Algorithmen für Chip-Entwurfswerkzeuge

Embedded Systems & Applications

Praktikumsleitfaden WS 2012/2013

Florian Stock, Prof. Andreas Koch

TECHNISCHE UNIVERSITÄT DARMSTADT

Eingebettete Systeme und ihre Anwendungen, Darmstadt

Inhaltsverzeichnis

1 Einleitung

Dieser Leitfaden soll Sie erfolgreich durch das Praktikum "Algorithmen für Chip-Entwurfswerkzeuge" führen. In dem Praktikum wird die praktische Arbeit in Gruppen einen signifikanten Stellenwert haben, die zweite Semesterhälfte sogar dominieren. Neben der Programmierarbeit werden Sie Ihre neuerworbenen Kenntnisse auch in Kolloquien und kurzen Vorträgen demonstrieren können. Diese und andere organisatorischen Details werden im ersten Teil des Leitfadens erklärt.

Der zweite Teil beschreibt die technischen Hintergründe (Grundlagen, Dateiformate, etc.) der Programmieraufgaben.

Wichtiger Hinweis: Die einzelne Aufgaben bauen aufeinander auf und werden im Laufe der Veranstaltung deutlich komplexer. Nehmen Sie sich für das Semester nicht noch diverse andere Praktika vor, wenn Sie ACE erfolgreich praktisch absolvieren wollen!

1.1 Abgabemodalitäten

Die Abgabtermine liegen jeweils an einem Montag, wobei die Abgaben selbst bis 23:59 Uhr dieses Tages eingehen können. Alle Abgaben erfolgen elektronisch durch E-Mail an die Adresse

`ace@esa.informatik.tu-darmstadt.de`

Programme sind dabei als **`.jar`**-Archiv zu packen. Ein solches Archiv enthält neben den **`.java`**-Quellen auch die vorkompilierten Klassen (**`.class`**-Dateien) und eine **`README`**-Datei. Letztere beschreibt die zur Kompilierung nötigen Kommandos (**`javac`** Aufrufe, kein Geklicke in einem IDE!) sowie den Aufruf des Programmes von der Kommandozeile aus. Auch die Kommandozeilenparameter sollen hier dokumentiert werden. Die Quellen die im jar-sind sollen in einer Sandbox compilierbar sein, d.h. das auch alle Bibliotheken die sie benutzen in dem **`.jar`** enthalten sein müssen. Weiterhin *muss* beschrieben werden, was *konkret* die einzelnen Gruppenmitglieder zu der abgegebenen Lösung beigetragen haben. Der Titel jeder Mail *muss* beginnen mit

`Gruppe` N **`Aufgabe`** M ....

Für einige der Lösungen sind weitere Abgaben erforderlich (Ergebnisdateien, PDFs, etc.). Diese werden in den jeweiligen Aufgabenstellungen gesondert beschrieben.

1.2 Kolloquien und Vorträge

Einige Tage nach jeder Abgabe findet mit jeder Gruppe ein ca. 30-minütiges Einzelkolloquium statt. Der genaue Zeitpunkt wird in der Kick-Off-Veranstaltung zu den praktischen Arbeiten gruppenspezifisch festgelegt.

In der Vorlesungszeit am Freitag findet dann eine zentrale Besprechung der Lösungen statt. Hier sollen alle Gruppen in 10 bis 15-minütigen Vorträgen über ihre Lösung der praktischen Aufgabe referieren.

Die Kurzvorträge, bei denen jedes der Gruppenmitglieder mindestens einmal präsentieren wird, sollen beispielsweise folgende Themen anschneiden:

1. Vorgehensweise bei der Lösung
2. Kernalgorithmus und Datenstrukturen
3. Aufbau des Programms
4. Ergebnisse
5. Erfahrungen und Kommentare

Zu einem solchen Vortrag sollen begleitende Folien vorbereitet werden. Diese können, soweit sie als PowerPoint, PDF oder einem der OpenOffice-Formate vorliegen, auf einem bereitgestellten Notebook präsentiert werden (Transfer dorthin via CD/DVD, USB-Stick). Alternativ kann natürlich auch ein selbst mitgebrachtes Notebook verwendet werden.

1.3 Programmierstil

Die von Ihnen erstellten Programme werden in der Endfassung erfahrungsgemäß zwischen 15.000 und 20.000 Zeilen Java umfassen. Um dem Betreuer das Verständnis und Ihnen die Wartung zu erleichtern, sollen Sie von Anfang an einen sauberen und disziplinierten Programmierstil praktizieren.

Bei der Implementierung sind die Konventionen aus *Writing Robust Java Code* weitgehend einzuhalten. Dieses Dokument liegt als PDF auch auf der Web-Seite der Vorlesung. Ergänzend soll folgendes beachtet werden:

- Achten Sie darauf, dass Klassen nicht zu komplex werden (zu viele Instanzvariablen, zu viele Methoden). Bei deutlich mehr als 20 dieser Konstrukte sollten Sie die Klasse aufteilen.
- Analoges gilt für die Komplexität von einzelnen Methoden. Auch hier sollten Sie bei mehr als 100 Programmzeilen Länge die Methode aufteilen.
- Verwenden Sie statt Abfragen von **`instanceof`** echte objekt-orientierte Konstrukte (z.B. polymorphe Methoden).

Der Test und die Abnahme Ihrer Programme wird vom Betreuer auf Linux mit dem SUN Java Development Kit (JDK) Version 1.6 erfolgen.

1.4 Dokumentation

Der Schwerpunkt Ihrer Arbeit in den ersten Phasen der Veranstaltung liegt auf der Programmierung, hier werden daher nur abgeschwächte Anforderungen an die Dokumentation gestellt. In der letzten Phase verschiebt sich dieses Gleichgewicht dagegen deutlich!

1.4.1 Aufgabe 1 bis 3

Die ersten drei Lösungen werden nur durch das oben beschriebene **`README`** und die in das Java-Programm eingebetteten JavaDoc-Direktiven und Kommentare dokumentiert. Achten Sie daher

darauf, dass Sie von diesen beiden Möglichkeiten ausreichend und aussagekräftig Gebrauch machen.

Kommentare sollen am Anfang jeder Datei, pro Klasse und pro Instanzvariable und Methode verfasst werden. Bei Verwendung relativ kurzer Methoden und aussagekräftiger Bezeichner können sich Kommentare innerhalb von Methoden nur auf die wirklich wichtigen Stellen beschränken.

Der Datei-Kommentar muss neben einer allgemeinen Beschreibung auch eine Historie von Änderungen enthalten. Jeder Eintrag in dieser Historie beschreibt unter Angabe von Datum/Uhrzeit und Namen des Autors auf 1-2 Textzeilen die Natur der Änderungen. Alternativ kann hier auch das Log Ihres Versionskontrollsystems (z.B. CVS oder besser SVN) beigelegt werden. In diesem Fall sollten Sie darauf achten, dass Sie ausreichend oft einchecken und *aussagekrätige* Kommentare bei jedem Check-In angeben!

Diese Angaben sind für den Betreuer wichtig, damit im Kolloquium die für ein Thema passenden Ansprechpartner gefunden werden!

1.4.2 Aufgabe 4

Die vierte und damit letzte Abgabe wird neben Verfeinerungen der Implementierung im wesentlichen aus einer umfassenden Dokumentation des Gesamtsystems bestehen. Zu den in diesem 20-30 seitigen Werk zu betrachtenden Aspekten gehören:

- Benutzerhandbuch, u.a. mit Beschreibung von Aufrufparametern
- Algorithmische Grundlagen
- Datenstrukturen und ihre Zusammenhänge untereinander
- Konkrete Realisierung in Java
- Profiling-Daten für die verschiedenen Betriebsmodi (sowohl in Bezug auf Speicher als auch Ausführungszeit)
- Messergebnisse analog zu Phase 1 bis 3 für die verschiedenen Betriebsmodi bei Anwendung auf die Beispielschaltungen `s27` bis `clma`
- Vergleich der Messergebnisse nach der Verfeinerung in Aufgabe 4 mit den Ergebnissen der Vorversionen aus den Phasen 1 bis 3

Zur Darstellung der teilweise komplexen Zusammenhänge ist es unerlässlich, dass Sie diese durch Zeichnungen veranschaulichen. Dabei ist es Ihnen freigestellt, ob Sie eine lesbare und konsistente eigene Notation oder eine Standardnotation wie UML verwenden. Seitenweise Zeichnungen reichen aber alleine *nicht* aus. Sie müssen durch entsprechende Beschreibungen im Fliesstext erläutert und in den Kontext der Gesamtdokumentation (Algorithmenbeschreibung etc.) eingebettet werden.

