $u$ existiert
- Die zusammenhängenden Teile von $G$ werden als die *Komponenten* des Graphen $G$ bezeichnet

Da in dem laufenden Beispiel bei einem Re-Mapping der Instanzen A, B, D oder E nur jeweils an einen Knoten des Graphen $G$ und bei einem Re-Mapping der Instanz C nur an einen Knoten in jeder Komponente von $G$ eine neue Verzögerungszeit annotiert

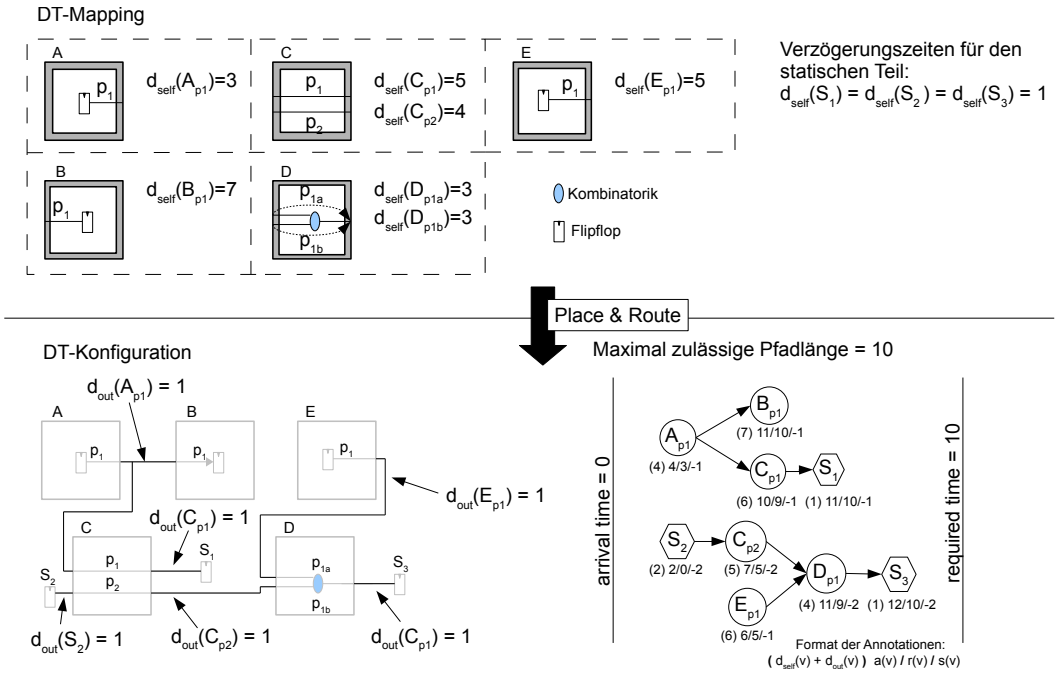



Abbildung 5.23: Alle Instanzen werden auf 1-DT-Versionen abgebildet

wird, kann aufgrund des annotierten Slacks zunächst ein Re-Mapping einer beliebigen Instanz erfolgen, bevor der Slack an den Knoten von $G$ erneut berechnet werden muss.

Abbildung 5.24 (a) zeigt den Graphen aus Abbildung 5.22 nach dem Re-Mapping der Instanz $C \in HC$ auf die 1-DT-Version, wenn die Verzögerungszeiten entsprechend Tabelle 5.1 angenommen werden. Die annotierten Verzögerungszeiten sind im nächsten Schritt noch ausreichend Slack für das Re-Mapping der Instanz $B$ ($\forall v \in nodes(B) : \Delta d(v,0,1) < s(v)$, da $\Delta d(B_{p1},0,1) < s(B_{p1})$ ) und $E$ ($\forall v \in nodes(E) : \Delta d(v,0,1) < s(v)$, da $\Delta d(E_{p1},0,1) < s(E_{p1})$ ). Nachdem in Abbildung 5.24 (b) das Re-Mapping von $C$, $B$ und $E$ vorgenommen wurde, ist der im Graphen verbleibende Slack für das Re-Mapping einer weiteren Instanz nicht mehr ausreichend ($\forall v \in nodes(A) : \Delta d(v,0,1) > s(v)$, da $\Delta d(A_{p1},0,1) > s(A_{p1})$ und $\forall v \in nodes(D) : \Delta d(v,0,1) > s(v)$, da $\Delta d(D_{p1},0,1) > s(D_{p1})$ ).

Wenn wir annehmen, dass die Verzögerungszeiten, die für die Verbindungsleitungen zwischen den Komponenten der Netzliste nach dem Re-Mapping in der DT-Konfiguration benötigt werden, den Verzögerungszeiten entsprechen, die vor dem Re-Mapping in der ursprünglichen FPGA-Konfiguration benötigt wurden und der Slack nach dem Re-Mapping an keinem Knoten des betrachteten Graphen kleiner 0 ist, können wir das Re-Mapping als *timing-closure-friendly* nach Definition 5.9 bezeichnen.

Abbildung 5.25 und Abbildung 5.26 zeigen jeweils ein DT-Mapping, aus dem unter die-

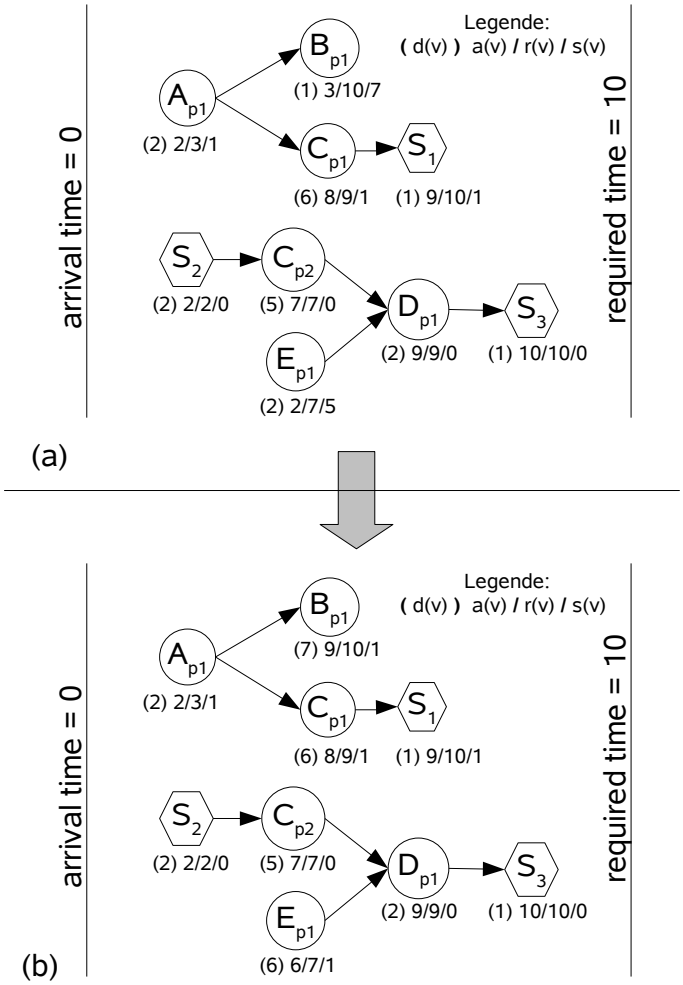



Abbildung 5.24: Der an den Knoten verbleibende Slack (a) nach dem Re-Mapping der Instanz C und (b) nach dem darauffolgenden Re-Mapping der Instanzen B und E auf die 1-DT-Version.

ser Bedingung eine DT-Konfiguration erstellt werden kann, die das in unserem Beispiel vorhandene Timing-Constraint „Maximal zulässige Pfadlänge = 10“ einhält. In beiden Lösungen resultiert das für ein Re-Mapping einer Instanz benötigte zusätzliche Delay an den Knoten dieser Instanz in einer Reduzierung des Slacks an den ausgetauschten Knoten. In dem in Abbildung 5.26 vorgeschlagenen Austausch geschieht das Re-Mapping der Instanzen A und D darüber hinaus zusätzlich zu Lasten des Slacks - und somit eines potentiell möglichen Re-Mappings - an den Knoten der Instanzen B, C und E, da der Slack an den Knoten $B_{p1}$, $C_{p1}$, $E_{p1}$ und $C_{p2}$ derart reduziert wird, dass kein Re-Mapping einer weiteren Instanz mehr möglich ist. Auch in Abbildung 5.24 wird der Slack durch den Austausch der Instanz C, also durch den gleichzeitigen Austausch der Knoten $C_{p1}$

und $C_{p2}$, derart reduziert, dass für ein Re-Mapping der Instanzen $A$ und $D$ nicht mehr ausreichend Slack an den Knoten zur Verfügung steht. Das Re-Mapping der Instanz $C$ hat allerdings *keine* Auswirkungen auf den Slack von $B_{p1}$ und $E_{p1}$. Der Austausch dieser Knoten im nächsten Schritt hat wiederum *keine* Reduzierung des Slacks an den Knoten $A_{p1}$ und $D_{p1}$ zur Folge.

Wir können die Delay-Werte $\Delta d(v_1), \Delta d(v_2), ..., \Delta d(v_n)$, die zu den Knoten $V = \{v_1, v_2, ..., v_n\}$ in $G(V, E)$ im jeweiligen Re-Mapping hinzugefügt wurden, als Vektor $\Delta D(V) = [\ \Delta d(v_1)\ \ \Delta d(v_2)\ \ ...\ \ \Delta d(v_n)]$ angeben. Die Summe aller Elemente dieses Vektors $|\Delta D(V)| = \sum_{k=1}^{n} \Delta d(v_k)$ ist dann das Delay, das insgesamt an den Knoten durch das jeweilige Re-Mapping zusätzlich benötigt wird. Für das laufende Beispiel führen wir nun eine Umbenennung der Knoten des Graphen entsprechend Abbildung 5.27 durch, in der Form, dass

$$\Delta D(V) = [\ \Delta d(v_1)\ \ \Delta d(v_2)\ \ \Delta d(v_3)\ \ \Delta d(v_4)\ \ \Delta d(v_5)\ \ \Delta d(v_6)\ \ \Delta d(v_7)\ \ \Delta d(v_8)\ \ \Delta d(v_9)\ ]$$

$$:= [\ \Delta d(A_{p1})\ \ \Delta d(B_{p1})\ \ \Delta d(C_{p1})\ \ \Delta d(S_1)\ \ \Delta d(S_2)$$
$$\Delta d(C_{p2})\ \ \Delta d(D_{p1})\ \ \Delta d(S_3)\ \ \Delta d(E_{p1})\ \ ]$$

gilt.

Für das in Abbildung 5.25 durchgeführte Re-Mapping der Instanzen C, B und E gilt dann entsprechend der letzten Spalte in Tabelle 5.1

$$\Delta D(V) = [\ \Delta d(A_{p1}) = 0\ \ \Delta d(B_{p1}) = 6\ \ \Delta d(C_{p1}) = 4\ \ \Delta d(S_1) = 0\ \ \Delta d(S_2) = 0$$
$$\Delta d(C_{p2}) = 3\ \ \Delta d(D_{p1}) = 0\ \ \Delta d(S_3) = 0\ \ \Delta d(E_{p1}) = 4\ \ ]$$

Demgegenüber gilt für das in Abbildung 5.26 durchgeführte Re-Mapping der Instanzen A und D

$$\Delta D(V) = [\ \Delta d(A_{p1}) = 2\ \ \Delta d(B_{p1}) = 0\ \ \Delta d(C_{p1}) = 0\ \ \Delta d(S_1) = 0\ \ \Delta d(S_2) = 0$$
$$\Delta d(C_{p2}) = 0\ \ \Delta d(D_{p1}) = 2\ \ \Delta d(S_3) = 0\ \ \Delta d(E_{p1}) = 0\ \ ]$$

Der unterschiedliche Umfang der insgesamt bei dem Re-Mapping der Instanzen C, B und E zusätzlich verwendeten Verzögerungszeit $|\Delta D(V)| = 17$ gegenüber der bei dem Re-Mapping der Instanzen A und D zusätzlich benötigten Verzögerungszeit $|\Delta D(V)| = 4$ zeigt, dass der Umfang der Verzögerungszeit, die für das Re-Mapping jeweils an den Knoten eingesetzt werden kann, von der Topologie der dem DT-Mapping zugrundeliegenden Netzliste abhängt.

### 5.3.3 Ein Re-Mapping-Algorithmus

Der Re-Mapping-Algorithmus soll nun parallel zu den folgenden Erläuterungen an dem laufenden Beispiel, das im letzten Abschnitt eingeführt wurde, demonstriert werden. Wir legen für das Beispiel daher zunächst die in Tabelle 5.2 angegebenen Funktionswahrscheinlichkeiten für die beteiligten Hard-Core-Versionen fest. In dieser Tabelle ist

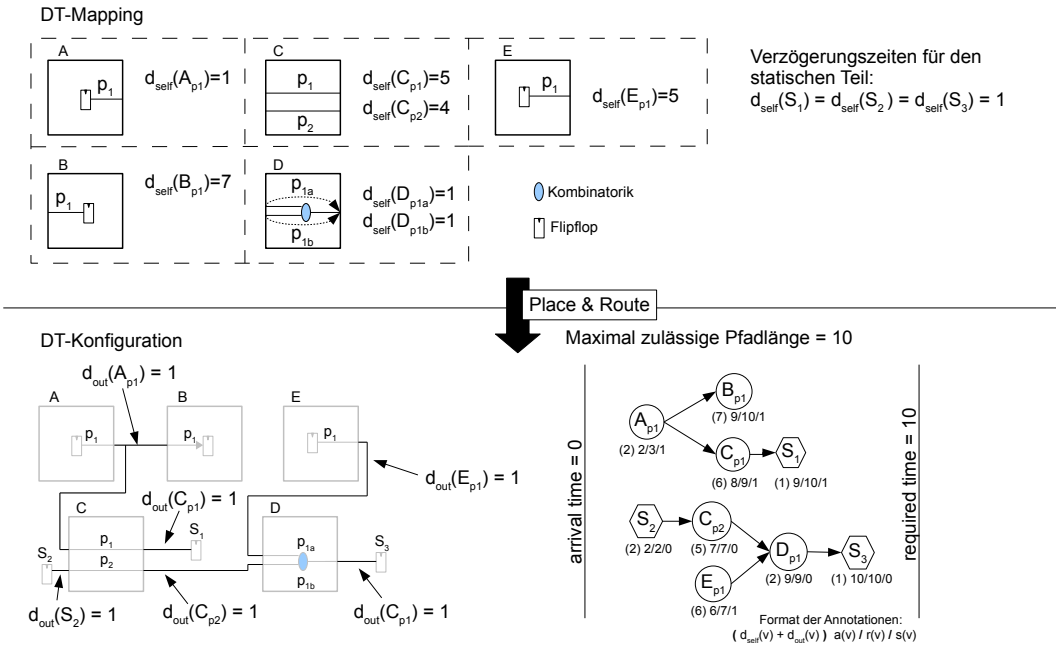



Abbildung 5.25: Die Instanzen C, B und E werden auf 1-DT-Versionen abgebildet

für jede der im laufenden Beispiel verfügbaren Versionen der Hard-Cores entsprechend Definition 5.5 auf Seite 77 die Wahrscheinlichkeit $p(hc, x)$ dafür angegeben, dass nur soviele defekte Logikblöcke innerhalb der Version $x \in states(hc)$ des Hard-Cores liegen, wie durch diese Version toleriert werden können. Das Verfahren für die Berechnung dieser Wahrscheinlichkeit wurde in Abschnitt 5.2.3 vorgestellt. Die in Tabelle 5.2 angegebenen Werte wurden hier so festgelegt, dass sie für die spätere Erläuterung des Re-Mapping-Algorithmus an dem laufenden Beispiel günstig sind.

Wenn die aktuelle Hard-Core-Version, die für das Mapping einer Instanz $hc \in HC$ verwendet wird, entsprechend Definition 5.3 mit $state(hc)$ und die nach dem Re-Mapping verwendete Hard-Core-Version mit $state'(hc)$ bezeichnet wird und $\phi_{ges}$ den Faktor bezeichnet, um den $p(HC)$ durch das Re-Mapping verbessert wird, gilt:

$$\phi_{ges} = \frac{\prod_{i=1}^{n} p(x_i, state'(x_i))}{\prod_{i=1}^{n} p(x_i, state(x_i))} = \prod_{i=1}^{n} \frac{p(x_i, state'(x_i))}{p(x_i, state(x_i))} \tag{5.12}$$

**Definition 5.11.** $\phi(hc, a, b)$ bezeichnet den Faktor, um den sich $\phi_{ges}$ verbessert, wenn vor dem Re-Mapping die Hard-Core-Version $a$ und nach dem Re-Mapping die Hard-Core-Version $b$ für die Instanz $hc$ verwendet wird.

Es gilt $\phi(hc, a, b) := \frac{p(hc,b)}{p(hc,a)}$

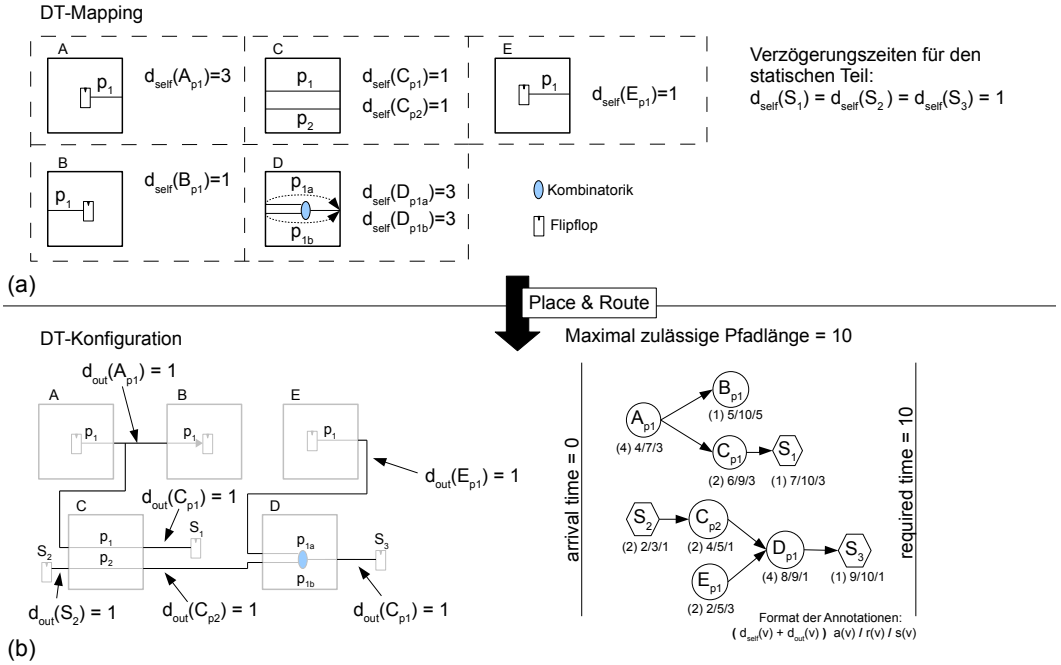



Abbildung 5.26: Die Instanzen A und D werden auf 1-DT- Versionen abgebildet

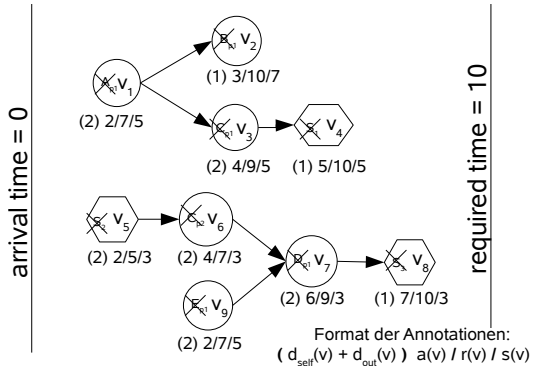



Abbildung 5.27: Umbenennung der Knoten aus Abbildung 5.22

Durch den in Formel 5.12 ersichtlichen Zusammenhang können wir $\phi_{ges}$ nun wie folgt definieren:

**Definition 5.12.** $\phi_{ges}$ bezeichnet den Faktor, um den $p(HC)$ durch ein Re-Mapping verbessert wird.

Es gilt $\phi_{ges} := \prod_{hc \in HC} \phi(hc, state(hc), state'(hc))$

In Tabelle 5.2 ist $\phi(hc, 0, 1)$ für jede Instanz $hc \in HC$ angegeben. Die Bezeichnungen der Versionen „0DT“ und „1DT“ werden hier jeweils abgekürzt als „0“ und „1“ angegeben.

| $hc \in HC$ | $p(hc, 0\,DT)$ | $p(hc, 1\,DT)$ | $\phi(hc, 0, 1) := \frac{p(hc, 1\,DT)}{p(hc, 0\,DT)}$ |
|---|---|---|---|
| A | 0,92 | 0,99 | 1,076 |
| B | 0,9 | 0,95 | 1,056 |
| C | 0,9 | 0,95 | 1,056 |
| D | 0,92 | 0,99 | 1,076 |
| E | 0,9 | 0,95 | 1,056 |

Tabelle 5.2: $p(hc, 0DT)$ und $p(hc, 1DT)$ sowie der daraus resultierende Faktor $\phi(hc, 0, 1)$ für alle Instanzen $hc \in HC$ des laufenden Beispiels

Wenn bei einem Re-Mapping alle der in der Netzliste enthaltenen 0-DT-Hard-Core-Versionen gegen n-DT-Hard-Core-Versionen mit dem höchsten verfügbaren Wert für $n$ ausgetauscht werden, erhalten wir den bestmöglichen Wert für $p(HC)$. Für das Re-Mapping aller Instanzen $hc \in HC$ (siehe Abbildung 5.23 auf Seite 87), im Folgenden als Lösung A bezeichnet, gilt dann

$$
\begin{aligned}
p(HC) &= \prod_{hc \in HC} p(hc, state'(hc)) = p(A, 1DT) * p(B, 1DT) * p(C, 1DT) * p(D, 1DT) * p(E, 1DT) \\
&= \prod_{hc \in HC} p(hc, state(hc)) * \phi_{ges} \\
&= \prod_{hc \in HC} p(hc, 0DT) * \prod_{hc \in HC} \phi(hc, 0, 1) \\
&\approx 0,617 * 1,363 \\
&\approx 0,84
\end{aligned}
$$

Für das Re-Mapping aus Abbildung 5.25 auf Seite 90, im Folgenden bezeichnet als Lösung B, ergibt sich im Vergleich zu Lösung A ein niedrigerer Wert für $p(HC)$ mit

$$
\begin{aligned}
p(HC) &= \prod_{hc \in HC} p(hc, 0DT) * \phi(C, 0, 1) * \phi(B, 0, 1) * \phi(E, 0, 1) \\
&\approx 0,617 * 1,056^3 \\
&\approx 0,727
\end{aligned}
$$

Für die letzte der vorgestellten Lösungen aus Abbildung 5.26 auf Seite 91, im Folgenden bezeichnet als Lösung C, erhalten wir den niedrigsten Wert für $p(HC)$ mit

$$
\begin{aligned}
p(HC) &= \prod_{hc \in HC} p(hc, 0DT) * \phi(A, 0, 1) * \phi(D, 0, 1) \\
&\approx 0,617 * 1,076^2 \\
&\approx 0,714
\end{aligned}
$$

Um dem Place&Route-Tool die Timing-Closure zu erleichtern, ist das Optimierungsziel für den Re-Mapping-Algorithmus das Re-Mapping mit dem höchsten Wert für $p(HC)$ unter der Bedingung, dass dieses Re-Mapping *timing-closure friendly* nach Definition 5.9 ist.

