Effiziente Generierung von Zustandsautomaten mit integriertem Datenpfad aus taktgebundenen VHDL-Verhaltensbeschreibungen

Walter Lange

WSI-2000-18

6. September 2000

Wilhelm-Schickard-Institut
Universität Tübingen
D-72076 Tübingen, Germany
e-mail: wlange@informatik.uni-tuebingen.de

© WSI 2000
ISSN 0946-3852

Effiziente Generierung von Zustandsautomaten mit integriertem Datenpfad aus taktgebundenen VHDL-Verhaltensbeschreibungen

Walter Lange

August 2000

Zusammenfassung

Die Entwicklung immer komplexerer und grösserer Schaltungen sowie die Forderung nach kürzeren Entwicklungszyklen zwingt den Entwickler neue, effiziente Wege zu suchen. Eine dieser Wege ist, Schaltkreise auf höheren Abstraktionsebene zu beschreiben. Wünschenswert wäre, wenn die Beschreibung auf höheren Abstraktionsebene automatisch auf die Physikalische Ebene synthetisiert würde. Die vorliegende Arbeit beschreibt ein neues Verfahren einer automatischen Generierung von Zustandsautomaten (FSM's) mit integriertem Datenpfad (DP) aus taktgebundenen VHDL-Verhaltensbeschreibungen auf algorithmischer Ebene. Die FSM mit integriertem DP entspricht einer Verhaltensbeschreibung auf RT-Ebene und kann effizient durch kommerziell verfügbare Werkzeuge weiter synthetisiert werden. Dieses Verfahren kann insbesondere für Datenübertragungs(DÜ)-Schaltungen und Protokoll-Logik eingesetzt werden.

Inhaltsverzeichnis

1 Einleitung **4**

2 Ziel-Systeme und Vorteile der Umwandlung **4**
2.1 Taktfreie Verhaltensbeschreibungen 4
2.2 Verfahren für taktgebundene Verhaltensbeschreibungen 5

3 VHDL-Parser und Grundlagen der FSM-Generierung **5**
3.1 VHDL-Parser 5
3.2 Die Programmparameter (Optionen) 5
3.3 Grundlagen der FSM Generierung 6
3.4 Die Eingabe: Schema der VHDL-Verhaltensbeschreibung 6

4 Konstruktion der FSM **8**
4.1 Allgemeines 8
4.2 Rahmen der Ausgangsbeschreibung 8
4.2.1 Erstellung des Programm-Rahmens 9
4.3 Umwandlungsalgorithmen 13
4.3.1 Ein einfaches “Wait” 13
4.3.2 UND-Verknüpfung eines “Wait” mit einer Signal-Bedingung . . . 14
4.3.3 Die gerollte “FOR”-Schleife 15
4.3.4 Die aufgerollte “FOR”-Schleife 18
4.3.5 Die “WHILE”-Schleife 18
4.3.6 “IF ... THEN ... ELSE” 20
4.3.7 Verschachtelungen 22
4.3.8 Zustandszusammenfassung bei Verschachtelungen 24

5 Zusammenfassung und Ausblick **30**

6 Literatur **32**

7 Anhang: Beispiele für die Generierung von FSM’s aus Verhaltensbeschreibungen **33**
7.1 Eingangsmodul AHT_IN 33
7.1.1 Eingangsbeschreibung 33
7.1.2 Ausgangsbeschreibung 34
7.2 Zellkopfübersetzungsmodul HT 37
7.2.1 Eingangsbeschreibung 37
7.2.2 Ausgangsbeschreibung 38
7.3 Routing-Steuerung (RC) 39
7.3.1 Eingangsbeschreibung 39
7.3.2 Ausgangsbeschreibung 42
7.4 Schieberegister-Steuerung (SRC) 46
7.4.1 Eingangsbeschreibung 46
7.4.2 Ausgangsbeschreibung 49

7.5 Verbindungs-Steuerung (CC) 51
7.5.1 Eingangsbeschreibung 51
7.5.2 Ausgangsbeschreibung 57
7.6 Datenaufnahme-Modul RD 66
7.6.1 Eingangsbeschreibung 66
7.6.2 Ausgangsbeschreibung 70
7.7 AHT-Ausgangs-Modul 73
7.7.1 Eingangsbeschreibung 73
7.7.2 Ausgangsbeschreibung 75

1 Einleitung

Die Entwicklung immer komplexerer und grösserer Schaltungen sowie die Forderung nach kürzeren Entwicklungszyklen (time to market) zwingt den Entwickler neue, effiziente Verfahren einzusetzen. Eines dieser Verfahren ist, das Verhalten von Schaltkreisen mittels einer "Hardware-Beschreibungssprache" (HDL: HW Description Language), z.B. VHDL zu beschreiben, zu simulieren oder formal zu verifizieren und danach zu synthetisieren.

Die in Verhaltens-VHDL beschriebenen Schaltkreise können mit kommerziell verfügbaren High-Level-Synthese-Werkzeugen synthetisiert werden. Sie transformieren die Verhaltensbeschreibung in eine RT-Beschreibung auf Struktur-Ebene.

Liegt jedoch eine Verhaltensbeschreibung für taktgebundene Schaltungen vor, wie sie sehr häufig für Datenübertragungs (DÜ)-Schaltungen angewendet wird, so ist der aufwendige Scheduling-Schritt der High-Level Synthese nicht nötig. In diesem Fall genügt es, wenn die verhaltensbasierte HW-Beschreibung mit Hilfe des hier beschriebenen Verfahrens in eine solche auf RT-Ebene umgewandelt wird, da verfügbare Logik-Synthese Werkzeuge wie z. B. der "Design Compiler^{TM}" (DC) der Fa. Synopsys^{TM} diese Beschreibung effektiv weiter synthetisieren können.

In der vorliegenden Arbeit wird diese Umwandlung, d.h. die Generierung eines Zustandsautomaten (FSM) mit integriertem Datenpfad für taktgebundene VHDL-Beschreibungen als neues kostengünstiges und sehr schnelles Verfahren dargestellt, das eventuell in ein High-Level-Synthesesystem eingebunden werden könnte.

Die Arbeit gliedert sich wie folgt: Im nächsten Kapitel werden die Ziel-Systeme und die Vorteile dieser Methode aufgeführt. In Kapitel 3 wird der VHDL-Parser und die FSM Generierung kurz beschrieben. In Kapitel 4 wird die Konstruktion der FSM detailliert dargestellt. Der Anhang enthält Beispiele von Eingangs- und Ziel- Beschreibungen.

2 Ziel-Systeme und Vorteile der Umwandlung

2.1 Taktfreie Verhaltensbeschreibungen

"Taktfreie" (free floating) Verhaltensbeschreibungen [Gutberl97] auf algorithmischer Ebene werden automatisch mit Hilfe von High-Level-Synthese (HLS)-Systemen in RT-Beschreibungen auf Struktur-Ebene umgewandelt und können danach weiter einer Logik-Synthese zugeführt werden.

Ziel-Systeme für die Synthese taktfreier VHDL-Verhaltensbeschreibungen sind z.B.: Schaltungen, die hauptsächlich arithmetische Operationen ausführen und bei denen es auf eine taktgenaue und kontinuierliche Datenübertragung nicht ankommt. Diese Systeme sind in der Praxis z.B. digitale Filter, Bild-Kompressions- und Dekompressions-Schaltungen (z.B.: für JPEG-Bildformate), Signalverarbeitungs- und ähnliche Schaltungen. Die High-Level Synthese (HLS) weist den arithmetischen Operationen im ersten Schritt Komponenten aus einer Komponenten-Bücherei zu. Im "Scheduling"-Schritt werden die Operationen Taktzyklen zugeordnet, und im "Allokation- Schritt werden die Operationen durch die ausgesuchten Komponenten ersetzt wobei meist eine Optimierung

der Schaltungsfläche bei vorgegebener Verzögerungszeit durchgeführt wird. Das HLS-Werkzeug liefert am Ausgang die Strukturbeschreibung des Datenpfads (DP) auf RT-Ebene, die aus VHDL-Strukturbeschreibungen der Komponenten, Register und Verbindungslogik bestehen. Zusätzlich wird der Kontrollpfad bzw. die FSM (Finite State Machine) automatisch generiert und optimiert. Ausgegeben wird in der Regel die VHDL-Strukturbeschreibung des DP mit externer VHDL-Beschreibung der FSM als Komponente des DP.

2.2 Verfahren für taktgebundene Verhaltensbeschreibungen

Ziel-Systeme für taktgebundene (cycle-fixed) Verhaltensbeschreibungen sind DÜ Systeme, z.B.: Router, Hubs, ATM-Verteiler usw. Bei diesen Systemen kommt es auf eine taktgenaue Datenübertragung an.

Die Steuerschaltungen und die Protokoll-Erkennung [SynPC99] für diese Systeme können in taktgebundener Verhaltens-VHDL beschrieben werden.

Taktgebundene Verhaltens-VHDL-Beschreibungen können mit der Option “cycle-fixed I/O mode” einer HLS unterzogen werden, um die Umwandlung in eine RT-Strukturbeschreibung zu erhalten. Allerdings sind diese Systeme hier unterfordert, da die Stärken der HLS-Systeme in effektivem Scheduling und Zuordnen von Komponenten (Assignement oder Binding) liegen. Das Scheduling ist aber im taktgebundenen Modus bereits vorgegeben.

Das neue, vorgestellte Verfahren wandelt taktgebundene VHDL-Verhaltensbeschreibungen in VHDL-Beschreibungen von FSM’s mit integriertem DP um, die der RT-Ebene zuzuordnen sind.

Die Vorteile der Umwandlung gegenüber einer HLS sind:

- Die Umwandlung geschieht wesentlich schneller als die HLS.
- Die Beschreibung ist vom Logik-Synthese-System DC von Synopsys besser optimierbar und liefert damit bessere Ergebnisse in Bezug auf Fläche und Verzögerungszeit als die Ausgabe der dem Autor bekannten kommerziellen HLS-Systeme.

3 VHDL-Parser und Grundlagen der FSM-Generierung

3.1 VHDL-Parser

Als Basisprogramm wird ein vorhandener VHDL Parser verwendet, der entsprechend den nachfolgenden Algorithmen die VHDL-Beschreibung einer FSM mit integriertem DP erstellt.

3.2 Die Programmparameter (Optionen)

Als Parameter werden angegeben:

1. Namen der Eingabedatei. Die Eingabedatei ist die verhaltensbasierte VHDL-Beschreibung. (z.B. ahtin.vhd)

2. Der Taktname. Er erscheint als IN-Port in der “Entity” und wird als Synchronisationstakt in den “Wait..” Instruktionen verwendet. (Z.B.:“clk”)

3. Der Name des Rücksetzsignals (Reset). Erscheint als IN-Port in der “Entity” und kann als synchrones Rücksetzsignal verwendet werden. (Siehe Synchronisations--Prozess). Zusätzlich wird die Aktiv-Polarität angegeben: D.h. “active high” oder “active low”.