Die Dokumentation soll als *eine* (1) konsistente Datei (fortlaufende Seitenzahlen, Inhaltsverzeichnis, etc.) im Adobe Portable Document Format (PDF) abgegeben werden. Bei Fragen zu der Erstellung von PDF wenden Sie sich ggf. an Kommilitonen oder den Betreuer.

Es ist Ihnen freigestellt, ob Sie dieses Abschlussdokument fortlaufend beginnend in den frühen Phasen pflegen, oder ob Sie es erst in der letzten Phase in Angriff nehmen. **Unterschätzen Sie aber keinesfalls den Aufwand zu seiner Erstellung!**

> In *allen* Phasen gilt: Fehlende, offensichtlich fehlerhafte oder unvollständige Dokumentation wird die Verweigerung der Abnahme und die Forderung von Nachbesserungen zur Folge haben.

1.5 Programmiertipps

Möglicherweise für Sie noch ungewohnt werden Ihre hier erstellten Programme tatsächlich in grösserem Umfang *rechnen*. Die Verfahren führen teilweise Millionen von Operationen auf Zehntausenden von Objekten aus. Laufzeiten bei der Bearbeitung größerer Schaltungen liegen selbst bei sehr geschickter Programmierung im Bereich von ca. einer halben *Stunde*. Ungeschicktere Ansätze lassen die Laufzeiten in die Dimension von Tagen und Wochen explodieren! Sie sollten also von Anfang an die Komplexität der von Ihnen verwendeten Algorithmen im Auge behalten. Dazu an dieser Stelle noch drei Tipps:

- Verwenden Sie bestehende Objekte/Datenstrukturen wenn möglich wieder und erstellen Sie nicht pausenlos Neue (insbesondere nicht in Schleifen)! Oft übersehen: Erzeugen Sie nicht innerhalb von Schleifen mit dem String-Konkatenationsoperator + ständig neue Strings. Falls dies für Debug- oder Log-Ausgaben doch nötig sein sollte, sehen Sie spezielle Debug- oder Log-Betriebsarten vor, die gezielt durch Kommandozeilenparameter aktiviert werden können. Im Normalbetrieb stört die so unterdrückte Erzeugung der Ausgaben dann nicht mehr.

- Ersetzen Sie sequentielle Suchen durch geschicktere Zugriffsverfahren wie beispielsweise Hash-Tabellen oder (noch besser) feste Verweise. Vermeiden Sie den häufigen Zugriff auf Daten über String-Vergleiche!

- Die Java 2 Collection-Klassen enthalten bereits eine ganze Reihe von Datenstrukturen wie Listen, Mengen, etc. Diese brauchen Sie also nicht von Grund auf neu zu erstellen. Konstrukte wie Iteratoren erlauben die Bearbeitung dieser Typen in übersichtlicher und konsistenter Weise. Allerdings sind diese Klassen in Bezug auf Rechenzeit weniger effizient als “primitive” Datenstrukturen (z.B. Felder fester Größe). Gegebenenfalls sollten Sie in wirklich zeitkritischen Bereichen (Profiling, siehe Abschnitt 1.8) auf diese unkomfortableren, aber schnelleren Implementierungen ausweichen.

Bei der Untersuchung des (Fehl-) Verhaltens Ihrer Implementierungen sind sogenannte Profiler sehr nützlich (siehe Abschnitt 1.8). Diese können die Schwerpunkte des Verbrauchs von Rechenzeit und Speicherplatz ermitteln und graphisch darstellen.

1.6 Testdaten

Für den Test Ihrer Programme werden Ihnen auf der Web-Seite der Veranstaltung Beispiele für Eingaben und Ausgaben in Form von entsprechenden Dateien zur Verfügung gestellt. Dazu gibt

es einen Minimalsatz von Testfällen, den Ihre Lösungen vollständig untersuchen müssen, sowie eine komplette Sammlung, mit der Sie noch weitergehende Tests durchführen können.

Für die einfacheren der Testfälle (z.B. die Schaltung s27) sind zur besseren Übersicht und dem Nachvollziehen der Ergebnisse auch noch graphische Darstellungen (in PostScript) beigelegt.

1.7 Team-Organisation

Durch die Komplexität der Aufgaben und die vorgesehenen Abgabetermine ist eine echte *Gruppenarbeit* unterläßlich. Teilen Sie also die Lösung jeder Aufgabe unter sich auf. In jeder Aufgabenstellung werden entsprechende Vorschläge enthalten sein.

Falls in Ihrer Gruppe eine Situation entstehen sollte, in der einzelne Mitglieder deutlich zuwenig (oder zuviel!) der anfallenden Arbeitslast bewältigen, sprechen Sie den Betreuer bitte *frühzeitig* auf die Problematik an. Nur so kann durch geeignete Maßnahmen in Ihrem Interesse gegengesteuert werden. Nach der Abgabe ist es dafür **zu spät** und Sie tragen die Konsequenzen selber (z.B. wenn sich eines Ihrer Team-Mitglieder wegen seiner Verpflichtungen beim Wasser-Polo nur stark eingeschränkt den Mühen der Programmierung widmen konnte, und Sie daher eine unvollständige Lösung abgeben mussten).

1.8 Werkzeuge

Zur Abnahme werden die abgegebenen Programme in einer sehr einfachen Umgebung kompiliert und getestet (`javac` von der Kommandozeile aus).

Ihre Programmierarbeiten können aber durch geeignete Werkzeuge deutlich schneller und komfortabler stattfinden. Die folgende Liste soll nur einige der Möglichkeiten aufzeigen:

- Für sichere Gruppenarbeit fast essentiell ist ein Versionsverwaltungssystem. Hier sei Subversion (`subversion.tigris.org`) empfohlen, für das es auch eine Reihe von graphischen Oberflächen gibt. Da die Software auch auf den RBG-Poolrechnern installiert ist, könnten Sie dort für Ihre Gruppe einen entsprechenden Server aufsetzen. Alleine schon beim Erstellen der für die Abgabe erforderlichen Historien (“Wer hat was gemacht?”) kann Ihnen ein solches Werkzeug viel Zeit ersparen.
- Komfortables Entwickeln und Debuggen von Java-Programmen ist in einer integrierten Umgebung möglich. Eclipse (`www.eclipse.org`) ist eines der mächtigsten derzeit verfügbaren Systeme. Es unterstützt auch teilweise die automatische Restrukturierung von Programmen (sogenanntes *Refactoring*), beispielsweise zum Aufteilen von zu lang gewordenen Methoden oder Klassen. Alternativen sind NetBeans und das seit kurzem auch frei verfügbare Inttelij IDEA.
- Automatische Regressionstests können zur Qualitätskontrolle während der Programmierung hilfreich sein. Systeme wie JUnit (`www.junit.org`) assistieren dabei durch das Bereitstellen eines Testrahmens.
- Profiler werten während der Ausführung von Java-Programmen automatisch gesammelte Statistikdaten aus. Sie erstellen daraus graphische Darstellungen von Rechenzeit- und Speicherverbrauch. Diese Angaben können dann als Grundlage für weitere Optimierun-

gen der Implementierung dienen. Eine ganze Reihe von Vorschlägen finden Sie z.B. auf `http://java-source.net/open-source/profilers`. Sehr einfache Tools zum Einstieg sind z.B. JIP für Laufzeit oder jmap/jhat für Speicherplatz. IDEs wie Eclipse und NetBeans bringen auch eingebaute Profiler mit. Falls Sie mit (komfortableren) kommerziellen Tools experimentieren wollen: Demo-Versionen gibt es z.B. von JProbe, YourKit, JProfiler, uvm.

- Die hier verwendeten Dateiformate sind so einfach, dass die zum Einlesen erforderlichen Lexer/Parser leicht manuell erstellt werden können. Als Alternative können hier aber auch Parser-Generatoren zum Einsatz kommen, die aus einer Grammatik automatisch entsprechenden Java-Code generieren. Hier sei beispielhaft auf das Werkzeug ANTLR (`www.antlr.org`) verwiesen.

Beachten Sie beim Durchgehen dieser Liste aber bitte, dass das wesentliche Lernziel der Veranstaltung *Algorithmen im Chip-Design* sein sollen, **nicht** aber Expertenkenntnisse verschiedener Programmierwerkzeuge! Wenn Sie sich nur Zeit für ein oder zwei der Werkzeuge nehmen wollen: Fangen Sie mit der Versionsverwaltung und dem Profiler an.

2 FPGA-Zielarchitektur

Im praktischen Teil der integrierten Veranstaltung befassen wir uns mit der Lösung verschiedener Entwurfsprobleme auf einer fiktiven, aber doch realistischen FPGA-Architektur. Die bearbeiteten Probleme liegen überwiegend im physikalischen Bereich, also dem Umgang mit Layouts und der Platzierung und Verdrahtung von Schaltungen. In diesem Dokument werden zunächst die Grundlagen der Zielarchitektur erklärt und dann die Dateiformate definiert, die die von Ihnen entwickelten Werkzeuge verwenden müssen.