Lösung B und Lösung C sind im Gegensatz zu Lösung A *timing-closure friendly* nach Definition 5.9. Wie im letzten Abschnitt gezeigt, können diese Lösungen nicht weiter verbessert werden, ohne dass diese Eigenschaft verloren geht. Die optimale Lösung für das laufende Beispiel ist bei dem oben angegebenen Optimierungsziel für den Re-Mapping-Algorithmus die Lösung B.

Eine Re-Mapping, das eine mögliche Lösung darstellt, lässt sich als Vektor beschreiben:

**Definition 5.13.** $ReMapping(HC) := [state'(hc_1) \ldots state'(hc_n)]$, mit $n = |HC|$ ist ein Vektor, der die Zustände der Instanzen $hc \in HC$ nach dem Re-Mapping enthält.

Ein optimales Re-Mapping kann nun wie folgt definiert werden:

**Definition 5.14.** Das optimale Re-Mapping ist das Re-Mapping mit der Eigenschaft *timing-closure friendly*, das den maximal erreichbaren Wert für $\phi_{ges}$ zur Folge hat. Für den maximal erreichbaren Wert für $\phi_{ges}$, im Folgenden bezeichnet mit $\phi_{max}$, gilt $\phi_{max} := max_{ReMapping(HC) \in \Psi} \prod_{hc \in HC} \phi(state(hc), state'(hc))$, wenn $\Psi$ die Menge aller möglichen Vektoren $ReMapping(HC)$ mit der Eigenschaft *timing-closure friendly* ist.

Da der Slack nach jedem Re-Mapping einer Instanz $hc \in HC$ erneut berechnet werden muss, bevor aufgrund des Slacks beurteilt werden kann, ob das Re-Mapping wei-

terer Instanzen erfolgen kann, soll für den Re-Mapping-Algorithmus im Folgenden neben dem Slack eine weitere Metrik für jeden Knoten herangezogen werden. Für diese Metrik, bezeichnet als Delay-Budget, gilt, dass die Nutzung des an den Knoten annotierten Delay-Budgets an allen Knoten gleichzeitig erfolgen kann, ohne den kritischen Pfad zu verletzen. Die im letzten Abschnitt eingeführte Notation $\Delta D(V) = [\ \Delta d(v_1)\ \ \Delta d(v_2)\ \ ...\ \ \Delta d(v_n)]$ soll nun als Vektor betrachtet werden, der jeweils ein Delay-Budget für die Knoten $v_1, v_2, ..., v_n$ angibt. Um das vorliegende Optimierungsproblem als *budget management* Problem formal zu beschreiben, übernehmen wir zunächst die folgenden Definitionen aus [CBSS02].

**Definition 5.15.** Eine *delay distribution* $D(V) = [\ d(v_1)\ \ d(v_2)\ \ ...\ \ d(v_n)\ ]$, mit $n = |V|$, ist ein Vektor, der die Delay-Werte der Knoten aus $G$ enthält.

**Definition 5.16.** Eine *slack distribution* $S(V) = [\ s(v_1)\ \ s(v_2)\ \ ...\ \ s(v_n)\ ]$, mit $n = |V|$, ist ein Vektor, der die Slack-Werte der Knoten aus $G$ enthält.

**Definition 5.17.** Eine *budget management instance* $\Delta D(V) = [\ \Delta d(v_1)\ \ \Delta d(v_2)\ \ ...\ \ \Delta d(v_n)\ ]$ mit $n = |V|$, ist ein Vektor mit Delay-Werten, die zu den Knoten in $G$ hinzugefügt werden. Die aktualisierte *delay distribution* $D_\Delta(V) = D(V) + \Delta D(V)$ führt zu einer aktualisierten *slack distribution* $S_\Delta(V)$.

Da die Verteilung des Slacks in der Schaltung von den Werten für $a(v)$ und $r(v)$ zu jedem Knoten $v \in V$ abhängt, ist für die Ermittlung von $S_\Delta(V)$ eine erneute Berechnung dieser Werte über zwei rekursive Durchläufe mit den Gleichungen 5.9 und 5.10 (Seite 84) notwendig. Weil ein negativer Slack an einem Knoten in $G$ zu einer Verletzung der vorgegebenen Timing-Constraints in der durch $G$ modellierten Schaltung führen würde, wird $G$ nur dann als *safe* bezeichnet, wenn $S(V) \geq 0$ gilt, wobei 0 hier für einen n-dimensionalen Vektor mit $n = |V|$ steht, dessen Elemente alle null sind.

**Definition 5.18.** Wenn eine *slack distribution* über eine *budget management instance* aktualisiert wird und $G$ nach dieser Aktualisierung noch immer *safe* ist, wird die *budget management instance* als *effective budget management instance* bezeichnet.

Auf den Graphen aus Abbildung 5.22 (Seite86) mit der *delay distribution*

$$D(V) = [\ d(A_{p1}) = 2\ \ d(B_{p1}) = 1\ \ d(C_{p1}) = 2\ \ d(S_1) = 1\ \ d(S_2) = 2\ \ d(C_{p2}) = 2\ \ d(D_{p1}) = 2\ \ d(S_3) = 1\ \ d(E_{p1}) = 2\ ]$$

bezogen gilt:

- Die *budget management instance*
  $\Delta D(V) = [\ \Delta d(A_{p1}) = 0 \quad \Delta d(B_{p1}) = 6 \quad \Delta d(C_{p1}) = 4 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0$
  $\Delta d(C_{p2}) = 3 \quad \Delta d(D_{p1}) = 0 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 4 \ ]$
  ist eine *effective budget management instance*, da die aktualisierte *delay distribution* (vgl. Delay-Annotationen an den Knoten des Graphen in Abbildung 5.25 auf Seite 90)
  $D_\Delta(V) = [\ d(A_{p1}) = 2 \quad d(B_{p1}) = 7 \quad d(C_{p1}) = 6 \quad d(S_1) = 1 \quad d(S_2) = 2$
  $d(C_{p2}) = 5 \quad d(D_{p1}) = 2 \quad d(S_3) = 1 \quad d(E_{p1}) = 6 \ ]$
  zu der aktualisierten *slack distribution*
  $S_\Delta(V) = [\ s(A_{p1}) = 1 \quad s(B_{p1}) = 1 \quad s(C_{p1}) = 1 \quad s(S_1) = 1 \quad s(S_2) = 0$
  $s(C_{p2}) = 0 \quad s(D_{p1}) = 0 \quad s(S_3) = 0 \quad s(E_{p1}) = 1 \ ]$
  führt und somit $S_\Delta(V) \geq 0$ gilt.

- Die *budget management instance*
  $\Delta D(V) = [\ \Delta d(A_{p1}) = 2 \quad \Delta d(B_{p1}) = 0 \quad \Delta d(C_{p1}) = 0 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0$
  $\Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 2 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 0 \ ]$
  ist ebenfalls eine *effective budget management instance*, da die aktualisierte *delay distribution* (vgl. Delay-Annotationen an den Knoten des Graphen in Abbildung 5.26 auf Seite 91)
  $D_\Delta(V) = [\ d(A_{p1}) = 4 \quad d(B_{p1}) = 1 \quad d(C_{p1}) = 2 \quad d(S_1) = 1 \quad d(S_2) = 2$
  $d(C_{p2}) = 2 \quad d(D_{p1}) = 4 \quad d(S_3) = 1 \quad d(E_{p1}) = 2 \ ]$
  zu der aktualisierten *slack distribution*
  $S_\Delta(V) = [\ s(A_{p1}) = 3 \quad s(B_{p1}) = 5 \quad s(C_{p1}) = 3 \quad s(S_1) = 3 \quad s(S_2) = 1$
  $s(C_{p2}) = 1 \quad s(D_{p1}) = 1 \quad s(S_3) = 1 \quad s(E_{p1}) = 3 \ ]$
  führt und somit $S_\Delta(V) \geq 0$ gilt.

- Die für das Re-Mapping aller Instanzen von Hard-Core-Versionen in der Netzliste mindestens benötigten Budgets für jeden Knoten des Graphen, d.h. die *budget management instance*
  $\Delta D(V) = [\ \Delta d(A_{p1}) = 2 \quad \Delta d(B_{p1}) = 6 \quad \Delta d(C_{p1}) = 4 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0$
  $\Delta d(C_{p2}) = 3 \quad \Delta d(D_{p1}) = 2 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 3 \ ]$
  ist keine *effective budget management instance*, da die aktualisierte *delay distribution* (vgl. Delay-Annotationen an den Knoten des Graphen in Abbildung 5.23)
  $D_\Delta(V) = [\ d(A_{p1}) = 4 \quad d(B_{p1}) = 7 \quad d(C_{p1}) = 6 \quad d(S_1) = 1 \quad d(S_2) = 2$
  $d(C_{p2}) = 5 \quad d(D_{p1}) = 4 \quad d(S_3) = 1 \quad d(E_{p1}) = 6 \ ]$
  zu der aktualisierten *slack distribution*
  $S_\Delta(V) = [\ s(A_{p1}) = -1 \quad s(B_{p1}) = -1 \quad s(C_{p1}) = -1 \quad s(S_1) = -1 \quad s(S_2) = -2$
  $s(C_{p2}) = -2 \quad s(D_{p1}) = -2 \quad s(S_3) = -2 \quad s(E_{p1}) = -1 \ ]$
  führt und somit $S_\Delta(V) \geq 0$ *nicht* gilt.

Ein Re-Mapping einer Instanz $hc \in HC$ auf eine Hard-Core-Version $x \in states(HC)$ kann genau dann ohne eine Verletzung der durch eine *effective budget management instance* vorgegebenen Delay-Budgets $\Delta d(v)$ für jeden Knoten $v \in nodes(hc)$ erfolgen, wenn das an den betroffenen Knoten angegebene Delay-Budget mindestens so groß ist

wie das für dieses Re-Mapping zusätzlich erforderliche Delay, d.h. $\forall v \in nodes(hc)$ : $\Delta d(v, state(hc), state'(hc)) \leq d(v, state(hc)) + \Delta d(v)$.

**Definition 5.19.** Ein Re-Mapping einer Instanz $hc \in HC$, dargestellt als Änderung des Zustands $state(hc)$ in einen Zustand $state'(hc)$ ist **in Bezug auf eine vorgegebene *effective budget management instance* zulässig**, wenn $\forall v \in nodes(hc)$ : $\Delta d(v, state(hc), state'(hc)) \leq d(v, state(hc)) + \Delta d(v)$ gilt und **optimal**, wenn von allen zulässigen $state'(hc)$ der Zustand gewählt wird, für den $p(hc, state(hc))$ maximal ist.

**Definition 5.20.** Ein Re-Mapping einer kompletten Netzliste $HC$ ist **optimal in Bezug auf eine vorgegebene *effective budget management instance***, wenn das Re-Mapping für jede Instanz $hc \in HC$ optimal ist.

Abbildung 5.28 zeigt das optimale Re-Mapping für das laufende Beispiel in Bezug auf die *effective budget management instance*

$$\begin{array}{lll} \Delta D(V) = & [ & \Delta d(A_{p1}) = 0 \quad \Delta d(B_{p1}) = 7 \quad \Delta d(C_{p1}) = 5 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0 \\ & & \Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 3 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 2 \qquad ] \end{array}$$

Für das Mapping der Instanzen A, C und E ist hier nach Definition 5.19 nur die Abbildung auf 0-DT-Versionen zulässig. Aus den zulässigen Mappings für die Instanzen B und D wählen wir in Abbildung 5.28 jeweils nach Definition 5.19 das optimale Mapping auf die 1-DT-Versionen aus.

Wir können das vorliegende Optimierungsproblem jetzt als ein *budget management* Problem betrachten. Gesucht ist eine *effective budget management instance* $\Delta D(V)$, bei der das optimale Re-Mapping in Bezug auf $\Delta D(V)$ dem optimalen Re-Mapping für das Optimierungsproblem laut Definition 5.14 entspricht. Bei dieser Betrachtung könnte der in Pseudo-Code angegebene Algorithmus 5.2 für die Optimierung verwendet werden: Nachdem in Zeile 3 durch zwei rekursive Durchläufe mit Hilfe der Gleichungen 5.9 und 5.10 (Seite 84) der Slack für jeden Knoten in $G$ berechnet wurde, wird in Zeile 3 eine für das Re-Mapping optimale *effective budget management instance* bestimmt, die dann in den Zeilen 3 bis 3 die Durchführung des optimalen Re-Mappings zur Folge hat.

Um die nicht triviale Bestimmung der in Zeile 3 geforderten optimalen *budget management instance* zu umgehen, verbessert die in der vorliegenden Arbeit entwickelte Heuristik das vorliegende Mapping iterativ. Algorithmus 5.3 skizziert die groben Schritte der Heuristik. Für jede Instanz $hc \in HC$ wird in Zeile 4 zunächst das in Bezug auf die Verbesserung von $p(HC)$ bestmögliche Re-Mapping für eine eventuelle Durchführung eingeplant. Als mögliche (*feasible*) Ziele für das Re-Mapping einer Instanz werden jeweils nur DT-Versionen berücksichtigt, für die ein ausreichender Slack auf jedem Pfad durch das Hard-Core vorhanden ist.

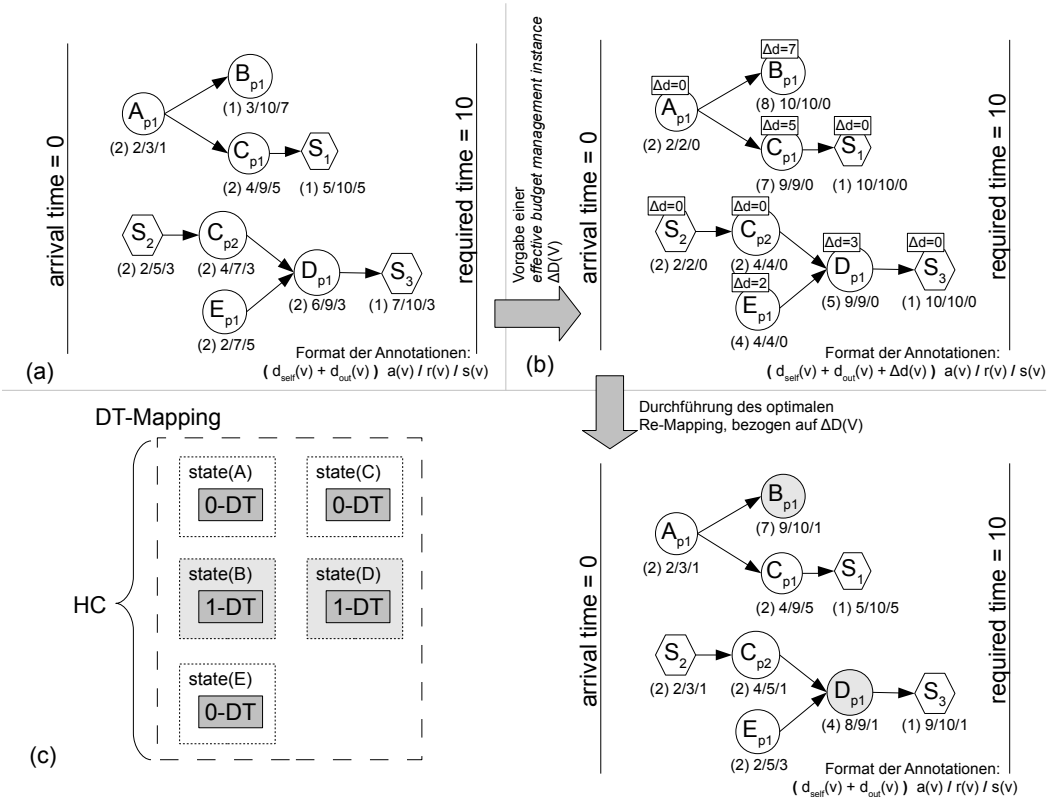



Abbildung 5.28: Beispiel für das optimale Re-Mapping der Instanzen aus Abbildung 5.19, bezogen auf eine vorgegebene *effective budget management instance* $\Delta D(V)$

**Definition 5.21.** $states_{feasible}(hc \in HC)$ bezeichnet die Menge der Hard-Core-Versionen, die für das Re-Mapping der Instanz $hc \in HC$ aufgrund des an den Knoten $nodes(hc) \in V$ annotierten Slacks ein mögliches Ziel darstellen.

Es gilt:

$states_{feasible}(hc) := \{x \in states(hc) | x \neq state(hc) \wedge \forall v \in nodes(hc) : d(v) + s(v) \geq d(v, x)\}$

Von den für ein Re-Mapping möglichen Zielen $states_{feasible}(hc)$ wird das Re-Mapping auf die Hard-Core-Version mit dem größten potentiellen Beitrag zu $\phi_{ges}$ als das bestmögliche Re-Mapping für die betrachtete Instanz $hc$ gewählt.

**Definition 5.22.** $bestState(hc \in HC)$ bezeichnet von den möglichen Zielen für das Re-Mapping einer Instanz die Hard-Core-Version, auf die ein Re-Mapping den größten Beitrag zu $\phi_{ges}$ liefert. Wenn kein Re-Mapping für die Instanz möglich ist, ist $bestState(hc) = na$ (***n**ot **a**vailable*).

**Algorithmus 5.2** : Pseudocode für einen (nicht realisierten) optimalen Re-Mapping-Algorithmus

```
1 Berechne den Slack für jeden Knoten in G;
2 Ermittle eine effective budget management instance ΔD(V), die das optimale
  Re-Mapping der Netzliste HC ermöglicht;
3 for alle Instanzen hc ∈ HC do
      // Führe für jede Instanz hc ∈ HC das optimale Re-Mapping in
         Bezug auf ΔD(V) durch
4     nextStates_inBudget := {x ∈ states(hc) | ∀v ∈ nodes(hc) : d(v,x) ≤ d(v) + Δd(v)};
5     state'(hc) := max_{x∈nextStates_inBudget} p(hc, x);
```

**Algorithmus 5.3** : Ein iterativer Re-Mapping-Algorithmus

```
1  while true do
2      Berechne den Slack für jeden Knoten in G;
3      for alle Instanzen hc ∈ HC do
4          Plane für die Instanz hc das bestmögliche Re-Mapping;
5      if Der Slack erlaubt für keine Instanz hc ∈ HC ein Re-Mapping then
6          exit; // Algorithmus ist beendet
7      else
8          Ermittle eine effective budget management instance für G;
9          for alle Instanzen hc ∈ HC do
10             if Re-Mapping ist für Instanz hc ∈ HC zulässig then
11                 Führe das geplante Re-Mapping für hc durch;
12         if Es wurde für kein hc ∈ HC ein Re-Mapping durchgeführt then
13             Schliesse eines der in Zeile 4 geplanten Re-Mappings komplett aus der
               Betrachtung durch den Algorithmus aus;
```

Es gilt: $bestState(hc) := \begin{cases} max_{x \in states_{feasible}(hc)} \phi(hc, state(hc), x) & wenn\ states_{feasible} \neq \{\} \\ na & sonst \end{cases}$

Wenn für keine der Instanzen $hc \in HC$ ein Re-Mapping möglich ist, d.h. $\forall hc \in HC$ gilt $bestState(hc) = na$, wird der Algorithmus in Zeile 4 beendet, anderenfalls wird in Zeile 4 eine *effective budget management instance* ermittelt, die dann in der Regel ein Re-Mapping einer oder mehrerer der Instanzen $hc \in HC$ in der For-Schleife ab Zeile 4 ermöglicht. Durch dieses Re-Mapping ändern sich die Voraussetzungen für die nächste Iteration des Algorithmus, d.h. die in Zeile 4 in der nächsten Iteration ermittelte *effective budget management instance* weicht von der *effective budget management instance*

in der aktuellen Iteration ab, wodurch dann in der Regel in der nächsten Iteration ein weiteres Re-Mapping einer oder mehrerer der Instanzen $hc \in HC$ durchgeführt werden kann.

Wenn in der For-Schleife ab Zeile 4 kein Re-Mapping durchgeführt werden konnte, d.h. in Zeile 4 gilt $\forall hc \in HC : state(hc) \neq bestState(hc)$, werden die Voraussetzungen für die nächste Iteration des Algorithmus in Zeile 4 gezielt verändert, um ein weiteres Re-Mapping in den nächsten Iterationen des Algorithmus zu ermöglichen. Dazu wird von den in Zeile 4 geplanten bestmöglichen Re-Mappings von Instanzen $hc \in HC$ ein Re-Mapping komplett aus der Betrachtung durch den Algorithmus entfernt. Dies wird erreicht, indem zunächst die Instanz $hc \in HC$ ermittelt wird, für die das geplante Re-Mapping den geringsten potentiellen Beitrag zu $\phi_{ges}$ liefert. Das ist genau die Instanz $hc$ für die

$$\phi(hc, state(hc), bestState(hc)) = min_{hc' \in HC \wedge bestState(hc') \neq \{\}} \phi(hc', state(hc'), bestState(hc'))$$

gilt. Das geplante Re-Mapping für diese Instanz wird dann aus der Betrachtung durch den Algorithmus entfernt, indem die Menge der für diese Instanz zur Verfügung stehenden Hard-Core-Versionen um die mit $bestState(hc)$ bezeichnete Hard-Core-Version verkleinert wird. Für die nächste Iteration gilt dann $states(hc) := states(hc) \setminus bestState(hc)$.

Algorithmus 5.4 zeigt eine ausführlichere Notation des Re-Mapping-Algorithmus. Die hier angegebene Heuristik berücksichtigt die Fläche nicht als begrenzt vorhandene Ressource. FPGA-Hersteller bieten in der Regel gleichartig aufgebaute FPGAs in verschiedenen Größen, bezeichnet als FPGA-Familien, an. Wenn der Place&Route-Vorgang nach dem Re-Mapping aufgrund einer zu geringen FPGA-Fläche nicht erfolgreich ist, muss ein FPGA aus derselben FPGA-Familie mit einer größeren Fläche als das ursprüngliche Ziel-FPGA gewählt werden. Da für die Serienproduktion mit einer DT-Konfiguration defekte FPGAs verwendet werden können, liegt der Stückpreis pro FPGA auch bei einem FPGA mit größerer Fläche voraussichtlich unter dem Stückpreis für das ursprüngliche Ziel-FPGA. Eine Alternative ist der Abbruch des Re-Mapping-Algorithmus, bevor die DT-Konfiguration nicht mehr auf dem ursprünglich vorgesehenen FPGA platziert werden kann, bzw. die Mitprotokollierung und anschließende Rücknahme der zuletzt ausgeführten Re-Mappings von Instanzen $hc \in HC$.

Um die nachfolgende Erläuterung des Verfahren für die Ermittlung der *effective budget management instance* in (Zeile 5 in Algorithmus 5.4) zu vereinfachen, erfolgt nun zunächst die Betrachtung von zwei *budget management* Problemen, die bereits ausführlich erforscht wurden (Referenzen zu Veröffentlichungen über diese und weitere *budget management* Probleme findet man z.B. in [GBH+06]).

