Tabelle 5.1: Entwicklung einer Integrierten Schaltung

Arbeitsschritt	Meilenstein
Idee	
Entwicklung eines Basis-Konzeptes	
Vorläufige Spezifikation	Vorläufige Spezifikation: Natürlichsprachliche Beschreibung, Erste Untersuchungen
Entwurf eines ersten VHDL-Verhaltensmodells *	
Simulation, Überprüfung des Modells auf Erfüllung der Anforderungen *	Verhaltensmodell bildet Spezifikations-grundlage
Partitionieren in später synthesefähige Teile * (Aufteilung, Detailuntersuchungen, Schnittstellendefinition der einzelnen Teile)	
Umwandlung dieser Teile in synthetisierbares VHDL (auf Register-Ebene) *	
Validation der VHDL-Beschreibung * (durch Simulation)	Synthetisierbares Verhaltensmodell, Endgültige Spezifikation
Einbau von Timing-Constraints *	
Umsetzung auf Gatter-Ebene mit ASIC-Synthesizer *	
Simulation auf Gatter-Ebene *	Unter Einbeziehung von Gatterlaufzeiten funktionsfähige Schaltung
Optimierung des Designs durch Einsatz zusätzlicher Hardware-Entwurfswerkzeuge *	
Schaffung von Testmöglichkeiten	
Simulation des testbaren Designs	
Layout	
Überprüfung des Layouts durch Simulation	Produktionsreifer Chip

*: Bestandteil der Diplomarbeit

Die Reihenfolge der einzelnen Schritte ist nicht immer bindend. Innerhalb des Entwurfes treten drei wesentliche Meilensteine auf, die jeweils den Abschluß einer Design-Phase darstellen. Diese Meilensteine sind der Abschluß der funktionalen Spezifikation, die komplette Umwandlung in von dem ASIC-Synthesizer umsetzbaren Code und die Erstellung eines unter Beachtung von Gatterlaufzeiten lauffähigen Modells. Häufig machen Ergebnisse eines späteren Schrittes ein Überdenken der vorangehenden Schritte einer Designphase notwendig. Unvorhersehbare Bottom-Up-Einflüsse erzwingen oftmals auch die Rückkehr zu einer früheren Design-Phase.

5.1.1 Entwicklung eines Verhaltensmodells

Ausgehend von einer meist vorläufigen Spezifikation einer (digitalen) Schaltung wird ein VHDL-Verhaltensmodell entwickelt, das von einem Simulator interpretiert und somit in seinem Verhalten überprüft werden kann. Die Spezifikation kann aus natürlichsprachlicher Beschreibung, Blockschaltbildern, Detailuntersuchungen, Flußdiagrammen usw. bestehen. Mithilfe des Verhaltensmodells kann wiederum geprüft werden, ob die Spezifikation der Aufgabenstellung gerecht wird.

Abhängig von der Art der Problemstellung und davon, inwieweit die Spezifikation bereits bindend ist, liegt es im Ermessensspielraum des Entwicklers, in welcher Art die erste Version des Verhaltensmodells erstellt wird.

Ist die Spezifikation in wesentlichen Teilen sehr ungenau, kann es ratsam sein, das Verhaltensmodell oder Teile davon zuerst in programmähnlicher Form zu entwickeln. Anhand dieses Verhaltensmodelles kann die Spezifikation ergänzt werden. Der sequentielle Ablauf ist am einfachsten zu beherrschen und schnell zu realisieren. Auch dann, wenn unterschiedliche Ansätze zur Realisierung denkbar sind, ist dies der schnellste Weg, sie funktional auszutesten.
Nach Beendigung des funktionalen Entwurfes muß das Verhaltensmodell in später synthesefähige Teile partitioniert werden und das so entstandene Verhaltensmodell erneut getestet werden.

Sind die zu implementierenden Algorithmen oder Befehlssätze usw. durch die Spezifikation fest vorgegeben, kann das Verhaltensmodell bereits im Hinblick auf eine spätere Synthetisierbarkeit entwickelt werden. Nachteilig ist, daß Spezifikationsänderungen einen größeren Teil der Entwurfsarbeit zunichte machen können, wenn bereits Aspekte der Synthese, wie Timingverhalten oder sparsamer Einsatz aufwendiger Einheiten durch optimierte Mehrfachbenutzung, berücksichtigt wurden.
Ergibt sich zwanglos eine Partitionierung des Problems in überschaubare und einzeln überprüfbare Komponenten, ist es sinnvoll, diese bereits in der Form nebenläufig arbeitender Prozesse einzeln zu entwerfen. Die Schnittstellen zwischen den Komponenten bestimmen wesentlich, wie hardwarenah entworfen werden muß. Sind die Aufgabengebiete der Komponenten gut zu entkoppeln und die Kommunikationsprotokolle der Komponenten untereinander unabhängig von ihrem internen Aufbau, können partielle Spezifikationsänderungen lokal in den betroffenen Komponenten realisiert werden, ohne das gesamte System verändern zu müssen.

Eine hardwarenahe Implementation des Verhaltensmodells empfiehlt sich, wenn bereits bekannt ist, daß die Schaltung besonders zeitkritische Teile oder aufwendige Bauteile wie große Speicher enthalten wird. In diesem Fall hängen viele Aspekte der Realisierung vom Zeitverhalten einzelner Komponenten ab, da für diese Einheiten bestimmte Standardzellen oder externe Einheiten vorgesehen werden müssen. Das Zusammenspiel der Bauteile kann erst anhand der Unterlagen des ASIC-Herstellers zu diesen Einheiten beurteilt werden. Damit können spätere sogenannte Bottom-Up-Einflüsse und damit unter Umständen Iterationen zwischen den Entwurfsebenen vermieden werden.

5.1.2 Vorgehen bei der Entwicklung des Histogrammers

Am Beginn der Arbeit stand eine Beschreibung der möglichen Einsatzgebiete des Histogrammprozessors [23]. In Diskussionen wurde der Funktionsumfang auf sinnvoll in Hardware realisierbare Funktionen beschränkt. Alle Divisionen, Fließkommaoperationen und das Ziehen von Wurzeln, die zur Berechnung von Statistikgrößen benötigt werden (s. Kapitel 2.6), wurden vom Histogrammer auf den Hostrechner verlagert, da der Hardwareaufwand für diese Funktionen groß, aber aufgrund der seltenen Verwendung dieser Operationen die erreichbare Beschleunigung gegenüber einer Realisierung per Software gering gewesen wäre. Des weiteren wurde die Datenbusbreite des Histogrammers verringert (s. Kapitel 4.5.1), um die Anzahl der Anschlüsse zu reduzieren. Anhand von Unterlagen zu den in einem ASIC verwendbaren RAM-Blöcken wurde abgeschätzt, welche Pixeltaktraten mit dem geplanten Konzept realisierbar sein würden (s. Kapitel 3.2.1). Die minimal geforderte Taktrate von 20 MHz schien damit realisierbar zu sein.

Ein vorläufiges Konzept und Pflichtenheft des Histogrammers [24] wurde dann vom MAZ erstellt. Neben einer mathematischen bzw. textuellen Beschreibung des Funktionsumfanges enthält es bereits Entwürfe einiger Strukturen auf Registertransfer-Ebene. Timing-Anforderungen sind dort nicht mit aufgenommen. Anhand dieses Pflichtenheftes sollte ein Verhaltensmodell des Histogrammers entworfen werden.

Zu Beginn der Implementation des Verhaltensmodells war bereits vom MAZ festgelegt worden, daß der Histogrammspeicher als Single-Port-RAM implementiert werden sollte. Die zu implementierenden Algorithmen waren ebenfalls festgelegt bzw. die Entscheidung für eine bestimmte Realisierung lag bereits vor Erstellung des Verhaltensmodells nahe. Dagegen lagen kaum bindende Aussagen über Schnittstellen vor.

Da von Beginn an keine grundlegenden Unklarheiten bezüglich der Art der Realisierung und dem Funktionsumfang bestanden, konnte ein bereits sehr hardwarenahes Verhaltensmodell erstellen werden. So war es möglich schon im Verhaltensmodell das Zusammenspiel der Einheiten auf die Anforderungen des RAMs abzustimmen.
Die Tatsache, daß sich der Histogrammer problemlos in eine Anzahl von Blöcken mit einer überschaubaren Anzahl von Wechselwirkungen untereinander zerlegen läßt (s. Abbildung 4.1: Blockschaltbild des Histogrammers), erleichterte die direkte Erstellung eines hardwarenahen Simulationsmodells.

Um bereits bei Erstellung des Verhaltensmodells auf die Anforderungen eines Standardzell-Entwurfssystems an den VHDL-Code Rücksicht nehmen zu können, wurde ein Vorversuch zur Synthese gemacht. Ausgehend von den Teilen Bildpositionszähler und -komparator wurde eine hardwarenahe Beschreibung erstellt, simuliert, synthetisierbar gemacht und das Syntheseergebnis untersucht.
Die Ergebnisse dieses Vorversuches und die Erstellung des Verhaltensmodells werden im folgenden dargestellt.
Parallel zur Erstellung des Verhaltensmodells wurden in einigen Teilen über den Konzept-Entwurf hinausgehende Vorschläge zu Erweiterungen oder Veränderungen der Spezifikation gemacht und nach Absprache realisiert.