Dieses Dokument setzt Grundkenntnisse über den Aufbau von FPGAs voraus, die Sie gegebenenfalls durch Lektüre der entsprechenden Kapitel aus dem Vorlesungsskript “Technische Grundlagen der Informatik I” von Prof. Koch wieder auffrischen können.

2.1 Blockmatrix

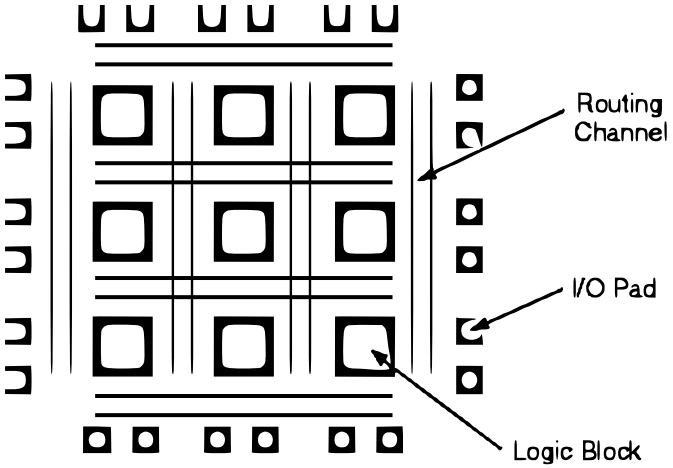

Abbildung 2.1: Struktur des FPGAs

Das fiktive FPGA (Abbildung 2.1) ist als symmetrische Matrix aufgebaut: Zwischen den so angeordneten Logikblöcken (Logic Blocks) verlaufen einzelne horizontale und vertikale Verdrahtungskanäle (Routing Channels), die insgesamt ein Verdrahtungsnetzwerk bilden. An den Aussenrändern liegen Ein/Ausgabeblöcke (I/O-Pads), die an die externen Pins des Chips angeschlossen sind.

2.1.1 Logikblöcke

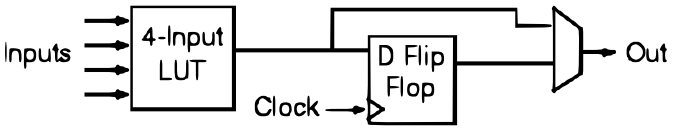

Abbildung 2.2: Aufbau eines Logikblockes

Die einzelnen Logikblöcke bestehen wie in Abbildung 2.2 gezeigt aus einer Wertetabelle (LUT) mit 4 1b Eingängen und einem 1b Ausgang. Dieser kann entweder direkt oder über ein optionales Flip-Flop an den Ausgang des Logikblockes angeschlossen werden. Bitte beachten Sie, dass die Zuordnung von den Eingängen des Logikblockes zu den Eingängen der Wertetabelle beliebig sein kann: Jede Vertauschung von Eingängen kann durch geeignete Anpassung des Inhaltes der Wertetabelle kompensiert werden.

Da wir uns bei den praktischen Arbeiten auf die physikalische Ebene konzentrieren, ist der Inhalt der Wertetabellen für Ihre Algorithmen bedeutungslos. Dagegen wird durch die Benutzung des Flip-Flops das Zeitverhalten der Schaltung, das Ihre Optimierungswerkzeuge ja berücksichtigen sollen, beeinflusst. Sie müssen daher zwischen kombinatorischer (ohne Flip-Flop) und sequentieller Nutzung (mit Flip-Flop) eines Logikblockes unterscheiden (siehe dazu auch Abschnitte 2.3 und 3.1.3).

2.1.2 Ein-/Ausgabeblöcke

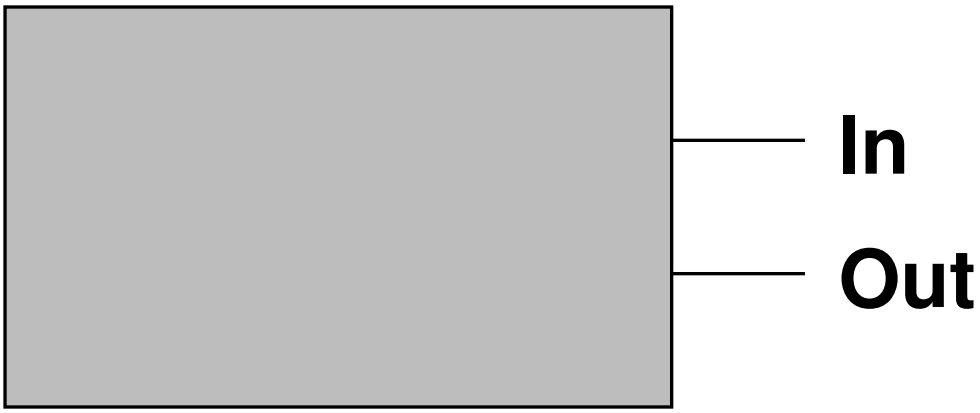

Abbildung 2.3: Ein-/Ausgabeblock

Jeder der an den Rändern der Logikblockmatrix gelegenen Ein-/Ausgabeblöcke hat einen Eingang und einen Ausgang zum Verdrahtungsnetzwerk (Abbildung 2.3). Wenn der Eingang beschaltet ist (siehe Abschnitt 3.1) agiert der Block als Ausgabeblock (ein Signal wird vom Verdrahtungsnetzwerk an den externen Chip-Pin angelegt). Bei Beschaltung des Ausgangs wird der Block als Eingabeblock verwendet (ein Signal wird von einem externen Chip-Pin an das Verdrahtungsnetzwerk angeschlossen). Bidirektionale Blöcke (Ein- *und* Ausgabe-Pins gleichzeitig verwendet) sind in unser FPGA-Architektur aus Vereinfachungsgründen nicht vorgesehen.

2.2 Konnektivität

Die in Abbildung 2.1 gezeigte Sicht der FPGA-Architektur ist stark abstrahiert. Von entscheidender Bedeutung ist die Konnektivität, also die Verbindbarkeit der unterschiedlichen Ressourcen untereinander sowie ihre geometrische Anordnung.

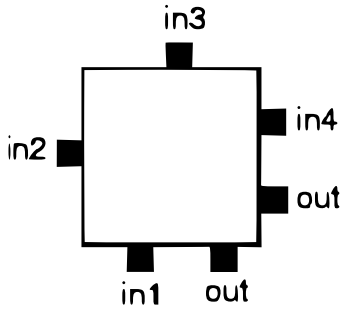

Abbildung 2.4: Anordnung der Pins eines Logikblocks

2.2.1 Logikblock-Pins

Beginnen wir zunächst mit der Beschreibung der Anschlüsse eines Logikblockes (Abbildung 2.4):

- Die vier Eingänge in1 bis in4 der Wertetabelle sind in der gezeigten Weise um den Logikblock herum angeordnet. Das heisst beispielsweise, dass in1 direkt nur in den *unterhalb* des Logik-Block gelegenen horizontalen Verdrahtungskanal geführt werden kann.
- Der Ausgang des Logikblocks out kann sowohl in den rechts vom Logikblock gelegenen vertikalen als auch in den darunter gelegenen horizontalen Kanal geführt werden.
- Der Takteingang Clock des Logikblockes wird geometrisch nicht explizit behandelt. Es wird davon ausgegangen, dass es dedizierte Verdrahtungsressourcen zur Führung des Taktes gibt (gängige Praxis bei realen FPGAs). Er muss allerdings von Ihren Werkzeugen trotzdem ausgewertet werden (Abschnitt 3.1.3).

2.2.2 Ein-/Ausgabeblock-Pins

Ein-/Ausgabeblöcke haben ihre Ein- und Ausgabe-Pins jeweils in Richtung des nächsten Verdrahtungskanals angeordnet. Beispielsweise können also an der Oberkante der Matrix gelegene Blöcke ihre beiden Pins in den darunter gelegenen horizontalen Kanal führen. Analog können Blöcke an der rechten Seite des Chips ihre Pins an den links daneben gelegenen vertikalen Kanal führen.

2.2.3 Verdrahtungskanäle

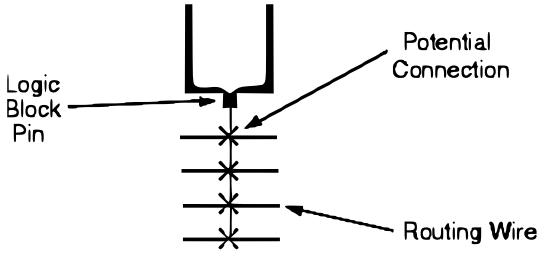

Abbildung 2.5: Aufbau eines Verdrahtungskanals

Die Anzahl von einzelnen Leitungen innerhalb eines Verdrahtungskanals, die sogenannte Kanalbreite, soll bei unser Zielarchitektur frei einstellbar sein (Abschnitt 3.5). Dabei soll in vertikaler und horizontaler Richtung immer die gleiche Anzahl von Leitungen verwendet werden.