4. Angabe der “sensitvity list” pro Prozess, falls sie vom Vorgabewert (s. u.) abweicht.

5. Der Name der Ausgangsdatei. Sie ist eine VHDL-RT-Beschreibung der FSM.

3.3 Grundlagen der FSM Generierung

Die VHDL-Eingabebeschreibung wird “geparst” und für jeden VHDL-Prozeß wird eine separate FSM entsprechend den unten beschriebenen Algorithmen generiert. Die Vorlage für die Eingabe-Verhaltensbeschreibung mit der FSM Generierung ist dem “Synopsys DC Tutorial” [SynDCTut99] entnommen und wurde vom Autor in verschiedenen Bereichen wie z.B.: “Verschachtelte Strukturen”, “sensitivity list”, Schleifen, “Synchronisations-Prozess” usw. modifiziert.

Mit der unten beschriebenen Methode der FSM Generierung wird eine **“Zustandsexplosion”** des endlichen Automaten dadurch vermieden, daß Zustände, die theoretisch möglich, aber praktisch unerreichbar sind, nicht erzeugt werden. Dies geschieht dadurch, daß in jedem Zustand der Folgezustand bzw. die Folgezustände explizit definiert werden.

3.4 Die Eingabe: Schema der VHDL-Verhaltensbeschreibung

Die Eingabe-Verhaltensbeschreibung wird in der HW-Beschreibungssprache VHDL’93 erstellt. Da der Sprachschatz von VHDL’93 sehr groß ist, (er umfaßt über 200 Sprachkonstrukte), und damit auch der Vielfalt der Beschreibungsmöglichkeiten kaum Grenzen zu setzen sind, wird ein Eingabeschema vorgegeben, das die Komplexität des Umwandlungsprogramms auf das Nötigste reduziert.

Anhand des folgenden schematischen Beispiels wird die Struktur der Eingabebeschreibung vorgestellt.

```
-- (1)
-- Copyright 2000 University of Tuebingen  ..
--
-- Verhaltensbeschreibung des HT (Header Translator)
--
-- Autor: W. Lange. Letzte Aenderung: ...

Library IEEE; -- (2)
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY ht IS -- (3)
  PORT (Clk_com     : IN  STD_LOGIC ;       -- Clock common
```

```
        Hd_Bus       : IN  STD_LOGIC_Vector(31 DOWNTO 0); -- Header
        ...
        Reset        : IN  STD_LOGIC);
END ht;

ARCHITECTURE behavior OF ht IS  -- (4)
  SIGNAL ....... -- (5)

BEGIN -- (6)
 htproc : PROCESS -- (7)
 -- (8)
  VARIABLE  Address    : STD_LOGIC_Vector(0 TO 15);
  ...

 BEGIN -- Process (9)
     -- Reset-Teil (10)
     Hd_fetch <= '0';  -- Setze die Signale auf null..
     ...
     WAIT UNTIL Clk_com'event and Clk_com = '1';

     -------------------- Haupt-Loop (11) -----------------------
     main_loop: LOOP
       -- Hole einen Header vom AHT1_In ab
       WAIT UNTIL  Clk_com'EVENT and Clk_com = '1'and Hd_rdy = '1';
       Header(31 downto 0) := Hd_Bus;

       WAIT UNTIL Clk_com'event and Clk_com = '1' and Hd_rdy = '0';
       ...
       IF <condition> THEN ... END IF;
       ...
       WHILE <condition> ... END loop;

       FOR i in 1 to <upper_limit> loop  ... END loop;
       ...
   END LOOP;
   WAIT UNTIL Clk_com'event and Clk_com = '1';
 END htproc;  -- 12

 second_proc: Process -- 13 --
  VARIABLE ...
  BEGIN
  ...
  END second_proc;

END behavior;           -- End architecture (14)
```

Die einzelnen Felder sind hier zur Veranschaulichung mit “– (xx)” numeriert und wie folgt beschrieben:

Die Verhaltensbeschreibung beginnt mit einem “Kommentarkopf” (1), bei dem jede Zeile mit einem Doppelstrich “–” beginnt.

Danach folgen die Library-Definitionen (2), beginnend mit dem Schlüsselwort “Library” und/oder “Use <library-definition>”

Die “Entity” (3) beginnt mit dem Schlüsselwort “Entity” und endet mit “END”.

Die “architecture” Deklaration (4) beinhaltet den Architektur-Namen und den Entity-Namen.

```
ARCHITECTURE <architecture-name> OF <entity-name> IS
```

Dahinter sind Deklarationen (5) von Signalen und Typen, die für alle Prozesse der Architektur übergeordnet gelten.

Architektur-Start (8) und Prozess-Start (9) sind mit dem Schlüsselwort “Begin” bezeichnet.

Ein Feld mit Deklarationen (8) für Variablen, Konstanten etc. hinter dem Prozess-Start gilt lokal für die Prozedur.

Direkt hinter dem Schlüsselwort “Begin” beginnt der Reset-Teil des Prozesses. Er endet an der Hauptschleife des Prozesses, die mit “While True loop” oder mit <label>: loop beginnt und mit “end loop;” endet. Der Prozeß endet mit “END <process-name>;” (12).

Es können weitere Prozesse (13) in der Architektur definiert sein, oder auch nur einzelne Instruktionen, die dann jeweils wie ein Prozeß behandelt werden.

Die Architektur wird mit dem Schlüsselwort “End <architecture-name>;” beendet. Die Angabe von <process-name> und <architecture-name> hinter “End” ist optional.

4 Konstruktion der FSM

4.1 Allgemeines

Die FSM eines Prozesses wird in in der Regel in einen Haupt-Teil und in FSM-Zweige unterteilt. Die **Zustandsmenge**, als Aufzählungstyp deklariert, steht zu Beginn der Konstruktion noch nicht fest, sie wird im Laufe der FSM-Entwicklung definiert, wobei sowohl für den FSM-Hauptteil, als auch für die Nebenteile separate Zustandsmengen festgelegt werden. Die Konstruktion beginnt mit der **Erstellung des Programm-Rahmens**. Ausgehend von der Eingangsbeschreibung werden in den Programmrahmen Schritt für Schritt die FSM-Teile entsprechend den Umwandlungsalgorithmen (siehe Kapitel 4) eingefügt.

4.2 Rahmen der Ausgangsbeschreibung

Die Ausgangsbeschreibung stellt die VHDL-Beschreibung eines endlichen Automaten (oder Finite State Machine: FSM) mit integriertem Datenpfad dar.

Im Prinzip besteht der Hauptteil jedes Prozesses aus einer großen Fallunterscheidung (CASE <state_variable> IS), die alle möglichen Zustandssymbole enthält und in der folgende Aktionen durchgeführt werden:

Der jeweils aktuelle Zustand des Prozesses wird in der <state_variable> abgefragt, die entsprechenden DP-Operationen die zu diesem Zustand gehören sind in VHDL beschrieben und der nächste Zustand wird in die <next_state_variable> gesetzt. Verzweigungen werden dadurch erzeugt, daß je nach Prüfung einer Bedingung verschiedene “nächste” Zustandssymbole in die <next_state_variable> gesetzt werden können.

In einem separaten Synchronisations-Prozeß wird jeweils bei Eintreffen der positiven oder negativen Takt-Flanke (Clock) der Inhalt der “nächsten” Zustandsvariablen

(<next_state_variable>) der aktuellen Zustandsvariablen <state_variable> zugeordnet. Damit werden pro Taktzyklus die Zustände der FSM weitergeschaltet.

4.2.1 Erstellung des Programm-Rahmens

Als erstes ist der sog. Programm-Rahmen der VHDLAusgangsbeschreibung zu erstellen, der folgende Teile beinhaltet:

1. Kommentarkopf: wird von der Eingangsbeschreibung kopiert
2. Library-Angaben: werden kopiert.
3. Die Entity-Beschreibung wird kopiert.
4. Die "architecture"-Zeile wird kopiert, jedoch der Architekturname durch "rtl" ersetzt.
5. Die Architektur-Deklarationen werden übernommen, jedoch die Definition des Zustandstyps mit den Zustandssymbolen (siehe Details) wird neu eingefügt sowie die Deklaration der Zustandsvariablen.
6. Die Prozess-Namen werden übernommen.
7. Der Synchronisations-Prozeß wird eingefügt.

Anhand des folgenden schematischen Beispiels wird die Gliederung des Rahmens der Ausgangsbeschreibung dargestellt.

```
-- (1)
-- Copyright 2000 University of Tuebingen  ..
--
-- Verhaltensbeschreibung des HT (Header Translator)
--
-- Autor: W. Lange. Letzte Aenderung: ...

Library IEEE; -- (2)
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY ht IS -- (3)
  PORT (Clk_com     : IN  STD_LOGIC ;       -- Clock common
        Hd_Bus       : IN  STD_LOGIC_Vector(31 DOWNTO 0); -- Header
        ...
        Reset        : IN  STD_LOGIC);
END ht;

ARCHITECTURE rtl OF ht IS -- (4a)
  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4 ..... );  -- (5a)
  SIGNAL state_0, next_state_0, state_1, next_state_1 : state_type;
  SIGNAL sync_sig : STD_LOGIC;  -- Synchronisations-Signal
```

```
BEGIN  -- (6a)
 htproc : PROCESS(sync_sig, Reset) -- (7a)
  VARIABLE  Address    : STD_LOGIC_Vector(0 TO 15);     -- (8a)
  ...

 BEGIN -- Process (9a)
   -- Reset-Teil (10a)
   IF (Reset = '1') THEN
     Hd_fetch <= '0';  -- Setze die Signale auf null..
     fwrq     <= '0';
     rt_read  <= '0';
     next_state <= ST0;  -- Zeile eingef"ugt
   ELSE
     CASE state_0 IS  -- (11a)
      WHEN ST0 =>
         IF (Hd_rdy = '1') THEN
            Header(31 downto 0) := Hd_Bus;
            Hd_fetch <= '1';          -- Sag Header abgeholt!
            next_state_0 <= ST1;
         ELSE
            next_state_0 <= ST0;
         END IF;

      WHEN ST1 =>
        IF (Hd_rdy = '0') THEN
          ...
          next_state_0 <= ST2;
        ELSE
          next_state_0 <= ST1;
        END IF;
        ...
        WHEN others =>
      END CASE;
   END IF;
 END htproc; -- (12a)

 second_proc: Process(Clk_com, Reset) -- (13a)
  VARIABLE ...
  BEGIN
  ...
  END second_proc;

  Sync_Proc: PROCESS  -- (15)
  BEGIN
        IF Reset = '1' THEN
          state_0 <= ST0;
          state_1 <= ST0;
          sync_sig <= '0';
          Wait until Clk_com'EVENT AND Clk_com = '1';
        ELSE
         if sync_sig = '0' then
           sync_sig <= '1';
         else
           sync_sig <= '0';
         end if;
```

```
          state_0 <= next_state_0;
          state_1 <= next_state_1;
         Wait until Clk_com'EVENT AND Clk_com = '1';
        END IF;
  END Process;
END rtl;           -- End architecture (14a)
```

Wie bei der Eingangsbeschreibung sind auch hier die einzelnen Felder zur Veranschaulichung numeriert. Die einander entsprechenden Felder werden mit der gleichen Ziffer mit angehängtem "a" bezeichnet ("– (xxa)").