5.2 Erstellung des Verhaltensmodells

Das Verhaltensmodell wurde auf einer Sparc-Workstation entwickelt. Zur Simulation des Verhaltensmodells wurde das an der Universität verfügbare Synopsys-System verwendet.

5.2.1 Erste Erfahrungen bei der Verhaltensmodellierung und Synthese

Als Vorversuch sollte zunächst getestet werden, wie ein VHDL-Modell beschaffen sein muß, um direkt synthetisierbar zu sein. Ein geeignetes Objekt dazu bot der Positionszähler für die aktuelle Bildposition (s. Kapitel 4.2). Dieser Zähler ist eigenständig funktionsfähig und benötigt nur wenige Signale zur Ansteuerung. Er beinhaltet Register, Inkrementer, Komparatoren und Logikgatter und ist leicht manuell auf Registertransfer-Ebene umzusetzen. Damit ist der Positionszähler eine ideale Komponente, um den Entwurf bis zur Schaltungssynthese zu testen.
Beim Versuch, den im Verhaltensmodell funktionsfähigen Bildzähler zu synthetisieren, ergab sich eine Reihe von Fehlermeldungen. Nähere Untersuchungen dieser Fehlermeldungen unter Zuhilfenahme der Handbücher zu Synopsys führten auf eine Reihe von Strukturregeln für den VHDL-Code. Nach Umstrukturierung des VHDL-Codes entsprechend den gefundenen Regeln war eine Schaltungssynthese durch das Synopsys-System möglich. Diese lieferte eine lauffähige, aber zu aufwendige Struktur auf Gatterebene. Erst weitere Optimierungen des VHDL-Codes lieferten eine genügend einfache Struktur auf Gatterebene. Die daraus abgeleiteten grundlegenden Regeln wurden im folgenden auch bei der Fertigstellung des Verhaltensmodells beachtet. Diese Regeln sind im folgenden Kapitel zusammengestellt.

5.2.2 Einige Richtlinien zur Erstellung des VHDL-Codes

Neben den grundlegenden Regeln zur Erstellung synthetisierbaren VHDL-Codes sollen hier Eigenheiten der VHDL-Programmierung genannt werden, die zu unerwarteten Ergebnissen führten.

1. Definition von Flip-Flops und Latches

Getaktet arbeitende Einheiten werden in VHDL in Form eines Prozesses definiert. Ein Prozeß wird durch den Simulator (sequentiell) ausgeführt, sobald sich eines der in der *Sensitivity*-Liste angegebenen Signale ändert. Damit ein solcher Prozeß synthetisiert werden kann, darf innerhalb des Prozesses nur **eine** Taktflanke eines Signales abgefragt werden. Die Abfrage muß laut Handbuch zu Synopsys dem folgenden festen Schema gehorchen:

a) Struktur für ein D-Flip-Flop:

```
DFF_Beispiel: PROCESS(Clk,Reset)
BEGIN
  IF Reset='1' THEN                      -- Asynchroner Reset
    ...                                  -- o.ä. asynchr. Zuweisungen
  ELSIF Clk'event AND (Clk='1') THEN     -- Abfrage auf (pos.) Taktflanke
    ...                                  -- getaktete Zuweisungen
  END IF;
END PROCESS;
```

b) Struktur für ein Latch:

```
Latch_Beispiel: PROCESS(Clk,D)
BEGIN
  ...                                    -- Ggf. asynchroner Reset
  IF Clk='1' THEN
    Q<=D;                                -- Zuweisungen: alle unabhängigen
    ...                                  -- Variablen in die Sensivity-Liste
  END IF;                                -- eintragen
END PROCESS;
```

Der Teil zur Reset-Abfrage darf auch entfallen, wenn keine Initialisierung der Flip-Flops vorgenommen werden soll. Das Schema zur Abfrage der Flanke ist fest vorgeschrieben. Weitere Verknüpfungen innerhalb der Flankenabfrage sind nicht zulässig. Deshalb müssen eventuelle weitere Bedingungen, von denen die Zuweisung eines neuen Wertes an ein Flip-Flop abhängen soll, innerhalb des Zuweisungsteils abgefragt werden. Diese Bedingungen werden dann über Multiplexer innerhalb des Datenpfades des Flip-Flops synthetisiert (s. Abbildung 5.2). Alternativ dazu kann eine bedingte Taktung durch separate Erzeugung von geeigneten Taktsignalen realisiert werden.
Ein *else*-Zweig zu einer Flankenabfrage ist unzulässig, da es kein Äquivalent in Hardware gibt. Dies entspräche nämlich einer Aktion, die immer dann stattfindet, wenn gerade kein Taktereignis eintrifft. Eine solche Aktion ist sinnlos, da die Dauer eines Taktereignisses nicht definiert (bzw. im Idealfall 0) ist.
Die Signale Reset und Takt beim D-Flip-Flop bzw. Reset, Takt und Dateneingang beim Latch müssen in die Sensitivity-Liste des Prozesses eingetragen werden. Auch weitere im Reset-Fall auszuwertende Signale müssen mit in die Sensitivity-Liste eingetragen werden.
Es gibt alternativ die Möglichkeit, die Taktung über *wait*-Statements zu steuern. Diese Methode unterliegt allerdings den selben Einschränkungen und wird von Synopsys nicht empfohlen.

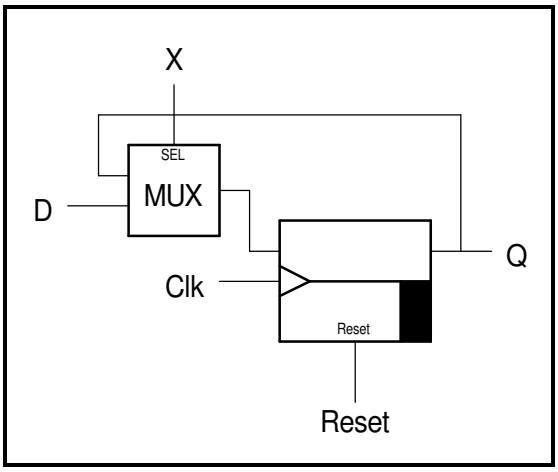

Abbildung 5.2: Bedingtes Laden eines Flip-Flops

2. Ausgabe von Variablen aus einem Prozeß

Aus jedem sinnvollen Prozeß und jeder *Entity* müssen Ergebnisse nach außen hin abgegeben werden. Sollen aus einer *Entity* Variablen eines Prozesses nach außen hin verfügbar gemacht werden, ist es naheliegend, nach der Berechnung des Variablenwertes innerhalb des Prozesses eine Zuweisung an das Ausgangs-Signal zu machen. Diese Zuweisung führte jedoch zur Synthese eines unnötigen zusätzlichen Flip-Flops für das Ausgangs-Signal. Eine Zuweisung außerhalb des Prozesses ist nicht möglich, da die Variable hier nicht zugreifbar („sichtbar“) ist.

Ein vollständiger Austausch der Variablen direkt gegen das Ausgangsignal ist nicht möglich, wenn der Wert innerhalb des Prozesses auch gelesen werden soll, da als Ausgang deklarierte Signale nicht gelesen werden können. Der für diesen Fall vorgesehene Ausgangstyp *buffer bit* ist nicht zusammen mit anderen Logiktypen wie *std_logic* anwendbar.
Diese Einschränkungen führten dazu, daß im VHDL-Code weitgehend auf die Benutzung von prozeß-lokalen Variablen verzichtet wurde, und anstatt dessen auf *Entity*-globale Signale zurückgegriffen wurde. Diese werden zur Ausgabe aus der *Entity* außerhalb der Prozesse an die zugeordneten Ausgangssignale zugewiesen.

Beispiel für Ausgabe eines Signals aus einem Prozeß:

```
ENTITY Test IS
  PORT(...
       Ausgabe: OUT std_logic;
  );
END Test;

ARCHITECTURE TestArch OF Test IS
  SIGNAL LesenUndSchreiben: std_logic;
BEGIN
  Ausgabe <= LesenUndSchreiben;
  PROCESS TestProc IS
    ...
    IF LesenUndSchreiben='1' THEN LesenUndSchreiben<='0';
    ...
  END PROCESS;
END TestArch;
```

3. Logiktypen

Beim ersten Entwurf des Positionszählers wurde mit den VHDL Standard-Datentypen Bit und Integer gearbeitet. Um dem Compiler anzuzeigen, wieviele Bit für ein Speicherregister benötigt werden, ist es bei Verwendung einer Integer-Variable notwendig, den Bereich der Variable auf den benötigten Wertebereich einzuschränken. Sonst werden Register und beteiligte Datenpfade 32 Bit breit ausgelegt, da Integer-Variablen in VHDL als Default 32 Bit breit repräsentiert werden.

Beispiel:
```
constant MaxXY: integer:=4095;          -- 12 Bit Zaehlerbreite
variable AuflX,AuflY: integer range 0 to MaxXY;
```

Ein Binärzähler läuft von seinem maximalen Zählwert auf Null über, wenn der Versuch gemacht wird, den Wert weiter zu inkrementieren. Dieses Verhalten ist oft sogar erwünscht. Wird jedoch bei einer Berechnung mit Integer-Variablen der vorgegebene Wertebereich überschritten, führt dies zu einer Fehlersituation während der Simulation, da der VHDL-Simulator bei Integer-Variablen Bereichsüberprüfungen vornimmt. Um das Verhalten digitaler Logik zu simulieren, muß bei Berechnungen mit dem Modulus gearbeitet werden.