Neben der Anzahl der Leitungen in einem Kanal ist bedeutsam, an welche davon ein individueller Logik- oder Ein-/Ausgabeblock-Pin angeschlossen werden kann. In unser Zielarchitektur gehen wir davon aus, dass ein Pin an jede der Leitungen innerhalb "seines" Kanals angeschlossen werden kann. Diese potentiellen Verbindungen sind in Abbildung 2.5 durch Kreuze markiert. In der Praxis werden diese Verbindungen durch programmierbare Schalter wie Multiplexer oder Pass-Transistoren realisiert. Letzteres soll bei unser Architektur der Fall sein. Weiterhin gilt:

- Bei **Eingangspins** darf nur *eine einzelne* der potentiellen Verbindungen benutzt werden. Es ist also illegal, einen Eingangspin an mehrere Leitungen seines Kanals anzuschliessen.
- **Ausgangspins** dagegen dürfen an beliebig viele Leitungen ihres Kanals angeschlossen werden.

2.2.4 Verbindungsblöcke

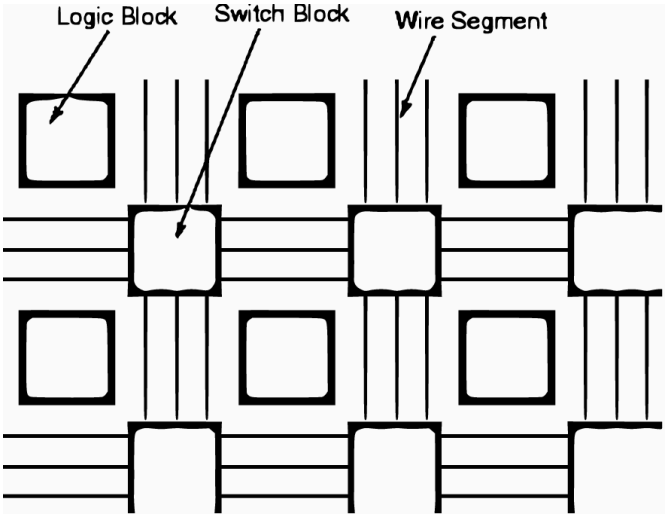

Abbildung 2.6: Leitungslängen in Kanälen

Die Leitungen in den Verdrahtungskanälen überspannen nicht das gesamte FPGA, sondern lediglich einen einzelnen Logikblock (Abbildung 2.6): An allen Kreuzungspunkten von horizontalen und vertikalen Kanälen befinden sich Verbindungsblöcke (Switch Blocks). Diese erlauben ein gezieltes Verbinden von Leitungen (Wire Segments) aus verschiedenen Kanalabschnitten.

Dabei sind allerdings nicht alle Arten von Verbindungen erlaubt (Abbildung 2.7). An jeder Kreuzung von zwei vertikalen (`T` und `B`) und zwei horizontalen (`L` und `R`) Leitungen sind programmierbare Schalter (Programmable Switches, in der Realität beispielsweise Pass-Transistoren) wie in der Abbildung gezeigt angeschlossen. Auf diese Weise kann ein eingehendes Signal gezielt auf bestimmte andere Leitungen verteilt werden (Schalter geschlossen) oder von diesen isoliert werden (Schalter offen). Bedingt durch die Art der Anordnung können aber nur solche Leitungen verbunden werden, die dieselbe Spurnummer (Track) haben. Beispiel: Ein auf Track 0 von links ankommendes Signal (`L0`) kann im Verbindungsblock an `R0`, `T0`, `B0` oder ei-

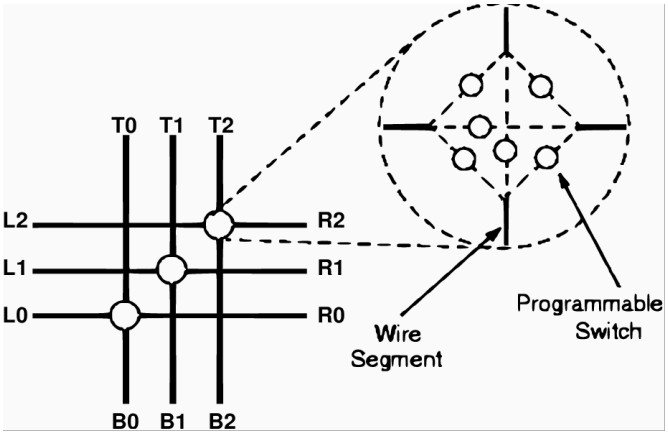

Abbildung 2.7: Innenleben eines Verbindungsblockes

ne beliebige Kombination davon angeschlossen werden. Alternativ kann es durch Öffnen aller Schalter auch an der Weiterleitung gehindert werden. Das ist z.B. nötig, wenn auf dem von rechts ankommenden Kanal R der Track 0 bereits für ein anderes Signal verwendet wurde.

Diese Art von Verbindungsblock war lange Zeit auf realen FPGAs üblich und wurde erst in den letzten Jahren durch flexiblere Topologien abgelöst. Letztere sind aber deutlich komplizierter zu modellieren und würden den Rahmen dieser Lehrveranstaltung sprengen.

2.3 Zeitverhalten

Die Rechenleistung einer Schaltung wird allgemein durch die Verzögerung von Signalen bestimmt.

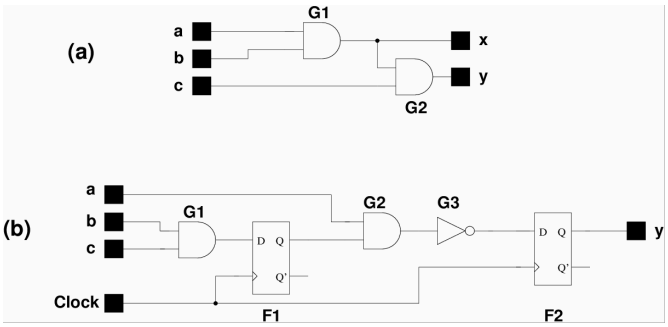

Abbildung 2.8: Verzögerung in Schaltungen

- Bei rein **kombinatorischen** Schaltungen (Abbildung 2.8.a) ohne sequentielle Elemente wie Speicher oder Flip-Flops bestimmt die langsamste Gesamtlaufzeit eines Signals von einem Eingang zu einem Ausgang die Geschwindigkeit der Berechnung. Ohne Berücksichtigung von Leitungslaufzeiten sollten in der Abbildung also die Wege von den Eingangsblöcken a und b zum Ausgangsblock y die grösste Verzögerung haben (zwei Gatterlaufzeiten).
- Bei **sequentiellen** Schaltungen (mit Flip-Flops, Abbildung 2.8.b) dagegen ist die Länge der Taktperiode entscheidend. Diese wird bestimmt durch die maximale Verzögerung zwischen einem Eingangsblock und dem Flip-Flop Eingang, dem Flip-Flop-Ausgang und einem Aus-

gangsblock, sowie zwischen einem Flip-Flop-Ausgang zu einem Flip-Flop-Eingang. In der Abbildung sollte die Taktperiode bestimmt werden durch die Summe der Verzögerungen von G2 und G3. Die beiden anderen Verzögerungen durch G1 bzw. vom Ausgang von F2 direkt zum Ausgangsblock sind beide kürzer.

Bei ihren praktischen Arbeiten sollen Sie allerdings ein etwas genaueres Verzögerungsmodell verwenden: Da wir auf der physikalischen Ebene arbeiten, können die Leitungsverzögerungen tatsächlich berücksichtigt oder doch zumindest abgeschätzt werden.

Folgende Parameter definieren auf unser Zielarchitektur die Verzögerung von Signalen:

- T_{ipad} ist die Verzögerung vom externen Chip-Pin zum Ausgangs-Pin des Eingabeblockes.
- T_{opad} ist die Verzögerung vom Eingabe-Pin des Ausgabeblockes zum externen Chip-Pin.
- T_{switch} ist die Verzögerung durch einen programmierbaren Schalter. Das heisst, dass jede Verbindung zwischen einem Logikblock oder Ein/Ausgabeblock und einer Leitung in einem Kanal diese Verzögerung in Kauf zu nehmen hat. Weiterhin fällt sie bei jedem Durchlauf eines Signals durch einen Verbindungsblock an.
- T_{comb} ist die Verzögerung vom Eingangs-Pin eines Logikblockes durch die Wertetabelle hin zum seinem Ausgangs-Pin. Sie bestimmt also die Durchlaufzeit durch einen kombinatorisch betriebenen Logikblock (ohne Flip-Flop).
- T_{FFin} ist die Verzögerungszeit vom Eingangs-Pin eines Logikblockes durch die Wertetabelle zum Eingangs-Pin des Flip-Flops. Sie wird also bei der Behandlung von sequentiell verwendeten Logikblöcken benötigt.
- T_{FFout} ist die Verzögerungszeit vom Ausgangs-Pin eines Logikblock-Flip-Flops zum Ausgangs-Pin des Logikblocks. Zusammen mit T_{FFin} wird sie bei der Behandlung von sequentiell verwendeten Logikblöcken benötigt.