Die Felder (1) bis (3) werden ungeändert nach (1a) bis (3a) übernommen.

Im Architekturfeld (4) wird der <architecture-name> in "rtl" geändert. (Siehe (4a)) Danach werden im Deklarationsteil (5a) die Zustandstypen mit den verwendeten Zustandssymbolen, die Zustandsvariablen und die Zustandsvariablen für die "nächsten" Zustände eingeführt.

Zustandstypen für den Hauptteil und für die Zweige der FSM:

```
  TYPE   state_type IS (ST0,ST1 ....   -- Symbole f"ur den Hauptteil
                       ST_A1,ST_A2..   -- Symbole f"ur die Zweige
                       ST_B1,ST_B2.. );
```

Man beginnt den Hauptteil mit dem Symbol ST0. Die folgenden Symbole ST1, .. werden während der Konstruktion der FSM hinzugefügt. Werden Zweige konstruiert, z.B. für Schleifen, IF..- Konstrukte usw., so werden jeweils FSM-Zweige gebildet. Der erste Zweig erhält die Symbole ST_A1, ST_A2,.. der zweite Zweig erhält die Symbole ST_B1,ST_B2,.. usw. Die Typ-Deklaration gilt gemeinsam für alle Prozesse, d.h. die **Aufzählungsmenge** der Symbole ist eine **Vereinigungsmenge** der Symbole für alle Prozesse.

Für jeden Prozeß muß ein Variablen-Paar aus aktueller Zustandsvariablen "state_x" und Variablen für den nächsten Zustand "next_state_x" generiert werden.

Die Zustandsvariablen und die Variablen für den nächsten Zustand werden beispielhaft für die ersten zwei Prozesse einer Architekturbeschreibung wie folgt definiert:

```
  SIGNAL state_0, next_state_0, state_1, next_state_1 : state_type;
```

Das Feld (6) "BEGIN" wird kopiert (siehe (6a)). Ebenso werden alle Kommentare (beginnend mit "- -" bis zum Zeilenende) kopiert.

Dem Feld "Process" wird die sog. Sensitivity-Liste (sensitivity list) zugefügt. Das ist ein Parameterfeld, das im Vorgabe-Fall (Default) das Synchronisations-Signal ("sync_sig") und den Reset-Namen (z.B.: Reset) enthalten, die zu diesem Prozeß gehören. Als Option kann zu einem Prozeß mit Namen <proc_name> eine von der Vorgabe abweichende Sensitivity-Liste zugefügt werden z. B.:

```
 -sens_list <proc_name> (<list>)
```

Das Variablenfeld (8) des Prozesses und das "Begin"-Feld (9) werden übernommen (8a).

Dem "Reset-Teil" (10) wird eine Zeile vorangestellt und ein "ELSE" angehängt:

```
IF (<reset-name> = '1') THEN -- falls ``active high'' angegeben,
                               -- sonst: = '0';
...
ELSE
```

Vor dem “ELSE” wird eine Zeile eingefügt, in der die <next_state_variable>, die zu diesem Prozeß gehört auf das erste Zustandsymbol im Hauptteil (main loop) gesetzt wird. Das erste Zustandssymbol ist in der Regel ST0, falls der “Reset”-Teil keine “Waits” enthält. Siehe Beispiel Connection Controller (CC).

Beispiel:

```
next_state <= ST0;  -- Zeile eingef"ugt
```

Die “Wait”-Instruktion am Ende des “Reset”-Feldes wird ignoriert.

Der Beginn der umfassenden Fallunterscheidung für die FSM-Zustände (CASE ..) wird eingefügt (Feld 11a)

```
CASE <actual state variable> IS  -- (11a)
 WHEN ST0 =>
```

Die Beschreibung der Umwandlung der einzelnen VHDL-Instruktionen in Zustände erfolgt im nächsten Kapitel (Umwandlungsalgorithmen).

Die “CASE”-Instruktion und die vorausgehende “IF”-Instruktion werden mit “WHEN others => END CASE; END IF;” unmittelbar vor der Beendigung des Prozesses abgeschlossen.

Der Rahmen der nächsten Prozesse, soweit vorhanden, werden äquivalent zum ersten Prozeß behandelt.

Schließlich wird der Synchronisations-Prozeß eingefügt entsprechend untenstehendem Beispiel (15).

Für jeden Prozeß wird in den Reset-Teil (nach dem IF-Konstrukt) eine Zeile eingefügt:

```
state_x <= ST0;
```

In den Umschalt-Teil (im ELSE-Block) wird pro Prozeß ebenfalls je eine Zeile eingefügt:

```
<actual_state_variable> <= <next_state_variable>;
```

Das **Synchronisations-Signal “sync_sig”** ändert seinen Zustand bei jeder positiven Taktflanke, es hat somit genau die halbe Taktfrequenz. Es ist dann nötig und wird in die Sensitivity-Liste übernommen, wenn bei der Simulation der Prozess nur einmal pro Taktzyklus durchlaufen werden soll, zum Beispiel zum Zählen von Schleifendurchläufen. Steht der Takt selbst in der Sensitivity-Liste, so wird der Prozess bei posituver und negativer Taktflanke durchlaufen und der Schleifenzähler wird doppelt inkrementiert.

Beispiel für einen Synchronisations-Prozeß mit 2 Prozessen:

```
Sync_Proc: PROCESS  -- (15)
BEGIN
      IF <reset_name> = '1' THEN
        state_0 <= ST0;
        state_1 <= ST0;
        sync_sig <= '0';
        Wait until Clk_com'EVENT AND Clk_com = '1';
      ELSE
       if sync_sig = '0' then
         sync_sig <= '1';
       else
         sync_sig <= '0';
       end if;
        state_0 <= next_state_0;
        state_1 <= next_state_1;
        Wait until Clk_com'EVENT AND Clk_com = '1';
      END IF;
END Process;
```

Zuletzt wird die Zeile "End <architecture_name>;" durch die Zeile "End rtl;" ersetzt.

4.3 Umwandlungsalgorithmen

Das Hauptprogramm, das in der Eingangsbeschreibung als "main-loop" bezeichnet wird, beginnt mit dem ersten Zustand (ST0).

Die ersten VHDL-Instruktionen werden nach Beginn der umfassenden Fallunterscheidung (CASE ..) hinter den Ausdruck "WHEN ST0 => solange kopiert, bis einer der folgenden Ausdrücke erscheint:

1. Ein einfaches "Wait ..."
2. UND-Verknüpfung eines"Wait mit einer Signal-Bedingung.
3. Eine "FOR"-Schleife
4. Eine "While"-Schleife
5. Eine "IF .. THEN .. ELSE"-Instruktion

All diese Instruktionen können auch verschachtelt auftreten.

Die Algorithmen für die Behandlung der o.g. Ausdrücke werden in den folgenden Unterkapiteln beschrieben.

4.3.1 Ein einfaches "Wait"

Der Parser stößt auf ein einfaches "Wait", d. h. im Programm wird an dieser Stelle auf die postitive oder negative Flanke des Taktsignals gewartet. Dadurch wird ein neuer Zustand erzeugt. Ein neues Zustandssymbol wird generiert (STx) und in die Liste der Zustandssymbole aufgenommen. "x" ist die nächste ganze Zahl in der Numerierung der Zustandssymbole.

Algorithmus:
A "Wait .." for the clock event only is detected.
The polarity of the clock does not matter.
1. Generate the next state symbol STx, write it into the enumeration field of the state_type.
2. Insert two new lines:
next_state_y <= STx;
WHEN STx =>
(y is the number of the process.)

Beispiel:

Eingangs-Datei:	Ausgangs-Datei
instruction_1;	instruction_1;
WAIT UNTIL clk'event and clk = '1';	next_state <=ST1;
instruction_2;	WHEN ST1 =>
	instruction_2;

4.3.2 UND-Verknüpfung eines "Wait" mit einer Signal-Bedingung

Der Parser findet die UND-Verknüpfung eines "Wait" mit einer Signal-Bedingung z.B.:

```
instruction_1;
WAIT UNTIL  clk'EVENT and clk = 'z' and <signal> = 'y';
instruction_2;
...
-- 'z', 'y' kann '0' oder '1' sein.
```

Es wird eine Warteschleife konstruiert, die solange im Zustand STx bleibt, bis die Bedingung <signal> = y; erfüllt ist. Danach wird die Instruktion 2 übernommen und das Folgezustandssymbol (Stx+1) der Zustandsvariablen "next_state" zugewiesen.

Algorithmus:
The following kind of instruction is detected:
"WAIT UNTIL clk'EVENT and clk = 'z' and <signal> = 'y';"
Insert the following lines:

```
WHEN STx =>
  if (<signal> = y) then
   instruction_2;
   ...
   next_state <= STx+1;
  else
   next_state <= STx;
  end if;
```

STx is the next state in the list of enumerated state-types

Beispiel:

Eingangs-Datei:	Ausgangs-Datei
	instruction_1;
instruction_1;	next_state <= ST1;
WAIT UNTIL clk_'event and clk= '1' and Hd_rdy ='1';	WHEN ST1 =>
instruction_2; ;	IF (Hd_rdy = '1') THEN
...	instruction_2;
	...
	next_state <= ST2;
	ELSE
	next_state <= ST1;

4.3.3 Die gerollte "FOR"-Schleife

Der Parser findet eine "FOR"-Schleife: z.B.:

```
  instruction_1;
load_loop2: For k in a TO b loop
-- or:   For k in a DOWNTO b loop
  instruction_2;
  Wait ...
  instruction_3;
  Wait ...
  instruction_4;
  Wait ...
  instruction_5;
  Wait ...
end loop;
```

Die Schleife wird dann "gerollt", d.h. die einzelnen Schleifendurchläufe werden sequentiell ausgeführt, wenn innerhalb der Schleife mindestens ein "Wait.." steht. Die Schleife kann "aufwärtszählend" (for k in a to b loop) oder "abwärtszählend" (for k in a downto b loop) sein. Das Zurücksetzen des Schleifenzählers sollte direkt for Beginn der Schleife erfolgen und nicht in der Schleife selbst. (Es kann sonst Probleme bei der Simulation geben.)