Beispiel:
```
X<=(X+1) mod (MaxXY+1);                 -- naechste Spalte
```

Für die Modellierung einer digitalen Schaltung reicht oft eine zweiwertige Logik mit den Werten 0 und 1 aus. Sobald jedoch eine digitale Schaltung daraufhin geprüft werden soll, ob sie intern mit sinnvollen Werten arbeitet, sind weitere Logikwerte hilfreich. Zur Implementation von Bussystemen ist mehrwertige Logik unverzichtbar.

In Standard-VHDL werden die Datentypen *boolean* und *bit*, sowie *integer* usw. unterstützt. Diese Datentypen basieren auf zweiwertiger Logik.
Die Zusatz-Library „IEEE.std_logic_1164“ stellt die Datentypen *std_logic* und *std_logic_vector* mit neunwertiger Logik zur Verfügung:

Zum komfortableren Debuggen finden sich die Logikwerte:
‘0’ Logisch 0
‘1’ Logisch 1
‘U’ Unbekannt (0, 1 oder wechselnde Polarität)
‘X’ Unbekannt fest (0 oder 1)

Zur Kennzeichnung unwichtiger Signalzustände innerhalb von Logiktabellen dient:
‘-’ Belanglos (meist wie X behandelt)

Zur Implementation von Bussystemen finden sich hier zusätzlich die Logik-Werte:
‘Z’ Hochohmig
‘W’ Schwach unbekannt
‘L’ Schwach 0
‘H’ Schwach 1

Die Library enthält nicht nur eine Implementation dieser Datentypen, sondern erweitert auch die auf den Standard-Datentypen definierten Operationen zur korrekten Behandlung dieser Datentypen.
Die dazu notwendige Logiktabelle für die boole’sche Operation AND ist in Tabelle 5.2 abgebildet.

Tabelle 5.2: AND-Funktion bei std_logic

AND	0,L	1,H	X	Z,W,-	U
0,L	0	0	0	0	0
1,H	0	1	X	X	U
X	0	X	X	X	U
Z,W,-	0	X	X	X	U
U	0	U	U	U	U

Herstellerspezifische Zusatzlibraries stellen weitere nützliche Funktionen, wie z.B. Typumwandlungen oder Vergleichsoperationen, zur Verfügung. Dies wird über das bei objektorientierten Sprachen gängige Overloading für den Anwender unsichtbar gemacht.
Die *std_logic*-Logiktypen umgehen auch die Probleme, die bei der Berechnung mit Integer-Variablen auftreten, wenn der Definitionsbereich überschritten wird. Ergebnisse von Berechnungen werden binär auf die Länge der Zielvariablen abgeschnitten. Hat die Zielvariable eine größere Länge als das Ergebnis, wird der Wert, je nachdem welche Arithmetik-Library eingebunden wurde, vorzeichenbehaftet oder nicht vorzeichenbehaftet erweitert.

4. Vergleich von std_logic-Datentypen

Entgegen der Erwartung, daß die mehrwertige Logik in den Zusatzlibraries vollständig für alle Operationen implementiert ist, wurde bei Vergleichsanweisungen keine vollständige Implementation der *std_logic*-Datentypen vorgenommen.

So ergibt

```
Erg <= '1' when Vektor="X01"
       else '0';
```

als Ergebnis auch dann '0', wenn der Vektor den Wert "001" hat. Offensichtlich findet in den aktuellen VHDL-Implementationen hier ein String-Vergleich statt, der keine Rücksicht auf die Bedeutung des Logikwertes 'X' nimmt. Die Werte 'X', 'U' und '-' werden dabei gleich behandelt. So kann zwar abgefragt werden, ob ein Signal zu einem bestimmten Zeitpunkt unbekannt ist, '-' und 'X' können jedoch nicht zum Ausmaskieren von Signalen verwendet werden.
Insbesondere für die Realisierung eines Steuerwerkes in der Form eines PLAs mit einer Vielzahl von Eingängen wäre dieses Konstrukt wünschenswert. Stattdessen muß eine explizite Abfrage der je Zustand relevanten Eingangsvariablen vorgenommen werden.

5. Generische Einheiten

VHDL sieht eine Reihe von Konstrukten vor, die einen generischen Entwurf von Komponenten ermöglichen. Generisch bedeutet, daß eine Untereinheit so allgemein entworfen wird, daß sie für unterschiedliche Anwendungsfälle in unterschiedlichen Ausführungen instanziiert werden kann. Beispielsweise kann ein vordefiniertes RAM durch Übergabe der Wortbreite und der Anzahl der Speicherzellen skaliert werden. Problematisch stellt sich der generische Entwurf dar, wenn beispielsweise abhängig von einem generischen Parameter ein Konstantenvektor unterschiedlich aufgebaut werden soll.

Beispiel: Ein Vektor mit skalierbarer Breite (maximal 12 Bit) soll daraufhin getestet werden, ob alle Bits gesetzt sind.

```
entity Vergleich is
  generic(Breite: Natural:=12);
  port(Arg: in std_logic_vector (Breite-1 downto 0);
       AlleEing1: out std_logic
      );
end Vergleich;

architecture Versuch1 of Vergleich is
begin                                    -- !!! Klappt nur bei Breite=12
  AlleEing1<='1' when Arg="111111111111"
             else '0';                   -- Alle Eingangsbits 1?
end Versuch1;

architecture Versuch2 of Vergleich is
begin                                    -- !!! Syntax-Fehler
  AlleEing1<='1' when Arg="111111111111"(Breite-1 downto 0)
             else '0';                   -- Alle Eingangsbits 1?
end Versuch2;

architecture Versuch3 of Vergleich is
constant allone : std_logic_vector(11 downto 0):="111111111111";
begin                                    -- !!! Korrekte Lösung
  AlleEing1<='1' when Arg=allone(Breite-1 downto 0)
             else '0';                   -- Alle Eingangsbits 1?
end Versuch3;
```

Gezeigt sind drei naheliegende Lösungsversuche.
Lösungsversuch 1 schlägt unter Umständen bei Argumentbreiten unterhalb der maximalen Breite von 12 Bit fehl, da das Argument vor dem Vergleich auf die Breite des

Konstantenvektors erweitert wird. Je nachdem, ob eine Bibliothek mit vorzeichenbehafteter oder nicht vorzeichenbehafteter Arithmetik für *std_logic*-Typen eingebunden wird, wird *Arg* unterschiedlich erweitert. Bei nicht vorzeichenbehafteter Arithmetik tritt niemals Gleichheit auf. Bei vorzeichenbehafteter Arithmetik würde die Erweiterung in Zweierkomplement-darstellung in diesem speziellen Fall dafür sorgen, daß der Vergleich wie gewünscht funktioniert.
Lösungsversuch 2 umgeht dieses Problem, indem der Versuch unternommen wird, den Vergleichsvektor zu skalieren. Diese Lösung wird jedoch vom VHDL-Simulator als syntaktisch falsch abgelehnt.
Lösungsversuch 3 ist eine Abänderung von Versuch 2, die vom VHDL-Simulator akzeptiert wird.
Weitere Lösungsmöglichkeiten sind die Verwendung einer Potenzierung zur Berechnung des Vergleichsvektors oder die Verwendung des *others*-Konstruktes. Diese Lösungsmöglichkeiten sind jedoch nur anwendbar, wenn der Vergleichsvektor so einfach aufgebaut ist wie im Beispiel.

6. With ... select-Konstrukt

Das *with ... select*-Konstrukt ist mit einer *case*-Anweisung in anderen Programmiersprachen vergleichbar. Das *with ... select*-Konstrukt ist sehr vorteilhaft für aufwendige Abfragen, da schon beim Syntaxcheck geprüft wird, ob alle möglichen Werte der abgeprüften Variable in der Abfrage abgedeckt werden. Seine allgemeine Verwendbarkeit in VHDL wird jedoch dadurch eingeschränkt, daß es nicht möglich ist, innerhalb der Abfragen eine Konkatenation von Bit-Vektoren (die beispielsweise durch Konstanten definiert sind) zu einem Vektor der benötigten Länge vorzunehmen. Dies liegt daran, daß die Konkatenation als Funktion interpretiert wird, aber in der Abfrage nur Konstanten zulässig sind.

Zusammenfassung

Die genannten Regeln entstammen zum Teil dem Handbuch zu Synopsys und sind zum Teil eigene Erfahrungen.
Nur bei einem Teil dieser Regeln handelt es sich um Einschränkungen des VHDL-Standards, die einer Vereinheitlichung der Programmiertechnik dienen und damit den Bau von Synthese-werkzeugen vereinfachen.
Als besonders gewöhnungsbedürftig stellt sich die implizite Definition von Flip-Flops dar.
Eine ausreichende Erläuterung der Verwendung von *std_logic*-Datentypen vermißt man in den Handbüchern zu Synopsys.