Ein einfaches *budget management* Problem ist die Ermittlung des maximal erreichbaren Wertes für die Summe des Delay-Budgets aller Knoten $v \in V$, bezeichnet als *effective budget B*. Die im Folgenden mit $\Delta_m D(V)$ bezeichnete optimale *budget management instance* für dieses Problem ist die *effective budget management instance*, die den maximal

**Algorithmus 5.4** : Der iterativer Re-Mapping-Algorithmus als ausführlicher Pseudo-Code dargestellt

1 **while** *true* **do**
2  Berechne den Slack für jeden Knoten in $G$;
3  **for** *alle Instanzen* $hc \in HC$ **do**
   `// Plane für jede Instanz` $hc$ `das bestmögliche Re-Mapping`
4  $states_{feasible}(hc) :=$
   $\{x \in states(hc) | x \neq state(hc) \wedge \forall v \in nodes(hc) : d(v) + s(v) \geq d(v, x)\}$
   `// Nur Re-Mappings für die der Slack ausreicht werden betrachtet`
   $bestState(hc) := \begin{cases} max_{x \in states_{feasible}(hc)} \phi(hc, state(hc), x) & wenn\ states_{feasible} \neq \{\} \\ na & sonst \end{cases}$
5  `// Wenn vorhanden, plane Re-Mapping mit der größten Auswirkung auf` $p(HC)$
6  **if** $\forall hc \in HC$ *gilt* $bestState(hc) = na$ **then**
7  exit; `// Algorithmus ist beendet`
8  **else**
9  Ermittle eine *effective budget management instance* $\Delta D(V)$ für $G$;
10  **for** *alle Instanzen* $hc \in HC$ *für die* $states_{feasible} \neq \{\}$ **do**
11  **if** $\forall v \in nodes(hc)$ *gilt* $d(v, bestState(v)) \leq d(v) + \Delta d(v)$ **then**
12  $state(hc) := bestState(hc)$;
13  **if** *Es wurde kein Re-Mapping durchgeführt* **then**
14  Bestimme die Instanz $hc \in HC$ für die $\phi(hc, state(hc), bestState(hc) = min_{hc' \in HC \wedge bestState(hc') \neq \{\}} \phi(hc', state(hc'), bestState(hc'))$ gilt und gebe das für die Instanz $hc$ geplante Re-Mapping auf:
  $states(hc) := states(hc) \setminus bestState(hc)$;

erreichbaren Wert für das *effective budget B* zur Folge hat. Für das maximal erreichbare *effective budget* $B_m$ gilt

$$B_m := |\Delta_m D(V)| = max_{\Delta D(V) \in \Psi} |\Delta D(V)| \tag{5.13}$$

wenn $\Psi$ die Menge aller möglichen *effective budget management instances* für $G$ ist. Abbildung 5.29 verdeutlicht diese Zusammenhänge.

Bei der Ermittlung des maximal erreichbaren Wertes für das *effective budget* wird keine Gewichtung der Knoten des Graphen $G$ vorgenommen, d.h. es wird implizit davon ausgegangen, dass ein Budget an jedem Knoten gleichermaßen für eine Optimierung

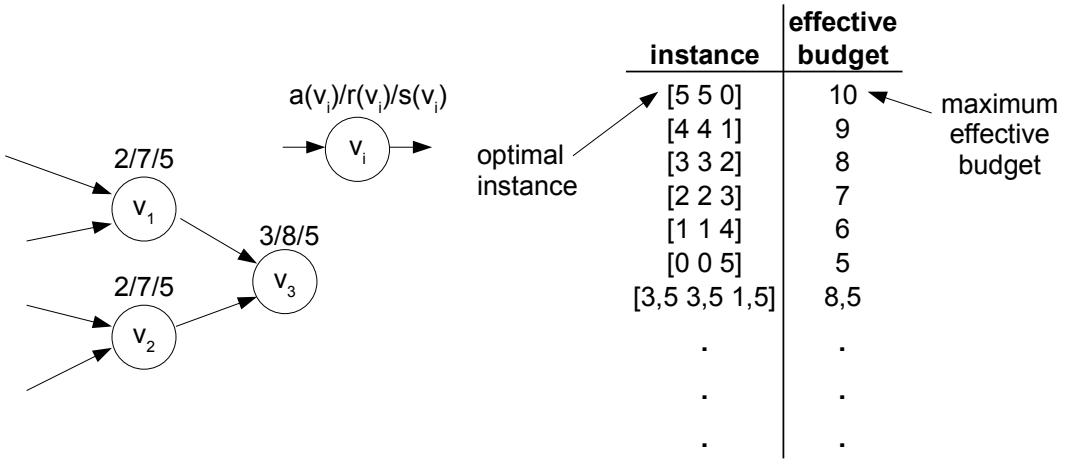



Abbildung 5.29: Mögliche "effective budget management instances" für ein Beispiel aus [CBSS02]

verwendet werden kann. Diese Annahme ist bei vielen Optimierungsproblemen, die als *budget management* Problem formuliert werden können, nicht korrekt. Wenn ein Vektor $w(V) = [\ w(v_1)\ \ w(v_2)\ \ ...\ \ w(v_n)\ ]$ Gewichte für die Knoten des Graphen $G$ enthält, kann das *budget management problem* daher alternativ mit dem gewichteten *effective budget* $\sum_{k=1}^{n} \Delta d(v_k) * w(v_k)$ formuliert werden. Eine optimale gewichtete *budget management instance* $\Delta_{wm}D(V)$ ist dann die *effective budget management instance*, die den maximal erreichbaren Wert für das gewichtete *effective budget* zur Folge hat. Für das maximal erreichbare *effective budget* $B_{wm}$ gilt

$$B_{wm} := |\Delta_{wm}D(V)| = max_{\Delta D(V) \in \Psi} \sum_{k=1}^{n} \Delta d(v_k) * w(v_k) \tag{5.14}$$

, wenn $\Psi$ die Menge aller möglichen *effective budget management instances* für $G$ ist. Beispiel: Für das in Abbildung 5.29 gezeigte Beispiel gilt für den Fall $w(v_1) = 2$, $w(v_2) = 2$ und $w(v_3) = 5$ die optimale *budget management instance* $\Delta_{wm}D(V) = [\ 0\ \ 0\ \ 5\ ]$ und somit für das maximal erreichbare gewichtete *effective budget* $B_{wm} = 5$.

Um das in Algorithmus 5.4 benötigte Budget zu berechnen, gehen wir wie in Algorithmus 5.5 angegeben vor.

In der For-Schleife ab Zeile 6 wird zunächst für jede Komponente $C(V^C, E^C)$ von $G(V, E)$ die optimale gewichtete *budget management instance* $\Delta_{wm}D(V^C)$ ermittelt. Wir definieren für $\phi(hc_v(v), state(hc_v(v)), bestState(hc_v(v)))$ die abgekürzte Schreibweise $\phi(v)$, um die Notation der für diese Ermittlung benötigten Gewichtungsfunktion zu vereinfachen.

**Algorithmus 5.5** : Die Ermittlung des Budgets in Algorithmus 5.4

```
1  for alle Komponenten C ⊆ G do
2      Initialisiere die Gewichte w(V^C) ← 0;
3      for alle Knoten v ∈ V^C do
4          if bestState(hc_v(v)) ≠ na) then
5              w(v) := log φ(v); // φ(v): siehe Definition 5.23
6          else
7              w(v) := 0;
8      Ermittle die gewichtete budget management instance Δ_wm D(V^C) für C;
9  Vereinige die budget management instances Δ_wm D(V_C) der Komponenten C ⊆ G zu der
   effective budget management instance ΔD(V) von G;
10 for alle Komponenten C ⊆ G do
       // Setze das Delay-Budget für alle Knoten der Komponente auf 0,
          wenn ein Re-Mapping trotz ausreichend vorhandenem
          Delay-Budget innerhalb der betrachteten Komponente nicht
          durchgeführt werden kann
11     for alle Knoten v ∈ V^C do
12         if d(v, bestState(v)) ≤ d(v) + Δd(v) then
13             if ∃u ∈ nodes(hc_v(v)) : d(u, bestState(u)) > d(u) + Δd(u) then
14                 for alle Knoten u ∈ V^C do
15                     Δd(u) = 0;
16                 break;
                   // Starte direkt die nächste Iteration der
                      For-Schleife ab Zeile 6
```

**Definition 5.23.** $\phi(v) := \phi(hc_v(v), state(hc_v(v)), bestState(hc_v(v)))$ bezeichnet den Beitrag zu $\phi_{ges}$, für den Fall, dass für die Instanz $hc_v(v) \in HC$ ein Re-Mapping auf die mit $bestState(hc_v(v))$ bezeichnete Hard-Core-Version erfolgt.

Für die Festlegung der Gewichte der Knoten $v \in V^C$ definieren wir die folgende Gewichtungsfunktion:

**Definition 5.24.** $w(v)$ bezeichnet das Gewicht des Knoten $v \in V$

$$\text{Es gilt } w(v) := \begin{cases} \log \phi(v) & wenn\ bestState(hc_v(v)) \neq na \\ 0 & sonst \end{cases}$$

Durch diese Gewichtung ist bei der Ermittlung der gewichteten *budget management instance* $\Delta_{wm}D(V^C)$ die Höhe des einem Knoten $v \in V^C$ zugeteilten Delay-Budgets abhängig von

- der Topologie des Schaltnetzes, in dem sich der Knoten befindet und
- dem Beitrag zu $\phi_{ges}$, den ein Re-Mapping der zu diesem Knoten gehörenden Instanz $hc_v(v) \in HC$ auf die als $bestState(hc_v(v))$ für das Re-Mapping eingeplante Hard-Core-Version hätte.

Die Logarithmisierung von $\phi(v)$ sorgt für eine additive Metrik an den Knoten $v \in V^C$ der Komponente $C(V^C, E^C)$.

Der Re-Mapping-Algorithmus soll im Folgenden an vier Beispielen erläutert werden, die unterschiedliche Abläufe im Algorithmus verursachen.

### Beispiel 1

Das laufende Beispiel soll nun als erstes Beispiel für den Ablauf des Re-Mapping-Algorithmus herangezogen werden. In der For-Schleife ab Zeile 5 des als Algorithmus 5.4 abgebildeten Re-Mapping-Algorithmus wird $bestState(hc) = 1$ für alle Instanzen $hc \in HC$ bestimmt, da der Slack für das Re-Mapping aller Instanzen ausreichend ist. Als Basis für die Ermittlung der *budget management instance* werden daraufhin in Algorithmus 5.5 in der ersten Iteration des Re-Mapping-Algorithmus die Gewichte wie in Tabelle 5.3 gezeigt ermittelt.

Der Graph $G = (V, E)$ für das laufende Beispiel enthält zwei Komponenten, im Folgenden bezeichnet als

$C_1 = (V_1^C, E_1^C)$; $V_1^C = \{A_{p1}, B_{p1}, C_{p1}, S_1\}$; $E_1^C = \{(A_{p1}, B_{p1}), (A_{p1}, C_{p1}), (C_{p1}, S_1)\}$

und

$C_2 = (V_2^C, E_2^C)$; $V_2^C = \{S_2, C_{p2}, D_{p1}, E_{p1}, S_3\}$; $E_2^C = \{(S_2, C_{p2}), (C_{p2}, D_{p1}), (E_{p1}, D_{p1}), (D_{p1}, S_3)\}$

Für diese Komponenten wird in Algorithmus 5.5 jeweils in Zeile 6 eine gewichtete *budget management instance* ermittelt. Diese Ermittlung kann z.B., wie nachfolgend dargestellt, mit dem in [CBSS02] vorgeschlagenen MISA-Algorithmus erfolgen.

Abbildung 5.30 skizziert die Ermittlung einer gewichteten *budget management instance* mit dem MISA-Algorithmus für die Komponente $C_2$. Wie in Abbildung 5.30 (a) skizziert, wird für diesen Graphen die Differenz $\epsilon$ von dem größten Slack, der an einem

| v | $A_{p1}$ | $B_{p1}$ | $C_{p1}$ | $S_1$ | $S_2$ | $C_{p2}$ | $D_{p1}$ | $S_3$ | $E_{p1}$ |
|---|---|---|---|---|---|---|---|---|---|
| w(v) | 0,032 | 0,024 | 0,024 | 0 | 0 | 0,024 | 0,032 | 0 | 0,024 |

Tabelle 5.3: Die Gewichte $w(v)$ für alle $v \in V$ in der ersten Iteration zu Beispiel 1

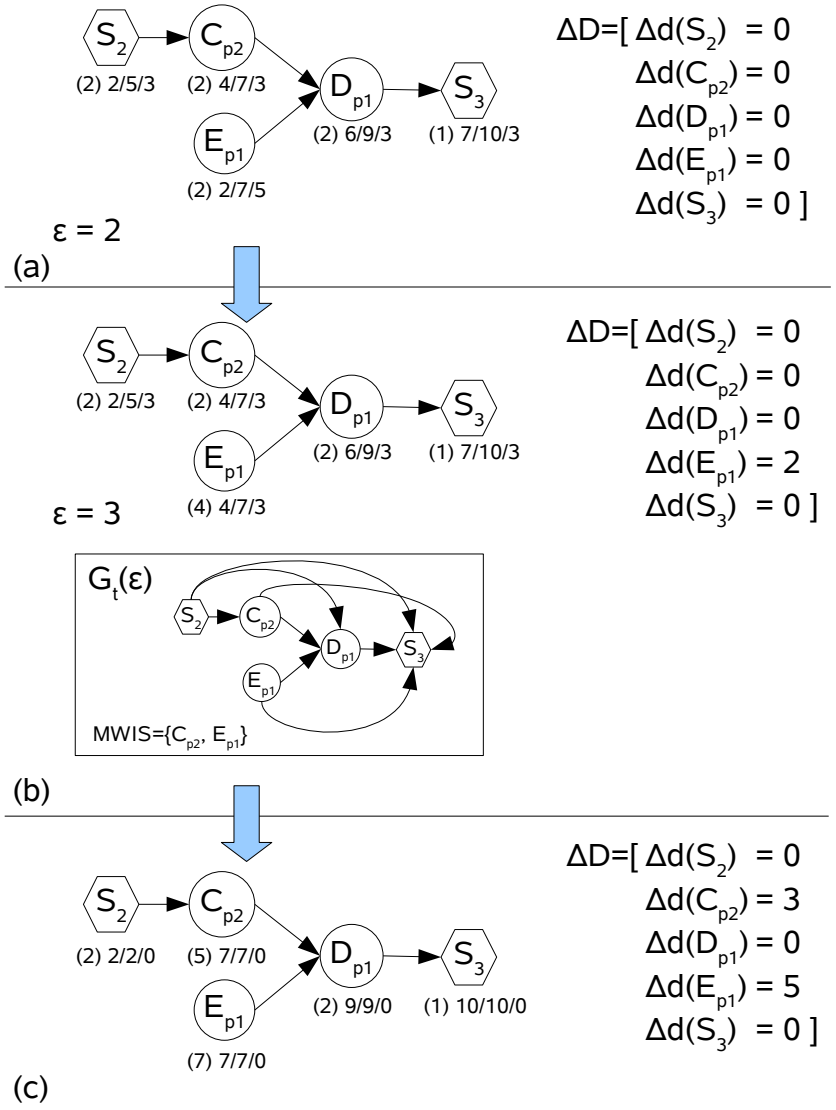



Abbildung 5.30: Der MISA-Algorithmus [CBSS02] wird für den Graphen $C_2 \subset G$ durchgeführt

Knoten annotiert wurde, in diesem Fall dem Wert 5, und dem zweitgrößten Slack, hier dem Wert 3, gebildet. In [CBSS02] wird ausführlich gezeigt, dass an den Knoten, an denen nicht der größte Slack annotiert ist, also in 5.30 (a) an den Knoten $S_2, C_{p2}, D_{p1}$ und $S_3$ der Slack *nicht* reduziert wird, wenn an den Knoten mit dem größten Slack, hier nur $E_{p1}$, der Slack um einen Wert $\leq \epsilon$ reduziert wird. Als Ergebnis der ersten Iteration des MISA-Algorithmus wird daher dem Knoten $E_{p1}$ der volle Umfang von $\epsilon$ als Delay-Budget zugewiesen. Zu Beginn der zweiten Iteration gilt

$\Delta_{wm}D(V^C) = [\ \Delta d(S_2) = 0 \quad \Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 0 \quad \Delta d(E_{p1}) = 2 \quad \Delta d(S_3) = 0\ ]$

In der zweiten Iteration des MISA-Algorithmus ist an jedem Knoten der gleiche Slack annotiert. Für diesen Fall ist der zweitgrößte Slack mit 0 definiert und $\epsilon$ entspricht, wie in Abbildung 5.30 (b) skizziert, dem an den Knoten annotierten Slack. Da hier an mehr als einem Knoten der „größte“ Slack annotiert ist, darf an allen Knoten der Slack in der Schaltung *insgesamt* nur um einen Wert $\leq \epsilon$ reduziert werden. Da eine Auswirkung der Reduzierung des Slacks an einem Knoten $v$ auf einen Knoten $u$ nur dann möglich ist, wenn es entweder einen Pfad von v nach u oder einen Pfad von u nach v gibt, kann der Slack an voneinander unabhängigen Knoten *gleichzeitig* reduziert werden. Für den Graphen in Abbildung 5.30 (b) kann der Slack somit einzeln an jedem Knoten, oder gleichzeitig an den Knoten $S_2$ und $E_{p1}$oder gleichzeitig an den Knoten $C_{p2}$ und $E_{p1}$ reduziert werden. Durch die oben definierte Gewichtungsfunktion, wird von den unabhängigen Mengen $\{S_2\}, \{C_{p2}\}, \{E_{p1}\}, \{D_{p1}\}, \{S_3\}, \{S_2, E_{p1}\}$ und $\{C_{p2}, E_{p1}\}$ die Menge $\{C_{p2}, E_{p1}\}$ ausgewählt, da sich nach der Aufsummierung der Gewichte für die Knoten dieser Menge mit $w(C_{p2}) + w(E_{p1}) = 0,047$ das höchste Gewicht ergibt. Diese Menge ist somit die maximal gewichtete unabhängige Menge (In [CBSS02] als MWIS (maximum weighted independent set) bezeichnet)[2]. Als Ergebnis der zweiten Iteration des MISA-Algorithmus wird den Knoten $E_{p1}$ und $C_{p2}$ der volle Umfang von $\epsilon$ als Delay-Budget zugewiesen. Zu Beginn der dritten Iteration gilt

$\Delta_{wm}D(V^C) = [\ \Delta d(S_2) = 0 \quad \Delta d(C_{p2}) = 3 \quad \Delta d(D_{p1}) = 0 \quad \Delta d(E_{p1}) = 5 \quad \Delta d(S_3) = 0\ ]$

Und der MISA-Algorithmus wird beendet, da kein weiterer Slack in der Schaltung vorhanden ist. Abbildung 5.31 skizziert die Durchführung des MISA-Algorithmus für die Komponente $C_1 \subset G$. Für eine ausführliche Erläuterung des MISA-Algorithmus sei an dieser Stelle auf die Veröffentlichung [CBSS02] zu diesem Algorithmus verwiesen.

Die für die Komponenten des Graphen $G(V, E)$ ermittelten gewichteten *budget management instances* werden in Algorithmus 5.5 in Zeile 6 zunächst zu einer *budget management instance* $\Delta D(V)$ des gesamten Graphen $G(V, E)$ vereinigt. Für das laufende Beispiel entsteht somit die *budget management instance*

$$\begin{aligned}\Delta D(V) = \ [\ & \Delta d(A_{p1}) = 0 \quad \Delta d(B_{p1}) = 7 \quad \Delta d(C_{p1}) = 5 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0 \\ & \Delta d(C_{p2}) = 3 \quad \Delta d(D_{p1}) = 0 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 5 \qquad ]\end{aligned}$$

In der For-Schleife ab Zeile 6 in Algorithmus 6 wird für jede Komponente überprüft, ob ein Re-Mapping, für das ein ausreichendes Delay-Budget innerhalb der Komponente vorliegt, aufgrund eines nicht ausreichenden Delay-Budgets an einem Knoten außerhalb der Komponente nicht durchgeführt werden kann.

[2]Da das Problem der Ermittlung dieser Menge für generelle Graphen NP-Hard ist, wird in [CBSS02] zunächst ein transitiver Graph $G_t(\epsilon)$ erzeugt. Die Ermittlung der maximal gewichteten Unabhängigen Menge ist dann optimal in $O(ne \log(n^2/e))$ Schritten lösbar, wobei $n$ die Anzahl der Knoten und $e$ die Anzahl der Kanten in dem Graphen bezeichnet. Für die Verbesserung der Laufzeit wird in [CBSS02] eine Heuristik vorgeschlagen, die in [Möh85] beschrieben wird und maximal $O(n \log n)$ Schritte benötigt.

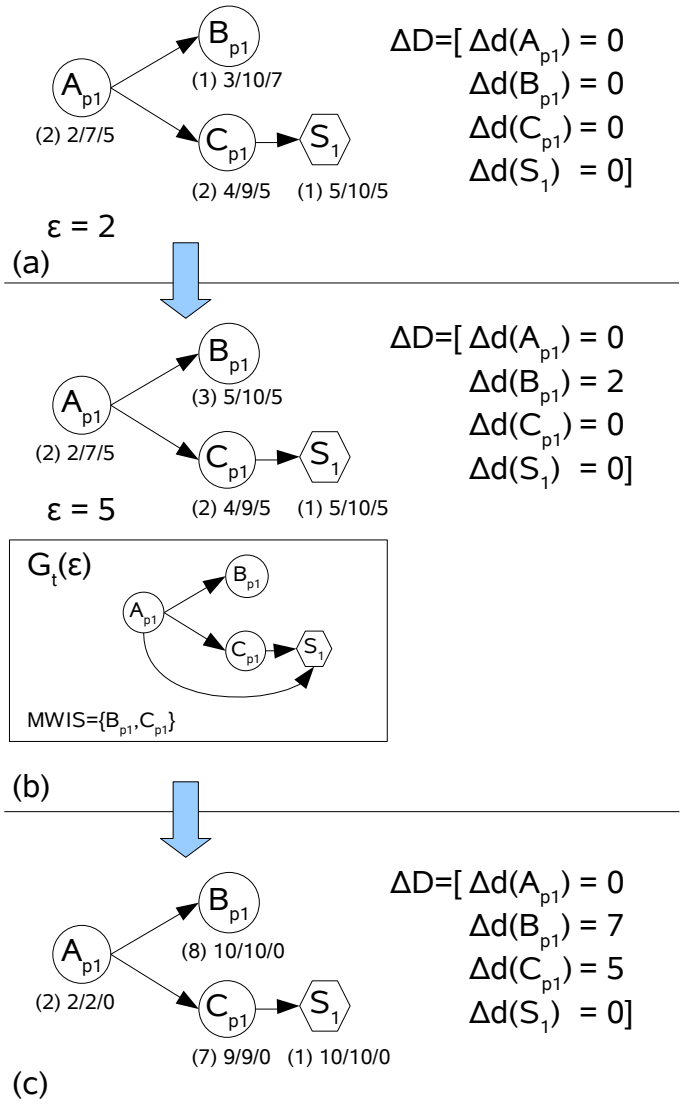



Abbildung 5.31: Der MISA-Algorithmus [CBSS02] wird für den Graphen $C_1 \subset G$ durchgeführt.

Für das Re-Mapping der Instanzen B und C ist das Delay-Budget innerhalb der Komponente $C_1$ des laufenden Beispiels ausreichend, da

$d(B_{p1}, 1) < d(B_{p1}, 0) + \Delta d(B_{p1})$ und $d(C_{p1}, 1) < d(C_{p1}, 0) + \Delta d(C_{p1})$ gilt (vgl. Tabelle 5.1 auf Seite 83). Für das Re-Mapping

$state(B) = 1$ und $state(C) = 1$

liegt daher innerhalb der Komponente $C_1$ ein ausreichendes Delay-Budget vor. Der ein-

zige Knoten, der von diesem Re-Mapping betroffen ist und außerhalb der Komponente $C_1$ liegt, ist der Knoten $C_{p2}$. Auch das Delay an diesem Knoten ist für das Re-Mapping ausreichend, da $d(C_{p2}, 1) < d(C_{p2}, 0) + \Delta d(C_{p2})$ gilt. Die For-Schleife ab Zeile 6 wird somit nicht ausgeführt.