Der Algorithmus für eine For-Schleife lautet:

Algorithmus:
A "FOR" loop with at least one inserted "Wait ..." is detected.
"count" is the name of the loop counter variable.
b_value is the target loop count.
Reset the loop counter: count := a;
Start the loop with an initial loop state ST_c1,
c is the next character in the alphabet not used
to number Sub-Programm-Blocks yet.
Build the Sub-FSM.
IF *"end loop" encoutered,*

IF *count*
(< if upcounting or > if downcounting) b_value
set next_state <= ST_c1
Increment if upcounting, decrement
if downcounting the loop counter "count"
ELSE
Reset the loop counter: count := a;
next_state <= STy;
STy is the next state after the loop.
FI;
FI;

Ein Schleifenzähler (loop counter): <variable_name> vom Typ "countertype" wird deklariert. Der Typ "integer" muß dann genommen werden, wenn die Laufvariable als Vektor-Index innerhalb der Schleife auftritt.

```
TYPE countertype  IS RANGE a TO b;
VARIABLE <counter_name>   :  countertype;
```

Im Reset-Teil wird <counter_name> := a; gesetzt. Am Ende der Schleife, (vor "end loop;") wird nachgeprüft,

1. ob die Schleife fortgesetzt wird (Schleifenzähler unter b-Wert bei Aufwärszählung und über b-Wert bei Abwärtszählung),

2. wenn ja, wird der Schleifenzähler Taktwert inkrementiert oder dekrementiert. Die "next_state"-Variable wird zusätzlich auf den Anfangszustand der Schleife gesetzt.

3. Falls der b-Wert des Schleifenzählers erreicht ist, wird die Zustandsvariable "next_state" wird auf den nächsten Zustand nach der Schleife gesetzt. Zusätzlich wird der Schleifenzähler zurückgesetzt: <counter_name> := a;

Der Schleifenbeginn erhält einen neuen Zustand, der nicht durch ein "Wait" verursacht wurde. Diese "Non-wait-states" können Ziel von Zustandsreduktionen sein, wenn die Ausführungszeit des Prozesses zeitkritisch ist. (Siehe Kapitel Zustandszusammenfassungen.)

Beispiel:

Eingangs-Datei:	Ausgangs-Datei
instruction_1;	instruction_1;
load_loop2: For k in 1 TO 12 loop	next_state <= ST_A1;
instruction_2;	WHEN ST_A1 =>
“Wait...”	instruction_2;
instruction_3;	next_state <= ST_A2;
“Wait...”	WHEN ST_A2 =>
instruction_4;	instruction_3;
“Wait...”	next_state <= ST_A3;
instruction_5;	WHEN ST_A3 =>
“Wait...”	instruction_4;
end loop;	next_state <= ST_A4;
	WHEN ST_A4 =>
	instruction_5;
	IF count < 12 THEN
	count := count+1;
	next_state <= ST_A1;
	ELSE
	count := 1;
	next_state <= ST_A5;
	END IF;

Die graphisch dargestellte FSM des Beispiels zeigt Abbildung 1.

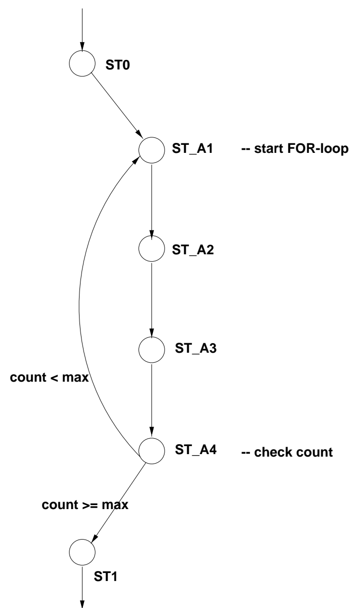

Abbildung 1: Die FSM einer gerollten FOR-Schleife

4.3.4 Die aufgerollte “FOR”-Schleife

Falls der Parser innerhalb einer Schleife kein “Wait” findet, so wird in der Regel als Vorgabe eine “aufgerollte” Schleife vom Logik-Synthesesystem erzeugt. Das bedeutet, die einzelnen Schleifendurchläufe werden als parallele Hardware aufgebaut.

In diesem Fall übernimmt der Parser die gesamte For-Schleife in einen Zustand.

4.3.5 Die “WHILE”-Schleife

Der Parser findet eine “WHILE”-Schleife: z.B.:

```
  instruction_1;
while <condition> loop
  instruction_2;
  Wait ...
  instruction_3;
  Wait ...
  ...
end loop;
```

Für die While-Schleife wird ein “Unter-Zweig” der FSM aufgebaut. D.h. die Schleife erhält eine “eigene” Menge von Zustandssymbolen, z.B. ST_A1, ST_A2, ... Die Schleife wird mit einem “Initial-Zustand” z.B.: ST_A1 begonnen.

Die Bedingung <condition> wird zu Beginn der Schleife abgefragt, ist sie wahr, wird der nächste Zustand der “neuen” Zustandsmenge (ST_A2) der “next_state” Variable zugewiesen, sonst wird der nächste Zustand der “alten” Zustandsmenge (STx+1) der “next_state” Variable zugewiesen. Abbildung 2 veranschaulicht die entstandene FSM.

Wie bei der “For-Schleife” erhält auch bei der “While”-Schleife der Schleifenbeginn einen neuen Zustand, der nicht durch ein “Wait” verursacht wurde. Diese “Non-wait-states” können Ziel von Zustandsreduktionen sein, wenn die Ausführungszeit des Prozesses zeitkritisch ist. (Siehe Kapitel Zustandszusammenfassungen.)

Algorithmus:
State STx is the actual state when encoutering the following While-loop:
“While <condition> loop ... end loop;”
Start a sub-branch of the FSM by defining a subset of state symbols
ST_c1, ST_c2, ...,
where c stands for a higher charater in the alphabet
than the one that might be used in the current state variable.
Start the loop with the “initial loop state ST_c1”
IF *<condition> = true*
  next_state <= ST_c2;
ELSE
  next_state <= STx+1;

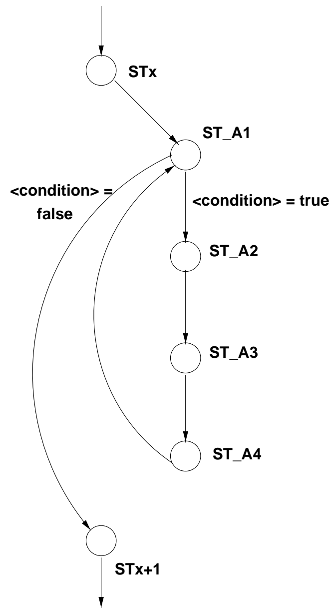

Abbildung 2: FSM einer While-Schleife

FI;
continue describing the states in the loop (ST_c2, ST_c3,..)
IF *"end loop;" encountered,*
set next_state <= ST_c1;
FI;

Beispiel:

Eingangs-Datei:	Ausgangs-Datei
instruction_1;	instruction_1;
while multicast = ’1’ loop	next_state <=ST_A1;
instruction_2;	WHEN ST_A1 =>
“Wait...”	IF multicast = ’1’ THEN
instruction_3;	instruction_2;
End loop;	next_state <= ST_A2;
	ELSE
	next_state <=ST2;
	END IF;
	WHEN ST_A2 =>
	instruction_3;
	next_state <= ST_A1;

4.3.6 “IF ... THEN ... ELSE”

Der Parser findet ein “IF ... THEN ... ELSE .. END IF;” oder ein “IF ... THEN ... END IF;” z.B.:

```
if <condition> then
  instruction 1
  Wait ...
  instruction 2
  Wait ...
else
  instruction 3
  Wait ...
  instruction 4
  Wait ...
  ...
end if;
```

“if <condition> then .. else”

Der Parser findet im Zustand STx ein “if .. then .. else”-Konstrukt. Für den “if ..” und für den “else ..”- Konstruktblock wird je ein “Unter-Zweig” der FSM aufgebaut. D.h. jeder Konstruktblock erhält eine “eigene” Menge von Zustandssymbolen, z.B. ST_B1, ST_B2, ...; für den “if condition = true ” -Block und ST_C1, ST_C2, ... für den “else”-Block. Die Bedingung <condition> wird im Zustand STx abgefragt. Ist sie wahr, wird der erste Zustand der “neuen” Zustandsmenge (ST_B1) der “next_state” Variable zugewiesen, sonst wird der erste Zustand der zweiten “neuen” Zustandsmenge (ST_C1) der “next_state” Variable zugewiesen. Der FSM-Zweig des “if condition = true” -Zweigs wird solange aufgebaut, bis der Parser das “ELSE” findet. Als Folgezustand wird der Zustand STx+1 genommen. Danach wird dasselbe-Verfahren für den ELSE-Zweig mit der zweiten “neuen” Zustandsmenge (ST_C1, ST_C2, ...) weitergeführt, bis das “END IF” gefunden wird. Als Folgezustand wird auch hier der Zustand STx+1 eingesetzt. Abbildung 3 zeigt die graphische Darstellung der FSM eines IF-THEN-ELSE Konstrukts.

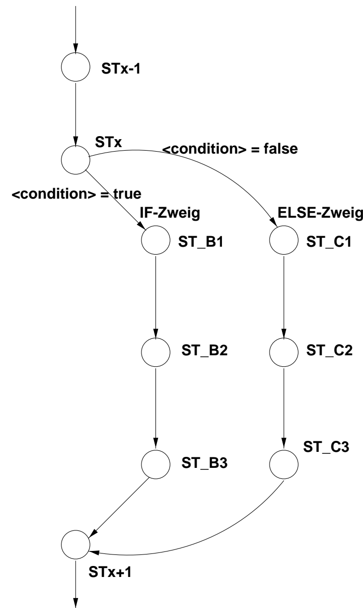

Abbildung 3: Die FSM eines IF-THEN-ELSE Konstrukts

Algorithmus für das "if .. then .. else" Konstrukt:
State STx is the actual state when encoutering the if-then-else construct.
Start a sub-branch of the FSM by defining a subset ST_c1, ST_c2, ...
of state variables, where c stands for the next charater in the alphabet
not used for sub-FSM's yet.
IF *<condition> = true,*
next_state <= ST_c1;
else next_state <= ST_d1;
where d stands for a charater in the alphabet different from c.
FI;
continue generating the states in the if .. then- block
(ST_c2, ST_c3,..) until "ELSE" is encountered,
set next_state <= STx+1;
continue generating the states in the else- block
(ST_d2, ST_d3,..) until "end if;" encountered,
set next_state <= STx+1;

"if <condition> then .. "

Der Parser findet im Zustand STx ein "if .. then .. "-Konstrukt. Es wird ein "Unter-Zweig" der FSM aufgebaut, bis das "END IF" gefunden wird. Der nächsten Zustand nach "END IF" ist STx+1.