Auf eine Definition eigener VHDL-Hilfskonstrukte beim Entwurf des Histogrammers wurde verzichtet, da der VHDL-Standard bereits sehr mächtig ist und beim Histogrammer keine für Chipentwürfe atypischen Probleme auftreten. Des weiteren wurden weitestmöglich VHDL-Standardfunktionen verwendet, um einen eventuellen späteren Übergang auf das am MAZ verwendete Entwicklungssystem Compass zu erleichtern.

Das Verhaltensmodell wurde zur besseren Übersichtlichkeit in eine Vielzahl in sich logisch zusammenhängender Teile aufgespalten und für jeden Teil ein File angelegt. Die Verknüpfung aller Files wird durch ein Top-Level-File vorgenommen, das nur die Aufgabe hat, die übergeordneten Komponenten innerhalb der einzelnen Files zu instantiieren und sie untereinander und mit den Anschlüssen des Histogrammers zu verbinden.
Das Verhaltensmodell ist in Anhang D abgedruckt.

5.3 Simulation des Verhaltensmodells

Zum Vortest des Verhaltensmodells für den gesamten Histogrammer wurden für seine aufwendigsten Teile Einzeltests durchgeführt. Danach wurde im Verlauf der Entwicklung des Verhaltensmodells eine Testumgebung erstellt, die alle Hauptaufgaben und -funktionen des Histogrammers innerhalb von 450 Takten anhand eines kleinen Beispielbildes aufruft (s. Anhang D, Anhang E).
Nach Fertigstellung des Verhaltensmodells wurde vom MAZ eine unabhängige Überprüfung sämtlicher Funktionskonfigurationen und -kombinationen und der Funktion bei Bearbeitung realistischer Bildgrößen vorgenommen, um die Übereinstimmung mit der spezifizierten Funktion sicherzustellen. Mithilfe der vom MAZ erstellten Testumgebung wird eine einfache Evaluation aller Fähigkeiten des Histogrammers ermöglicht.

Nach Erstellung des Verhaltensmodells wurde die Vereinbarung getroffen, den Histogrammer mit dem Compass-System zu synthetisieren, da das MAZ den Chip selbst fertigen möchte und der Aufwand für eine Portierung in einem späteren Entwicklungsstadium größer gewesen wäre. Unterschiede zwischen den Zusatz-Libraries von Synopsys und dem am MAZ benutzten Compass-System machten eine nur geringfügige Überarbeitung des Verhaltensmodells notwendig. Die Unterschiede betrafen die Verwendung der für *std_logic*-Datentypen zur Verfügung stehenden Funktionen. Die endgültige Version des Verhaltensmodells nutzt nur sowohl bei Synopsys als auch bei Compass verfügbare Konstrukte, da dies keine wesentliche Einschränkung bedeutet. Eine anschließende Simulation mit dem an das Compass-System angpaßten VHDL-Simulator Voyager der Firma Ikos ergab das gleiche Verhalten wie zuvor unter Synopsys.

5.4 Erstellung einer synthesefähigen Beschreibung

Da bereits bei der Erstellung des Verhaltensmodells Wert auf die Verwendung synthesefähiger Konstrukte gelegt wurde, entfiel der sonst notwendige Schritt der kompletten Umwandlung in eine hardwarenahe Beschreibung. Die Synthese wurde mit dem am MAZ verwendeten Compass-System durchgeführt.

Das Compass-System ist voll über eine grafische Oberfläche bedienbar und unterstützt einen hierarchischen grafischen Entwurf. Um diese Eigenschaft nutzen zu können, muß ein Blockschaltbild des Chips als Grafik erstellt werden (s. Anhang C). Dieser sogenannte Top-Level dient der Verbindung aller in oberster Ebene definierten Komponenten. Diese Komponenten können entweder über VHDL-Programme definiert werden oder mit dem Speichergenerator oder anderen Hilfswerkzeugen wie dem Datenpfad-Compiler erstellt worden sein. Ein Symbol (sog. Icon) mit den in den Komponenten definierten Ein- und Ausgängen wird automatisch generiert und bei Veränderungen der *Port*-Deklarationen sofort an den aktuellen Stand angepaßt. Da am MAZ bereits zu Dokumentationszwecken ein solcher Top-Level erstellt wurde, der für jedes File des Verhaltensmodells eine Komponente enthält, wurde hierauf aufbauend die Umwandlung in ein synthesefähiges Modell vorgenommen.
Die Verbindung der Komponenten innerhalb dieser Grafik kann entweder über grafische Verbindung der Signale (*wires* und *busses*) geschehen oder über die Vergabe von Signalnamen. Signale und Anschlußpunkte gleichen Namens werden als miteinander verbunden behandelt. Für die Taktleitung und die Reset-Leitung wurde namentliche Verbindung vorgenommen. Alle anderen Einheiten wurden grafisch verbunden, um sämtliche Abhängigkeiten zwischen den Komponenten leicht erkennen zu können. Um die Anzahl der Einzel-Leitungen nicht zu groß werden zu lassen, wurden logisch zusammengehörige Leitungen zu Bussen zusammengefaßt.

Die Erstellung des grafischen Top-Levels unter Compass und vor allem die Anpassung bei Änderungen von Komponenten erwies sich trotz angenehmer Bedienung des Systems als wesentlich aufwendiger als die Definition der Verbindungen zwischen den Komponenten direkt über VHDL. Dies liegt daran, daß bei Veränderungen der *Port*-Deklarationen teilweise auch die Icons verändert und die Leitungen zu der betroffenen Komponente verschoben werden müssen. Da als Bus zusammengefaßte unterschiedliche Signale innerhalb des Busses nicht über ihren ursprünglichen Namen verfügen, erfordert die Streichung oder Hinzufügung eines Signals unter Umständen die Anpassung mehrerer Signale des Busses an allen angeschlossenen Komponenten.

Bei der Synthese und der Untersuchung der Syntheseergebnisse traten einige weitere Einschränkungen und Eigenheiten des Synthesewerkzeuges von Compass [5] zutage. Die meisten Einschränkungen gelten auch für Synthese mit Synopsys.

1. Zugriff auf Register

Bei einem Schreibzugriff auf ein als *std_logic_vector* definiertes Register darf schreibend nicht auf echte Teile des Registers zugegriffen werden. Bei jeder Zuweisung innerhalb des Prozesses müssen alle Bits zugewiesen werden.

2. Benutzung von Arrays

Es können keine zwei- oder mehrdimensionalen Arrays synthetisiert werden. Da ein Bit-Vektor wie *std_logic_vector* bereits als eindimensionales Array behandelt wird, kann also auch kein Array aus Bit-Vektoren synthetisiert werden.
Konstantenarrays werden überhaupt nicht unterstützt. Es ist also nicht möglich, über eine im VHDL-Code definierte Tabelle eine PLA-ähnliche Struktur zu definieren.
Anwenderdefinierte Enumerationstypen können nicht zur Indizierung eines Arrays verwendet werden. Stattdessen müssen Integers oder symbolische Integer-Konstanten eingesetzt werden.

Diese Einschränkungen führten zu Änderungen des Steuerwerks für den Speicherzugriff, da dieses zuvor ein Konstantenarray nutzte, das über einen anwenderdefinierten Enumerationstyp adressiert wurde.

3. Synthese von RAMs

Die Synthese von RAMs aus VHDL-Code wird nur bei RAM-Größen von einigen Bit bis zu zwei Kilobit unterstützt. Die derart aus Arrays generierten RAMs werden flächenintensiv aus einzelnen Flip-Flops und Decodern erstellt.
Zur Verwendung eines vom Chip-Hersteller definierten und auf den Produktionsprozeß optimierten skalierbaren RAM-Blockes muß über den sogenannten Cell-Compiler ein RAM der gewünschten Art und Größe generiert werden. Zwar wird auch für dieses RAM ein VHDL-Modell erzeugt, dieses kann jedoch nicht ohne weiteres als *component* in den VHDL-Code integriert werden. Zur korrekten Einbindung muß das RAM innerhalb des grafischen Top-Levels oder in einer grafischen Unterkomponente eingebunden werden. Der RAM-Generator erzeugt dazu ein Icon des RAMs mit seinen Anschlüssen (s. Anhang C).
Um den generierten Speicher wahlweise für die Simulation gegen die Speichereinheit des Verhaltensmodells austauschen zu können, wurde eine zusätzliche Ebene unterhalb des Top-Levels erstellt, die die vier Speicherblöcke und eine VHDL-Komponente enthält (s.

Anhang C). Die VHDL-Komponente beinhaltet Latches für RAM-Adressen und Daten, da das verwendete asynchrone RAM [4] keine Adresslatches enthält und trotz separater Dateneinund ausgänge während eines Schreibzugriffs die Datenausgänge hochohmig schaltet.

4. Reset von Flip-Flops

Bei D-Flip-Flops, die nur bedingt bei einem Takt neu geladen werden sollen und gleichzeitig nicht über einen asynchronen Reset verfügen, synthetisiert Compass einen zusätzlichen über die Reset-Leitung der vorhergehenden Stufen gesteuerten Multiplexer. Der Multiplexer verbindet während des Reset-Impulses den D-Eingang des Flip-Flops mit seinem Ausgang. Diese Konstruktion soll offensichtlich verhindern, daß bei Beginn des asynchronen Resets entstehende Hazards in anderen Schaltungsteilen das Flip-Flop in einen undefinierten Zustand bringen können. Bei Synthese eines solchen Multiplexers warnt das Compass-System den Benutzer. Da im Histogrammer keine Flip-Flops ihren Zustand auch während eines Resets beibehalten müssen, werden alle nur bedingt veränderten D-Flip-Flops über den Reset-Eingang des Histogrammers initialisiert.