Die genauen Werte sollen wie in Abschnitt 3.5 gezeigt aus einer Datei gelesen werden. Die dort gegebenen Werte sind auch untereinander konsistent, sie können für die ganze Veranstaltung als Referenz benutzt werden.

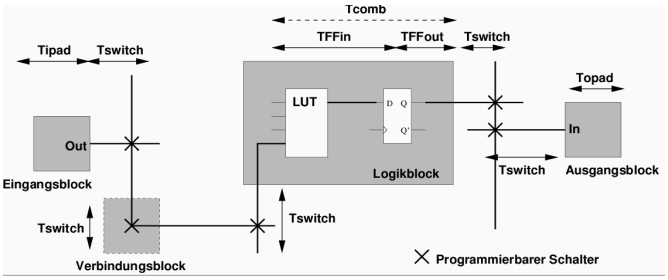

Abbildung 2.9: Signalverzögerungen

Abbildung 2.9 zeigt ein Beispiel für die Zuordnung der einzelnen Zeiten an eine Signalführung: Das Eingangssignal kommt von einem externen Chip-Pin und gelangt über einen Eingabeblock zu dessen Ausgang (T_{ipad}). Dieser wird über einen programmierbaren Schalter an eine vertikale Leitung angeschlossen (T_{switch}). Durch die geeignete Beschaltung im darunter gelegenen Verbindungsblock wird das Signal auf eine horizontale Leitung geführt (T_{switch}). Nun wird es an den

Eingangs-Pin eines Logikblockes angeschlossen (T_{switch}). Es durchläuft dort die Wertetabelle und erreicht den Eingangs-Pin des Flip-Flops (T_{FFin}). Hier endet der erste Teil des Signalpfades (Flip-Flops trennen ja kombinatorische Pfade auf). Der zweite Teil beginnt mit dem Weg vom Ausgang des Flip-Flops zum Ausgangs-Pin des Logikblockes (T_{FFout}). Dieser wird wieder an eine vertikale Leitung angeschlossen (T_{switch}). Von dort wird das Signal aber unmittelbar in den Eingangs-Pin eines Ausgangsblocks abgegriffen (T_{switch}) und erreicht dann den externen Chip-Pin (T_{opad}).

Im kombinatorischen Fall (kein Flip-Flop im Logikblock) würde das Signal bei Erreichen des Eingangs-Pins des Logikblockes nach T_{comb} am Ausgangs-Pin des Logikblockes ankommen (gestrichelt dargestellt).

3 Dateiformate

Die folgenden Abschnitte beschreiben die Dateiformate, die die von ihnen im Laufe der Veranstaltung zu entwickelnden Werkzeuge bearbeiten müssen. Diese Formate sind historisch gewachsen und haben diverse Eigenheiten, die aber in den folgenden Abschnitten erklärt werden.

3.1 Netzlisten

Diese Dateien, üblicherweise mit der Namenserweiterung `.net`, beschreiben die Struktur einer Schaltung, enthalten aber keine Angaben über Platzierung von Elementen oder ihre konkrete Verdrahtung.

Als atomare Elemente werden Ein-/Ausgabeblöcke und Logikblöcke unterstützt. Verbindungen zwischen den Elementen werden durch benannte Netze realisiert, die mit den Pins der atomaren Elemente assoziiert sind. Alle Pins, die mit dem gleichen Netznamen assoziiert sind, gelten als elektrisch verbunden. Die Assoziation erfolgt dabei über die Position des Netznamens in der Pin-Liste jedes atomaren Elements (ähnlich der Position der Parameter bei einem Funktionsaufruf).

Die `.net` Textdateien enthalten jeweils 2-3 Zeilen lange Einträge. Zwischen den Einträgen dürfen Leerzeilen stehen, innerhalb eines Eintrags aber nicht. Vorkommende Namen dürfen dabei aus allen darstellbaren Zeichen mit Ausnahme des Leerzeichens und der Klammern `()` bestehen. `[7335]` und `_i__332` sind also gültige Namen. Ein `#` leitet einen Kommentar bis zum Ende der Zeile ein.

3.1.1 Eingabeblöcke

`.input` *blockname*
`pinlist`: *netname*

Diese Direktive beschreibt einen neuen Eingabe-Block *blockname*. Sein einziger Ausgangspin ist an das Netz *netname* angeschlossen.

Beispiel:

```
.input datain
  pinlist: datanet
```

3.1.2 Ausgabeblöcke

`.output` *blockname*
`pinlist`: *netname*

Diese Direktive beschreibt einen neuen Ausgabe-Block *blockname*. Sein einziger Eingangspin ist an das Netz *netname* angeschlossen.

Beispiel:

```
.input dataout
```

```
pinlist: [5589]
```

3.1.3 Logikblöcke

.clb *blockname*
pinlist: *in1net in2net in3net in4net outnet clocknet*
subblock: *blockname in1pin in2pin in3pin in4pin outpin clockpin*

Die Direktive beschreibt einen Logikblock und seine Verbindungen. Da das **.net**-Format mehr Flexibilität bietet, als in unserer Veranstaltung tatsächlich gebraucht wird, ist dieser Eintrag etwas unhandlich.

blockname ist der Name des Blockes, der sowohl in der ersten als auch in der dritten Zeile eingetragen werden muss. In der **pinlist:** Zeile sind die Namen der Netze anzugeben, die an die Pins angeschlossen sind. Dabei ist die Reihenfolge zu beachten. Pins, die an kein Netz angeschlossen sind (=offen bleiben), werden mit dem Schlüsselwort **open** versehen. In der **subblock:** Zeile wird für alle angeschlossenen Pins ihre Nummer (durchzählen, beginnend bei Null) in der **pinlist:** Zeile angegeben. Auch hier werden nicht angeschlossene Pins als **open** markiert.

Die Beschaltung des Clock-Pins bestimmt darüber, ob ein Logikblock kombinatorisch oder sequentiell (mit Flip-Flop zwischen Wertetabelle und Ausgangs-Pin) verwendet wird: Ist hier ein Taktnetz angeschlossen, wird das Flip-Flop verwendet, ist der Clock-Pin in beiden Zeilen als **open** deklariert, wird der Block kombinatorisch betrieben.

Beispiel:

```
.clb _i_b_99425
     pinlist: n3552 data open open qqh clk
     subblock: _i_b_99425 0 1 open open 4 5
```

Hier wird ein Logikblock namens **_i_b_99425** beschrieben. Die 1. und 2. Eingangs-Pins (mit den Nummern 0 und 1) des Blocks sind mit den Netzen **n3552** und **data** verbunden. Die 3. und 4. Eingangs-Pins werden nicht benutzt. Der Ausgangspin des Blocks wird mit dem Netz **qqh** verbunden. Da der Logikblock ein Netz an seinem Takteingang hat (das Netz **clk**), wird er sequentiell betrieben.

Beachten Sie weiterhin, dass es durchaus Logikblöcke *ohne* beschaltete Eingangs-Pins geben kann. Diese agieren als sogenannte Konstantengeneratoren und liefern, abhängig vom Inhalt der Wertetabellen, den festen Wert 0 oder 1 an ihren Ausgang.

3.1.4 Taktnetz

.global *clkname*

Durch diese Direktive wird ein Netz als Taktnetz markiert. Konkret bedeutet dies, dass das so markierte Netz nicht explizit platziert oder verdrahtet werden soll, sondern auf speziellen Ressourcen auf dem Chip geführt wird. Hinweis: Der Name des Taktnetzes ist (im Gegensatz zu **open**) kein reserviertes Wort. Ein Taktnetz kann also beispielsweise auch `foobar` heissen. Eine Abhängigkeit von Netz- und Blocknamen gibt es nicht. Die Tatsache, dass ein Netz ein Taktnetz

(es kann theoretisch auch mehrere geben) ist, hängt nur davon ab, dass an den Clock-Port von CLBs angeschlossen ist.