Auch bei der Überprüfung der Komponente $C_2$ wird die For-Schleife ab Zeile 6 nicht ausgeführt. Innerhalb dieser Komponente liegt für das Re-Mapping

$state(C) = 1$ und $state(E) = 1$

ein ausreichendes Delay-Budget vor. Der einzige Knoten, der von diesem Re-Mapping betroffen ist und außerhalb der Komponente $C_2$ liegt, ist der Knoten $C_{p1}$. Auch das Delay an diesem Knoten ist für das Re-Mapping ausreichend, da $d(C_{p1}, 1) < d(C_{p1}, 0) + \Delta d(C_{p1})$ gilt.

Nach der Ausführung von Algorithmus 5.5 wird ab Zeile 5 in Algorithmus 5.4 für jede Instanz $hc \in HC$ das geplante Re-Mapping $state(hc) := bestState(hc)$ durchgeführt, wenn die ermittelte *budget management instance* dies zulässt. Das Re-Mapping für das laufende Beispiel entspricht dem Re-Mapping aus Abbildung 5.25 auf Seite 90 und somit dem optimalen Re-Mapping für das vorliegende Problem. Da in der nächsten Iteration des Re-Mapping-Algorithmus für keine Instanz $hc \in HC$ ein weiteres Re-Mapping durchgeführt werden kann, wird der Algorithmus in Zeile 5 beendet. Abbildung 5.32 auf der nächsten Seite skizziert den Ablauf des Re-Mapping-Algorithmus für das erste Beispiel.

### Beispiel 2

Um die Arbeitsweise des Algorithmus über mehrere Iterationen und die Auswirkung der For-Schleife ab Zeile 6 in Algorithmus 5.5 zu verdeutlichen, soll das laufende Beispiel leicht verändert werden. Für Beispiel 2 sollen, abweichend von den in Tabelle 5.2 auf Seite 92 angegebenen Werten, die in Tabelle 5.4 angegebenen Werte gelten. Aus der Veränderung von $p(D, 0DT)$ entsteht eine Veränderung für $\phi(D, 0, 1)$, die sich im weiteren Verlauf des Algorithmus auf das Gewicht des Knoten $D_{p1}$ auswirkt. Abbildung 5.33 zeigt den Ablauf des Re-Mapping-Algorithmus unter diesen Voraussetzungen.

In Beispiel 2 wird aufgrund der veränderten Gewichtsverteilung für die Komponente $C_2$ die gewichtete *budget management instance*

$$\Delta_{wm}D(V_2^C) = [\ \Delta d(S_2) = 0 \quad \Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 3 \quad \Delta d(E_{p1}) = 2 \quad \Delta d(S_3) = 0\ ]$$

ermittelt und mit der im Vergleich zu Beispiel 1 unveränderten gewichteten *budget management instance* $\Delta_{wm}D(V_1^C)$ zu der *effective budget management instance*

$$\begin{aligned}\Delta D(V) = [\ & \Delta d(A_{p1}) = 0 \quad \Delta d(B_{p1}) = 7 \quad \Delta d(C_{p1}) = 5 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0 \\ & \Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 3 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 2 \quad ]\end{aligned}$$

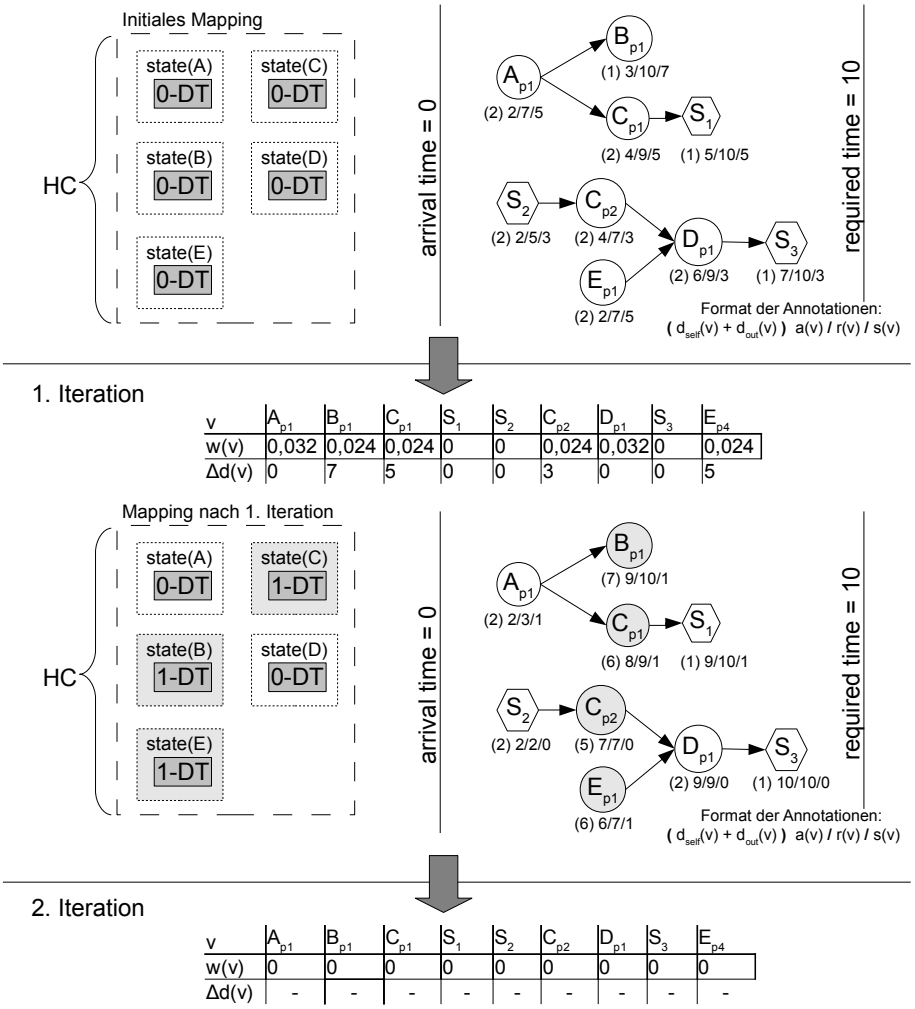



Abbildung 5.32: Der Ablauf des Re-Mapping-Algorithmus für Beispiel 1

vereinigt. Das optimale Re-Mapping in Bezug auf diese *effective budget management instance* zeigt Abbildung 5.28 auf Seite 97. Da für das Re-Mapping $state(B_{p1}) = 1$ und $state(C_{p1}) = 1$ innerhalb der Komponente $C_1$ ein ausreichendes Delay-Budget vorliegt, aber ein Re-Mapping der Instanz C nicht möglich ist, weil $d(C_{p2}, 1) > d(C_{p2}, 0) + \Delta d(C_{p2})$ gilt, wird innerhalb der For-Schleife ab Zeile 6 in Algorithmus 5.5 bei der Betrachtung der Komponente $C_1$ das Delay-Budget für alle Knoten $v \in V_1^C$ auf 0 gesetzt. Die veränderte *budget management instance*

$$\Delta D(V) = [\ \Delta d(A_{p1}) = 0 \quad \Delta d(B_{p1}) = 0 \quad \Delta d(C_{p1}) = 0 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0$$
$$\Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 3 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 2 \quad ]$$

lässt daraufhin in der ersten Iteration des Re-Mapping-Algorithmus nur ein Re-Mapping der Instanz $D$ zu. In der zweiten Iteration des Re-Mapping-Algorithmus reicht der Slack

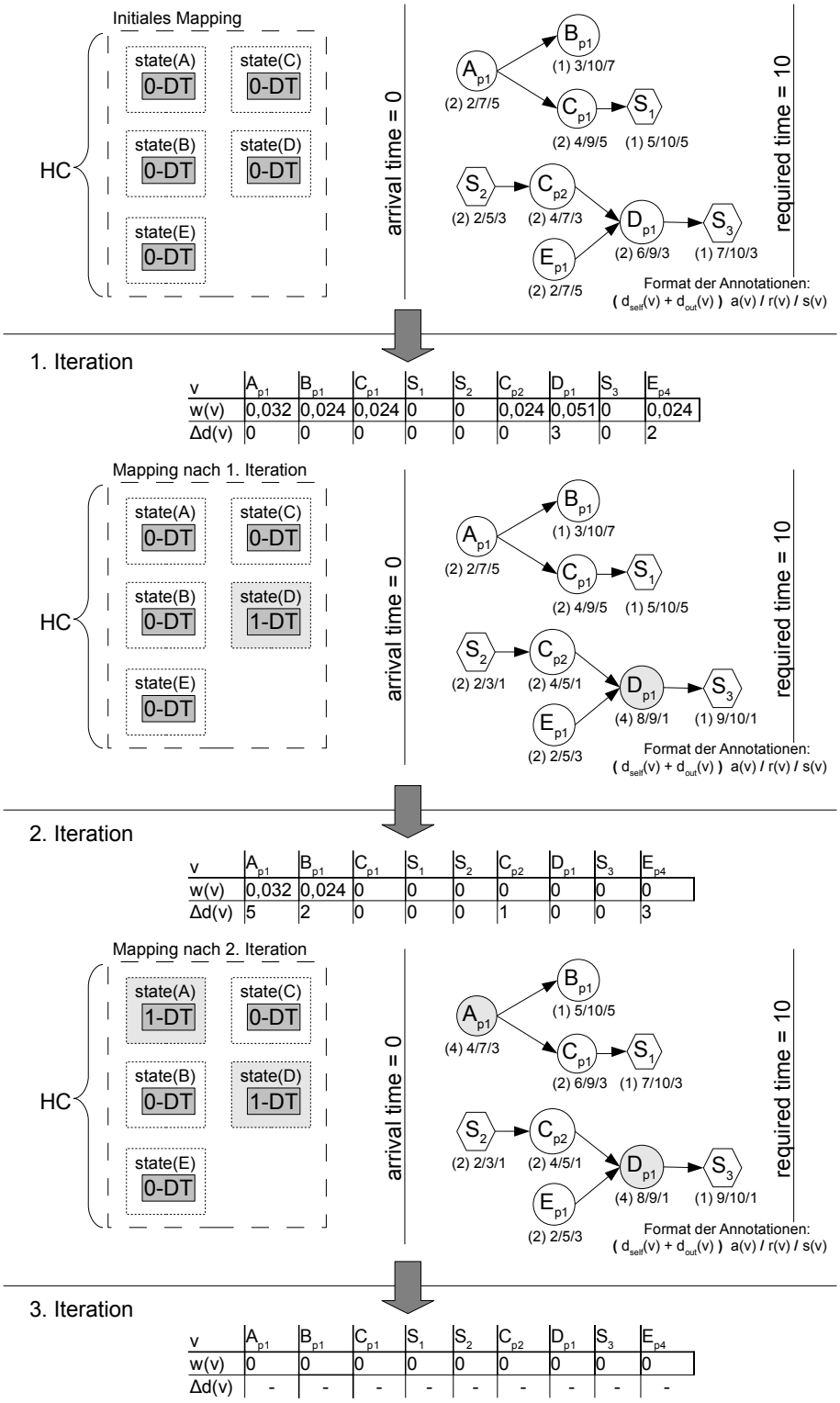



Abbildung 5.33: Der Ablauf des Re-Mapping-Algorithmus für Beispiel 2

| $hc \in HC$ | $p(hc, 0\,DT)$ | $p(hc, 1\,DT)$ | $\phi(hc, 0, 1) := \frac{p(hc, 1\,DT)}{p(hc, 0\,DT)}$ |
|---|---|---|---|
| A | 0,92 | 0,99 | 1,076 |
| B | 0,9 | 0,95 | 1,056 |
| C | 0,9 | 0,95 | 1,056 |
| **D** | **0,88** | **0,99** | **1,125** |
| E | 0,9 | 0,95 | 1,056 |

Tabelle 5.4: Zu Beispiel 2: $p(hc, 0DT)$ und $p(hc, 1DT)$ sowie der daraus resultierende Faktor $\phi(hc, 0, 1)$ für alle Instanzen $hc \in HC$

an den Knoten des Graphen $G$ nur für das Re-Mapping der Instanzen $A$ und $B$. In Algorithmus 5.5 wird daher die *budget management instance*

$$\begin{aligned}\Delta D(V) = \; [ \;& \Delta d(A_{p1}) = 5 \quad \Delta d(B_{p1}) = 2 \quad \Delta d(C_{p1}) = 1 \quad \Delta d(S_1) = 0 \quad \Delta d(S_2) = 0 \\ & \Delta d(C_{p2}) = 0 \quad \Delta d(D_{p1}) = 3 \quad \Delta d(S_3) = 0 \quad \Delta d(E_{p1}) = 3 \quad ]\end{aligned}$$

ermittelt[3]. Die den Knoten zugewiesenen Delay-Budgets lassen nun ein Re-Mapping der Instanz A zu. Da in der nächsten Iteration des Re-Mapping-Algorithmus für keine Instanz $hc \in HC$ ein weiteres Re-Mapping aufgrund des vorhandenen Slacks möglich ist, wird der Re-Algorithmus beendet.

### Beispiel 3

Ein weitere interessanter Ablauf des Re-Mapping-Algorithmus entsteht, wenn zusätzlich zu den für Beispiel 2 erfolgten Änderungen gegenüber dem laufenden Beispiel eine Veränderung von $d(C_{p2}, 1)$ und daraus resultierend von$\Delta d(C_{p2}, 0, 1) = 1$ entsprechend den Angaben in Tabelle 5.5 vorgenommen wird.

Abbildung 5.34 zeigt den Ablauf des Re-Mapping-Algorithmus für Beispiel 3. In der zweiten Iteration sorgt die Gewichtsverteilung dafür, dass der verbleibende Slack - abweichend von Beispiel 2 - für das Re-Mapping der Instanzen $B$ und $C$ genutzt wird.

[3]Die Delay-Budgets für die Knoten $v \in V_2^C$ können sich je nach Vorgehen des MISA-Algorithmus bei gleichgewichteten unabhängigen Knotenmengen von den hier angenommenen unterscheiden. Da diese Budgets in keinem Fall für ein Re-Mapping ausreichend sind, ändert sich für den Re-Mapping-Algorithmus auch bei einer abweichenden Budget-Verteilung für diese Knoten nichts.

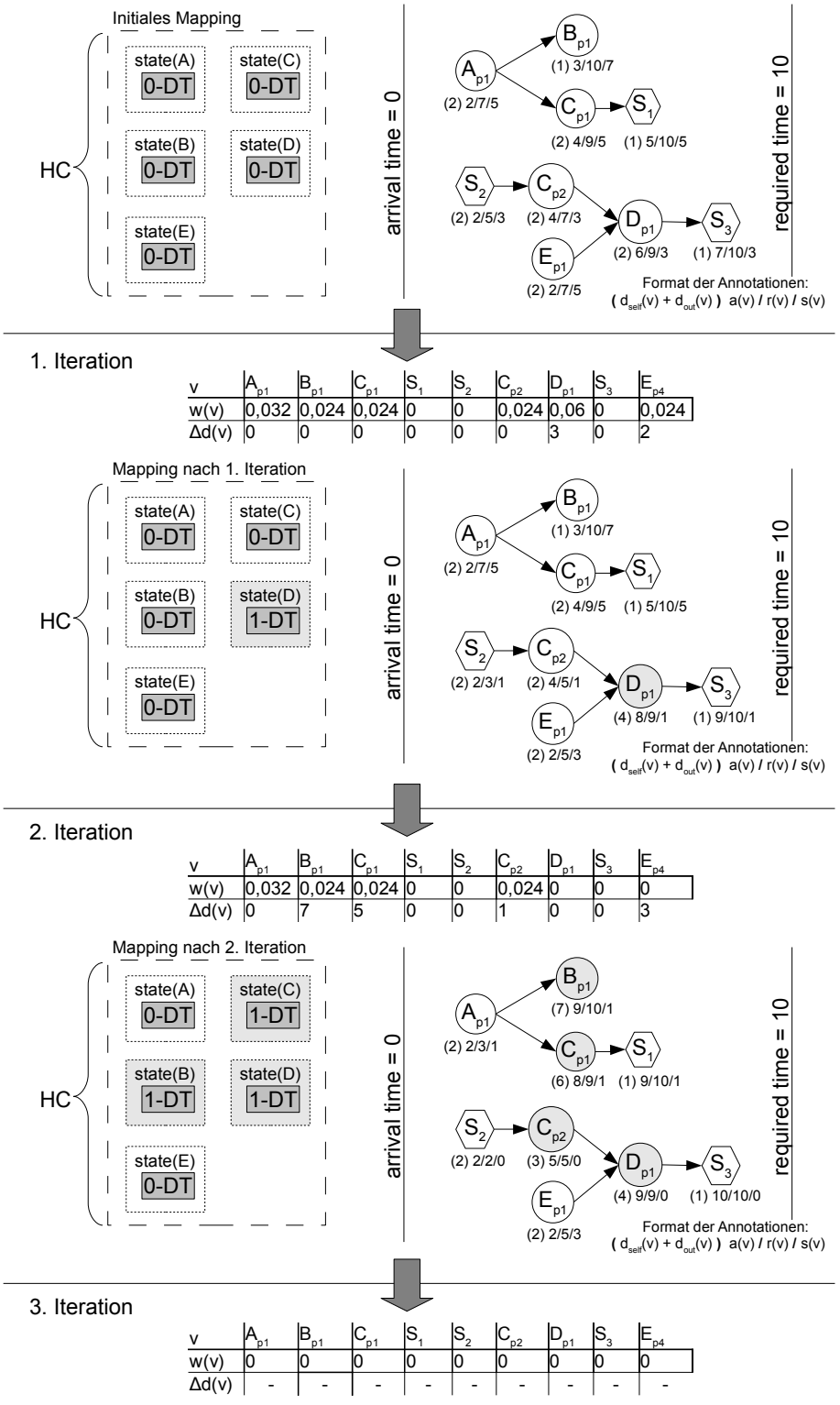



Abbildung 5.34: Der Ablauf des Re-Mapping-Algorithmus für Beispiel 3

| Pfad<br>$v \in V_{dyn}$ | Delay in 0-DT-Version von v<br>$d(v,0)$ | Delay in 1-DT-Version von v<br>$d(v,1)$ | Zusätzlich erforderliches Delay für eine Änderung von $state(hc_v(v))$<br>$\Delta d(v,0,1) := d(v,1) - d(v,0)$ |
|---|---|---|---|
| $A_{p1}$ | 1 | 3 | 2 |
| $B_{p1}$ | 1 | 7 | 6 |
| $C_{p1}$ | 1 | 5 | 4 |
| $C_{p2}$ | **1** | **2** | **1** |
| $D_{p1}$ | 1 | 3 | 2 |
| $E_{p1}$ | 1 | 5 | 4 |

Tabelle 5.5: Zu Beispiel 3: Die Verzögerungszeiten der Knoten $v \in V_{dyn}$ und das zusätzlich erforderliche Delay für ein Re-Mapping

### Beispiel 4

Der Re-Mapping-Algorithmus hat für alle bisher gezeigten Beispiele das optimale Re-Mapping nach Definition 5.14 auf Seite 93 ermittelt. Die Heuristik liefert aber für umfangreichere Graphen in der Regel keine optimale Lösung, da bei der Vergabe des Delay-Budgets an einen Knoten im Re-Mapping-Algorithmus nur die Komponente des Graphen betrachtet wird, in der sich dieser Knoten befindet. Im Folgenden wird zur Veranschaulichung ein einfaches Beispiel konstruiert, für das der Re-Mapping-Algorithmus ein Re-Mapping ermittelt, das nicht optimal ist.

Wir verändern dazu das laufende Beispiel erneut und ersetzen einige der in Tabelle 5.2 auf Seite 92 angegebenen Werte. Abbildung 5.35 zeigt den Ablauf des Re-Mapping-Algorithmus für Beispiel 4. Die in Tabelle 5.6 angegebenen Werte führen zu der in Abbildung 5.35 in der 1. Iteration angegebenen Gewichtung.

Wie bereits zu Beginn von Abschnitt 5.3.3 erläutert, gibt es für das laufende Beispiel verschiedene Möglichkeiten für ein Re-Mapping, das nach Definition 5.9 auf Seite 83 timing-closure friendly ist. Eine Möglichkeit für ein Re-Mapping zeigt Abbildung 5.25, eine andere Möglichkeit wurde in Abbildung 5.26 vorgestellt.

Wie leicht überprüft werden kann, ergibt sich der maximal mögliche Wert für $\phi_{ges}$ mit den veränderten Werten aus Tabelle 5.6 und somit das optimale Re-Mapping nach Definition 5.14 auf Seite 93, wenn das Re-Mapping entsprechend Abbildung 5.26 durchgeführt wird. In diesem Fall gilt

$\phi_{ges} = \phi(A,0,1) * \phi(D,0,1) = 1,1^2 = 1,21.$

| $hc \in HC$ | $p(hc, 0\,DT)$ | $p(hc, 1\,DT)$ | $\phi(hc, 0, 1) := \frac{p(hc, 1\,DT)}{p(hc, 0\,DT)}$ |
|---|---|---|---|
| **A** | **0,9** | **0,99** | **1,1** |
| B | 0,9 | 0,95 | 1,056 |
| C | 0,9 | 0,95 | 1,056 |
| **D** | **0,9** | **0,99** | **1,1** |
| E | 0,9 | 0,95 | 1,056 |

Tabelle 5.6: Zu Beispiel 4: $p(hc, 0DT)$ und $p(hc, 1DT)$ sowie der daraus resultierende Faktor $\phi(hc, 0, 1)$ für alle Instanzen $hc \in HC$

Das über den Re-Mapping-Algorithmus ermittelte Re-Mapping (siehe Abbildung 5.35) ist nicht optimal nach Definition 5.14. Für $\phi_{ges}$ gilt bei diesem Re-Mapping

$\phi_{ges} = \phi(B, 0, 1) * \phi(C, 0, 1) * \phi(E, 0, 1) = 1,056^3 \approx 1,18$

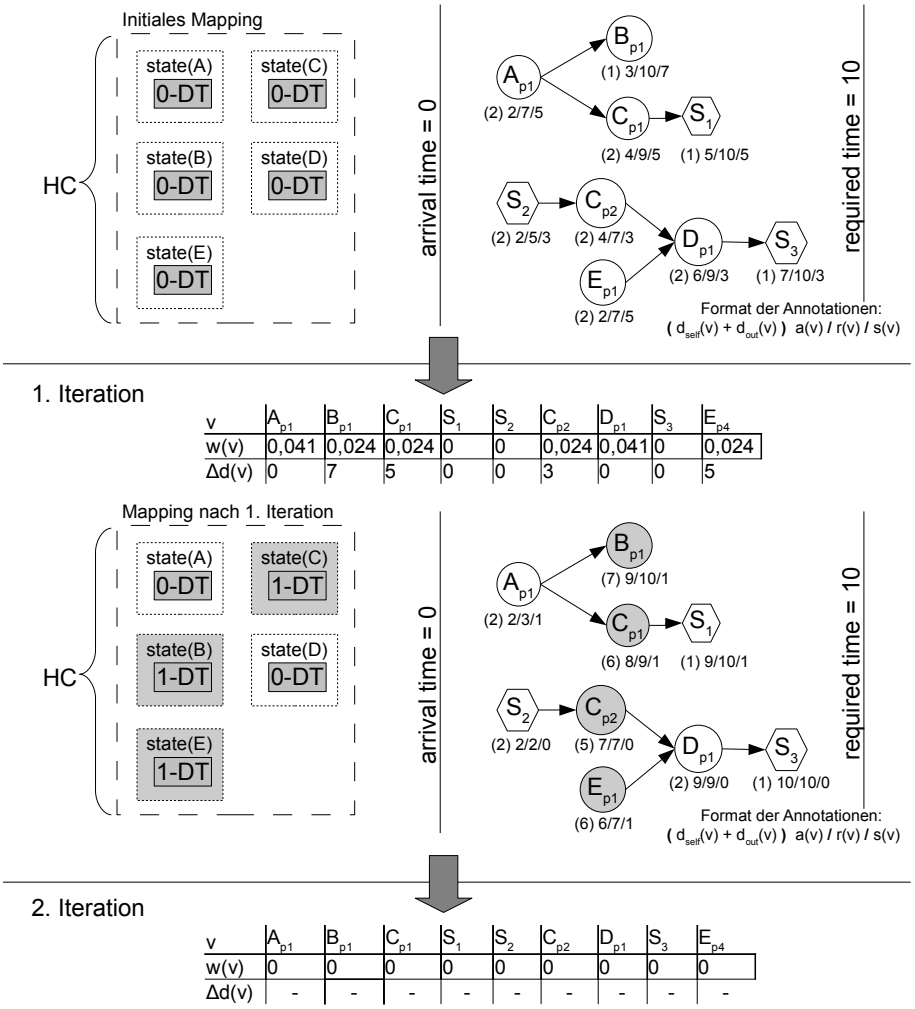



Abbildung 5.35: Der Ablauf des Re-Mapping-Algorithmus für Beispiel 4

## 5.4 Programmierung teildefekter FPGAs in der Serienproduktion

Wenn in der Serienproduktion für jedes FPGA eine Defect Map vorliegt, kann - anhand der über die in die DT-Konfiguration eingesetzten DT-Hard-Core-Versionen bekannten Informationen - ermittelt werden, ob die Defekte ausgeglichen werden können. Wenn dies der Fall ist, muss eine Auswahl der speziell für das vorliegende FPGA einzusetzenden Varianten der Versionen erfolgen. Beide Schritte können bei dem Hard-Core basierten Verfahren, wie im Folgenden dargestellt, mit einem geringen Zeitaufwand durchgeführt werden.