Nach dem Einschalten der Spannungsversorgung hat ein symmetrisch aufgebautes Flip-Flop naturgemäß einen undefinierten Zustand. Erst Fertigungsungenauigkeiten sorgen dafür, daß ein Großteil der Flip-Flops bevorzugt in einen seiner Zustände kippt. Dieses Verhalten kann gut an SRAMs beobachtet werden. Für den Test eines komplexen Bausteins wie dem Histogrammer wirft solches Verhalten jedoch Probleme auf, da eine Nichtübereinstimmung zweier Bausteine dann nicht unbedingt ein Hinweis auf Fehlverhalten eines der beiden Bausteine ist. Auch diese Überlegung legt nahe, nach Möglichkeit alle Flip-Flops, deren Startzustand bei korrekten Testsequenzen direkt Ausgangssignale beeinflußt, durch den auch bei Test-Start zu gebenden Reset-Impuls in einen definierten Zustand zu bringen.

Ein Verstoß gegen die hier genannten Regeln führte nicht immer zu Fehlermeldungen, sondern teilweise auch zu Warnungen und erst später auftretenden Fehlern. Ein Verstoß gegen Regel 1 führte unter Synopsys sogar zu einem Absturz des Synthesizers.

5.5 Simulation der Netzliste

Nach Erstellung einer synthesefähigen VHDL-Beschreibung kann diese schließlich vom ASIC-Synthesizer synthetisiert werden. Dabei wird die technologie-unabhängige VHDL-Beschreibung in eine Schaltung aus Standardzellen umgewandelt. Diese Standardzellen sind vom Chip-Hersteller zur Verfügung gestellte Grundbauelemente, die jeweils eine logische Funktion realisieren und i.a. zwei bis zu wenigen zehn Transistoren beinhalten. Mit der Abbildung auf Standardzellen findet eine Festlegung auf die Zieltechnologie statt. Das Ergebnis der Synthese ist eine sogenannte Netzliste. Die Netzliste bildet die Funktionalität des Chips auf eine Verschaltung der Standardzellen ab.
Zur Prüfung des Syntheseergebnisses wurde der Histogrammer anhand dieser Netzliste erneut simuliert. Dabei sollte sichergestellt werden, daß die Schaltung auch unter Einbeziehung von Gatterlaufzeiten funktionsfähig ist. Eine Einbeziehung realistischer Laufzeiten ist erst bei der Netzlistensimulation möglich. Die Simulation des Verhaltensmodells dagegen berücksichtigt keine Laufzeiten, da weder der Aufbau der Schaltung auf Gatterebene bekannt ist, noch die Zieltechnologie festgelegt ist.

5.5.1 Benutzung des VHDL-Simulators zur Netzlisten-Simulation

Damit der Anwender eines Entwurfssystems nicht für die Simulation der Netzliste einen neuen Simulator erlernen und eine neue Testumgebung für seinen Chip erstellen muß, gibt es die Möglichkeit, den VHDL-Simulator auch zur Netzlisten-Simulation zu verwenden. Dazu kann das Compass-System Netzlisten nach VHDL umwandeln. In den so generierten VHDL-Beschreibungen wird jede Standardzelle der Netzliste auf ein vom Chip-Hersteller geliefertes VHDL-Äquivalent der Standardzelle abgebildet. Diese in VHDL programmierten Standardzell-Äquivalente simulieren auch das Zeitverhalten der Standardzellen und melden bei der Simulation Verstöße gegen das vom Hersteller vorgeschriebene Timing-Schema. Alle im VHDL-Quellcode definierten Signalnamen werden in der VHDL-Netzliste beibehalten.
Aus der Netzliste jeder Komponente innerhalb des grafischen Top-Levels wird eine separate VHDL-Beschreibung erstellt. Auch aus dem grafischen Top-Level selbst kann Compass eine VHDL-Beschreibung erzeugen, die sämtliche innerhalb der Grafik definierten Einheiten einbindet und miteinander verschaltet.
Das RAM des Histogrammers wurde über den zum Compass-System gehörigen RAM-Generator als Makrozelle erzeugt. Der RAM-Generator erzeugt direkt ohne den Umweg über eine Netzliste ein physikalisches Layout des RAMs. Zur Einbindung eines RAMs in den aus der Netzliste erzeugten VHDL-Code erstellt auch der RAM-Generator ein VHDL-Verhaltensmodell des RAMs. In diesem Verhaltensmodell wird ebenfalls die Einhaltung der vom Chip-Hersteller garantierten Timing-Parameter des RAMs bei seiner Ansteuerung während der Simulation überprüft. Das Modell des RAMs wird bei Erstellung des VHDL-Codes aus dem Top-Level automatisch mit in den erzeugten Code eingebunden.
Um einen schrittweisen Übergang von der Simulation des Orginal-Codes zur Netzlisten-Simulation zu ermöglichen, kann für jede Komponente des grafischen Top-Levels gewählt werden, ob der aus der Netzliste erzeugte VHDL-Code oder der Quellcode eingebunden werden soll. Um ein bereits realistisches Timing bei der Netzlisten-Simulation zu erhalten, werden kapazitive Lasten beachtet. Innerhalb jeder Komponente errechnet der Synthesizer die Lastkapazität für jeden Gatterausgang, indem er die Default-Werte für alle angeschlossenen Eingänge aufsummiert. Diese Lastkapazitäten werden zur Bestimmung der Schaltverzögerung der Gatter beachtet. Allerdings werden die Kapazitäten nicht aus den einzelnen Komponenten in das VHDL-Modell für den Top-Level übergeben, so daß auf oberster Ebene als Default jeweils nur ein Gattereingang für die Lastkapazität angenommen wird.

Bei Einbindung eines der Quellcode-Files müssen aus dem VHDL-Modell des Top-Levels manuell die Angaben der Lastkapazitäten für die entsprechende Komponente gelöscht werden, da Kapazitätsangaben im Quellcode nicht vorgesehen sind.
Nach Anfertigung eines Layouts mithilfe der Netzlisten können die aus dem Layout gewonnenen Daten genutzt werden, um genaue Lastkapazitäten für alle Gatter zu berechnen und eine Netzlisten-Simulation mit präziserem Timing zu machen.

Bei dem Versuch einer Simulation des gesamten Histogrammers anhand des aus der Netzliste erzeugten VHDL-Codes ergab sich, daß die Komponenten „histstat“ (Statistikteil) und „histwin“ (Fensterkomparatoren, Positionszähler) (s. Anhang D) zu groß waren, um mit dem VHDL-Simulator „Voyager“ der Firma Ikos simuliert zu werden. Diese Teile konnten daher bei der ersten Simulation nicht durch aus der Netzliste generiertes VHDL eingebunden werden. Sie mußten als VHDL-Quellcode eingebunden werden. Während synthetisierte Einheiten um die Gatterlaufzeiten verzögert auf Taktsignale reagieren, tritt bei Simulation des Quellcodes keine Verzögerung auf. Dieser Unterschied führte jedoch im Zusammenspiel des Bildzählers und der Fensterkomparatoren mit den übrigen Einheiten zu Verletzungen der Data-Hold-Zeit in den Modellen der Flip-Flops, da deren Eingangsdaten nach dem Taktimpuls mindestens noch für die Data-Hold-Zeit von einer Nanosekunde konstant bleiben müssen. Der Statistikteil verfügt nicht über getaktete Ausgänge und konnte deshalb durch unverzögerte Reaktion auf ein Taktsignal keine Timing-Verletzungen hervorrufen.
Erst eine am MAZ neu hinzugekaufte Zusatzsoftware zum Simulator (Voyager „nsim“), ermöglichte nach anfänglichen Schwierigkeiten die Simulation des gesamten Histogrammers anhand der Netzliste. Die Zusatzsoftware befähigt den Simulator, zusätzlich zu der Interpretation von VHDL-Code auch direkt auf einer Netzliste im standardisierten EDIF-Format zu arbeiten und so den Umweg der Netzlisten-Simulation über VHDL-Modelle zu umgehen. Allerdings müssen zuvor Konversionsprogramme gestartet werden, um die vom Compass-System erstellte EDIF-Datei für den Simulator geeignet zu modifizieren und einige Hilfsfiles zu erzeugen. Weiterhin muß manuell für jede Komponente des Chips eine Konfigurationsdatei erstellt werden.
Abbildung 5.3 zeigt, welche Ausgabefiles des Compass-Systems für eine Netzlistensimulation benötigt werden.
Zu der Netzlisten-Simulations-Software erwarb das MAZ auch eine Beschleuniger-Hardware, die eine Steigerung der Simulationsgeschwindigkeit auf ein Vielfaches ermöglichen soll. Da jedoch die Simulation des Histogrammers per Software mit dem Testfile „histosti“ deutlich weniger Zeit in Anspruch nahm, als die zuvor nach Änderungen jeweils notwendigen Schritte (Synthese, Analyze und Elaborate der Files), wurde die Beschleuniger-Hardware nicht genutzt.