Algorithmus für das "if .. then" Konstrukt:
State STx is the actual state when encoutering the

if-construct.
Start a sub-branch of the FSM by defining
a subset ST_c1, ST_c2, ...
of state variables, where
c stands for the next charater in the alphabet
not used for sub-FSM's yet.
IF *<condition> = true,*
  set next_state <= ST_c1;
    else set next_state <= STx+1;
FI;
continue describing the states in the if .. then block
(ST_c2, ST_c3,..).
IF *"end if;" encountered, set next_state <= STx+1;*
FI;

Beispiel für einen if .. then .. else block. Die "instruction_1" wird im Zustand ST1 ausgeführt:

Eingangs-Datei:	Ausgangs-Datei
	instruction_1;
instruction_1;	IF multicast = '1' THEN
if multicast = '1' then	instruction_2
instruction_2	next_state <= ST_B1;
"Wait..."	ELSE
instruction_3	instruction_4
else	next_state <=ST_C1;
instruction_4	END IF;
"Wait..."	WHEN ST_B1 =>
instruction_5	instruction_3
end if;	next_state <= ST2;
	WHEN ST_C1 =>
	instruction_5
	next_state <= ST_2;

4.3.7 Verschachtelungen

Der Parser findet innerhalb von Schleifen oder "IF ... THEN ... ELSE .. END IF;"-Blöcken weitere, geschachtelte Schleifen oder "IF"-Blöcke. Es werden rekursiv jeweils Unter-FSM's mit neuen Untermengen von Zustandssymbolen definiert.

Trifft das Umwandlungsprogramm im Laufe des Haupt-FSM-Aufbaus auf eine Schleife oder ein IF-Konstrukt, so wird das nächste Zustandssymbol STx+1 auf einen "LIFO"-Speicher (Last-in-first-out) oder Stack gelegt und die Unter-FSM mit einer neuen Zustandssymbol-Menge ST_A1, ST_A2, ... usw. entsprechend obiger Algorithmen aufgebaut. Wird wiederum eine Schleife oder ein IF-Konstrukt vor der "End"-Instruktion gefunden, so wird erneut das nächste Zustandssymbol (z.B. ST_A2) auf den Stack gelegt

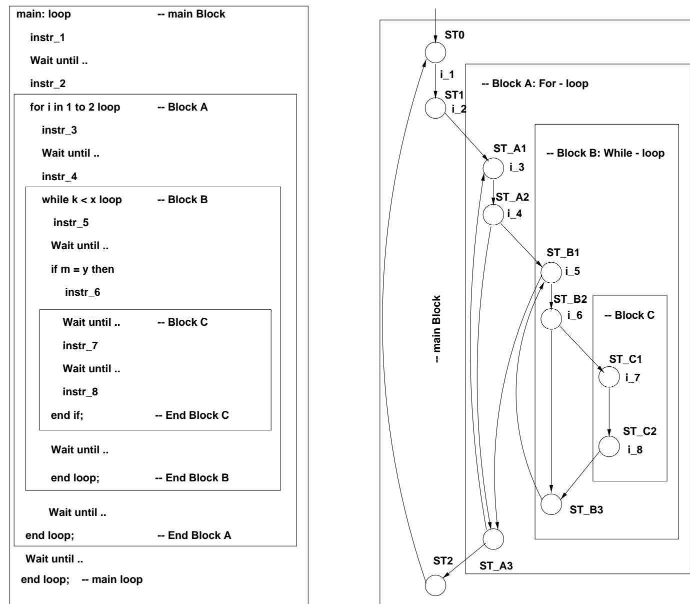

Abbildung 4: Allgemeines Beispiel für Verschachtelungen: Auf der linken Seite ist die VHDL-Beschreibung gezeigt, rechts ist die dazugehörige FSM dargestellt.

und eine neue Unter-FSM mit der Zustandssymbol-Menge ST_B1, ST_B2 aufgebaut, usw. Findet schließlich das Umwandlungsprogramm das erste END-Konstrukt (END loop; oder END IF;), so wird die laufende Sub-FSM abgeschlossen, das obenauf liegende Zustandssymbol wird vom Stack genommen und der Aufbau der FSM wird damit fortgesetzt.

Im Beispiel Abbildung 4 sind neben dem Haupt-Block noch 3 weitere Blöcke, Block A, Block B und Block C gezeigt. Der Parser findet im Haupt-Block eine "For .." -Schleife, er legt die A-Menge von Zustandssymbolen an (ST_A1,..) und beginnt die Konstruktion der Sub-FSM des A-Blocks. Danach trifft er auf eine "While .." -Schleife. Er legt die B-Menge von Zustandssymbolen an (ST_B1,..) und beginnt die Konstruktion der Sub-FSM des B-Blocks. Schließlich findet er ein "IF" -Konstrukt, legt die C-Menge von Zustandssymbolen an (ST_C1,..) und konstruiert die FSM des C-Blocks.

Die prinzipielle Übersetzung nach VHDL ohne Deklarationen, Reset Teil, etc. für das Beispiel in Abbildung 4 lautet:

```
Case state
  When ST_0 =>
    instr_1
    next_state <= ST1;
  When ST1 =>
    instr_2
    next_state <= ST_A1;

    When ST_A1 =>         -- For-Loop beginnt hier
    instr_3
    next_state <= ST_A2;

    When ST_A2 =>
    instr_4
    next_state <= ST_B1;

       When ST_B1 =>       -- WHile loop
```

```
      if k < x then instr_5; next_state <= ST_B2;
       else next_state ST_A3;  end if;

      When ST_B2 =>
      if m = y then instr_6; next_state <= ST_C1;
      else  next_state <= ST_B3; end if;

         When ST_C1 =>
         instr_7; next state <= ST_C2;

         When ST_C2 =>
         instr_8;  next state <= ST_B3;

      When ST_B3 =>
      next state <= ST_B1;

   When ST_A3 =>
   if i < 2 then
    if clk  = '0'  then i := i + 1; end if;
    next_state <= ST_A1;
   else i := 1; next_state <= ST2;  end if;

 When ST2 =>
 next_state <= ST1;

 When others =>

 end CASE;
```

Um eine bessere Übersichtlichkeit des Programms zu erhalten, werden die Beschreibungen der einzelnen Sub-FSM's eingerückt.

Im Folgenden ist der generelle Algorithmus für eine Verschachtelung beschrieben.

Algorithmus für die FSM-Generierung bei verschachtelten VHDL-Beschreibungen:

WHILE
a loop or an if-construct and not "END .." detected:
1. Save the next state symbol of the current
FSM or SUB-FSM on a stack-memory
and add it to the list of enumerated state types.
2. Start a sub-branch of the FSM or SUB-FSM
using a new subset ST_c1, ...
of state variables according to the algorithm
defined for the loop or if-construct.
(c is the next character in the alphabet
not used for sub-FSM's yet.
END WHILE
WHILE *"END ... detected:*
Take the next state symbol from the stack
and continue the current FSM.
END WHILE

4.3.8 Zustandszusammenfassung bei Verschachtelungen

Obwohl die Zustandsautomaten aus **taktgebundenen** Verhaltensbeschreibungen generiert werden, können bei der Anwendung der o.g. Algorithmen zusätzlich Zustände entstehen, nämlich die bei der For- und While-Schleife erwähnten sog. "Non-wait-states". Die Non-wait-states sind in manchen Fällen nicht zu vermeiden, in jedem Fall verzögern sie die Ausführungszeit des Systems.

Bei der Schachtelung von Schleifen und IF-Konstrukten, wenn die Schachtelungsgrenzen am Anfang oder am Ende von Funktions-Blöcken zusammenfallen, ist es möglich, Non-wait-states zusammenzufassen und damit die Ausführungszeit des Systems zu reduzieren.

Im Folgenden sind einige Verfahren mit Beispielen aufgezeigt, um unnötige Zustände durch Zusammenfassen zu reduzieren.

Zustandszusammenfassung am Ende von Unter-FSM's

Am Ende von Verschachtelungen werden Zustände bei Anwendung obiger Algorithmen automatisch zusammengefaßt, wenn die End-Instruktionen der verschachtelten Blöcke direkt, ohne Unterbrechung durch ein "Wait ...", aufeinanderfolgen.

Diese Vorschrift wird rekursiv angewendet.

Das Beispiel Abbildung 5 ist abgeleitet von Abbildung 4 mit dem Unterschied, daß die Block-Enden von Block B und von Block C direkt aufeinderfolgen.

In diesem Fall werden die Zustände ST_B2 und ST_B3 zusammengefaßt, Zustand ST_B3 fällt weg.

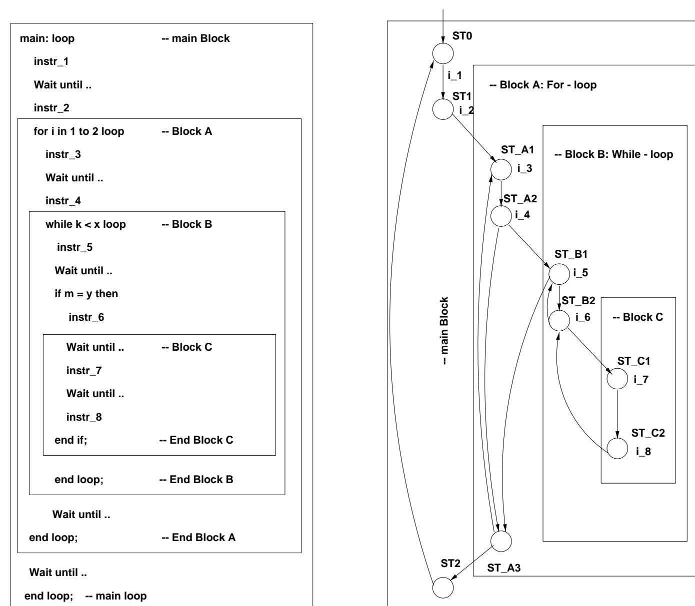

Abbildung 5: Beispiel für Verschachtelungen mit Zustandszusammenfassung: Auf der linken Seite ist die VHDL-Beschreibung gezeigt, rechts ist die FSM mit Zustandszusammenfassung dargestellt.