3.1.5 Beispiel für Netzlistendatei

```
# Netlist: BLIF/s27.net
.global clock

.input s27_in_2_
pinlist: s27_in_2_

.input s27_in_1_
pinlist: s27_in_1_

.input s27_in_3_
pinlist: s27_in_3_

.input s27_in_0_
pinlist: s27_in_0_

.input clock
pinlist: clock

.output out:s27_out
pinlist: s27_out

.clb s27_out  # Only LUT used.
pinlist: s27_in_3_ n_n41 n_n42 [13] s27_out open
subblock: s27_out 0 1 2 3 4 open

.clb n_n40  # Both LUT and FF used.
pinlist: s27_in_1_ s27_in_3_ [13] [11] n_n40 clock
subblock: n_n40 0 1 2 3 4 5

.clb n_n41  # Both LUT and FF used.
pinlist: s27_in_3_ [13] open open n_n41 clock
subblock: n_n41 0 1 open open 4 5

.clb n_n42  # Both LUT and FF used.
pinlist: s27_in_3_ n_n42 [13] open n_n42 clock
subblock: n_n42 0 1 2 open 4 5

.clb [13]  # Only LUT used.
pinlist: s27_in_2_ s27_in_0_ n_n40 n_n41 [13] open
subblock: [13] 0 1 2 3 4 open

.clb [11]  # Only LUT used.
pinlist: s27_in_2_ n_n40 n_n41 open [11] open
subblock: [11] 0 1 2 open 4 open
```

3.2 Platzierung

Die Platzierung von Schaltungen wird durch Textdateien mit der Erweiterung **`.p`** beschrieben.

Sie bestehen aus einem drei Zeilen umfassenden Kopf mit folgendem Aufbau:

`Netlist file:` *netfilename* **`Architecture file:`** *archfilename*
`Array size:` *x-size* **`x`** *y-size* **`logic blocks`**
(die dritte Zeile ist leer)

netfilename ist der Name der Netzlistendatei (Abschnitt 3.1), deren Platzierung im Folgenden beschrieben wird. *archfilename* ist der Name der Architekturparameterdatei (Abschnitt 3.5) die für die Platzierung verwendet wurde. In der nächsten Zeile werden die Abmessungen des für diese Platzierung verwendeten Ziel-FPGAs in Logikblöcken (zunächst die Breite, dann die Höhe) angegeben.

Beispiel:

```
Netlist file: BLIF/s27.net   Architecture file: prak10.arch
Array size: 3 x 3 logic blocks
```

Daran schliessen sich die eigentlichen Platzierungsdaten an: Je atomarem Element der Netzliste (Ein-/Ausgabeblock, Logikblock) wird eine Zeile geschrieben. Diese hat den folgenden Aufbau:

blockname x-coord y-coord subblk **`#`** *blocknum*

blockname ist der Name des Blocks, *x-coord* und *y-coord* seine Koordinaten (siehe unten), *subblk* die Unterblocknummer (wichtig für Ein-/Ausgabeblöcke) und *blocknum* eine eindeutige interne Blocknummer (kann hilfreich beim Debugging sein). Optional können noch, durch **`#`** eingeleitet, Kommentare bis zum Zeilenende geschrieben werden.

Die Koordinatenangaben eines Blocks können natürlich nur innerhalb eines Koordinatensystems interpretiert werden. Abbildung 3.1 zeigt dieses System am Beispiel eines 2x2 grossen FPGAs. Die XY-Koordinaten des linken unteren Logikblockes (CLB) sind immer (1,1). Neben dem linken Rand der CLB-Matrix liegen die linken Ein-/Ausgabeblöcke. Sie haben die X-Koordinate 0, der unterste Block hat die Y-Koordinate 1. Die Koordinaten (0,0) sind frei, hier liegt kein Block. Analoges gilt für die anderen Ränder und Ecken des FPGAs.

Wie bereits in Abbildung 2.1 gezeigt kommen auf eine Logikblocklänge zwei Ein-/Ausgabeblöcke. Diese werden in Abbildung 3.1 zu einem Block zusammengefasst, der allerdings die beiden Sub-Blöcke 0 und 1 enthält. Die Platzierung eines Ein-/Ausgabeblocks besteht also immer aus *x-coord y-coord* und *subblk*. Dabei muss die Nummerierung der Sub-Ein-/Ausgabeblöcke immer bei 0 beginnen. Wenn also nur ein einzelner Ein-/Ausgabeblock auf einer Koordinate liegt, *muss* dieser die Sub-Blocknummer 0 haben! Bei Logikblöcken muss als *subblk* immer der Wert 0 eingetragen werden.

3.2.1 Beispiel für eine Platzierungsdatei

```
Netlist file: s27.net   Architecture file: ../prak10.arch
Array size: 3 x 3 logic blocks
```

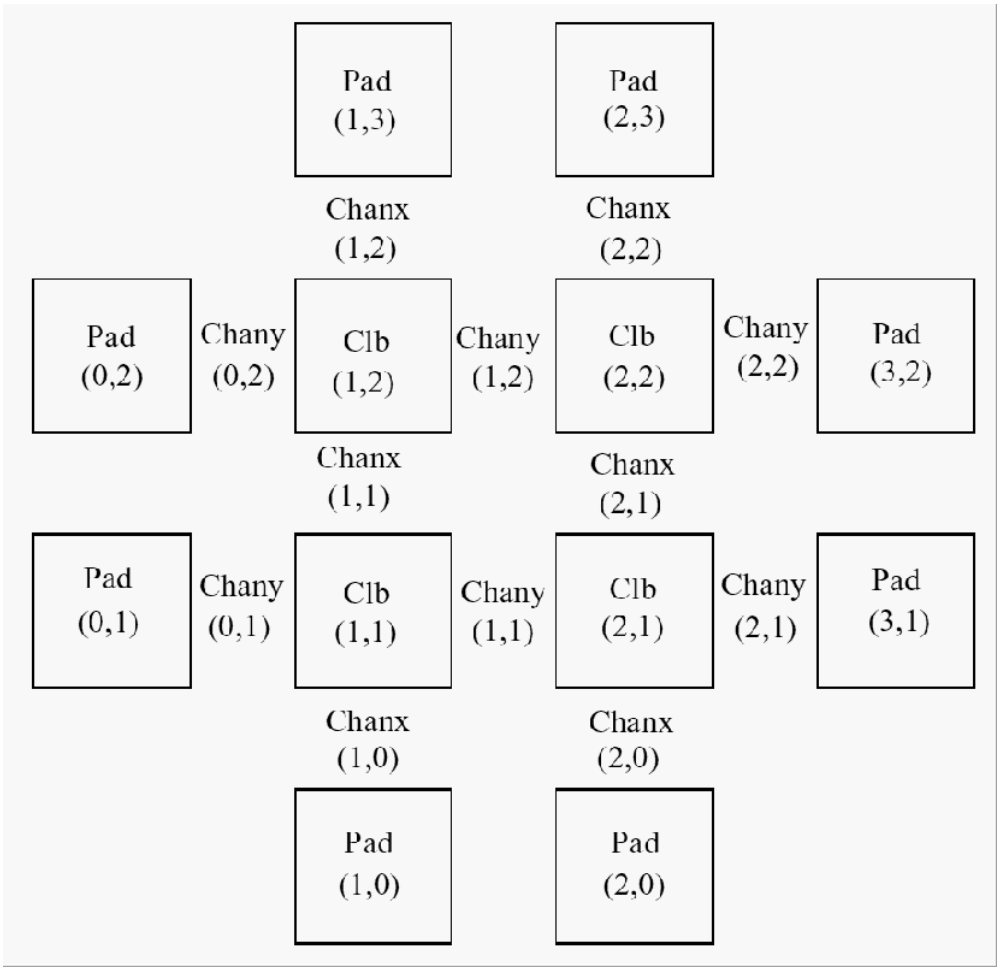

Abbildung 3.1: Koordinatensystem

```
#block name x   y   subblk  block number
#---------- --  --  ------  ------------
s27_in_2_   2   0   1       #0
s27_in_1_   4   2   0       #1
s27_in_3_   3   0   0       #2
s27_in_0_   2   0   0       #3
clock       0   2   0       #4
out:s27_out 1   0   0       #5
s27_out     1   1   0       #6
n_n40       3   2   0       #7
n_n41       3   1   0       #8
n_n42       1   2   0       #9
[13]        2   1   0       #10
[11]        2   2   0       #11
```

Abbildung 3.2 zeigt eine graphische Darstellung dieser Platzierung. Hier sieht man auch die Trennung der Ein-/Ausgabeblöcke in die beiden Sub-Blöcke.