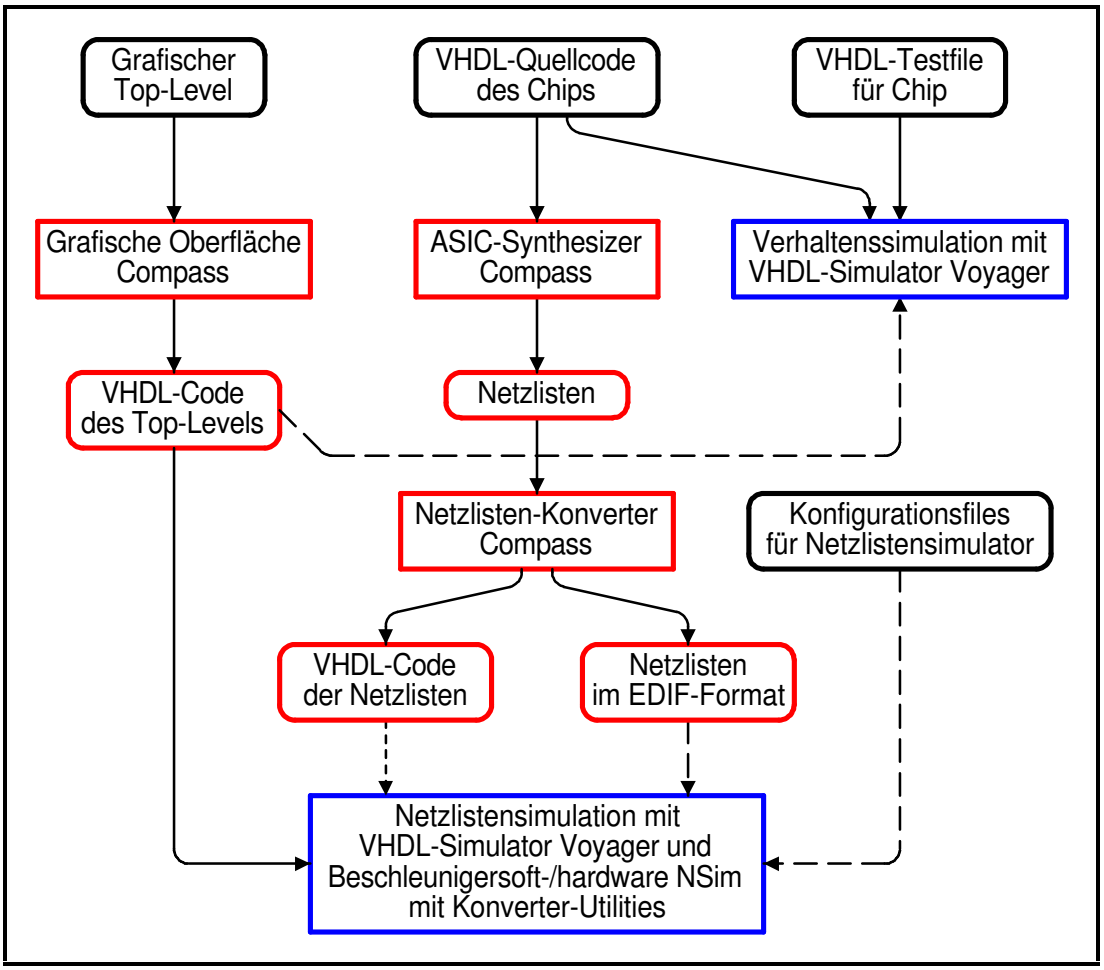

Abbildung 5.3: Eingabedaten für Netzlistensimulation

5.5.2 Anpassungen des VHDL-Modells

Bei der Simulation eines Entwurfes auf Basis der Netzliste können Timing-Probleme auftreten. Es wird angestrebt, diese Timing-Probleme durch Veränderung des VHDL-Quellcodes zu beseitigen, damit durch direkte Synthese des Quellcodes eine funktionsfähige Schaltung entsteht. Eingriffe in die vom Synthesizer erzeugte Netzliste dagegen müßten nach jeder Synthese manuell wiederholt werden.

In aufwendigen Entwürfen besteht die Gefahr, daß Signale innerhalb des Chips mit mehr Eingängen belastet werden, als eine einzige Ausgangsstufe treiben kann. Bei MOS-Schaltungen entsteht dabei eine deutliche Reduktion der Schaltgeschwindigkeit. Diese Verzögerung eines Signales wird durch den Innenwiderstand der Treiberstufe und die Lastkapazität bestimmt, die durch die Summe der Kapazitäten der angesteuerten Eingänge gebildet wird. Die Laufzeit t_{DELAY} eines Signals durch ein Gatter ergibt sich in erster Linie durch die Signallaufzeit t_{INT} innerhalb des Gatters und die Verzögerung durch kapazitive Belastung C_{LOAD} des Ausgangs:

$$t_{DELAY}=t_{INT}+R*C_{LOAD}$$

Der Proportionalitätsfaktor R ist ein Widerstand, der von der Treiberstärke abhängt.

Wird ein Ausgang zu stark belastet, muß ein Teil der Eingänge zusammengefaßt und über einen zusätzlichen Treiber an den Ausgang angeschlossen werden. Dies macht der ASIC-Synthesizer normalerweise automatisch.

Zu den am stärksten belasteten Signalen gehört meist das Taktsignal. Das Taktsignal steuert in einem synchronen Entwurf jedes Flip-Flop und dient als zentrale Zeitreferenz. Dabei ist nicht nur die Frequenz des Taktsignals entscheidend, sondern insbesondere auch der Zeitpunkt, zu dem getaktet wird. Es ist notwendig, daß alle Flip-Flops mit minimalem Zeitversatz getaktet werden, damit Strukturen wie z.B. Schieberegister funktionieren. Die Verschiebung (Skew) des Taktsignales zwischen den einzelnen Flip-Flops muß dafür geringer sein als die Differenz zwischen der Laufzeitzeit eines Flip-Flops und der Datenhaltezeit (des folgenden Flip-Flops) nach einem Taktimpuls. Diese Differenz beträgt etwa 1,5 Nanosekunden. Um eine möglichst geringe Taktverschiebung zu gewährleisten, verbindet das Compass-System alle Taktsignale miteinander und fügt für Taktsignale keine Treiber ein. Ein zur Ansteuerung aller Takteingänge ausreichend starker Treiber soll vom Anwender in Form einer Pad-Zelle innerhalb des Top-Levels eingefügt werden. Es zeigte sich jedoch, daß nur innerhalb jeweils einer Komponente strikt alle Takteingänge miteinander verbunden werden. Sobald eine Komponente über ca. 100 Flip-Flops enthält, fügt Compass innerhalb dieser Komponente einen zentralen Takttreiber ein. Seine Treiberstärke wird entsprechend der Lastkapazität ausgewählt. Aufgrund der dann aber schon sehr hohen Lastkapazität für diesen Treiber entsteht eine nicht vernachlässigbare Verzögerung des Taktes um ca. 5 ns gegenüber dem extern an die Komponente angelegten Takt. Diese Verzögerung sorgt dafür, daß die Zusammenarbeit mit Komponenten ohne Takttreiber schon bei der Netzlisten-Simulation fehlschlägt.

Eine solche Situation trat bei den Komponenten „histwin“ und „histstat“ des Histogrammers auf, die jeweils mehrere hundert synchron getaktete Flip-Flops beinhalten. Laut Auskunft der Herstellerfirma des Synthesewerkzeuges kann das Problem der synchronen Taktung großer Entwürfe zufriedenstellend nur durch Erwerb des zum Entwicklungssystem passenden sogenannten Clock-Tree Compilers gelöst werden.
Der Clock-Tree Compiler ist ein Werkzeug, das für die Taktleitung eines synchron arbeitenden Designs einen ausgewogenen Baum aus Takttreibern erstellt. Die Blätter dieses Takt-Baumes bestehen aus jeweils einer Anzahl von Takteingängen der Flip-Flops. Alle Blätter haben den gleichen Abstand zur Wurzel, an der zentral das Taktsignal eingespeist wird (s. Abbildung 5.4). Die Verzweigungen zu den einzelnen Blättern werden alle mit der gleichen Kapazität belastet. Eine Feinabstimmung dieser Kapazitäten findet durch das Einfügen von Kondensatoren statt. Dadurch und durch ein angepasstes Layout des Chips kann garantiert werden, daß nur sehr geringfügige Laufzeitunterschiede des Taktsignals zu den einzelnen Takteingängen bestehen. Angeboten wird der Clock-Tree Compiler zu einem mit dem Entwicklungssystem selbst vergleichbaren Preis.
Am MAZ war der Clock-Tree Compiler nicht verfügbar. Deshalb wurde zum Abgleich der Laufzeiten des Taktsignals innerhalb des Top-Levels manuell ein externer Takttreiber für alle diejenigen Komponenten eingefügt, innerhalb derer der ASIC-Synthesizer keinen Takttreiber eingefügt hatte (s. Anhang C). Treiber wurden vom ASIC-Synthesizer nur innerhalb der Komponenten „histwin“ und „histstat“ eingefügt, da diese jeweils über 100 Flip-Flops beinhalten. Die Stärke des externen Treibers wurde so gewählt, daß die durch die Flip-Flops der übrigen Komponenten insgesamt gebildete kapazitive Last zur gleichen Taktverzögerung führt, wie sie innerhalb der Einheiten „histwin“ und „histstat“ auftritt. Die Lastkapazität mußte manuell in das aus dem Top-Level automatisch generierte VHDL-Modell eingetragen werden, da die kapazitive Last der Komponenten für die Simulation auf Gatterebene im Top-Level nicht beachtet wird.
Ein manuelles oder nicht ausgewogenes automatisches Einfügen von Takttreibern führt insbesondere beim Layout von großen Entwürfen zu Problemen, da die Leitungsführung der Taktleitung zu den einzelnen Flip-Flops einen wesentlichen Einfluß auf die Laufzeit des Taktsignales hat. Dies resultiert aus dem nicht vernachlässigbaren Widerstand der Leiterbahnen. Unter Umständen werden viele Optimierungszyklen notwendig.