Algorithmus für eine Verschachtelung mit Zustandszusammenfassung am Ende von Unter-FSM's:

WHILE

a loop or an if-construct and not "END .." detected:

1. Save the next state symbol of the current

FSM or SUB-FSM on a stack-memory
and add it to the list of enumerated state types.
2. Start a sub-branch of the FSM or SUB-FSM
according to the algorithm defined for the
loop or if-construct found,
using a new subset ST_c1, ... of state
variables. (c is the next character
in the alphabet not used
for numeration of Sub-FSM's yet.)
END WHILE
WHILE *"END ... detected:*
IF *a second "END .." detected,*
Take the next state symbol from the stack
and discard it.
ELSE
Take the next state symbol from the stack
and continue the current FSM with that state symbol.
END WHILE

Zustandszusammenfassung am Anfang von Unter-FSM's

Am Anfang von Verschachtelungen können Zustände bei Anwendung obiger Algorithmen dann zusammengefaßt werden, wenn die End-Instruktionen der verschachtelten Blöcke zusammengefaßt werden und die Anfänge der Blöcke direkt, ohne Unterbrechung durch ein "Wait ...", aufeinanderfolgen.

Diese Vorschrift kann rekursiv angewendet werden.

Das Beispiel Abbildung 6 ist abgeleitet von Abbildung 5 mit folgenden Unterschieden:

1. die Block-Anfänge von Block B und von Block C folgen direkt aufeinder.

2. Der Block D wurde hinzugefügt um eine "Deadlock"-Situation zu vermeiden, die ohne Block D dann auftreten würde, wenn die Bedingung "m = y" nicht zutrifft.

In der Abbildung 6 können die Zustände B2 und B1 zusammengefaßt werden. Der Zustand B2 fällt weg und der Block C geht in den Block B über. Der Block D wird zum Block C. Die Abbildung 7 zeigt die entsprechende Blockaufteilung und die reduzierte FSM.

Die prinzipielle Übersetzung nach VHDL ohne Deklarationen, Reset Teil, etc. für das Beispiel in Abbildung 7 lautet:

```
Case state
  When ST_0 =>
    instr_1
    next_state <= ST1;
  When ST1 =>
    instr_2
    next_state <= ST_A1;  --

    When ST_A1 =>         -- i wird am Ende abgefragt
    instr_3
    next_state <= ST_A2;    -- hier sparen

    When ST_A2 =>
    instr_4
    next_state <= ST_B1;
```

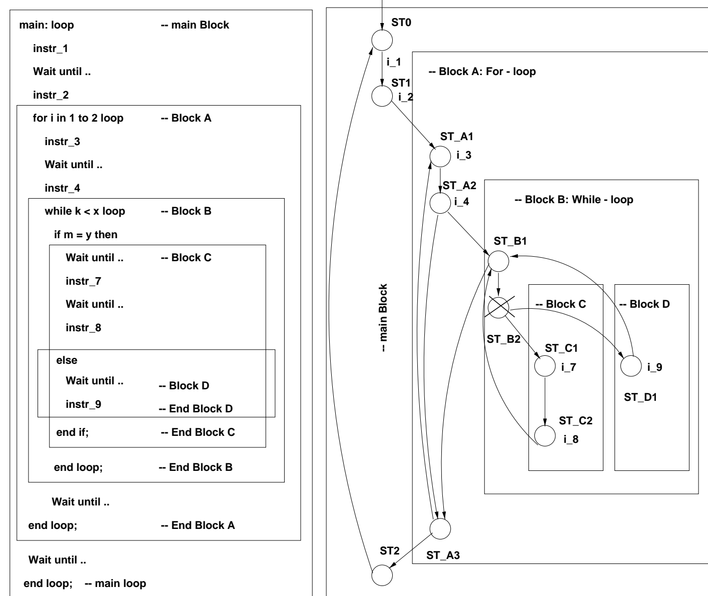

Abbildung 6: Beispiel für Verschachtelungen mit Zustandszusammenfassung am Anfang von Unter-FSM's: Auf der linken Seite ist die VHDL-Beschreibung gezeigt, rechts ist die FSM dargestellt. Der Zustand ST_B2 kann gestrichen werden.

```
      When ST_B1 =>       -- While loop
      if k < x then
         if m = y then instr_6
          next_state <= ST_B2;
          else next_state <= ST_C1; end if;
       else next_state ST_A3;  end if;

         When ST_C1 =>       -- else block
         instr_9; next_state <= ST_B1;

      When ST_B2 =>
      instr_7;  next state <= ST_B3;

      When ST_B3 =>
      instr_8;  next state <= ST_B1;

   When ST_A3 =>
   if i < 2 then
    if clk  = '0'  then i := i + 1; end if;
    next_state <= ST_A1;
   else i := 1; next_state <= ST2;  end if;

 When ST2 =>
 next_state <= ST1;

 When others =>

 end CASE;
```

Algorithmus für eine Verschachtelung mit Zustandszusammenfassung am Anfang und am Ende von Unter-FSM's:

WHILE
a loop or an if-construct and not "END .." detected:
IF *a further direct consecutive loop or*
if-construct detected,
IF *for the detected loops or*

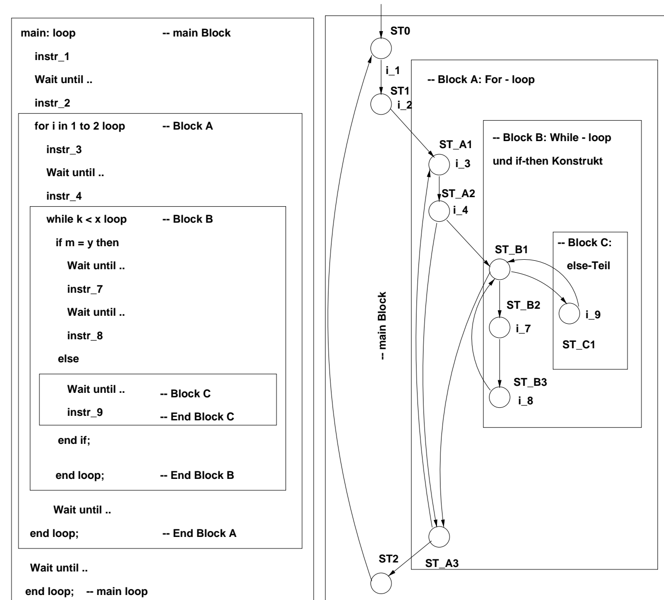

Abbildung 7: Beispiel für Verschachtelungen mit Zustandszusammenfassung am Anfang von Unter-FSM's: Auf der linken Seite ist die VHDL-Beschreibung gezeigt, rechts ist die FSM dargestellt. Der Block D geht in den Block C über.

if-constructs the "END's .." follow
consecutively then
start constructing the FSM according to
the algorithm for the loops or if-constructs
using only one sub-FSM.
FI
ELSE
construct the FSM's according to the algorithm
for nested VHDL-descriptions.
FI
END WHILE
WHILE *"END ... detected:*
IF *a second "END .." detected,*
Take the next state symbol from the stack
and discard it.
ELSE
Take the next state symbol from the stack
and continue the current FSM with that state symbol.
END WHILE

Ein Beispiel aus der Praxis

Folgendes Beispiel ist der Modulsammlung für die "ATM-Switch-Steuerung" ent-

nommen. Es ist der “Multicast-Prozess” (Prozeß Nr. 4) der Verbindungssteuerung CC (Connection Controller). Das gesamte Modul ist im Anhang dargestellt. Das Beispiel zeigt drei Verschachtelungen und jeweils zwei Zustandszusammenfassungen am Anfang und am Ende von Unter-FSM’s.

Die Verhaltensbeschreibung ohne Deklarationen und Reset-Teil:

```
main_loop: loop

  L1: FOR i in 1 to 13 loop    -- 13 Kanaele-loop
    wait until Clk_com'event and Clk_com = '1' and unicast_lock  = '0';
    inp_addr := std_logic_vector(conv_unsigned(i, 4)); -- library
    fct..

    IF active(i)  = '1' THEN
      multicast_lock  <= '1';
      While multicast(i) = '1' loop
        IF active(i)  = '1' then    -- wird wiederholt, es muss sein

         -- OA(i) wird immer neu gestezt von RC
         out_addr    := OA(i);         -- conv to std_logic_vector
         OAd  := natural(CONV_INTEGER(out_addr));

         wait until Clk_com'event and Clk_com = '1' and belegt_reg(OAd) = '0';
         inpaddr_sig  <= inp_addr;
         outaddr_sig  <= out_addr;
         set_load  <= '1';  -- an switch process;

         wait until Clk_com'event and Clk_com = '1' and set_load_ok = '1';
         set_load  <= '0';  -- an switch process;
         act_ok(i) <= '1';

         wait until Clk_com'event and Clk_com = '1' and active(i)  = '0';
         act_ok(i) <= '0';
         ELSE
          wait until Clk_com'event and Clk_com = '1';
        END IF; -- if active(i) = '1';
      End loop; -- while loop
    multicast_lock <= '0';
    set_conf  <= '1';  -- an switch process;
    wait until Clk_com'event and Clk_com = '1' and setconf_ok  = '1';
    set_conf <= '0';

    start_xmit(i) <= '1';
    wait until Clk_com'event and Clk_com = '1' and xmit_bsy(i) = '1';
    start_xmit(i) <= '0';

    END IF;  -- if active(i) = '1';
  END loop;  -- 1 to 13 loop

End loop; -- main loop
```

Die Ausgangsdatei ohne Deklarationen und Reset-Teil:

```
--  main_loop: loop
CASE state_4 IS

 WHEN ST0 =>
  if  broadcast_lock  = '0' then
     next_state_4 <= ST_A1;
    else
     next_state_4 <= ST0;
  end if;

    WHEN ST_A1 =>
   --      L1: FOR i in 1 to 13 loop  ST_A gehoert zur For-loop
     IF active(i)  = '1' THEN
       multicast_lock  <= '1';
       inp_addr := std_logic_vector(conv_unsigned(i, 4)); -- library fct..
       next_state_4 <= ST_B1;
     else
        next_state_4 <= ST_A2;
     end if;

       WHEN ST_B1 =>
         IF multicast(i) = '1' then        --  While multicast(i) = '1' loop
           IF active(i)  = '1' THEN        --  If active-loop
             next_state_4 <= ST_C1;
           end if;
         else
           next_state_4 <= ST_B2;   -- check!
         end if;
```

```
          WHEN ST_C1 =>      -- While loop & act(i)-loop
            out_addr     := OA(i);
            OAd  := natural(CONV_INTEGER(out_addr));
            if belegt_reg(OAd)  = '0' then
              -- OA(i) wird immer neu gestezt von RC
              inpaddr_sig  <= inp_addr;
              outaddr_sig  <= out_addr;
              set_load  <= '1';  -- an switch process;
              next_state_4 <= ST_C2;
            else
              next_state_4 <= ST_C1;
            end if;

          WHEN ST_C2 =>
            if set_load_ok  = '1' then
              set_load  <= '0';  -- an switch process;
              act_ok(i)    <= '1';
              next_state_4 <= ST_C3;
            else
              next_state_4 <= ST_C2;
            end if;

          WHEN ST_C3 =>
            if active(i)  = '0' then
              act_ok(i) <= '0';
              next_state_4 <= ST_B1;
            else
              next_state_4 <= ST_C3;
            end if;                -- end if active(i) = '1';

        WHEN ST_B2 =>
          set_conf  <= '1';  -- an switch process, set configure
          next_state_4 <= ST_B3;

        WHEN ST_B3 =>
           if setconf_ok  = '1'  then
              multicast_lock <= '0';
              set_conf <= '0';
              start_xmit(i) <= '1';
              next_state_4 <= ST_B4;
            else
              next_state_4 <= ST_B3;
            end if;

        WHEN ST_B4 =>
          if xmit_bsy(i) = '1'  then
            start_xmit(i) <= '0';
            next_state_4 <= ST_A2;
          else
            next_state_4 <= ST_B4;
          end if;

     WHEN ST_A2 =>
       if i < 13  then
          then i := i + 1;
       else
         i := 1;
       end if;
         next_state_4 <= ST0;

         when others =>
   end CASE;
```

Bild 8 zeigt die graphische Darstellung der FSM für obiges Beispiel.