Wie in Abbildung 5.36 skizziert, können n-DT-Versionen in einigen Fällen mehr als n defekte Logikblöcke ausgleichen. Bei der hier skizzierten 1-DT-Version, ein Beispiel, das bereits in Abbildung 5.9 auf Seite 67 zur Verdeutlichung der Erstellung einer n-DT-Version verwendet wurde, kommunizieren die rund dargestellten Logikblöcke über spezielle Direktverbindungen miteinander und müssen daher direkt übereinander angeordnet sein. Die erstellten Varianten für diese 1-DT-Version erlauben in jedem Fall den Ausgleich von *einem* defekten Logikblock. Es können allerdings in günstigen Fällen bis zu vier defekte Logikblöcke ausgeglichen werden. Für die Überprüfung, ob ein FPGA mit defekten Logikblöcken genutzt werden kann, ist es daher nicht ausreichend, die Anzahl der Defekte innerhalb eines n-DT-Bereiches in der DT-Konfiguration zu überprüfen. Die zu jedem Logikblock gespeicherte Information, welche Varianten für den Fall, dass dieser Logikblock defekt ist, genutzt werden können, ist in Abbildung 5.36 in Form einer Tabelle dargestellt. Diese Informationen, die aus Abbildung 5.10 auf Seite 69 übernommen wurden, können verwendet werden, um über die Bildung einer Schnittmenge die einzusetzende Variante zu ermitteln. Es sei

- $D$ die Menge aller Defekte $d \in D$ die in der Defect Map verzeichnet sind,
- $I$ die Menge aller Instanzen $i \in I$ von Hard-Core-Versionen, die in der DT-Konfiguration enthalten sind,
- $defectsIn(i) \subseteq D$ eine Menge, in der alle Defekte enthalten sind, die im Bereich der Instanz $i \in I$ liegen,
- $X_{min}(i)$ ($Y_{min}(i)$) die am weitesten links (oben) und $X_{max}(i)$ ($Y_{max}(i)$) die am weitesten rechts (unten) gelegene Koordinate einer Instanz $i \in I$,
- $X(d)$ ($Y(d)$) die x-(y-)Koordinate eines Defektes $d \in D$ und
- $lookup(i, X, Y)$ die Menge aller Varianten der Instanz $i \in I$, die den Logikblock an der angegebenen Position nicht verwenden.

Die Menge der Varianten, die verwendet werden kann, um die in einer Instanz $i \in I$ liegenden defekten Logikblöcke $defectsIn(i) \subseteq D$ auszugleichen, kann dann als

$$\begin{aligned}\{&lookup(i, X(d_1) - X_{min}(i), Y(d_1) - Y_{min}(i)) \cap \\ &lookup(i, X(d_2) - X_{min}(i), Y(d_2) - Y_{min}(i)) \cap \\ &\ldots \cap \\ &lookup(i, X(d_n) - X_{min}(i), Y(d_n) - Y_{min}(i))\}\end{aligned}$$

beschrieben werden, wenn es $n$ Defekte im Bereich der Instanz gibt.

**Beispiel:**

Für die in Abbildung 5.36 dargestellte 1-DT-Version kann, bei der abgebildeten Platzierung einer Instanz $i$ dieser Version in der DT-Konfiguration (derart dass $X_{min}(i) = 6$ und $Y_{min}(i) = 2$ gilt), die zweite Variante genutzt werden, wenn auf dem FPGA defekte Logikblöcke an den abgebildeten Positionen $d1 : X7Y2$, $d2 : X7Y5$ und $d3 : X3Y7$ vorliegen. In diesem Fall befinden sich zwei Defekte im Bereich der Instanz, somit gilt:

$$\begin{aligned} \{lookup(i, 7-6, 2-2) \cap lookup(i, 7-6, 5-2)\} = \\ \{lookup(i, 1, 0) \cap lookup(i, 1, 3)\} = \\ \{\{2\} \cap \{2; 4\}\} = \{2\} \end{aligned}$$

Der Algorithmus 7 überprüft, ob ein FPGA mit defekten Logikblöcken genutzt werden kann und ermittelt die für die in der DT-Konfiguration enthaltenen Instanzen einsetzbaren Varianten. Die folgenden Vereinbarungen erleichtern die Darstellung des Algorithmus als Pseudocode:

- $variante(i)$ bezeichnet die in der DT-Konfiguration eingesetzte Variante der Instanz $i \in I$ .
  Beispiel: Für die in Abbildung 5.36 dargestellte Instanz $i$ gilt vor dem Austausch der Variante $variante(i) = 1$ und nach dem Austausch $variante(i) = 2$.
- $blockStatus(i, v, X, Y)$ gibt „used“ zurück, wenn der Logikblock der Instanz $i$ in der Variante $v$ an der angegebenen Position verwendet wird, sonst „unused“.
  Beispiel: Für die Instanz in Abbildung 5.26 gilt $blockStatus(i, 1, 0, 0) = unused$, da der Logikblock $X0Y0$ in der Variante 1 nicht verwendet wird.
- $head(X)$ bezeichnet das erste Element einer Liste $X$
  Beispiel: Für die Instanz in Abbildung 5.36 kann die erste der unter $lookup(i, 1, 3)$ gespeicherten Varianten, die den Logikblock $X1Y3$ nicht nutzen, angegeben werden mit $head(lookup(i, 1, 3)) = head(2, 4) = 2$.
- Die Partitionen einer partitionierten Version (vgl. Abbildung 5.7 auf Seite 65) werden in der DT-Konfiguration als einzelne Instanzen betrachtet.

Nach der Initialisierung der Mengen $defectsIn(i)$ und $variante(i)$ für alle $i \in I$ wird in der For-Schleife in den Zeilen 7-15 jeder Defekt genau der Instanz $i \in I$ zugeordnet, in deren Bereich er sich befindet. Die Anweisung „break;“ in Zeile 12 sorgt dafür, dass nach der Zuordung eines Defektes zu einer Instanz die innere For-Schleife beendet wird, da jeder Defekt nur innerhalb genau einer Instanz auftreten kann.

Wenn sich nur ein Defekt in dem Bereich einer Instanz befindet, ist die Ermittlung einer Variante, die diesen Defekt ausgleichen kann, aufgrund der gespeicherten Varianten

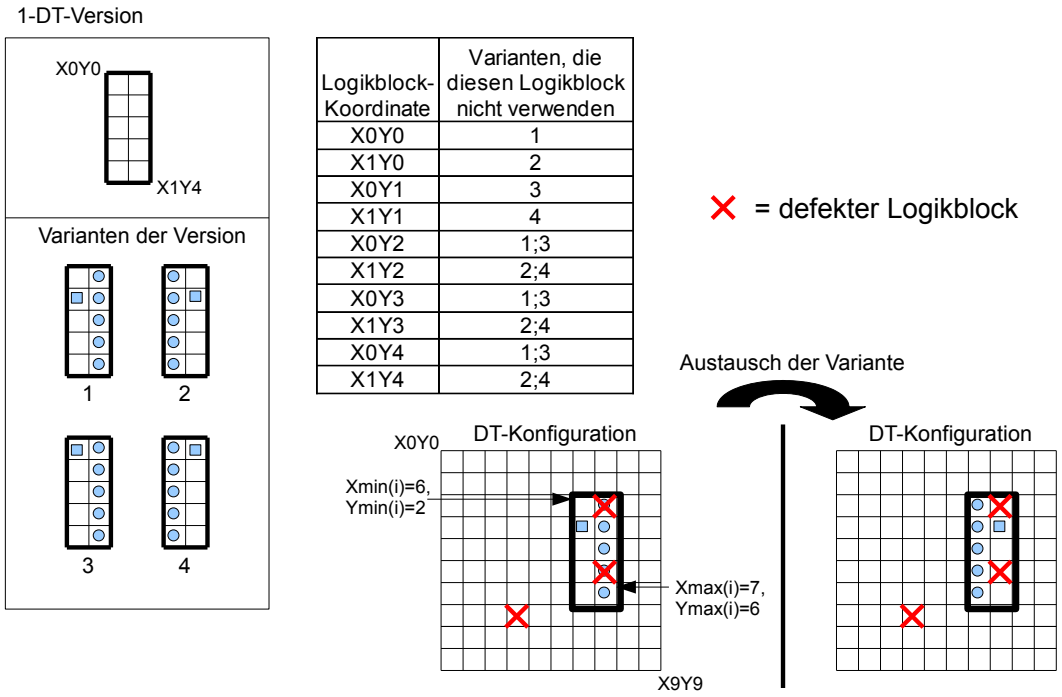



Abbildung 5.36: Die Spezialisierung der DT-Konfiguration durch den Austausch der Varianten für die Hard-Core-Version aus Abbildung 5.9

in *lookup*(.), durch eine einfache Errechnung der Koordinaten innerhalb der Instanz und der Auswahl der ersten aufgeführten Variante mit *head*(*lookup*(.)) möglich (Zeilen 18 - 21). Wenn mehr als ein Defekt innerhalb des Bereiches der Instanz liegen (d.h. $|defectsIn(i)| > 1$), wird der Suchbereich für eine passende Variante in der For-Schleife zwischen Zeile 24 und 36 auf genau die Varianten beschränkt, die den ersten Defekt aus der Menge $defectsIn(i)$ ausgleichen können. Für die Suche einer passenden Variante sind dadurch maximal $n * (x - 1)$ Vergleiche notwendig, wenn $n$ die Anzahl der Varianten im Suchbereich und $x$ die Anzahl der Defekte innerhalb der Instanz bezeichnet. Der in Zeile 25 initial mit „true“ belegten Variablen „varianteIsUsable“, wird in Zeile 28 „false“ zugewiesen, sobald ein defekter Logikblock von der untersuchten Variante verwendet wird. Nur, wenn alle defekten Logikblöcke innerhalb der untersuchten Variante nicht verwendet werden, wird diese Variante in Zeile 33 für die aktuell betrachtete Instanz ausgewählt. Sobald für eine der in der DT-Konfiguration enthaltenen Instanzen $i \in I$ keine passende Variante ermittelt werden konnte, wird der Algorithmus in Zeile 39 abgebrochen. Das vorliegende FPGA ist in diesem Fall nicht verwendbar.

Wenn der Algorithmus erfolgreich durchgeführt wurde, ist die Variable „varianteIsUsable“ mit „true“ belegt. In diesem Fall wurde für alle Instanzen $i \in I$ eine passende Variante ermittelt und unter *variante*($i$) gespeichert.

**Algorithmus 5.6** : Ermittlung der einzusetzenden Varianten

```
// Initialisierung
for jede Instanz i ∈ I do
    defectsIn(i) = {};
    variante(i) = bereits in die DT-Konfiguration eingesetzte Variante;
// Defekte den Instanzen zuordnen
for jeden Defekt d ∈ D do
    for jede Instanz i ∈ I do
        if (X_min(i) ≤ X(d) ≤ X_max(i)) and (Y_min(i) ≤ Y(d) ≤ Y_max(i)) then
            defectsIn(i) = defectsIn(i) ∪ d; break;
// Ermittlung einer Variante zum Ausgleich der Defekte
for jede Instanz i ∈ I do
    if |defectsIn(i)| = 1 then
        d = head(defectsIn(i));
        variante(i) = head(lookup(X(d) − X_min(i), Y(d) − Y_min(i)));
        varianteIsUsable = true;
    else if |defectsIn(i)| > 1 then
        d1 = head(defectsIn(i)); for v = (jede Variante
        v ∈ lookup(X(d) − X_min(i), Y(d) − Y_min(i))) do
            varianteIsUsable = true; for jeden d ∈ defectsIn(i)\head(defectsIn(i)))
            do
                if blockStatus(i, v, X(d) − X_min(i), Y(d) − Y_min(i)) == used then
                    varianteIsUsable = false; break;
            if varianteIsUsable == true then
                variante(i) = v;
                break;
    // Keine passende Variante vorhanden: Abbruch
    if varianteIsUsable==false then
        exit;
```

## 5.5 Zusammenfassung

Zu Beginn dieses Kapitels wurde ein grober Überblick zu dem vorgeschlagenen Hard-Core basierten Ansatz vorgestellt. Im weiteren Verlauf wurden dann die einzelnen Komponenten dieses Ansatzes erläutert. Neben der Bereitstellung von DT-Hard-Core-Versionen durch die IP-Anbieter wurde die Erweiterung des Design-Flows um einen Re-

Mapping-Algorithmus ausführlich dargestellt. Die im letzten Abschnitt dieses Kapitels vorgestellte Programmierung der teildefekten FPGAs in der Serienproduktion wurde durch die vorhergehenden Phasen des Ansatzes derart vereinfacht, dass die Erstellung eines speziellen Bitstreams für ein FPGA mit defekten Logikblöcken mit einem geringen Zeitaufwand erfolgen kann.

# 6 Bewertung des Ansatzes

In diesem Kapitel wird der Hard-Core basierte Ansatz aus verschiedenen Perspektiven bewertet. Neben der im nächsten Abschnitt diskutierten Verbesserung der Ausbeute durch den Ansatz, die vor allem für den FPGA-Hersteller von Interesse ist, sind die zusätzlichen Kosten für den Kunden des FPGA-Herstellers interessant, der den Hard-Core basierten Ansatz für eine Serienproduktion einsetzen möchte. Ein Ansatz für die Modellierung der eingesparten Kosten bei der Verwendung von defekten FPGAs wird in Abschnitt 6.2 vorgestellt.

Zum Abschluss des Kapitels erfolgt in Abschnitt 6.3 ein ausführlicher Vergleich des Hard-Core basierten Ansatzes mit den in Kapitel 4 vorgestellten vergleichbaren Arbeiten.

## 6.1 Verbesserung der Ausbeute

Wenn wir von einem Yield bezüglich der *random defects* $Y_{RANDOM}$ zwischen 50% und 80% für ein FPGA in der Herstellung ausgehen und für eine grobe Abschätzung der durchschnittlichen Anzahl von Defekten auf dem FPGA $\lambda$ das Poisson-Modell, also $Y_{RANDOM} = e^{-\lambda}$, verwenden, können wir den minimalen Wert für $\lambda$ als

$$\lambda_{min} = -\ln 0,8 \tag{6.1}$$

und den maximalen Wert für $\lambda$ als

$$\lambda_{max} = -\ln 0,5 \tag{6.2}$$

angeben. Wenn wir weiterhin davon ausgehen, dass die durchschnittliche Anzahl von Defekten, die sich ausschließlich auf einen Logikblock auswirken $\lambda_{LBs}$ zwischen 10% und 40% der durchschnittlichen Anzahl von Defekten auf dem gesamten FPGA beträgt, können wir die Wahrscheinlichkeiten bestimmen, dass bis zu $k$ Defekte auftreten, die sich ausschließlich auf einen Logikblock auswirken.

Mit Gleichung 2.5 (Seite 27) gilt

$$P_{LBs}(X = k) = \frac{e^{-\lambda}\lambda_{LBs}^{k}}{k!} \tag{6.3}$$

für die Wahrscheinlichkeit, dass genau $k$ Defekte auftreten, die sich ausschließlich auf einen Logikblock auswirken,

$$\sum_{x=0}^{k} P(X = x) \tag{6.4}$$

für die Wahrscheinlichkeit, dass 0 bis k Defekte auftreten, die sich ausschließlich auf einen Logikblock auswirken. Wenn wir die durchschnittliche Anzahl von denjenigen Defekten, die eine Auswirkung auf mehr als einen Logikblock haben, mit $\lambda_{rest} = \lambda - \lambda_{LBs}$ bezeichnen, gibt

$$P_{rest}(X = 0) = \frac{e^{-\lambda}\lambda_{rest}^{0}}{0!} = e^{-\lambda} \tag{6.5}$$

die Wahrscheinlichkeit dafür an, dass keiner dieser Defekte auftritt und

$$\sum_{x=0}^{k} P(X = x) * P_{rest}(X = 0) \tag{6.6}$$

die Wahrscheinlichkeit für bis zu $k$ Defekte auf einem FPGA, die jeweils ausschließlich einen Logikblock des FPGAs betreffen.

Tabelle 6.1 zeigt diese Wahrscheinlichkeiten für die oben angenommenen Randfälle (alle Wahrscheinlichkeiten sind gerundet dargestellt). Die Ergebnisse der obigen Abschätzung können wie folgt interpretiert werden:

Der *random yield* lässt sich bei einem ursprünglichen Wert von $Y_{RANDOM} = 0,5$ auf $Y_{RANDOM} = 0,66$ ($Y_{RANDOM} = 0,536$) verbessern, wenn 40% (10%) der auf dem FPGA im Durchschnitt vorhandenen Defekte auf einen Logikblock begrenzt sind und FPGAs mit defekten Logikblöcken nicht als Ausschuss, sondern als verwendbare Chips betrachtet werden. Wenn man von $Y_{RANDOM} = 0,8$ ausgeht, kann eine Verbesserung auf $Y_{RANDOM} = 0,875$ ($Y_{RANDOM} = 0,818$) erreicht werden. Die Bereitstellung von solchen DT-Versionen, die mehr als zwei defekte Logikblöcke tolerieren können, ist bei den oben angenommenen Parametern nicht notwendig, da nur ein sehr geringer Anteil der FPGAs drei Defekte enthält, die sich jeweils ausschließlich auf einen Logikblock auswirken und die Wahrscheinlichkeit dafür, dass drei Defekte in einer DT-Version auftreten somit sehr gering ist. Die Ausbeute an FPGAs, die für den Hard-Core basierten Ansatz verwendet werden können, lässt sich steigern, wenn der Ansatz in Verbindung

mit einem Verfahren verwendet wird, das Defekte außerhalb der Logikblöcke tolerieren kann.

| | $Y_{RANDOM}=0{,}5$<br>$\lambda_{LBs}=0{,}1*\lambda_{max}$<br>$P_{rest}(X=0)=0{,}536$ | | $Y_{RANDOM}=0{,}8$<br>$\lambda_{LBs}=0{,}1*\lambda_{min}$<br>$P_{rest}(X=0)=0{,}818$ | |
|---|---|---|---|---|
| | a) | b) | a) | b) |
| k=0 | 0,933 | 0,5 | 0,978 | 0,8 |
| k=1 | 0,065 | 0,535 | 0,022 | 0,818 |
| k=2 | 0,002 | 0,536 | < 0,001 | 0,818 |
| k=3 | < 0,001 | 0,536 | < 0,001 | 0,818 |

| | $Y_{RANDOM}=0{,}5$<br>$\lambda_{LBs}=0{,}4*\lambda_{max}$<br>$P_{rest}(X=0)=0{,}66$ | | $Y_{RANDOM}=0{,}8$<br>$\lambda_{LBs}=0{,}4*\lambda_{min}$<br>$P_{rest}(X=0)=0{,}875$ | |
|---|---|---|---|---|
| | a) | b) | a) | b) |
| k=0 | 0,758 | 0,5 | 0,915 | 0,8 |
| k=1 | 0,210 | 0,639 | 0,082 | 0,871 |
| k=2 | 0,029 | 0,658 | 0,004 | 0,875 |
| k=3 | 0,003 | 0,66 | < 0,001 | 0,875 |

Tabelle 6.1: Unter verschiedenen Annahmen für $Y_{RANDOM}$ und $\lambda_{LB}$:
a) $P_{LBs}(X = k)$: Die Wahrscheinlichkeit dass genau k Defekte, auf einen Logikblock begrenzt sind; b) $\sum_{x=0}^{k} P_{LBs}(X = x) * P_{rest}(X = 0)$: Die Wahrscheinlichkeit für bis zu k Defekte auf einem FPGA, die jeweils ausschließlich einen Logikblock betreffen.

## 6.2 Reduzierung der Kosten für die Massenproduktion

Wenn die FPGAs entsprechend der Anzahl der auf ihnen vorhandenen defekten Logikblöcke klassifiziert werden und in der Serienproduktion der jeweilige Anteil an FPGAs mit einer festen Anzahl von defekten Logikblöcken bekannt ist, kann der für die Serienproduktion nutzbare Anteil an FPGAs mit defekten Logikblöcken vorhergesagt werden. Die Berechnung dieser Vorhersage wird im folgenden Abschnitt erläutert.

Der Kunde des FPGA-Herstellers, der FPGAs in einer Massenproduktion einsetzen will, benötigt ein Kostenmodell, anhand dessen er berechnen kann, unter welchen Voraussetzungen die Kosten für die Nutzung von FPGAs in dem von ihm geplanten Produkt durch die Nutzung von FPGAs mit defekten Logikblöcken gesenkt werden können. Ein Ansatz für die Modellierung der eingesparten Kosten bei der Verwendung von defekten FPGAs wird in Abschnitt 6.2.2 vorgestellt.