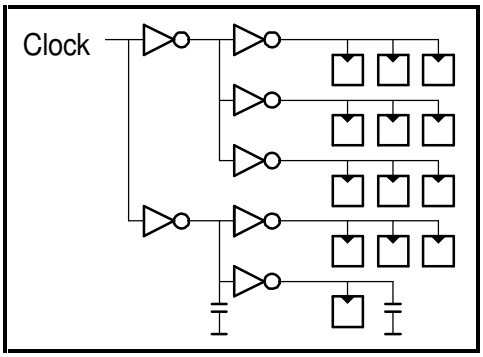

Abbildung 5.4: Clock-Tree

Bei der Netzlistensimulation traten weitere Timing-Probleme im Bereich der RAM-Ansteuerung auf.
Die zulässige Taktrate des Histogrammers wurde ursprünglich durch die minimale Dauer eines Schreibzugriffes auf das Histogramm-RAM (13 ns) begrenzt, da für den Schreibzugriff bei Histogrammerstellung und Kachelmodus nur ein halber Takt vorgesehen war (vgl. Kapitel 4.3.3). Um die zulässige Taktrate des Histogrammers zu erhöhen und gleichzeitig für den Histogramm-Inkrementer mehr Zeit zur Verfügung zu stellen, wurde der Schreib-Impuls von einem halben Takt auf einen ganzen Takt verlängert. Dies war möglich, da das verwendete asynchrone RAM keine Precharge-Time benötigt. Eine Einhaltung der vom RAM geforderten Haltezeit für Daten und Adressen nach einem Schreibzugriff (1 ns) konnte durch die Laufzeit innerhalb der Elemente der Adreß- und Daten-Erzeugung gewährleistet werden. Wo diese Verzögerung nicht von vornherein ausreichte, wurden die maßgeblichen Steuersignale der Einheiten während eines Schreibzugriffs auf das RAM gelatcht und erst durch das Ende des Schreib-Signals des RAMs wieder für Veränderungen freigegeben.
Weiterhin war sicherzustellen, daß auf der Read-/ Write-Leitung des RAMs keine Hazards entstehen. Dazu wurde ein separates Schaltwerk zur Ansteuerung dieser Leitung erstellt.

Alle Einheiten des Histogrammers sind über einen zentralen bidirektionalen Datenbus mit dem Businterface verbunden. Bei einem solchen Bus kann es zu Situationen kommen, in denen der Bus von keiner Einheit angesteuert wird (floating). Wenn dieser Zustand längere Zeit andauert, können sich die Pegel auf dem Bus auf einen beliebigen Wert zwischen den Logikpegeln einstellen. Dies führt zu hoher Stromaufnahme der angeschlossenen Eingangsstufen. Soll ein solcher Zustand verhindert werden, muß immer ein Treiber aktiv sein. Das bedeutet aber, daß bei einem Wechsel der Datenrichtung auf dem Bus gleichzeitig die zuvor aktivierten Treiber hochohmig geschaltet und andere Treiber niederohmig geschaltet werden müssen. Bei diesem Vorgang kann jedoch nicht garantiert werden, daß nicht kurzzeitig beide Einheiten den Bus treiben („Bus-Contention“). Diese Situation führt durch Kurzschlüsse auf den Datenleitungen ebenfalls zu einer hohen Spitzenstromaufnahme der Schaltung und sollte vermieden werden.
Über sogenannte Bus-Repeater können zeitweise nicht angesteuerte Busse auf definierten Pegeln gehalten werden. Diese Bus-Repeater sind aus einem rückgekoppelten Treiber mit sehr schwacher Ausgangsstufe aufgebaut. Sie sorgen dafür, daß die zuletzt aufgeprägten Pegel erhalten bleiben. Bus-Repeater sollen (laut Compass) eingesetzt werden, wenn ein Bus länger als 100 ns nicht getrieben wird. Solche Bus-Repeater wurden manuell innerhalb des Top-Levels des Histogrammers für den zentralen Datenbus und den Bus für RAM-Schreibdaten eingebaut. Die Datenausgänge der im Histogrammer verwendeten RAMs sind während eines Schreibzugriffs hochohmig, obwohl die RAMs über getrennte Ein- und Ausgänge verfügen. Deshalb wurden auch hier Bus-Repeater eingesetzt.

5.5.3 Einbau von Timing-Constraints

Bei der Synthese eines Entwurfes verwendet das Compass-System als Default Schaltungstechniken mit mittlerem Platzaufwand und mittlerer Laufzeit, da für sehr schnelle Strukturen im allgemeinen auch ein hoher Platzaufwand notwendig ist. Für einen Addierer wird beispielsweise ein sogenannter Conditional-Sum-Adder synthetisiert [5], dessen Flächenbedarf mit dem Produkt aus Bitanzahl und Logarithmus der Bitanzahl steigt und dessen Laufzeit proportional dem Logarithmus der Bitanzahl wächst [29]. Die erreichbare Taktfrequenz eines Chips ist nicht von vornherein festgelegt, sondern hängt stark vom Synthesewerkzeug ab. Um sicherzustellen, daß der Chip eine vorgegebene minimale Taktfrequenz erreichen kann, ist es möglich, innerhalb des synthesefähigen VHDL-Modells sog. Timing-Constraints in Form von Compilerdirektiven unterzubringen. Compilerdirektiven werden in Form von Kommentaren in den VHDL-Quellcode eingebracht.

Es gibt mehrere Typen von Timing-Constraints [5]:
- **Taktfrequenz**: Setzt eine minimal zu erreichende Arbeitsfrequenz bezüglich eines gegebenen Taktsignals
- **Ausgangs-Verzögerung**: Maximale zeitliche Verzögerung zwischen einem gegebenen Taktsignal und der Gültigkeit gegebener Ausgänge
- **Setup-Zeit**: Maximale Signallaufzeit zwischen gegebenen Eingängen und denjenigen Speicherelementen, die einen vorgebenen Takt verwenden
- **Maximale kombinatorische Verzögerung**: Maximale Laufzeit von einer Anzahl von Eingängen zu einer Anzahl von Ausgängen

Es können maximale, nicht aber minimale Laufzeiten vorgegeben werden. Es ist also nicht möglich, Verzögerungsketten zu erzeugen. Der Synthesizer versucht, die vorgegebenen Anforderungen mit minimalen Mehraufwand an Logikzellen gegenüber der Default-Lösung zu erreichen. Sämtliche Constraints können nur auf Ein- und Ausgangs-Signale von VHDL-*Entities* angewendet werden. Signale innerhalb einer *Entity* können nicht referenziert werden. Timing-Constraints können lokal zu einer *Entity* oder global definiert werden. Bei Angabe widersprüchlicher Constraints gelten die jeweils strengeren Constraints.
Die Angabe einer minimal zu erreichenden Taktfrequenz setzt Constraints für alle Pfade. Diese sind jedoch unter Umständen zu streng, da der Compiler annehmen muß, daß für die Pfade zwischen allen getakteten Einheiten eine Laufzeit von maximal einem Takt zur Verfügung steht.
Eine Einheit im Histogrammer, bei der eine Angabe der Taktfrequenz zu strenge Constraints setzen würde, ist beispielsweise der Statistikteil. Obwohl alle Flip-Flops des Statistikteils mit jedem Takt getaktet werden, übernehmen sie neu errechnete Werte nur bei Eintreffen eines Pixeltaktes, also höchstens alle zwei Takte. Deshalb ist es möglich, für die Berechnung neuer Eingangswerte zwei Takte zur Verfügung zu stellen. Hier ist es also sinnvoll, nicht die minimale Taktfrequenz, sondern (lokal) die maximalen Laufzeiten vorzugeben. Eine globales Constraint für die Taktfrequenz darf demnach nicht verwendet werden.