Folgende Zustandszusammenfassungen werden durchgeführt:

1. Für die For-Schleife und das 1. IF-Konstrukt.
2. Für die While Schleife und das 2. IF-Konstrukt.

5 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wird die **Generierung von Zustandsautomaten (FSM's)** aus VHDL-Verhaltensbeschreibungen für taktgebundene Schaltungen detailliert beschrieben. Die beschriebenen Algorithmen sind auf eine VHDL-Mindestmenge beschränkt, die ausreicht, um umfangreiche und komplizierte Verhaltensbeschreibungungen zu übersetzen.

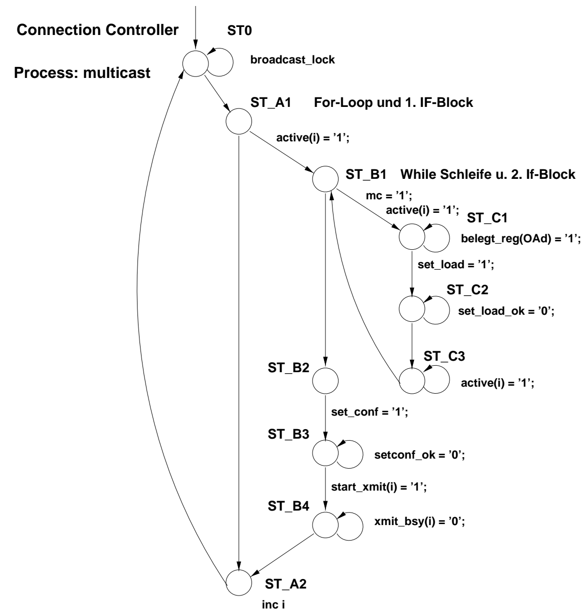

Abbildung 8: Beispiel der FSM des CC-Multicast Prozesses.

Der ganze synthetisierbare VHDL-Sprachumfang konnte nicht berücksichtigt werden, es würde den Umfang dieser Arbeit um ein Vielfaches übertreffen.

Obwohl die Zustandsautomaten aus taktgebundenen VHDL-Verhaltensbeschreibungen generiert werden, können bei der Anwendung der gezeigten Algorithmen zusätzlich Zustände entstehen, die zwar keine Fehler erzeugen, aber die dennoch die Ausführungszeit des Systems unnötig verzögern. Es werden einige Verfahren aufgezeigt, die in solchen Fällen eine Zustandsreduzierung durch Zusammenfassen von Zuständen ermöglichen.

Taktgebundene Verhaltensbeschreibungen werden bei Datenübertragungs-Systemen häufig erstellt und benötigen keine High-Level-Synthese wenn das vorliegende FSM--Generierungsverfahren angewandt wird. Das Generierungsprogramm könnte jedoch in ein High-Level-Synthese- System integriert werden und dann zum Tragen kommen, wenn ein Benutzer eine HW-Beschreibung im“cycle fixed I/O Modus” synthetisieren möchte.

Das Verfahren ist schnell und das Ergebnis der Umwandlung, eine VHDL-Verhaltensbeschreibung auf RT-Ebene, wird mit kommerziell verfügbaren Logik-Synthese-Systemen, z.B. mit dem DC von Synopsys^{TM}, sehr effektiv weiter synthetisiert.

Dieses Verfahren kann auch für andere HW-Beschreibungssprachen z. B. für “HW-C” oder “e” eingesetzt werden, wenn die oben beschriebenen Algorithmen in den entsprechender Parser eingesetzt werden.

6 Literatur

Literatur

[LaRo97] W. Lange, W. Rosenstiel, Modellierung einer ATM-Switch-Steuerung. WSI 97-6 Wilhelm-Schickard-Institut Universität Tübingen (Bericht) WSI 1997 ISSN 0964-3852

[DeMi94] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill (1994)

[Gaj92] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, Steve Y-L Lin, High-Level Synthesis Introduction to Chip and System Design, Kluwer Academic Publishers (1992)

[Gutberl97] P. Gutberlet, Spezifikation und Synthese aus taktfreien Verhaltensbeschreibungen. Dissertation der Fakultät für Informatik der Universität Tübingen. (1997)

[HanBring98] C. Hansen, O. Bringmann, Spezifikation von VHDL-Verhaltensbeschreibungen für das High-Level-Synthesesystem CADDY-II. FZI-Bericht 5-13-10/98

[SynDCTut99] Synopsys Design Compiler Tutorial. "TIME_STATE_MACHINE.vhd" Version 1999.10

[SynPC99] Synopsys Protocol Compiler User Guide. Version 1999.10

7 Anhang: Beispiele für die Generierung von FSM's aus Verhaltensbeschreibungen

Die Eingangsbeschreibungen sind Verhaltensbeschreibungen für eine ATM-Switch-Steuerung (siehe [LaRo97]).

Die Ausgangsbeschreibungen sind "von Hand" entwickelt worden und können deshalb möglicherweise von Dateien, die mit den vorn beschriebenen Algorithmen erstellt wurden leicht abweichen.

7.1 Eingangsmodul AHT_IN

7.1.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright 2000 University of Tuebingen
-- Wilhelm Schickard Institute for Computer Sciences
-- Dept. Computer Engineering, Prof. Dr. W. Rosenstiel
-- Autor:  Walter Lange, September 2000
--
-- Reine Verhaltensbeschreibung des AHT_IN
-- ahtin.vhd
-- AHT_In hat 2 Prozesse: einen fuer die Aufnahme und asynchrone Weiterleitung
-- des Zell-Headers, der zweite Prozess nimmt die Payload auf,  und die
-- asynchrone Weitergabe des Payload Registers an den Cell-Fifo.
-- Version mit Synchronisation fuer Cell_Sync_In (start_loop)
-- Aenderungen fuer Synopsys:
-- design_reset eingefuehrt
-- Attribute fuer "dont_unroll" of load_loops
-- Das handshaking zum Payload FIFO geht hier so nicht. Synopsys BC bringt
-- Fehler. Man sollte synchron
-- zum Payload FIFO uebertragen. (Siehe Payload-Prozess)
-- Autor: W. Lange September 2000


Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY aht_in IS
  PORT (Data_In     : IN UNSIGNED(7 DOWNTO 0);
       Clk_AHT_In   : IN STD_LOGIC;
       Cell_Sync_In : IN STD_LOGIC;
       Reset        : IN STD_LOGIC;   -- fuer Synopsys eingefuehrt
       Hd_fetch     : IN STD_LOGIC;
       Cell_fetch   : IN STD_LOGIC;
       Buffer_full  : IN STD_LOGIC;
       Hd_Bus       : OUT UNSIGNED(31 DOWNTO 0);
       Cell_Bus     : OUT UNSIGNED(31 DOWNTO 0);
       Hd_rdy       : OUT STD_LOGIC;
       Cell_rdy     : OUT STD_LOGIC);
END aht_in;
---------------------------------

ARCHITECTURE ahtlar OF aht_in IS
 SIGNAL Start_Payl:  STD_LOGIC;

BEGIN
  -- Der AHTIN Process nimmt den Header auf und leitet ihn asynchron
  -- an HT weiter.

  AHTIN: PROCESS
   VARIABLE Hd_reg     : UNSIGNED(31 downto 0);  -- Reg fuer Header B.

  BEGIN -- Process

  --  Reset: Setze alle Variablen und Indices zurueck ----------------------
    Hd_rdy <= '0';
    Start_Payl <= '0';   -- fuer BC zurueckgenommen
    Hd_reg := "00000000000000000000000000000000";
    Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';

  -- Cell_Sync_In erscheint, eine neue Zelle kommt an...

  main_loop: loop   -- main loop

    Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1' and Buffer_full ='0';
```

```
    Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1' and (Cell_Sync_In = '1');
        Hd_reg(31 downto 24) := Data_In;

        Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
    -- Cell_Sync ist da, lade den Header..     --------------------------
Hd_reg(23 downto 16) := Data_In;
Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
Hd_reg(15 downto 8)  := Data_In;
Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
--        Start_Payl <= '1';   -- 2  Kontrollschritte vorverlegt fuer BC
Hd_reg(7 downto 0)   := Data_In;
Hd_Bus <= Hd_reg;    -- Setze das Hdreg. auf den Bus und
        Hd_rdy <= '1';       -- Vergiss den HEC...
Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
        -- Das HEC Byte bleibt unberuecksichtigt.
Start_Payl <= '1';    -- fuer BC
Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1' and Hd_fetch = '1';
Hd_rdy <= '0';
        Start_Payl <= '0';
Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
    End loop; -- Main Loop
  End Process;

  Payload: PROCESS
  VARIABLE Cell_reg    : UNSIGNED(31 downto 0);  -- Reg fuer Cell Bytes

  -- Der Payload Process nimmt den Header auf und leitet ihn asynchron
  -- an den Cell-Fifo weiter.