### 6.2.1 Eine Vorhersage des nutzbaren Anteils an defekten FPGAs

Bei einem Chip mit $N$ Logikblöcken insgesamt, von denen $d_{LBgesamt}$ Logikblöcke defekt sind, gibt es $\begin{pmatrix} N \\ d_{LBgesamt} \end{pmatrix}$ verschiedene Möglichkeiten, im Folgenden als *Defektverteilungen* bezeichnet, für die Platzierung dieser defekten Logikblöcke auf dem Chip. Jede dieser Defektverteilungen wird im Folgenden im mathematischen Sinn als Ergebnis eines Zufallsexperiments betrachtet. Die Menge der möglichen Defektverteilungen entspricht dann der Menge der möglichen Ergebnisse dieses Zufallsexperiments $\Omega$, d.h. es gilt $|\Omega| = \begin{pmatrix} N \\ d_{LBgesamt} \end{pmatrix}$(Beispiel, siehe unten[1]).

Wir definieren nun das Ereignis $DT \subseteq \Omega$ als

> DT = „Die betrachtete DT-Konfiguration kann die Defekte auf dem FPGA tolerieren“

Bei Annahme einer Gleichverteilung der Ergebnisse kann die Auftrittswahrscheinlichkeit $P(DT)$ für das Ereignis $DT$ mit

$$P(DT) = \frac{Anzahl\ der\ Defektverteilungen\ für\ die\ DT\ gilt}{Anzahl\ der\ möglichen\ Defektverteilungen} = \frac{|DT|}{|\Omega|} \tag{6.7}$$

angegeben werden.

Es sei nun $HC$ die Menge aller Instanzen von Hard-Core-Versionen, die in der DT-Konfiguration platziert sind. Um $|DT|$ zu bestimmen, wird die DT-Konfiguration zunächst in $|HC| + 1$ disjunkte Bereiche aufgeteilt. Die Anzahl der innerhalb einer Instanz (= ein Bereich) $hc \in HC$ liegenden defekten Logikblöcke soll im Folgenden mit $d_{LB}(hc)$ und die Anzahl der defekten Logikblöcke im nicht genutzten Bereich des FPGAs mit $d_{LB}(unused)$ bezeichnet werden. Alle Varianten $\omega \in DT$ erfüllen dann die Bedingungen[2]

$$\sum_{hc \in HC} d_{LB}(hc) + d_{LB}(unused) = d_{LBgesamt} \tag{6.8}$$

und

$$\forall hc \in HC : d_{LB}(hc) < d_{LBmax}(hc) \tag{6.9}$$

[1] Beispiel: Auf einem FPGA der Größe N=4, also einem FPGA, auf dem 4 Logikblöcke vorhanden sind, gibt es bei $d_{LBgesamt} = 2$ Defekten Logikblöcken $|\Omega| = \begin{pmatrix} 4 \\ 2 \end{pmatrix} = 6$ mögliche Verteilungen dieser Defekte auf dem FPGA.

[2] Alle Logikblöcke, die außerhalb einer Instanz liegen, in der Defekte toleriert werden können, werden aufsummiert und als 0-DT-Instanz betrachtet.

, wobei $d_{LBmax}(hc)$ die Anzahl der defekten Logikblöcke beschreibt, die in der Instanz $hc$ maximal toleriert werden können und $d_{LBgesamt}$ wie oben die Anzahl der defekten Logikblöcke auf dem Chip insgesamt bezeichnet.

Abbildung 6.2 skizziert ein rekursives Verfahren, mit dem alle möglichen Verteilungen auf die disjunkten Bereiche des FPGAs ermittelt werden können. Die erste Spalte gibt die Anzahl der Defekte an, die insgesamt in den Instanzen $hc \in HC$ liegen. Die restlichen Defekte liegen entsprechend im nicht genutzten Bereich des FPGAs und sind in der letzten Spalte angegeben. Nach Gleichung 6.8 gilt für jede Zeile $d_{LB}(unused) = d_{LBgesamt} - \sum_{hc \in HC} d_{LB}(hc)$. In dem in der Abbildung gezeigten Beispiel wird von $d_{LBgesamt} = 3$ defekten Logikblöcken auf dem FPGA ausgegangen. In jeder neuen Zeile wird gegenüber der vorhergehenden Zeile ein weiterer defekter Logikblock in den im FPGA genutzten Bereich eingefügt, wobei stets die in Gleichung 6.9 angegebene Bedingung eingehalten wird. Wenn das Verfahren auf eine Verteilung trifft, die bereits ermittelt wurde (in Abbildung 6.2 grau dargestellt), wird die Rekursion für den betreffenden Zweig nicht fortgesetzt.

Es sei nun $V$ die Menge aller möglichen Verteilungen der Defekte auf die Instanzen und den in der DT-Konfiguration nicht genutzten Bereich. Eine Verteilung $v \in V$ kann als Tupel mit $n = |HC| + 1$ Elementen angegeben werden: $v = (d_{LB}(hc_1), d_{LB}(hc_2), \ldots, d_{LB}(hc_n), d_{LB}(unused))$.

Das Ereignis $DT_v$ sei nun wie folgt definiert:

> $DT_v$ = „Die betrachtete DT-Konfiguration kann die Defekte auf dem FPGA tolerieren *und* die Defekte sind -wie durch $v \in V$ angegeben- auf die Instanzen und die nicht genutzte Fläche des FPGAs verteilt“

Gleichung 6.7 kann jetzt formuliert werden als

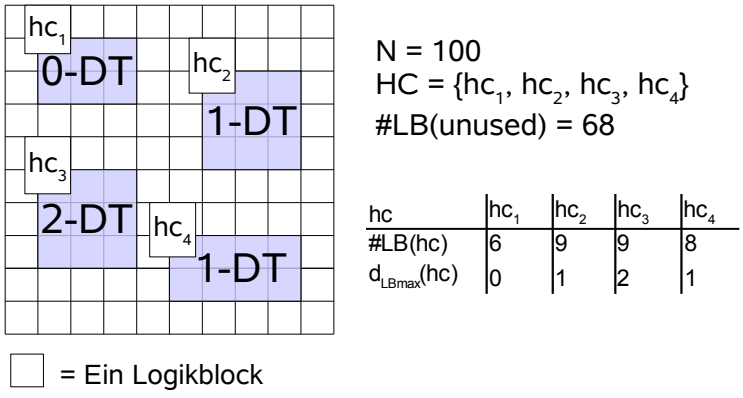



Abbildung 6.1: Beispiel: Eine DT-Konfiguration

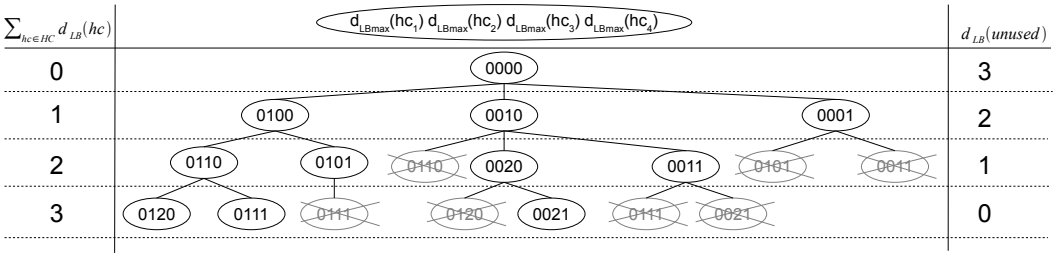



Abbildung 6.2: Ein rekursives Verfahren für die Ermittlung aller möglichen Verteilungen der Defekte auf die Instanzen und den in den nicht genutzten Bereich, für die in Abbildung 6.1 gezeigte DT-Konfiguration, wenn $d_{LBgesamt} = 3$ gilt.

$$P(DT) = \frac{\sum_{v \in V} |DT_v|}{|\Omega|} \tag{6.10}$$

wobei $|DT_v|$ durch

$$|DT_v| = \prod_{hc \in HC} \left( \begin{array}{c} \#LB(hc) \\ d_{LB}(hc, v) \end{array} \right) * \left( \begin{array}{c} \#LB(unused) \\ d_{LB}(unused, v) \end{array} \right) \tag{6.11}$$

bestimmt werden kann, wenn durch $\#LB(hc)$ die Anzahl der Logikblöcke in der Instanz $hc \in HC$ und durch $\#LB(unused)$ die Anzahl der in der DT-Konfiguration nicht genutzten Logikblöcke angegeben ist und $d_{LB}(hc, v)$ die Anzahl der Defekte innerhalb der Instanz $hc$ in der Verteilung $v$ sowie $d_{LB}(unused, v)$ die Anzahl der Defekte in den nicht genutzten Logikblöcken in der Verteilung $v$ angibt.

Die wie in 6.2 skizziert ermittelte Menge aller möglichen Verteilungen $V$ für das Beispiel aus Abbildung 6.1 ist

$V = \{(0,0,0,0,3), (0,1,0,0,2), (0,0,1,0,2), (0,0,0,1,2), (0,1,1,0,1), (0,1,0,1,1), (0,0,2,0,1),$
$(0,0,1,1,1), (0,1,2,0,0), (0,1,1,1,0), (0,0,2,1,0)\}$

Die Berechnung von $DT_v$ soll hier nur beispielhaft für eines dieser $v \in V$ angegeben werden:

$|DT_{(0,1,0,0,2)}| = \prod_{hc \in HC} \left( \begin{array}{c} \#LB(hc) \\ d_{LB}(hc, v) \end{array} \right) * \left( \begin{array}{c} \#LB(unused) \\ d_{LB}(unused, v) \end{array} \right) = \left( \begin{array}{c} 6 \\ 0 \end{array} \right) * \left( \begin{array}{c} 9 \\ 1 \end{array} \right) * \left( \begin{array}{c} 9 \\ 0 \end{array} \right) * \left( \begin{array}{c} 8 \\ 0 \end{array} \right) *$
$\left( \begin{array}{c} 68 \\ 2 \end{array} \right) = 20502$

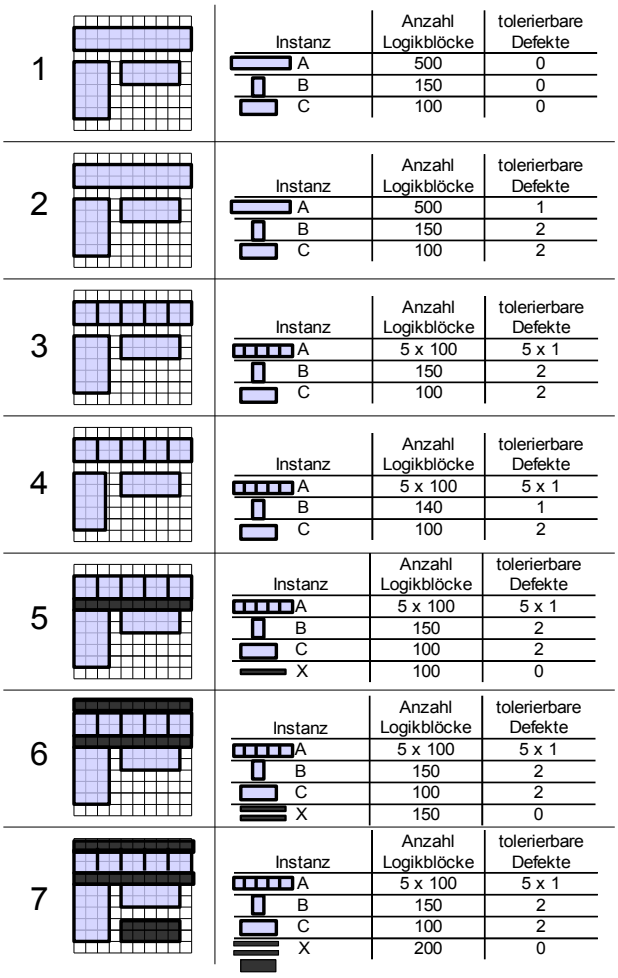
Bei $d = 3$ defekten Logikblöcken kann mit Gleichung 6.10 dann die Wahrscheinlichkeit $P(DT) \approx 0,667$ ermittelt werden. Aus diesem Ergebnis folgt, dass 66,7 % der FPGAs mit drei defekten Logikblöcken verwendet werden können, wenn die in Abbildung 6.1 gezeigte DT-Konfiguration für die Serienproduktion verwendet wird. Der nutzbare Anteil der FPGAs mit $d$ Defekten wird im Folgenden mit $NA(d)$ bezeichnet (Für das Beispiel gilt also $NA(3) = 66,7\%$).

Abbildung 6.3 zeigt die Ergebnisse von 7 verschiedenen DT-Konfigurationen, für die der nutzbare Anteil $NA(d)$ jeweils für $d = 1$ bis $d = 5$ defekte Logikblöcke auf dem FPGA errechnet wurde. Die Ergebnisse zeigen den Einfluss verschiedener Faktoren auf den nutzbaren Anteil der defekten FPGAs. Die ersten beiden der vorgestellten Konfigurationen ermöglichen den Vergleich zwischen einer FPGA-Konfiguration, die keine DT-Hard-Core-Versionen verwendet, und einer DT-Konfiguration, wobei die Anzahl der auf dem FPGA genutzten Logikblöcke in beiden Konfigurationen gleich ist. Der nutzbare Anteil lässt sich, wie in der dritten Konfiguration zu sehen, deutlich verbessern, wenn anstelle der 1DT-Version, die für die Instanz A eingesetzt ist, eine partitionierte 1-DT-Version verwendet werden kann. Die unterschiedlichen nutzbaren Anteile zwischen der vierten und der dritten Konfiguration zeigen, dass die Verwendung einer 2-DT-Version gegenüber der Verwendung einer etwas kleineren 1-DT-Version bereits eine deutliche Auswirkung auf das Ergebnis hat.

Für die Bereitstellung eines festen Interfaces werden Anschlüsse an die Umgebung der Hard-Core-Version benötigt, deren Position in allen Varianten dieser Version gleich bleibt. Ob diese Anschlüsse ohne die Verwendung von zusätzlichen Logikblöcken festgelegt werden können, hängt von der Anzahl der benötigten Anschlüsse und den Möglichkeiten der FPGA-Architektur ab. Die Realisierung des Interfaces mit Hilfe von zusätzlichen Logikblöcken wurde bereits in Abbildung 5.14 auf Seite auf Seite 73 gezeigt. Diese Art der Realisierung ist auch dann möglich, wenn für die durch das Hard-Core benötigten Anschlüsse keine gleichbleibenden Punkte außerhalb der Logikblöcke festgelegt werden können. Sie verursacht allerdings auch einen zusätzlichen nicht defekttoleranten Bereich auf dem FPGA. Die Auswirkungen dieses zusätzlichen 0-DT-Bereiches ist in den letzten drei Konfigurationen in Abbildung 6.3 erkennbar. Auch Hard-Cores, für die keine defekttoleranten Versionen zur Verfügung stehen oder für die ein Austausch aufgrund der Performance-Anforderungen an das FPGA-Design nicht stattfinden kann, vergrößern den 0-DT-Bereich in einer DT-Konfiguration.

### 6.2.2 Einsparungen durch die Nutzung von defekten FPGAs

Wenn FPGAs mit defekten Logikblöcken – anders als bisher – in der Praxis tatsächlich verwendet werden können, sind verschiedene Preiskategorien, wie sie bereits jetzt für




| **DT-Konfiguration** | **Anzahl der Defekte auf dem FPGA** | | | | |
|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** |
| 1 | 25.0 | 6.2 | 1.6 | 0,4 | < 0,1 |
| 2 | 100 | 75.0 | 49.6 | 29.7 | 16.3 |
| 3 | 100 | 95,1 | 85,7 | 73,0 | 58,8 |
| 4 | 100 | 93,1 | 80,7 | 65,2 | 49,1 |
| 5 | 90 | 76 | 60 | 44,1 | 29,9 |
| 6 | 85 | 67,3 | 49,3 | 33,2 | 20,3 |
| 7 | 80 | 59,0 | 39,8 | 24,3 | 13,1 |

Abbildung 6.3: Die Berechnung von NA(d) für 7 verschiedene DT-Konfigurationen für d=1..5

FPGAs mit verschiedenen „speedgrades“ vorgesehen sind[3], denkbar. Je nach Anzahl der Defekte auf dem FPGA würden dann verschiedene Kategorien festgelegt. Die Anzahl der Fehler könnte dann als „defectgrade“ für das betreffende FPGA gelten und sich auf den Preis für dieses FPGA auswirken. Eine Voraussetzung für dieses Vorgehen ist ein Diagnose-Verfahren, dessen Kosten vermutlich auf die defekten FPGAs umgelegt würden, das also letztendlich vom Kunden des FPGA-Herstellers bezahlt werden müsste. Zusätzliche Kosten entstehen durch ein geeignetes Verfahren für die Bereitstellung einer *defect map* für jedes FPGA.

Für den Kunden des FPGA-Herstellers entsteht eine Reduzierung der Gesamtkosten für eine Serienproduktion durch die günstigeren Stückpreise für die defekten FPGAs. Auf der anderen Seite entstehen durch das Hard-Core basierte Verfahren allerdings auch Kosten, die nun im Folgenden genauer betrachtet werden sollen.

Da die Erstellung der DT-Versionen, wie in Abschnitt 5.2 beschrieben, nur teilweise automatisiert erfolgen kann, entstehen dem IP-Core-Anbieter zusätzliche Kosten für die Entwicklung von DT-Versionen der angebotenen IP-Cores, die vermutlich an den Kunden weitergegeben werden. Der zusätzliche Zeitaufwand für die Erstellung einer DT-Konfiguration im Design-Flow erzeugt ebenfalls Kosten, die voraussichtlich jedoch gering sind, da das Design im ersten Schritt komplett mit den nicht defekttoleranten Versionen der IP-Cores erstellt werden kann. Erst wenn die FPGA-Konfiguration erfolgreich getestet wurde, erfolgt die automatisierte Erstellung der DT-Konfiguration durch den Re-Mapping-Algorithmus.

In der Serienproduktion entstehen zusätzliche Kosten vor allem durch den zusätzlich benötigten Zeitaufwand, der durch eine Überprüfung und die aufwändigere Programmierung der defekten FPGAs entsteht. Bevor ein defektes FPGA in der Serienproduktion eingesetzt werden kann, muss durch die Überprüfung festgestellt werden, ob das FPGA zusammen mit der für die Serienproduktion vorliegenden DT-Konfiguration verwendet werden kann. Diese Überprüfung wird durch den Algorithmus durchgeführt, der in Abschnitt 5.4 vorgestellt wurde (7, Seite 118). Wenn die Überprüfung erfolgreich war, wird durch diesen Algorithmus für jede DT-Hard-Core-Version, die in der DT-Konfiguration enthalten ist, die einzusetzende Variante ausgegeben. Der zusätzliche Aufwand für die Programmierung des FPGAs besteht nun lediglich im Zusammenfügen der partiellen Bitstreams für die DT-Konfiguration und die Varianten, die für die Programmierung bereitgehalten werden.

Durch den nutzbaren Anteil an den defekten FPGAs, der im letzten Abschnitt berechnet wurde, ist eine Vorhersage der Anzahl von den zu überprüfenden FPGAs möglich. Wenn $n_{usable}$ FPGAs in der Serienproduktion verbaut werden sollen, müssen wahrscheinlich

[3] Im Test der FPGAs wird die maximale Taktfrequenz bestimmt, mit der die jeweiligen Test-Schaltungen getaktet werden können. Aufgrund der Ergebnisse wird jedem FPGA ein *speedgrade* zugeteilt, dieser Vorgang wird als *speed binning* bezeichnet.

$n_{inspect} = \frac{n_{usable} * 100}{NA(d)}$

FPGAs mit $d$ Defekten überprüft werden, bevor ausreichend FPGAs für die Serienproduktion zur Verfügung stehen[4]. Das durch den Re-Mapping-Algorithmus verfolgte Ziel, ist die Erstellung einer DT-Konfiguration, die die Verwendung eines möglichst hohen Anteils an defekten FPGAs ermöglicht, wodurch gleichzeitig die Anzahl der innerhalb der Serienproduktion zu überprüfenden FPGAs möglichst gering gehalten wird. Im welchen Umfang dieses Ziel erreicht wird, hängt insbesondere von den Hard-Core-Versionen, die von dem IP-core Anbieter bereitgestellt werden, und dem Slack, der in der Schaltung für den Austausch der Hard-Core-Versionen genutzt werden kann, ab. Der in der Schaltung vorhandene Slack kann durch eine Reduzierung der Taktfrequenz vergrößert werden. In Schaltungen, in denen eine Reduzierung der Taktfrequenz nicht möglich ist, und in der aufgrund der festgelegten Taktfrequenz nur wenig Slack zur Verfügung steht, versucht der Re-Mapping-Algorithmus eine im Hinblick auf den nutzbaren Anteil von defekten FPGAs möglichst günstige DT-Konfiguration zu erstellen. Da der in der FPGA-Konfiguration verfügbare Slack für den Austausch von Hard-Core-Versionen stark von den verwendeten Algorithmen für die Synthese und den Place&Route-Vorgang abhängt und diese Algorithmen im Rahmen der vorliegenden Arbeit nicht näher betrachtet wurden, ist eine allgemeine Aussage zu den erreichbaren DT-Konfigurationen für ein vorliegendes Design an dieser Stelle nicht möglich.

Zusammengefasst entstehen also Einsparungen durch die Verwendung von günstigeren FPGAs in der Serienproduktion. Diesen Einsparungen stehen zusätzlich entstehende Kosten durch

- die zusätzliche erforderlichen DT-Hard-Core-Versionen,
- den zusätzlichen Zeitaufwand für die Erstellung einer DT-Konfiguration und
- für die Überprüfung und angepasste Programmierung der FPGAs in der Serienproduktion

gegenüber.

## 6.3 Vergleich mit anderen Arbeiten

Der Ansatz soll nun mit den im Kapitel „Vergleichbare Arbeiten“ vorgestellten Ansätzen verglichen werden. Zunächst werden die Kriterien, die in [DH06] für den Vergleich von Verfahren zur Verbesserung der Ausbeute von FPGA eingeführt wurden, übernommen, um einen Vergleich mit den im Kapitel „Vergleichbare Arbeiten“ vorgestellten

[4]Beispiel: Wenn 20000 FPGAs für eine Serienproduktion benötigt werden, und $NA(1) = 95\%$ errechnet wurde, müssen wahrscheinlich $\frac{2000*100}{95} \approx 21053$ FPGAs mit einem defekten Logikblock überprüft werden.

Ansätzen zu vereinfachen. Djupdal et al. schlagen in [DH06] die obersten sieben der in der ersten Spalte der Tabellen in Abbildung 6.4 gezeigten Kriterien für den Vergleich von Verfahren zur Verbesserung der Ausbeute durch die Tolerierung von Defekten auf FPGAs vor. Eine ausführliche Erläuterung dieser Kriterien erfolgt in Tabelle 6.2 und Tabelle 6.3. Die Kriterien „Timing-closure friendly“ und „Reconfigurability“ werden zusätzlich eingeführt und erläutert. Zusammen mit der Erläuterung der in den Tabellen angegebenen Bewertungen der vergleichbaren Arbeiten erfolgt nun eine Einordnung des Hard-Core basierten Ansatzes bezüglich der angegebenen Kriterien.

### 6.3.1 Defect Coverage

Durch ein individuelles Place&Route einer FPGA-Konfiguration für jedes teildefekte FPGA auf Grundlage einer vorherigen Diagnose, können alle Defekte ausgeglichen werden, die durch eine Verwendung von alternativen Verbindungsleitungen oder Logikblöcken ausgleichbar sind. Dies gilt sowohl für die komplette Erstellung einer FPGA-Konfiguration für jedes defekte FPGA (In Abbildung 6.4: Chip specific bitfile) als auch für die dynamische Erstellung einer Konfiguration zur Laufzeit in alternativen FPGA-Architekturen (Dynamic PAR), wie sie von Macias und Durbeck für die Cell Matrix beschrieben wird [MD04]. Applikationsspezifische FPGAs (ASFPGA), also z.B. das von Xilinx verwendete Verfahren für die Nutzung von leicht defekten FPGAs [eas08], können aus Sicht des FPGA-Designers wie ein defektfreies FPGA programmiert werden, d.h. alle Defekte werden vor dem FPGA-Designer verborgen.