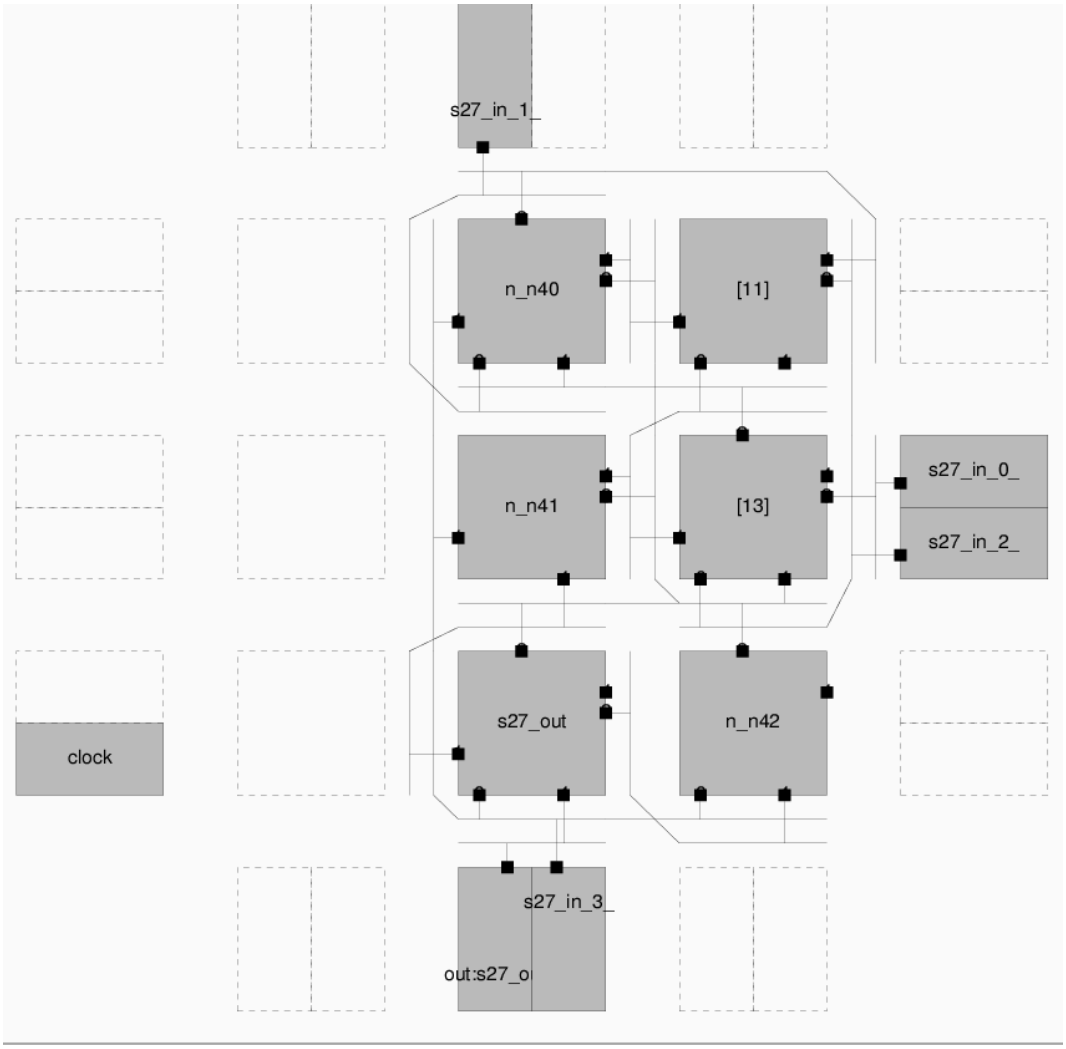

Abbildung 3.2: Platzierte und verdrahtete Beispielschaltung

3.3 Verdrahtung

Die Verdrahtung einer bereits platzierten Netzliste wird durch eine weitere Textdatei mit der Erweiterung `.r` beschrieben. Wie üblich wird auch hier durch `#` ein Kommentar bis zum Zeilenende eingeleitet.

Die Datei hat, ähnlich der Platzierungsdatei, einen 4-zeiligen Kopf:

`Array size:` *x-size* **`x`** *y-size* **`logic blocks.`**
(Leerzeile)
`Routing:`
(Leerzeile)

Nun folgt für jedes Netz ein Datenblock. Dieser beginnt mit den Zeilen

`Net` *netnumber* **`(`***netname* **`)`**
(Leerzeile)

Dabei ist *netnumber* eine eindeutige Netzkennnummer, die hier im wesentlichen für Debugging-Zwecke ausgegeben wird. Hier kann z.B. eine fortlaufende Numerierung verwendet werden. Praktisch wichtiger ist der dann in Klammern folgende Netzname (aus der Netzliste). Nun folgt eine weitere Leerzeile.

Für jedes verdrahtete Netz werden jetzt alle dafür verwendeten Verdrahtungsressourcen angegeben (eine je Zeile). Ein Netz hat immer eine angeschlossene Quelle (Ausgang) und ein oder mehrere Senken (Eingänge).

Die Quelle eines Netzes ist der Ausgang eines Logikblocks oder Eingabeblocks. Ein Logikblock als Quelle wird durch eine Zeile

SOURCE (*x-coord*,*y-coord*) **Class: 1**

beschrieben. Ein Eingabeblock verwendet die ähnliche Zeile

SOURCE (*x-coord*,*y-coord*) **Pad:** *subblk*

Beide Elemente werden also durch ihre Koordinaten im System von Abbildung 3.1 identifiziert. Für den Eingabeblock muss in Form von *subblk* auch noch die Sub-Blocknummer (0 oder 1) angegeben werden.

Der nächste Schritt eines Netzes läuft dann immer über einen Ausgabe-Pin am Quellelement. Für Logikblöcke wird dafür die Zeile

OPIN (*x-coord*,*y-coord*) **Pin: 4**

ausgegeben (Pin 4 ist immer der Ausgabe-Pin). Bei Eingabeblöcken lautet die Zeile

OPIN (*x-coord*,*y-coord*) **Pad:** *subblk*

Hier muss also wieder die Sub-Blocknummer des Eingabeblocks angegeben werden. In beiden Fällen müssen die Koordinaten in der **SOURCE** und der **OPIN** Zeile gleich sein.

Danach folgen in der Regel ein oder mehrere Angaben über die verwendeten Verdrahtungskanäle und die Spurnummer der Leitung darin. Diese haben die Form

CHANX (*x-coord*,*y-coord*) **Track:** *tracknum*

für horizontale Kanäle und

CHANY (*x-coord*,*y-coord*) **Track:** *tracknum*

für vertikale Kanäle. In beiden Fällen geben die Koordinaten eine Position gemäß dem System in Abbildung 3.1 an. Dabei ist zu erkennen, dass bei gleichen Koordinaten die zu einem Logikblock gehörenden Kanäle rechts und oberhalb davon angeordnet sind (Abbildung 3.3).

An den Rändern und Ecken der Matrix ergeben sich natürlich Ausnahmen. So hat der Ein-/Ausgabeblock (2,3) in Abbildung 3.1 keine zugeordneten Verdrahtungskanäle (beide würden ausserhalb des FPGAs liegen). Innerhalb der Kanäle haben die dem zugeordneten Block am nächsten liegenden Leitungen die Track-Nummer 0 und werden von dort an nach aufsteigender Entfernung durchnumeriert.

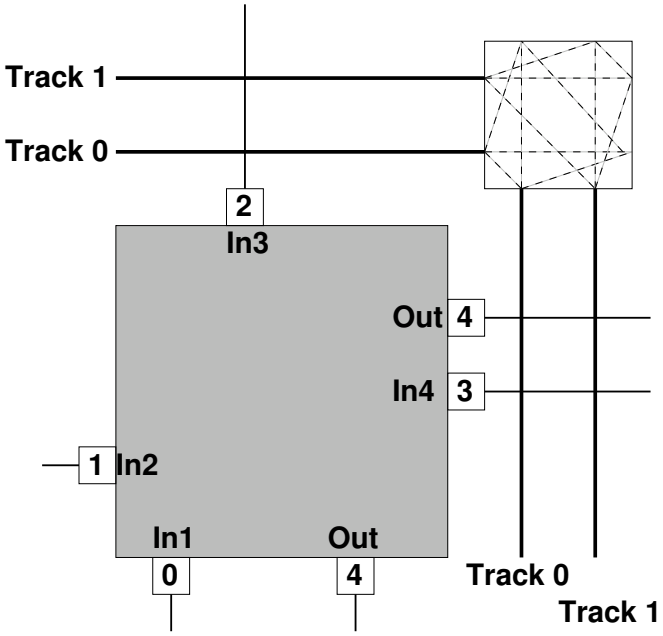

Abbildung 3.3: Block und dazugehörige Verdrahtungskanäle

Durch die **CHANX** und **CHANY** Zeilen wird das Netz nun kanalweise über das FPGA geführt. Angaben über die Verbindungsblöcke werden dabei nicht explizit gemacht, können aber bei Bedarf aus den vorhandenen Informationen gewonnen werden.