5.5.3.1 Wesentliche Timing-Constraints im Histogrammer

Die aufwendigsten Teile des Histogrammers sind die Teile „histwin“ (Fensterkomparatoren, Positionszähler), „histstat“ (Statistikteil), „histadd“ (ALU) und „histrmux“ (Multiplexer zur Ankopplung des RAMs an den Datenbus). Innerhalb dieser Teile wurden Timing-Constraints eingefügt, um die Funktion bei einer Taktfrequenz von 50 MHz Systemtakt bzw. 25 MHz Pixeltakt sicherzustellen. Im folgenden wird auf die Timing-Constraints für die drei Komponenten „histwin“, „histadd“ und „histstat“ näher eingegangen.

Die Komponente „histwin“ war auf Netzlisten-Basis laut Timing-Report des ASIC-Synthesizers bereits ohne Timing-Constraints mit 55 MHz funktionsfähig. Die Einfügung der Compilerdirektive (Frequency 50 CLK) ergab daher keine Veränderung des Syntheseergebnisses.

Die Komponente „histadd“ enthält die ALU für die Modifikation des RAM-Inhaltes. Die für die Berechnung des RAM-Inhaltes jeweils zur Verfügung stehende Zeit hängt von den Timing-Anforderungen des RAMs, der Taktfrequenz und Signallaufzeiten in weiteren Einheiten ab und kann deshalb bei Synthese der ALU nicht vom ASIC-Synthesizer berechnet werden. Die maximale Laufzeit der ALU muß manuell berechnet und dem Synthesizer in Form maximaler kombinatorische Verzögerungen angegeben werden.

Die für die Berechnung neuer Werte innerhalb der ALU verfügbare Zeit ist (vgl. Kapitel 4.3.3):

$t_{inc} = 2*t_{CLK} - t_{ACC} - t_{LATCH} - t_{SETUP}$

t_{CLK}	20 ns	Dauer einer Taktperiode bei 50 MHz
t_{ACC}	15 ns	Zugriffszeit des RAMs
t_{LATCH}	5 ns	Verzögerungen durch die Adreß- und Datenlatches der RAMs
t_{SETUP}	4,5 ns	Setup-Zeit für Schreibzugriff auf das RAM

$t_{inc} = 40 \text{ ns} - 15 \text{ ns} - 5 \text{ ns} - 4{,}5 \text{ ns} = 15{,}5 \text{ ns}$

Um die mit dem RAM theoretisch mögliche maximale Taktfrequenz von 66 MHz ($=1/t_{ACC}$) zu erreichen, müßte die Laufzeit der ALU und der Latches begrenzt werden auf:

$t_{inc}' + t_{LATCH}' = t_{ACC} - t_{SETUP} = 10{,}5 \text{ ns}.$

Die Laufzeit der ALU zwischen RAM-Leseport und RAM-Schreibport konnte auf 14,6 ns begrenzt werden. Laut Timing-Report des ASIC-Synthesizers benötigt der Addierer zur Aufsummierung im Kachelmodus 12,6 ns maximale Laufzeit. Für den Inkrementer konnte eine maximale Laufzeit von unter 12 ns erzielt werden. Zwei Nanosekunden Laufzeit benötigen die Multiplexer und Tristate-Treiber zur Ankopplung an den RAM-Datenbus. Diese bei maximaler Optimierung bestenfalls erzielbare Laufzeit der ALU wurde erst erreicht, nachdem eine niedrigere Laufzeit im Constraint angegeben wurde. Bei mittlerer Optimierung war die Laufzeit des Addierers und damit die Laufzeit der ALU bei gleichem Aufwand an Gattern um eine Nanosekunde höher.

Bei Synthese eines mit Timing-Constraints versehenen Quelltextes meldet der ASIC-Synthesizer nicht, ob die Constraints eingehalten werden konnten. Diese Information kann über das Kommando „report timing“ erhalten werden. Bei Untersuchung erster Syntheseergebnisse ergab sich, daß der Synthesizer das Timing für die ALU nicht einhielt. Obwohl die in oberster Ebene der Komponente vorgegebene maximale Laufzeit von 15,5 ns durch den Histogramm-inkrementer nach diesem Syntheseversuch um ca. 20 ns überschritten wurde, nahm der Synthesizer keine Veränderung der Strukturen in untergeordneten *Entities* der ALU vor. So wurden die einzelnen Teilinkrementer innerhalb des Histogramminkrementers weiterhin als (vereinfachte) defaultmäßig verwendete Conditional-Sum-Adder ausgeführt. Erst Timing-Constraints innerhalb aller untergeordneten *Entities* ermöglichten eine geringfügige Verringerung der Laufzeit. Die dann für die Einzelinkrementer erzeugte Struktur entsprach der in Kapitel 4.3.4 gezeigten. Die trotz optimierter Inkrementer nahezu unverändert hohe Laufzeit war unter der Annahme erklärlich, daß der Carry-Pfad weiterhin seriell durch alle Teilinkrementer lief. Offensichtlich betrachtet der Synthesizer alle untergeordneten *Entities* bei

der Optimierung als black-box, und konnte deshalb die serielle Struktur nicht parallelisieren. Um die maximale Länge der Carry-Pfade zu verringern, wurden innerhalb des Quellcodes des Histogramm-Inkrementers manuell mehrere parallel zu errechnende Carry-Signale eingeführt. Jedoch meldete der ASIC-Synthesizer auch nach dieser Maßnahme eine nahezu unverändert hohe Laufzeit.
Zur näheren Prüfung der Ursachen für die weiterhin zu hohe gemeldete Laufzeit wurde der im Compass-System als zusätzliches Progamm integrierte sog. Timing-Verifier verwendet. Dieses Programm ermöglicht es, die maximale Laufzeit und den längsten Pfad zwischen zwei beliebigen Punkten in der Schaltung zu bestimmen oder zu einem beliebigen Zielpunkt denjenigen Punkt zu bestimmen, von dem aus die Laufzeit zum Zielpunkt maximal ist.
Der Timing-Verifier meldete genauso hohe maximale Laufzeiten wie der Synthesizer. Eine genauere Untersuchung des vom Timing-Verifier ausgegebenen längsten Pfades innerhalb des Histogramminkrementers behauptete, daß trotz parallel generierter Carry-Signale weiterhin der gesamte Carry-Pfad seriell durchlaufen würde. Tatsächlich jedoch konnte dies nicht sein, da konfigurationsabhängig jeweils nur wenige Inkrementer als Ripple-Carry-Kette verschaltet waren. Es gab keine Kombination der Steuerleitungen, in der alle Inkrementer seriell nacheinander durchlaufen wurden. Daß weiterhin dieser Pfad ausgemessen wurde, resultiert offensichtlich daraus, daß weder ASIC-Synthesizer noch Timing-Verifier prüfen, ob es eine Belegung der Eingangsleitungen gibt, mit der der gefundene maximale Pfad tatsächlich geschaltet werden kann.
Da es auf diese Weise nicht möglich war, anhand der Timing-Reports sicherzustellen, daß der Inkrementer in jedem Fall die maximale Laufzeit unterschreitet, wurde das Problem durch vollständigen Verzicht auf die Benutzung von Ripple-Carries umgangen.
Zur Ansteuerung jedes Teilinkrementers in jeder Konfiguration wird ein paralleles Carry-Signal generiert. Die Carry-Signale werden, wie in Kapitel 4.3.4 gezeigt, aus den Signalen für das Clipping bei Überlauf erzeugt. Auf den Carry-Ausgang der Inkrementer konnte so verzichtet werden und es enstand kein Mehraufwand an Gattern.

Laut Handbuch kann der Timing-Verifier keine kombinatorische Logik auswerten, in der Schleifen auftreten, wie z.B. innerhalb von Flip-Flops. Die Unfähigkeit, rückgekoppelte Strukturen auszuwerten, hängt wahrscheinlich mit dem gefundenen Verhalten zusammen.

In der Komponente „histstat“ wurden Constraints für Setup-Zeiten eingefügt, um dem ASIC-Synthesizer anzuzeigen, wie lange die betroffenen Signale vor einem Takt stabil anstehen. Diese Constraints wurden jedoch vom Synthesizer nicht akzeptiert. Auch Constraints für Ausgangs-Verzögerungen führten regelmäßig zu der Meldung: „can’t compute constraint“. Um die Laufzeit der den Registern vorgeschalteten Recheneinheiten dennoch einschränken zu können, wurden die Recheneinheiten separat von den Registern in eine *Entity* eingebunden und Constraints für die kombinatorischen Verzögerungen eingesetzt. Für die Setup-Zeit von Flip-Flops und vorgeschalteten Multiplexern wurde dazu ein mit dem Timing-Verifier überprüfter Schätzwert angenommen. Die für die Pixelfrequenz notwendigen Rechenzeiten wurden jedoch schon ohne Constraints deutlich unterschritten. Lediglich der Quadrierer brauchte allein 35 ns Rechenzeit, also fast doppelt so lange wie vorgesehene Rechenzeit von einem Systemtakt (20 ns). Dieses Ergebnis konnte jedoch auch durch Constraints nicht verändert werden. Offensichtlich findet zumindest bei mittlerem CPU-Aufwand für Optimierung keine Optimierung von Multiplizierern statt. Dies hängt möglicherweise damit zusammen, daß die Realisierung von schnellen Multiplizierern in Hardware sehr aufwendig ist. Zur Abhilfe wurde eine zusätzliche Pipelinestufe für den Quadrierer eingefügt. So steht ein voller Pixeltakt von minimal 40 ns zur Verfügung.