Ansätze, in denen kein komplettes Place & Route durchgeführt wird, um Zeit in einer Serienproduktion einzusparen, können nur eine eingeschränkte Anzahl von Fehlern korrigieren. In Verfahren, die mit prekompilierten Schaltungsteilen arbeiten, wie z.B. dem tile basierten Verfahren von Lach et al. [LMSP98b], kann pro Schaltungsteil nur eine begrenzte Anzahl von Defekten korrigiert werden. Auch die Verfahren, die zusätzliche redundante Logikblöcke in die Architektur einfügen, wie die in Abschnitt 4.2.2 erläuterten Ansätze, können nur eine begrenzte Anzahl von Defekten auf dem FPGA ausgleichen.

In dem von Doumar et al. [DI00a] vorgeschlagenen Verfahren, im Folgenden als „Shifting-Ansatz“ bezeichnet, kann durch Verschiebungen der FPGA-Konfiguration (*shifting*) in den meisten Situationen nur ein Defekt ausgeglichen werden. Bei den in Abschnitt 4.2.1 vorgestellten Verfahren, die mit redundanten Zeilen und/oder Spalten arbeiten, ist die Anzahl der ausgleichbaren Defekte durch die Anzahl der für das FPGA eingeplanten redundanten Zeilen und/oder Spalten vorgegeben und somit auch hier stark begrenzt.

Ansätze, die mit lokaler Redundanz (Local Redundancy) arbeiten, werden in Abbildung 4.1 (Seite 40) als Verfahren eingeordnet, die mit feingranularer Redundanz innerhalb

(a) Veränderung der Programmierung des FPGA

| | *Chip specific bitfile* | *Precompiled subcircuits* | | *Adaptive configuration* | |
|---|---|---|---|---|---|
| | | *tile based* | *Hard-Core based* | *Shifting* | *Dynamic PAR* |
| Defect coverage | High | Medium | Medium | Low | High |
| Area overhead | - | Medium | Medium | Medium | High |
| Timing overhead | Low | Medium | Medium | Medium | High |
| Bitfile size | Medium | High | High | Medium | Low |
| Extra HW required | - | - | - | Low | High |
| Maturity | High | Medium | Medium | Medium | Low |
| Mass production friendly | Low | High | High | High | High |
| Timing-closure friendly | High | Medium | High | Medium | Low |
| Reconfigurability | High | High | High | High | High |

(b) andere Verfahren

| | Node redundancy | | Local redundancy | ASFPGA |
|---|---|---|---|---|
| | Redundant row / col bus based \| segmented | Single nodes bus based \| segmented | | |
| Defect coverage | Low \| Low | Medium \| Medium | Medium | High |
| Area overhead | Low \| Low | Low \| Low | Low | - |
| Timing overhead | - \| Medium | - \| Medium | Low | - |
| Bitfile size | Medium \| Medium | Medium \| Medium | Medium | Medium |
| Extra HW required | Low \| High | Low \| High | Low | - |
| Maturity | High \| Low | Medium \| Low | Medium | High |
| Mass production friendly | High \| High | High \| High | High | High |
| Timing-closure friendly | High \| Medium | High \| Medium | Medium | High |
| Reconfigurability | High \| High | High \| High | High | Low |

Abbildung 6.4: Ein Vergleich von Verfahren zur Verbesserung der Ausbeute durch die Tolerierung von Defekten auf FPGAs

der Logik-Blöcke oder innerhalb anderer Komponenten eines FPGAs arbeiten. Je nach betrachtetem Verfahren ist hier die Art der tolerierbaren Defekte eingeschränkt.

#### „Defect Coverage“ durch den Hard-Core basierten Ansatz

Wie viele Defekte durch den Hard-Core basierten Ansatz ausgeglichen werden können hängt ab von

- den durch den IP-Core-Anbieter zur Verfügung gestellten Hard-Core-Versionen und
- dem innerhalb der Schaltung zur Verfügung stehenden Slack, d.h. der durch den FPGA-Designer gewünschten maximalen Taktfrequenz, mit der das Design betrieben werden soll.

Wie bei den anderen Verfahren, die auf Logikblock-Ebene arbeiten, können nur defekte Logikblöcke und keine Defekte in den Verbindungsleitungen des FPGAs toleriert werden.

### 6.3.2 Area Overhead

Die individuelle Konfiguration eines FPGA (*chip specific bitfile*) und applikationsspezifische FPGAs sind Verfahren, die ohne Redundanz arbeiten. Da fast alle der übrigen der hier betrachteten Verfahren so konstruiert wurden, dass sie mit möglichst wenig redundanten Komponenten auf dem FPGA auszukommen, wurde nur ein Verfahren in Abbildung 6.4 mit „high“ für dieses Kriterium bewertet. Das dynamische Place & Route zur Laufzeit des FPGAs ist ein Verfahren, das potentiell den höchsten zusätzlichen Flächenverbrauch pro Logikblock von allen betrachteten Verfahren hat, da zusätzlich redundante Informationen in jedem Logikblock gespeichert werden müssen, um eine Programmierung der benachbarten Logikzellen durchführen zu können.

Auch für das tile basierte Verfahren ist die Redundanz vergleichsweise hoch, da für jedes tile mindestens ein Reserve-Logikblock benötigt wird. Die Anzahl der Reserve-Logikblöcke, die für den Shifting-Ansatz in die FPGA-Konfiguration integriert werden müssen, ist, wenn bei dem tile basierten Verfahren eine sehr feingranulare Partitionierung erfolgreich ist (z.B. die Partitionierung der gesamten FPGA-Konfiguration in tiles der Größe 2x2), geringer. Bei größeren tiles ist die benötigte Anzahl an Reserve-Logikblöcken vergleichbar. Die Anzahl der benötigten Reserve-Logikblöcke im Shifting-Ansatz ist abhängig von der eingesetzten Verteilung (vgl. Abbildung 4.9 auf Seite 50). Bei Verfahren, die mit zusätzlich in die FPGA-Architektur eingefügten redundanten Logikblöcken arbeiten, ist die zusätzlich für die Reserve-Logikblöcke benötigte Fläche gering, da nur wenige Reserve-Logikblöcke eingefügt werden.

#### „Area Overhead“ durch den Hard-Core basierten Ansatz

Das Hard-Core basierte Verfahren ist bzgl. der Anzahl der neu in die Schaltung eingefügten Reserve-Logikblöcke mit dem tile basierten Verfahren und dem Shifting-Ansatz vergleichbar.

### 6.3.3 Timing Overhead

Bei den Verfahren, die mit zusätzlich in die Architektur des FPGA eingefügten Reserve-Logikblöcken arbeiten (*Node redundancy*), ist das zusätzliche Delay abhängig von der Verbindungsstruktur. In einem Bus-basierten FPGA, also in einem FPGA, in dem Verbindungsleitungen vorhanden sind, welche die gesamte Höhe und die gesamte Breite des FPGAs umfassen, ist die Realisierung eines solchen Verfahrens ohne zusätzliches Delay möglich, da das Delay der Verbindungsleitungen der Logikblöcke untereinander sich nicht verändert, wenn die Funktion eines defekten Logikblocks auf dem FPGA durch einen Reserve-Logikblock übernommen wird, der sich in derselben Zeile oder Spalte befindet.

Sobald Verbindungen zwischen den Logikblöcken möglich sind, die über Verbindungssegmente erfolgen, müssen die Verbindungssegmente verlängert werden, um ein Verfahren mit redundanten Spalten oder Zeilen zu ermöglichen, wie in Abbildung 4.4 auf Seite 44 skizziert wurde. In diesem Fall wird das Delay der verlängerten Verbindungssegmente erhöht. Auch bei Verfahren, bei denen nicht eine komplette Zeile oder Spalte für den Ausgleich eines defekten Logikblocks eingesetzt wird, sind zusätzliche Schalter und Leitungen erforderlich, die das Delay in der Schaltung erhöhen.

Die Verwendung von fehlerkorrigierenden Codes oder andere Verfahren, die mit feingranularer Redundanz innerhalb der FPGA-Architektur arbeiten, haben in der Regel nur eine geringe Auswirkung auf die Verzögerungszeiten innerhalb des FPGAs. Ansätze, die mit zusätzlichen Schaltern arbeiten, wie z.B. das Verfahren von Yu und Lemieux [RWL05] , in dem zusätzliche Multiplexer in die FPGA-Architektur eingefügt werden (vgl. Abbildung 4.3 auf Seite 42) können die Verzögerungszeiten innerhalb eines FPGAs jedoch signifikant verändern.

Im Gegensatz zu den applikationsspezifischen FPGAs ist die FPGA-Konfiguration als Ergebnis eines Place&Route-Vorgangs bei einer individuellen Konfiguration pro defektem FPGA evtl. nicht optimal, da die defekten Komponenten auf dem FPGA für den jeweiligen Place&Route-Vorgang nicht genutzt werden können.

Auch die übrigen der in Abbildung 6.4 (a) betrachteten Verfahren, die mit einer Veränderung der FPGA-Konfiguration arbeiten, wirken sich auf die Verzögerungszeiten der Verbindungsleitungen zwischen den Logikblöcken aus. Experimente zu dem tile basierten Ansatz von Lach et al. ergaben eine zusätzliche Verzögerungszeit von bis zu 45%, verglichen mit dem nicht defekttoleranten Original Design [LMSP98b].

Bei Architekturen, die wie die Cell Matrix [DM01] ein dynamisches Place&Route auf dem Chip erlauben, kann von einem hohen Timing Overhead ausgegangen werden, da nur einfache Place&Route-Algorithmen dynamisch auf dem Chip ausgeführt werden können, und da der Area-Overhead in den Logikblöcken, der für ein dynamisches Place&Route erforderlich ist, für eine zusätzliche Verlängerung der Verzögerungszeiten sorgt.

#### „Timing Overhead“ durch den Hard-Core basierten Ansatz

Da auch bei dem Hard-Core basierten Ansatz Redundanz in Form von nicht genutzten Logikblöcken in das Design eingefügt wird, ist das Verfahren auch bzgl. dieses Kriteriums mit dem tile basierten Verfahren vergleichbar einzuordnen. Da die Verzögerungszeiten jedoch erst nach dem kompletten Place&Route-Vorgang feststehen, ist bei dem tile basierten Verfahren keine gezielte Erstellung eines Designs bzgl. vorgegebener Timing-Constraints möglich.

Der Hard-Core basierte Ansatz ist das einzige der betrachteten Verfahren, in dem zusätzliche Reserve-Logikblöcke abhängig von dem vorhandenem Slack in den Verbindungsleitungen, die über diese Logikblöcke verlaufen, eingefügt werden.

### 6.3.4 Bitfile size

Eine Besonderheit der Ansätzen, die mit prekompilierten Schaltungsteilen arbeiten ist der zusätzliche Speicher, der für die Bereithaltung der Varianten dieser Schaltungsteile benötigt wird. Neben der eigentlichen FPGA-Konfiguration, werden im tile basierten Ansatz von Lach et al. zusätzliche Teil-Konfigurationen bereitgehalten, die für die Anpassung der FPGA-Konfiguration an defekte FPGAs benötigt werden.

Fast alle der übrigen vorgestellten Verfahren haben keine Auswirkungen auf die Größe des Bitfiles, das für die Programmierung der FPGAs verwendet wird. Die einzige Ausnahme bilden Verfahren, bei den der Place&Route-Vorgang dynamisch auf dem Chip durchgeführt wird, da bei der Programmierung eines solchen Chips, der ein dynamisches Place&Route unterstützt, keine Platzierungsinformationen übermittelt werden müssen.

#### „Bitfile size“ bei dem Hard-Core basierten Ansatz

Da auch das Hard-Core basierte Verfahren mit prekompilierten Schaltungsteilen arbeite, wird auch hier ein zusätzlicher Speicher benötigt. Gegenüber dem tile basierten Verfahren sind hier allerdings Einsparungen vorhanden, wenn diese Schaltungsteile (Hard-Core-Versionen) mehrfach in der Schaltung verwendet werden.

### 6.3.5 Extra HW required

Die größte Veränderung der bekannten FPGA-Architekturen ist für die Realisierung eines dynamischen Place&Route-Vorgangs auf dem Chip notwendig.

Bei der Beurteilung zu den Verfahren, die zusätzliche Reserve-Logikblöcke in die FPGA-Architektur einfügen (*node redundancy*) muss wiederum zwischen der Erweiterung der Bus-basierten Architektur und der Erweiterung einer FPGA-Architektur im

island-style unterschieden werden. In einer Bus-basierte Architektur erfordert ein Verfahren, dass mit redundanten Zeilen arbeitet lediglich eine Deaktivierung der defekten Zeile und eine Möglichkeit zur Änderung der Adressierung der Zeilen des FPGAs. Bei einer island-style-Architektur müssen zusätzliche Verbindungsleitungen in das FPGA eingebaut werden. Während die Firma Altera, einer der führenden FPGA-Hersteller, in FPGAs, die eine Bus-basierte Architektur aufweisen, ein Verfahren mit redundanten Spalten bereits seit über 10 Jahren einsetzt, und den Einsatz dieses Verfahrens angekündigt hat, um den Yield für die nächste - in einer 40nm-Technologie hergestellte - FPGA-Generation zu verbessern, ist noch kein Verfahren für island-style FPGAs für den Massenmarkt realisiert worden, das mit redundanten Spalten arbeitet. Dazu heißt es in [RH97]:

> „Altera has demonstrated a workable method with their long segment architectures based on column redundancy. An open challenge for island-style architectures, with short-length segments is to determine if there is a practical way to do a similar kind of redundancy. Although there has been some work on this subject, none of has yet to be proven practical because of routing issues. Specifically, short routing segments required significant overhead in area to be made redundant.“

Die in Abbildung 6.4 (Seite 133) unter local redundancy eingeordneten Ansätze sind in der Regel darauf ausgelegt, möglichst wenig zusätzliche Fläche im FPGA zu beanspruchen. Auch der von Doumar et al. vorgeschlagene Shifting-Ansatz [DI00a] erfordert nur eine geringfügige Veränderung des Konfigurationsspeichers auf dem FPGA, um das Shiften der Konfigurationsdaten zu ermöglichen.

Die übrigen der in Abbildung 6.4 (a) betrachteten Verfahren setzen keine Veränderung oder Erweiterung der FPGA-Architektur voraus.

#### „Extra HW required“ bei dem Hard-Core basierten Ansatz

Der Hard-Core basierte Ansatz setzt keine Veränderung oder Erweiterung der bestehenden FPGA-Architektur voraus.

### 6.3.6 Maturity

Das die komplett individuelle Konfiguration für jedes defekte FPGA durchführbar ist, wurde im Teramac-Projekt demonstriert und kann daher in Abbildung 6.4 als ausgereift betrachtet werden, auch wenn der Einsatz dieses Verfahrens in einer Serienproduktion nicht möglich ist. Applikationsspezifische FPGAs werden, wie auf Seite 41 beschrieben, von der Firma Xilinx angeboten. Redundante Spalten werden von der Firma Altera zur Verbesserung des Yield eingesetzt.

Alle übrigen Verfahren, insbesondere auch Verfahren, die auf einer island-style FPGA-Architektur mit redundanten Spalten arbeiten, wurden bisher noch nicht für die Nutzung von defekten FPGAs in einer hohen Stückzahl eingesetzt.

Ein dynamisches Place&Route auf dem FPGA, wie es für die Cell Matrix von Durbeck und Macias vorgeschlagen wurde, setzt die Realisierung von sehr großen Arrays von Zellen voraus, und wird daher von den Autoren für kommende Technologie-Generationen vorgeschlagen [DM01].

#### „Maturity“ des Hard-Core basierten Ansatzes

Das Hard-Core basierte Verfahren setzt keine Änderung der bekannten FPGA-Architekturen voraus. Es wird bisher noch nicht für die Nutzung von defekten FPGAs eingesetzt.

### 6.3.7 Mass production friendly

Die Generierung einer speziellen FPGA-Konfiguration für jedes FPGA (Chip specific bitfile) eignet sich aufgrund des damit verbundenen Zeitaufwands nicht für die Serienproduktion eines Produktes, in dem ein FPGA verwendet wird. Alle übrigen der innerhalb dieser Arbeit betrachteten Verfahren eignen sich für den Einsatz in einer Serienproduktion.

### 6.3.8 Timing-closure friendly

Um die Timing-Closure, also die Einhaltung aller Timing-Constraints, für ein Design zu erreichen, wenn eine nach dem Place&Route-Vorgang durchgeführte Timing-Analyse eine Verletzung der Timing-Constraints ergibt, kann der FPGA-Designer:

- die Modellierung der Schaltung und / oder
- die Belegung der Pins des FPGAs

so verändern, dass für die nachfolgende Synthese und das darauffolgende Place&Route die Einhaltung der Timing-Constraints vereinfacht wird.

Eine weitere Möglichkeit für die Einhaltung der Timing-Constraints zu sorgen, ist die Vorgabe dieser Timing-Constraints für die Algorithmen, die für die Synthese und/oder den Place&Route-Vorgang verwendet werden. Ob die Timing Closure erreicht werden kann, hängt in diesem Fall ab von

1. den Freiheiten, die dem Place&Route-Werkzeug gegeben werden können und
2. der FPGA-Architektur, die bei einigen Ansätzen so verändert wird, dass die FPGA-Komponenten (teilweise) eine höhere Verzögerungszeit aufweisen.

Sowohl in den tile-basierten Verfahren wie auch in dem Shifting-Ansatz werden Reserve-Logikblöcke über das gesamte FPGA verteilt. Durch diese, aus Sicht des Place&Route-Vorgangs „störenden“, Logikblöcke wird in vielen Fällen eine Platzierung von den Pfaden, die nur wenig Slack enthalten, erschwert.

Da für die Realisierung eines dynamischen Place&Route-Vorgangs ein Hardware-Overhead in jedem Logikblock notwendig ist, der in Architekturen wie z.B. der Cell Matrix [DM01] beschrieben wird, ist die Verzögerungszeit auf jedem Pfad durch das Array bei diesen Ansätzen höher, als bei einem herkömmlichen FPGA. Die Timing Closure bzgl. von Timing-Constraints, wie sie bei einer vergleichbaren Schaltung auf einem traditionellen FPGA realisiert werden können, ist in diesem Fall nicht erreichbar.

Verfahren, die mit lokaler Redundanz arbeiten und Verfahren, die in FPGA-Architekturen mit segmentierten Verbindungsleitungen zusätzliche Logikblöcke einfügen, verändern ebenfalls die minimalen Verzögerungszeiten, die in diesen Architekturen erreicht werden können.

#### „Timing-closure friendly“ bei dem Hard-Core basierten Ansatz

Das Hard-Core basierte Verfahren bietet als erstes Verfahren für die Nutzung von defekten FPGAs die Möglichkeit den in den Pfaden vorhandenen Slack zu berücksichtigen, wenn durch einen Austausch der Hard-Core-Versionen Reserve-Logikblöcke eingefügt werden. Nachdem ein DT-Mapping erstellt wurde, kann der Place&Route-Vorgang mit den notwendigen Timing-Constraints durchgeführt werden. Durch eine abschließende Timing-Analyse nach dem Place&Route-Vorgang wird sichergestellt, dass die maximale Taktfrequenz durch das Design eingehalten wird. Das die gleiche Taktfrequenz erreicht werden kann, die vor dem Re-Mapping durch das Place&Route-Werkzeug erreicht wurde, kann allerdings auch im Hard-Core basierten Ansatz nicht garantiert werden.

### 6.3.9 Reconfigurability

Die applikationsspezifischen FPGAs (ASFPGA), wie sie z.B. von der Firma Xilinx angeboten werden [eas08], können nur mit FPGA-Konfigurationen verwendet werden, mit denen sie vorher bei dem FPGA-Hersteller getestet wurden, d.h. für den Fall, dass während der Serienproduktion eine Nachbesserung des Designs erforderlich ist, können die bereits mit der FPGA-Konfiguration getesteten und erworbenen FPGAs nicht mehr verwendet werden. Ein nachträgliches Update einer FPGA-Konfiguration nach einer Auslieferung des Produktes an den Kunden ist ebenfalls nicht möglich.

#### „Reconfigurability“ bei dem Hard-Core basierten Ansatz

Die Rekonfigurierbarkeit der FPGAs bleibt im Hard-Core basierten Ansatz erhalten. Eine neue DT-Konfiguration kann mit einer hohen Wahrscheinlichkeit für die Konfigu-

ration derselben FPGAs verwendet werden, die auch in Verbindung mit der vorherigen DT-Konfiguration genutzt wurden. Falls während der Serienproduktion eine Nachbesserung der DT-Konfiguration erforderlich ist, können die bereits erworbenen defekten FPGAs weiterhin verwendet werden, da die einfache Überprüfung der Eignung der FPGAs für die vorliegende DT-Konfiguration – anders als das aufwändige Testen der defekten FPGAs – bei dem Hersteller des Serienproduktes erfolgen kann.