Eine Verbindung endet schliesslich an einer Senke: Dem Eingabe-Pin eines Logik- oder Ausgabeblockes. Hier steht dann analog zu **OPIN**

IPIN (*x-coord*,*y-coord*) **Pin:** *pinnr*

bei einem Logikblock und

IPIN (*x-coord*,*y-coord*) **Pad:** *subblk*

bei einem Ausgangsblock. Bei letzterem muss wieder die Sub-Blocknummer (0 oder 1) angegeben werden. Im Fall des Logikblocks muss auch noch die Nummer des Eingangspins 0...3 angegeben werden. Abbildung 3.3 zeigt die Lage der Pins und die Zuordnung zu den Logikblockeingängen. Das Ausgangssignal hat immer die Nummer 4, auch wenn es an zwei separaten Pins (unten und rechts) anliegt [1].

Nach der Angabe des Senken-Pins durch **IPIN** folgt nun eine Angabe über die Senke (Logikblock oder Ausgangsblock) selbst. Für einen Logikblock ist dies eine Zeile

SINK (*x-coord*,*y-coord*) **Class: 0**

für einen Ausgangsblock eine Zeile

SINK (*x-coord*,*y-coord*) **Pad:** *subblk*

subblk ist wieder die bereits bekannte Angabe über den verwendeten Sub-Block des Ausgabeblocks.

[1] Falls doch einmal der genau Ausgangs-Pin bestimmt werden soll, muß die folgende CHANX (=unterer Pin) oder CHANY-Zeile (=rechter Pin) ausgewertet werden.

Mit der **SINK**-Zeile ist eine Verbindung innerhalb des Netzes vollständig verdrahtet. Da ein Netz aber mehrere Senken haben kann, müssen auch noch eventuelle Verzweigungen beschrieben werden. Auch dies geschieht mit den bereits vorgestellten Einträgen: Bei einer Verzweigung folgt auf das **SINK** des bereits verdrahteten Netzes die Beschreibung des Abgreifpunktes der neuen Verbindung in Form einer **CHANX**, **CHANY** oder auch **OPIN** Angabe. In den beiden ersten Fällen wird das Signal an einem bestehenden Track abgegriffen, im letzten Fall wieder am ursprünglichen Ausgangspin. In allen Fällen wird vom Angreifpunkt an der Weg des Signals bis hin zu seiner **SINK** dann wieder wie üblich beschrieben. Im Netz wird so pro Senke ein neuer Verbindungsast angelegt, der wie oben beschrieben an die bestehende Verdrahtung "angeflanscht" wird. Wenn alle Verbindungen eines Netzes verdrahtet wurden, werden zwei Leerzeilen ausgegeben. Dann wird das nächste Netz beginnend mit einer **Net**-Zeile beschrieben.

Netze, die in der Netzliste mittels der Direktive **.global** als globale Netze (z.B. Taktnetze) deklariert wurden, haben eine andere **Net**-Zeile:

Net *netnumber* **(***netname***): global net connecting:**

Ihr interner Aufbau unterscheidet sich aber vollständig von normalen Netzen. Da sie zum Glück für unsere Arbeiten hier ohne Belang sind, können die entsprechenden Zeilen (alle beginnend mit **Block**) überlesen werden. Die Auswertung geht dann mit der nächsten **Net**-Zeile weiter.

3.3.1 Beispiel für eine Verdrahtungsdatei

Aus Platzgründen wird hier nur der Beginn und das Ende einer Platzierungsdatei wiedergegeben.

```
Array size: 3 x 3 logic blocks.

Routing:

Net 0 (s27_in_2_)

SOURCE (2,0)  Pad: 1
  OPIN (2,0)  Pad: 1
 CHANX (2,0)  Track: 0
 CHANY (1,1)  Track: 0    # Abgriffspunkt fuer Verzweigung
  IPIN (2,1)  Pad: 1
  SINK (2,1)  Class: 0
 CHANY (1,1)  Track: 0    # Hier Verzweigung von oben
 CHANY (1,2)  Track: 0
  IPIN (2,2)  Pad: 1
  SINK (2,2)  Class: 0
.
.
.
Net 10 (n_n40)

SOURCE (3,2)  Class: 1
  OPIN (3,2)  Pin: 4
 CHANX (3,1)  Track: 1
 CHANX (2,1)  Track: 1    # Abgriffspunkt fuer Verzweigung
```

```
  IPIN (2,1)  Pin: 2
  SINK (2,1)  Class: 0
 CHANX (2,1)  Track: 1    # Hier Verzweigung von oben
  IPIN (2,2)  Pin: 0
  SINK (2,2)  Class: 0
```

Diese Datei passt zu den bisher gezeigten Netzlisten- und Platzierungsdateien. Die Verdrahtung ist in Abbildung 3.2 abgebildet. Hier lassen sich auch mit Hilfe der Blocknamen die Netze zum besseren Verständnis manuell nachverfolgen.

3.4 Timing-Analysen

Für die Ergebnisse der Timing-Analyse (Bestandteil der ersten Aufgabe) gibt es kein vorgeschriebenes Dateiformat. Das folgende Beispiel für einen Pfad ist lediglich ein Vorschlag. Die Verzögerungsangaben sind hier in Picosekunden gemacht.

From			To			Item	Total
Type	Name	Location	Type	Name	Location	Delay	Delay
INPAD	pad1	(0,1).0	OPIN	pad1	(0,1).0	500	500
OPIN	pad1	(0,1).0	INPIN	data1	(1,1).1	1000	1500
INPIN	data1	(1,1).1	OUTPIN	data1	(1,1).4	900	2400
OUTPIN	data1	(1,1).4	IPIN	pad2	(1,0).0	1000	3400
IPIN	pad2	(1,0).0	OUTPAD	pad2	(1,0).0	300	3700

Location ist dabei die Koordinatenangabe mit der Pin-Nummer für Logikblöcke und mit der Sub-Blocknummer für Ein-/Ausgabeblöcke.

3.5 Architekturparameter

Die konkrete Ausprägung der Zielarchitektur wird durch die Werte einiger Parameter bestimmt. Im einzelnen handelt es sich dabei um:

X Ausdehnung der Logikblockmatrix in X-Richtung.

Y Ausdehnung der Logikblockmatrix in Y-Richtung.

W Anzahl von Leitungen in einem Verdrahtungskanal.

Tipad Verzögerung durch einen Eingangsblock.

Topad Verzögerung durch einen Ausgangsblock.

Tswitch Verzögerung durch einen programmierbaren Schalter.

Tcomb Verzögerung durch einen kombinatorischen Logikblock.

TFFin Verzögerung bis zum Flip-Flop-Eingang in sequentiellem Logikblock.

TFFout Verzögerung vom Flip-Flop-Ausgang zum Logikblockausgang.

Zur Vereinfachung der Benutzung Ihrer Programme sollen Vorgabewerte für diese Parameter aus einer Datei gelesen werden. Dabei handelt es sich um eine einfache Textdatei mit der Erweiterung **.arch**, die einen Wert pro Zeile in der o.g. Reihenfolge enthält. Alle Verzögerungszeiten

sind dabei in Picosekunden angeben. Wie üblich soll durch `#` ein Kommentar bis zum Zeilenende möglich sein.

3.5.1 Beispiel für Architekturdatei

Die hier gegebenen Werte können auch für die Tests Ihrer Entwurfswerkzeuge verwendet werden.

```
# prak10.arch
# 8x8 Matrix
8
8
# 6 Leitungen je Kanal
6
# Tipad 0.5ns
500
# Topad 0.3ns
300
# Tswitch 0.5ns
500
# Tcomb 0.9ns
900
# TFFin 0.8ns
800
# TFFout 0.5ns
500
```

Die Vorgabewerte aus dieser Datei sollen durch entsprechende Kommandozeilenparameter bei Aufruf der von Ihnen zu entwickelnden Programme individuell überschreibbar sein. Hier zu sollen Argumente der Form `-Tswitch 600 -W 10 -X 4 -Y 4` etc. dienen.

4 Hinweise zum Thema Plagiarismus

Im Rahmen dieser Veranstaltung wird eine vorher festgelegte Arbeitsgruppe bewertet. Fremde Code-Bibliotheken dürfen Sie nur für die Realisierung nebensächlicher Funktionen verwenden (z.B. log4j für Logging, ANTLR für Lexer/Parser, etc.), wobei Sie alle diese externen Quellen korrekt zitieren müssen. Bitte fragen Sie in Zweifelsfällen bei Ihrem Betreuer nach! Zusammenarbeit über Gruppengrenzen hinweg ist in Form der wechselseitigen Vorträge und durch Diskussion von Lösungs*ideen* erlaubt. Es dürfen aber keine Artefakte wie Programm-Code, Dokumentationsteile (Text, Zeichnungen) oder ähnliches ausgetauscht werden.

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Mit der Abgabe einer Lösung (Hausaufgabe, Programmierprojekt, etc.) bestätigen Sie, dass Ihre Gruppe die alleinigen Autoren des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitiert haben. Weiterführende Informationen zu diesem Thema finden Sie unter `http://www.informatik.tu-darmstadt.de/Plagiarism`.

Viel Erfolg!