Auch ein nachträgliches Update der FPGA-Konfiguration nach der Auslieferung des Produktes an den Kunden kann mit einer hohen Wahrscheinlichkeit schnell durchgeführt werden, wenn eine Möglichkeit vorhanden ist, auf die *defect map* des in diesem Produkt verbauten FPGAs zuzugreifen. Falls die neue DT-Konfiguration auf dem defekten FPGA nicht platziert werden kann, ist ein individuelles Place&Route für das FPGA erforderlich.

| Kriterium | Bedeutung |
|---|---|
| **Defect coverage** | Da aus strategischen Gründen keine Daten bezüglich der Ausbeute an FPGAs durch die FPGA-Hersteller veröffentlicht werden, ist keine Verteilung der Defekt-Typen bekannt. Anstelle einer Beurteilung, welcher Anteil der in einer Produktion tatsächlich auftretenden Defekte durch das bewertete Verfahren abgedeckt wird, kann also an dieser Stelle nur ein Vergleich, bzgl. der Anzahl und der Art der Defekte, die toleriert werden können, erfolgen. |
| **Area overhead** | Unter diesem Punkt wird diskutiert, in welchem Umfang und in welcher Form Redundanz auf dem FPGA benötigt wird, um einen Ausgleich von defekten Komponenten zu ermöglichen. Es wird nur die zusätzliche Fläche für redundante Komponenten betrachtet und nicht der zusätzliche Flächen-Aufwand, der durch das betrachtete Verfahren entsteht (vgl. Kriterium „Extra HW required“) |
| **Timing overhead** | Die mit vielen der betrachteten Verfahren einhergehende Vergrößerung von Verzögerungszeiten einzelner Verbindungen innerhalb der FPGA-Konfiguration kann zu einer Reduzierung der maximal möglichen Taktfrequenz führen mit der diese Konfiguration genutzt werden kann. Wenn das erwartete Zeitverhalten des FPGAs durch diese Vergrößerung nicht eingehalten werden kann, ist eine manuelle Änderung der Schaltungsbeschreibung notwendig. Falls auch durch Änderungen in der Schaltungsbeschreibung das erwartete Zeitverhalten zusammen mit dem Verfahren zur Tolerierung von Defekten nicht garantiert werden, ist eine Verwendung des Verfahrens nicht möglich. |
| **Bitfile size** | Wenn die Programmierung eines FPGAs unabhängig vom ATE in dem Produkt in dem das FPGA verwendet wird zur Laufzeit erfolgen soll, ist die Menge der für diese Programmierung benötigten Daten ein relevantes Kriterium für die Bewertung des Verfahrens, da diese Daten zusätzlichen Speicher in dem Produkt erfordern. |
| **Extra HW required** | Wenn das betrachtete Verfahren Defekte umgeht, erfolgt dies oft durch eine Anpassung des FPGAs oder der FPGA-Konfiguration an die neuen Gegebenheiten. Die für diese Anpassung erforderlichen zusätzlichen Schaltelemente müssen bei einigen der vorgestellten Verfahren in die FPGA-Architektur integriert werden. Da diese Schaltelemente nur im Falle eines Defektes genutzt werden können, sind aus Sicht dieses Kriteriums Verfahren vorteilhaft, die innerhalb der FPGA-Architetur nur einen geringen zusätzlichen Aufwand erfordern. |

Tabelle 6.2: Die in Abbildung 6.4 verwendeten Kriterien

| Kriterium | Bedeutung |
|---|---|
| **Maturity** | Dieses Kriterium ermöglicht die Unterscheidung zwischen Verfahren, die bereits in der Praxis eingesetzt wurden, Verfahren, die bereits durch verschiedenen Forschergruppen untersucht wurden und als weitgehend ausgereift angesehen werden können und Verfahren, die noch weit von einer Nutzung in einem kommerziell einsetzbaren Produkt entfernt sind. |
| **Mass production friendly** | Unter diesem Kriterium wird betrachtet, ob das Verfahren für FPGAs angewendet werden kann, die in einer Serienproduktion eingesetzt werden. |
| **Timing-closure friendly** | Wenn für ein FPGA-Design eine minimale Taktfrequenz vorgegeben ist, mit der das Design betrieben werden soll, oder andere Zeit-Constraints vorgegeben sind, besteht eine Aufgabe des FPGA-Schaltungsdesigners darin, die Schaltung so zu entwerfen, dass die Zeit-Constraints mit der erstellten FPGA-Konfiguration eingehalten werden. Diese Aufgabe wird allgemein als Timing-Closure-Problem bezeichnet. Moderne Entwurfswerkzeuge unterstützen den Designer bei der Timing-Closure durch die Möglichkeit Zeit-Constraints vorzugeben, die bei der Durchführung des Place&Route-Vorgangs berücksichtigt werden..<br>Unter diesem Kriterium wird betrachtet, ob der Designer durch das betrachtete Verfahren dabei unterstützt wird, die Timing-Closure, also die Einhaltung aller Zeit-Constraints, zu erreichen. |
| **Reconfigurability** | Unter diesem Kriterium wird betrachtet, ob das FPGA nur einmalig programmiert werden kann, oder ob eine Rekonfiguration möglich ist, um die FPGA-Konfiguration für ein Produkt zu korrigieren oder zu optimieren. |

Tabelle 6.3: Die in Abbildung 6.4 verwendeten Kriterien (Fortsetzung)

## 6.4 Zusammenfassung

In diesem Kapitel wurde zunächst beschrieben, wie die Verbesserung der Ausbeute durch den Hard-Core basierten Ansatz mit dem Poisson-Modell grob abgeschätzt werden kann, sofern die dafür benötigten Parameter bekannt sind.

Für den Kunden des FPGA-Herstellers, der defekte FPGAs in einer Serienproduktion einsetzen möchte, sind günstigere Stückpreise für die FPGAs zu erwarten. Diesen Einsparungen stehen allerdings zusätzliche Kosten gegenüber, die entstehen, wenn der Hard-Core basierte Ansatz für die Nutzung dieser FPGAs verwendet wird. In Abschnitt 6.2 wurde ein Ansatz für die Modellierung der eingesparten Kosten bei der Verwendung von defekten FPGAs vorgestellt.

Zum Abschluss des Kapitels erfolgte ein ausführlicher Vergleich des Hard-Core basierten Ansatzes mit den in Kapitel 4 vorgestellten vergleichbaren Arbeiten.

# 7 Zusammenfassung

Die Verwendung von IP-Cores im FPGA-Design hat vor allem im letzten Jahrzehnt laufend zugenommen. Moderne FPGAs verfügen über ausreichend Logik-, Speicher- und Routing-Ressourcen, um umfangreiche eingebettete Systeme, bestehend aus Prozessoren, Software und anwendungsspezifischer Hardware, auf einem FPGA zu realisieren. Dieses Wachstum an Komplexität und Umfang führt in vielen Fällen zu einer Annäherung an die Machbarkeitsgrenzen bei der Herstellung von FPGAs. Diese Annäherung geschieht auf Kosten der erreichbaren Ausbeute bei der Chip-Herstellung. Durch die Nutzung von leicht defekten FPGAs ist dann eine deutliche Steigerung der Ausbeute möglich.

Innerhalb der vorliegenden Arbeit wurde erstmalig ein Hard-Core basierter Ansatz für die Nutzung von FPGAs mit defekten Logikblöcken entwickelt. Dieser Ansatz zeichnet sich durch die folgenden Eigenschaften aus:

- Es ist keine Veränderung der bestehenden FPGA-Architekturen notwendig
- Die Rekonfigurierbarkeit der defekten FPGAs bleibt erhalten
- Das Verfahren berücksichtigt bei der Erstellung der DT-Konfiguration den in der FPGA-Konfiguration vorhandenen Slack und unterstützt dadurch die Timing-Closure für das FPGA-Design
- Der Aufwand für die Nutzung von defekten FPGAs wird auf drei Schritte verteilt:

    1. Bereitstellung von defekttoleranten IP-Core-Versionen
    2. Erstellung einer defekttoleranten FPGA-Konfiguration (bezeichnet als DT-Konfiguration)
    3. Anpassung der DT-Konfiguration an jedes defekte FPGA innerhalb der Serienproduktion

Die Anpassung der DT-Konfiguration an jedes defekte FPGA innerhalb der Serienproduktion ist der einzige der oben angegebenen Schritte, der für jedes defekte FPGA erneut ausgeführt werden muss. Durch die vorliegenden prekompilierten Varianten der Hard-Core-Versionen kann dieser Schritt mit einem geringen Aufwand durch den in Abschnitt 5.4 dargestellten Algorithmus erfolgen.

Im Gegensatz zu anderen Verfahren, die mit prekompilierten Schaltungsteilen arbeiten, wie z.B. dem von Lach et al. [LMSP98b] vorgeschlagenen Verfahren, kann die Bereitstellung der prekompilierten Schaltungsteile im Hard-Core basierten Ansatz durch den IP-Core-Anbieter erfolgen. Durch diese Auslagerung der in Abschnitt 5.2 erläuterten Bereitstellung von prekompilierten Schaltungsteilen in Form von DT-Hard-Core-Versionen kann die Erstellung einer DT-Konfiguration innerhalb des Design-Flows wesentlich einfacher automatisiert werden, als bei Verfahren, die eine Partitionierung der Schaltung innerhalb des Design-Flows erfordern.

Da von den FPGA-Herstellern keine Informationen herausgegeben werden, aus denen Rückschlüsse auf den bei der Herstellung der FPGAs erzielten Yield möglich sind, konnte im Rahmen dieser Arbeit nur eine grobe Abschätzung der möglichen Steigerung des Yields, die entsteht, wenn FPGAs mit defekten Logikblöcken genutzt werden können, erfolgen. Diese Abschätzung wurde basierend auf einem gängigen Ausbeutemodell vorgenommen und führt zu dem Ergebnis, dass der *random yield* bei einem ursprünglichen Wert von $Y_{RANDOM} = 0,5$ auf $Y_{RANDOM} = 0,66$ verbessert wird, wenn 40% der auf dem Chip im Durchschnitt auftretenden Defekte auf einen Logikblock begrenzt sind und FPGAs mit defekten Logikblöcken nicht als Ausschuss, sondern als verwendbare Chips betrachtet werden. Wenn man von $Y_{RANDOM} = 0,8$ ausgeht, kann unter den gleichen Bedingungen eine Verbesserung auf $Y_{RANDOM} = 0,875$ erreicht werden. Um einen Anreiz für den Kunden des FPGA-Herstellers zu schaffen, defekte FPGAs in der Serienproduktion einzusetzen, müssten defekte FPGAs jedoch günstiger als voll funktionsfähige FPGAs angeboten werden.

Das Hard-Core basierte Verfahren wurde in Abschnitt 6.3 als ein aktives Verfahren klassifiziert. Eine Voraussetzung für alle aktiven Verfahren, die eine Nutzung von defekten FPGAs ermöglichen, ist die vorherige Diagnose der defekten FPGAs. Wegen der kostspieligen Testzeit wird ein Test bei heutigen Test-Verfahren allerdings sofort abgebrochen, wenn ein Fehler detektiert wurde. Die so eingesparte Testzeit ist signifikant, da sehr viele Defekte bereits mit den ersten der angelegten Testmuster detektiert werden können. Eine Diagnose, durch die zusätzliche Kosten für defekte FPGAs entstehen, wird daher zur Zeit generell als nicht wirtschaftlich durchführbar angesehen [Tri08]. In Kapitel 3 wurden Forschungsansätze zu BIST-Verfahren für FPGAs vorgestellt, die eine kostengünstigere Diagnose defekter Logikblöcke versprechen. Erst wenn ein wirtschaftliches Diagnose-Verfahren zur Verfügung steht und von den FPGA-Herstellern eingesetzt wird, kann das Hard-Core basierte Verfahren für die Nutzung von defekten FPGAs in der Serienproduktion verwendet werden.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[Act97] Actel Inc. *Introduction to Actel FPGA Architecture*, 1997.

[alt00] Altera's patented redundancy technology dramatically increases yield on high-density apex(tm) 20ke devices. Technical report, Altera Inc., 2000.

[Alt07] Altera Inc. *Stratix III Programmable Power*, 2007.

[alt08] Leveraging the 40-nm process node to deliver the world's most advanced custom logic devices. Technical report, Altera Inc., 2008.

[AS01] Miron Abramovici and Charles E. Stroud. Bist-based test and diagnosis of fpga logic blocks. In *IEEE Transactions On VLSI Systems*, 2001.

[ASE04] Miron Abramovici, Charles E. Stroud, and John M. Emmert. Online BIST and BIST-based diagnosis of FPGA logic blocks. *IEEE Trans. VLSI Syst*, 12(12):1284–1294, 2004.

[ASH+99] Miron Abramovici, Charles E. Stroud, Carter Hamilton, Sajitha Wijesuriya, and Vinay Verma. Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications. In *ITC*, pages 973–982, 1999.

[ASSE00] Miron Abramovice, Charles Stroud, Brandon Skaggs, and John Emmert. Improving on-line BIST-based diagnosis for roving STARs. In *Proc. 6th IEEE International On-Line Testing Workshop*, July 03 2000.

[AT05] Ghazanfar Asadi and Mehdi Baradaran Tahoori. Soft error rate estimation and mitigation for SRAM-based FPGAs. In *FPGA*, pages 149–160, 2005.

[Bir98] John Birkner. Hdl ip cores in fpgas to drive pace of innovation, Januar 1998.

[BR96] Stephen Brown and Jonathan Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Transactions on Design and Test of Computers*, 13:42–57, August 15 1996.

[BR99] Vaughn Betz and Jonathan Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 59–68, 1999.

[BRM99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[CAC+97] W. Bruce Culbertson, Rick Amerson, Richard J. Carter, Philip Kuekes, and Greg Snider. Defect tolerance on the teramac custom computer. In *FCCM*, pages 116–124, 1997.

[Car06] C. Carmichael. Triple module redundancy design techniques for vortex series fpga. Technical report, Xilinx Application Notes 197, v1.0.1, Juli 2006.

[CBSS02] Chen, Bozorgzadeh, Srivastava, and Sarrafzadeh. Budget management with applications. *ALGRTHMICA: Algorithmica*, 34, 2002.

[CCP06] Deming Chen, Jason Cong, and Peichan Pan. *FPGA Design Automation: A Survey*. now Publishers Inc., 2006.

[CEB06] Jason Cheatham, John M. Emmert, and Stan Baumgart. A survey of fault tolerant methodologies for fpgas. *ACM Transactions on Design Automation of Electronic Systems*, 11:501–533, 2006.

[CGJW02] Michael Caffrey, Paul Graham, Eric Johnson, and Michael Wirthlin. Single-event upsets in sram fpgas. In *MAPLD*, 2002.

[CK06] Nathaniel Couture and Kenneth B. Kent. Periodic licensing of FPGA based intellectual property. In Steven J. E. Wilton and André DeHon, editors, *FPGA*, page 234. ACM, 2006.

[DH06] Asbjoern Djupdal and Pauline C. Haddow. Yield enhancing defect tolerance techniques for fpgas. In *Proceedings of the 2006 MAPLD International Conference*, 2006.

[DI00a] Abderrahim Doumar and Hideo Ito. Defect and fault tolerance sram-based fpgas by shifting the configuration data. In *IEICE Trans. Inf. & Syst.*, volume E83-D, May 2000.

[DI00b] Abderrahim Doumar and Hideo Ito. Design of switching blocks tolerating defects/faults in FPGA interconnection resources. In *DFT*, pages 134–142. IEEE Computer Society, 2000.

[DI03] Abderrahim Doumar and Hideo Ito. Detecting, diagnosing, and tolerating faults in sram-based field programmable gate arrays: A survey. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, 11(3):386–405, June 2003.

[DM01] Liesa J.K. Durbeck and Nicholas J. Macias. The cell matrix: an architecture for nanocomputing. *Nanotechnology*, 12:217–230, 2001.

[DP94] Serge Durand and Christian Piguet. Fpga with self-repair capabilities. In *Proc. International ACM / SIGDA Workshop on FPGAs*, 1994.

[DST99] Shantanu Dutt, Vimalvel Shanmugavel, and Steve Trimberger. Efficient incremental rerouting for fault reconfiguration in field programmable gate arrays. In *International Conference on Computer-Aided Design (ICCAD '99)*, pages 173–177, Washington - Brussels - Tokyo, November 1999. IEEE.

[eas08] Easypath product brochure. Technical report, Xilinx Inc., 2008.

[FP92] Albert V. Ferris-Prabhu. *Introduction to Semiconductor Device Yield Modeling*. Artech House, 1992.

[GBH+06] Soheil Ghiasi, Elaheh Bozorgzadeh, Po-Kuan Huang, Roozbeh Jafari, and Majid Sarrafzadeh. A unified theory of timing budget management. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(11):2364–2375, 2006.

[GCZJ03] Paul Graham, Michael Caffrey, Jason Zimmerman, and D. Eric Johnson. Consequences And Categories Of SRAM FPGA Configuration SEUs. Technical report, Los Alamos National Laboratory, 2003.

[GLPS97] Patrick Girard, Christian Landrault, Serge Pravossoudovitch, and D. Severac. A gate resizing technique for high reduction in power consumption. In Brock Barton, Massoud Pedram, Anantha Chandrakasan, and Sayfe Kiaei, editors, *ISLPED*, pages 281–286. ACM, 1997.

[HD98] Fran Hanchek and Shantanu Dutt. Methodologies for tolerating cell and interconnect faults in FPGAs. *IEEE Trans. Computers*, 47(1):15–33, 1998.

[HGWS99] Carter Hamilton, Gretchen Gibson, Sajitha Wijesuriya, and Charles E. Stroud. Enhanced bist-based diagnosis of FPGAs via boundary scan access. In *VTS*, pages 413–419. IEEE Computer Society, 1999.

[HKSW98] Heath, Kuekes, Snider, and Williams. A defect-tolerant computer architecture: Opportunities for nanotechnology. *SCIENCE*, 280:1716–1721, 1998.

[HSN+93] F. Hatori, T. Sakurai, K. Nogami, K. Sawada, M. Takahashi, M. Ichida, M. Uchida, I. Yoshii, Y. Kawahara, T. Hibi, Y. Saeki, H. Muroga, A. Tanaka, and K. Kanzaki. Introducing redundancy in field programmable gate arrays. In *IEEE Custom Integrated Circuit Conference*, pages 7.1.1–7.1.4, 1993.

[HTA94] Neil J. Howard, Andrew M. Tyrrell, and Nigel M. Allinson. The yield enhancement of field-programmable gate arrays. *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, 2(1):115–123, March 1994.

[Kea02] Tom Kean. Cryptographic rights management of FPGA intellectual property cores. In *Proceedings of the ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 113–118, 2002.

[KI94] Jason L. Kelly and Peter A. Ivey. Defect tolerant SRAM based FPGAs. In *ICCD*, pages 479–482. IEEE Computer Society, 1994.

[KK98] Israel Koren and Zahava Koren. Defect tolerance in vlsi circuits: Techniques and yield analysis. In *Proceedings of the IEEE*, volume 86, September 1998.

[KK07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2007.

[KL04] A. J. Kleinosowski and David J. Lilja. The nanobox project: Exploring fabrics of self-correcting logic blocks for high defect rate molecular device technologies. In *ISVLSI*, pages 19–24. IEEE Computer Society, 2004.

[LMSP98a] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Efficiently supporting fault-tolerance in fpgas. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field Programmable Gate Arrays*, pages 105–115, 1998.

[LMSP98b] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Low overhead fault-tolerant fpga systems. *IEEE Transactions on very large scale integration (vlsi) systems*, 6(2):212–220, 1998.

[LMSP00] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Enhanced fpga reliability through efficient run-time fault reconfiguration. In *IEEE Transactions On Reliability*, volume 49, September 2000.

[MD02] Nicholas J. Macias and Lisa J. K. Durbeck. Self-assembling circuits with autonomous fault handling. In *Evolvable Hardware*, pages 46–55. IEEE Computer Society, 2002.

[MD04] N.J. Macias and L. J. K. Durbeck. Adaptive methods for growing electronic circuits on an imperfect synthetic matrix. *Biosystems*, 73:173–204, 2004.

[MD05] Nicholas J. Macias and Lisa J. K. Durbeck. A hardware implementation of the cell matrix self-configurable architecture: The cell matrix MOD 88. In *Evolvable Hardware*, pages 103–106. IEEE Computer Society, 2005.

[Mey03] Volker Hans-Walther Meyer. *Test produktionsbedingter Laufzeitfehler in hochintegrierten, digitalen Schaltungen*. PhD thesis, Universität Bremen, 2003.

[MG03] Mahim Mishra and Seth Copen Goldstein. Defect tolerance at the end of the roadmap. In *ITC*, pages 1201–1211, 2003.

[Möh85] R.H. Möhring. *Algorithmic Aspects of Comparability Graphs and Interval Graphs*. D. Reidel Publishing Company, 1985.

[RBHSC82] Sr. Robert B. Hitchcock, Gordon L. Smith, and David D. Cheng. Timing analysis of computer hardware. *IBM Journal of Research and Development*, 26(1):100–105, January 1982.

[REGSV93] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field programmable gate arrays. In *Proceedings of the IEEE*, pages 1013 – 1029, 1993.

[RH97] Jonathan Rose and Dwight D. Hill. Architectural and physical design challenges for one-million gate FPGAs and beyond. In *FPGA*, pages 129–132, 1997.

[RWL05] Tero Rissa, Steven J. E. Wilton, and Philip Heng Wai Leong, editors. *Defect-Tolerant FPGA Switch Block and Connection Block with Fine-Grain Redundancy for Yield Enhancement*. IEEE, 2005.

[SD06] Vishal Suthar and Shantanu Dutt. Mixed PLB and interconnect BIST for FPGAs without fault-free assumptions. In *VTS*, pages 36–43. IEEE Computer Society, 2006.

[SKCA96] Charles E. Stroud, Srinivasa Konala, Ping Chen, and Miron Abramovici. Built-in self-test of logic blocks in FPGAs (finally, a free lunch: BIST without overhead!). In *VTS*, pages 387–392, 1996.

[SLA97] Charles E. Stroud, Eric Lee, and Miron Abramovici. BIST-based diagnostics of FPGA logic blocks. In *ITC*, pages 539–547. IEEE Computer Society, 1997.

[SLKA96] C. Stroud, E. Lee, S. Konala, and M. Abramovici. Using ILA testing for BIST in FPGAs. In *International Test Conference (ITC '96)*, pages 68–75, Washington - Brussels - Tokyo, October 1996. IEEE.

[SMSP98] N. R. Shnidman, William H. Mangione-Smith, and Miodrag Potkonjak. On-line fault detection for bus-based field programmable gate arrays. *IEEE Trans. VLSI Syst*, 6(4):656–666, 1998.

[SNLA02] C. Stroud, J. Nall, M. Lashinsky, and M. Abramovici. BIST-based diagnosis of FPGA interconnect. In *International Test Conference 2002 (ITC '02)*, pages 618–627, Washington - Brussels - Tokyo, October 2002. IEEE.

[Sri03] Ankur Srivastava. Simultaneous vt selection and assignment for leakage optimization. In *Proceedings of the international symposium on Low power electronics and design*, pages 146–151, 2003.

[SS99] Andreas Steininger and Christoph Scherrer. On the necessity of on-line-BIST in safety-critical applications - A case study. In *FTCS*, pages 208–215, 1999.

[SWHA98] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici. Built-in self-test of FPGA interconnect. In *International Test Conference 1998 (ITC '98)*, pages 404–411, Washington - Brussels - Tokyo, October 1998. IEEE.

[Syn04] Synopsys. *Synopsys Online Documentation - PrimeTime User Guide: Fundamentals*, 2004.

[Tri95] Steven Trimberger. Effects of FPGA architecture on FPGA routing. In *Design Automation Conference*, pages 574–578, 1995.

[Tri08] Steven M. Trimberger. Xilinx, Inc., April 2008. Private Communication.

[VDS04] Vinay Verma, Shantanu Dutt, and Vishal Suthar. Efficient on-line testing of FPGAs with provable diagnosabilities. In *Proceedings of the 41st Annual conference on Design Automation (DAC-04)*, pages 498–503, New York, June 7–11 2004. ACM Press.

[WL94] M.S. Won and C. Lytle. Continuous interconnect in the flex 8000 architecture. In *WESCON/94 Idea/Microelectronics Conference Record*, pages 673–676, September 1994.

[Xil07a] Xilinx. *Constraints Guide*, 9.1i edition, 2007.

[Xil07b] Xilinx Inc. *Virtex-II Platform FPGAs: Complete Data Sheet (v3.5)*, November 2007.

[Xil08] Xilinx Inc. *Virtex-5 Family Overview*, 2008.

# Lebenslauf

| | |
|---|---|
| 1974 | geboren in Oldenburg (Oldb) |
| 1980 - 1984 | Grundschule Osternburg in Oldenburg |
| 1984 - 1986 | Orientierungsstufe Osternburg in Oldenburg |
| 1986 - 1990 | Realschule Osternburg in Oldenburg |
| 1990 - 1993 | Fachgynasium Technik an der BBS II in Oldenburg |
| 1993 - 1994 | Zivildienst |
| 1994 - 2002 | Informatikstudium an der Carl von Ossietzky Universität Oldenburg<br>Nebenfach: Musik<br>Abschluss: Diplom Informatiker |
| 2002 - 2008 | Wissenschaftlicher Mitarbeiter in Forschung und Lehre<br>an der Carl von Ossietzky Universität Oldenburg<br>sowie am „Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme“(OFFIS)<br>Arbeitsgruppe: System Design Methodik |
| 2008 - 2009 | Abschluss der Promotion |
| seit 2009 | Lehrer am Friedrich List Berufskolleg in Bonn |