  BEGIN -- Process
    --  Reset: Setze alle Variablen und Indices zurueck -------------
      Cell_rdy  <= '0';
      Cell_reg := "00000000000000000000000000000000";
    Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
    -- Die Payload wird geladen

  main_loop2: loop               -- main loop

    Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1' and Start_Payl = '1';

    load_loop2: For k in 1 TO 12 loop
     --  4 Byte der Zelle werden geladen..
  Cell_reg(31 downto 24) := Data_In;
          Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
Cell_reg(23 downto 16) := Data_In;
 Cell_rdy  <= '0';
          Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
Cell_reg(15 downto 8)  := Data_In;
          Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
Cell_reg(7 downto 0)   := Data_In;
            Cell_Bus <= Cell_reg;   -- Lade 4 Bytes
            Cell_rdy <= '1';
          Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
    End loop;   -- loop2: Payload loop   -------------------------
            Cell_rdy <= '0';
    Wait UNTIL Clk_AHT_In'EVENT AND Clk_AHT_In = '1'; -- for BC
    END loop; -- main loop
  END Process;
 END ahtlar;
```

7.1.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 2000 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, June 2000
--
-- Verhaltensbeschreibung des AHT_In fuer
-- Synthese mit Synopsys - DC
-- ahtin_rtl.vhd
-- Diese Beschreibung ist als FSMD ausgelegt.
-- Es gibt  einen RTL-Prozess und einen Synch-Prozess,
-- um mit dem Takt zu synchronisieren.
-- Die Prozesse haben Sensitivity-Lists,
-- Das bedeutet, das auch bei negativer Clock-Flanke
-- der Prozess durchlaufen wird.
-- (Kann Probleme beim Zaehlen geben!)
-- Daher: Nur bei Clock down zaehlen!
-- Es werden zwei Register verwendet und ein Zaehler,
-- der die Payload Worte bis 6 (x8) hochzaehlt.
-- Aenderungen fuer Synopsys:
-- Reset wird eingefuehrt
-- Das handshaking zum Payload FIFO geht hier so nicht. Synopsys BC bringt
-- Fehler. Man sollte synchron
-- zum Payload FIFO uebertragen. (Siehe Payload-Prozess)
-- Author: W. Lange, Last update Sept. 2000
```

```
-- Architecture - name zu ``rtl'' ge"andert.
-- Sync - Signal eingefuehrt: Sept. 2000
-- Dadurch wird man von der negativen clock-Flanke unabhaengig.
-- Sensitvity list geaendert: nur sync signal und reset hineingenommen


Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY aht_in IS
  PORT (Data_In     : IN UNSIGNED(7 DOWNTO 0);
       Clk_AHT_In   : IN STD_LOGIC;
       Cell_Sync_In : IN STD_LOGIC;
       Reset        : IN STD_LOGIC;   -- fuer Synopsys eingefuehrt
       Hd_fetch     : IN STD_LOGIC;
       Cell_fetch   : IN STD_LOGIC;
       Buffer_full  : IN STD_LOGIC;
       Hd_Bus       : OUT UNSIGNED(31 DOWNTO 0);
       Cell_Bus     : OUT UNSIGNED(31 DOWNTO 0);
       Hd_rdy       : OUT STD_LOGIC;
       Cell_rdy     : OUT STD_LOGIC);
END aht_in;
---------------------------------

ARCHITECTURE rtl OF aht_in IS
  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4,ST5,ST6,
                        ST_A1,ST_A2,ST_A3,ST_A4
                        );
  SIGNAL state_0, state_1, next_state_0, next_state_1 : state_type;
  SIGNAL start_payld : STD_LOGIC;
  signal sync_sig : STD_LOGIC;

BEGIN
  -- Der AHTIN Process nimmt den Header auf und leitet ihn asynchron
  -- an HT weiter. Er ist hier als RTL-Prozess beschrieben.

  AHTIN: PROCESS(sync_sig, Reset)

   VARIABLE Hd_reg     : UNSIGNED(31 downto 0);  -- Reg fuer Header B.


  BEGIN -- Process ahtin

  --  Reset: Setze alle Variablen und Indices zurueck ----------------------

    IF (Reset = '1') THEN
Hd_rdy <= '0';
next_state_0 <= ST0;
Hd_reg   := "00000000000000000000000000000000";
        start_payld <= '0';
        next_state_0 <= ST0;

     ELSE

     CASE state_0 IS
        WHEN ST0 =>                 -- Warte bis Buffer frei ist
  IF (Buffer_full ='0') THEN
    next_state_0 <= ST1;
  ELSE next_state_0 <= ST0;
  END IF;
WHEN ST1 =>
  IF (Cell_Sync_In = '1' and Buffer_full ='0') THEN
            -- Cell_Sync_In erscheint, eine neue Zelle kommt an...
    Hd_reg(31 downto 24) := Data_In;
    next_state_0 <= ST2;
  ELSE next_state_0 <= ST1;
  END IF;
WHEN ST2 =>
    Hd_reg(23 downto 16) := Data_In;
    next_state_0 <= ST3;
 --          start_payld <= '0';
WHEN ST3 =>
    Hd_reg(15 downto 8) := Data_In;
    next_state_0 <= ST4;
WHEN ST4 =>
    Hd_reg(7 downto 0) := Data_In;
      Hd_Bus <= Hd_reg;    -- Setze das Hdreg. auf den Bus und
            Hd_rdy <= '1';       -- Vergiss den HEC..
            start_payld <= '1';
      next_state_0 <= ST5;
WHEN ST5 =>
          next_state_0 <= ST6;
           -- Das HEC Byte bleibt unberuecksichtigt.
           -- Es werden keine Daten gelesen
--            start_payld <= '1';
WHEN ST6 =>
  IF (Hd_Fetch ='1') THEN
            -- Cell_Sync_In erscheint, eine neue Zelle kommt an...
    Hd_rdy <= '0';
    next_state_0  <= ST0;
            start_payld <= '0';
         ELSE
```

```
            next_state_0 <= ST6;
  END IF;
        WHEN OTHERS =>
      END Case;
     END IF;
  End Process;

  Payload:  PROCESS(sync_sig, Reset)
   TYPE countertype  IS RANGE 0 TO 12;
   VARIABLE Cell_reg   : UNSIGNED(31 downto 0);  -- Reg fuer Cell Bytes
   VARIABLE Cell_reg2  : UNSIGNED(31 downto 0);  -- Reg 2 fuer Cell Bytes
   VARIABLE count      :  countertype;

   BEGIN -- Process
  --  Reset: Setze alle Variablen und Indices zurueck ----------------------

    IF (Reset = '1') THEN
Cell_rdy <= '0';
next_state_1 <= ST0;
        count := 1;
  Cell_reg  := "00000000000000000000000000000000";
   Cell_reg2 := "00000000000000000000000000000000";
        next_state_1 <= ST0;
     ELSE

     CASE state_1 IS

WHEN ST0 =>
        -- Start state
           IF (start_payld = '1') THEN
             next_state_1 <= ST_A1;
           ELSE
             next_state_1 <= ST0;
           END IF;

WHEN ST_A1 =>
        -- Die Payload wird eingelesen
     Cell_reg(31 downto 24) := Data_In;
            next_state_1 <= ST_A2;
WHEN ST_A2 =>
   Cell_reg(23 downto 16) := Data_In;
           Cell_rdy <= '0';
           next_state_1 <= ST_A3;
WHEN ST_A3 =>
   Cell_reg(15 downto 8) := Data_In;
           next_state_1 <= ST_A4;
WHEN ST_A4 =>
   Cell_reg(7 downto 0) := Data_In;
        -- hier wird Cell_reg2 als zweites Register genommen
           Cell_reg2 := Cell_reg;
           Cell_Bus <= Cell_reg2;   -- Lade 4 Bytes
           Cell_rdy <= '1';
           IF count < 12 THEN
               count := count + 1;
               next_state_1 <= ST_A1;
           ELSE
               next_state_1 <= ST2;     -- ST5
               count := 1;
           END IF;

  WHEN ST2 =>
           -- Halte Cell_rdy  noch einen Takt lang
           next_state_1 <= ST3;

  WHEN ST3 =>
           Cell_rdy <= '0';
           next_state_1 <= ST0;

        WHEN OTHERS =>
     END Case;

     END IF;
  End Process;

  Sync_Proc: PROCESS
  BEGIN
        IF (Reset = '1') then
         state_0 <= ST0;
         state_1 <= ST0;
         sync_sig <= '0';
         Wait until Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
        ELSE
         if sync_sig = '0' then
           sync_sig <= '1';
         else
           sync_sig <= '0';
         end if;
 state_0 <= next_state_0;
         state_1 <= next_state_1;
         Wait until Clk_AHT_In'EVENT AND Clk_AHT_In = '1';
        end IF;
  END Process;
```

```
END rtl;
```

7.2 Zellkopfübersetzungsmodul HT

7.2.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, 2000
--
-- Verhaltensbeschreibung des HT (Header Translator)
-- ht_rtl/ht.vhd
-- Reset wird eingefuehrt.
-- Autor: W. Lange. September 2000

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


ENTITY ht IS
  PORT ( Clk_com     : IN  STD_LOGIC ;       -- Clock common
Hd_Bus       : IN  STD_LOGIC_Vector(31 DOWNTO 0); -- Header - 4 Bytes
Hd_rdy       : IN  STD_LOGIC;   -- AHT1 has Header for you
Hd_fetch     : OUT STD_LOGIC;   -- Header arrived in HT: Header fechted
RT_Addr      : OUT STD_LOGIC_Vector(15 downto 0);
rt_read      : OUT STD_LOGIC;
RT_data      : IN  STD_LOGIC_Vector(39 downto 0);
        rtdta_strobe : IN  STD_LOGIC;
fwrq         : OUT STD_LOGIC;                      -- Fifo write re.
  headout      : Out STD_LOGIC_Vector(47 downto 0);   -- Fifo input
        head_taken   : IN  STD_LOGIC;
ffull        : IN  STD_LOGIC;                       -- fifo ist voll
        Reset        : IN  STD_LOGIC);
END ht;
---------------------------------

ARCHITECTURE htar OF ht IS

BEGIN
 htproc : PROCESS

  VARIABLE  Address    : STD_LOGIC_Vector(0 TO 15);
  VARIABLE  Header     : STD_LOGIC_Vector(47 DOWNTO 0);  -- Header w. RTag
  VARIABLE  rtag       : STD_LOGIC_Vector(15 DOWNTO 0);  -- Hilfsvar. f.rtag
  VARIABLE  vpivci     : STD_LOGIC_Vector(23 DOWNTO 0);  -- Hilfsvar. f. vpivci
  VARIABLE  rtdata     : STD_LOGIC_Vector(39 downto 0);  -- Hilfsvar.
  VARIABLE  rdwrt      : STD_LOGIC;

 BEGIN -- Process -------------------------------------------------------

     Hd_fetch <= '0';  -- Setze die Signale auf null..
     fwrq     <= '0';
     rt_read  <= '0';
     --   Headfifo_reset;    -- Reset Header - Puffer (noch nicht eingebaut)

     WAIT UNTIL Clk_com'event and Clk_com = '1';

     ------------------------ Haupt-Loop ----------------------------------
     main_loop: LOOP

     -- Hole einen Header vom AHT1_In ab                   -----------------
     ------------------------ 1. Takt ----------------------------------
     WAIT UNTIL  Clk_com'EVENT and Clk_com = '1'and Hd_rdy = '1';

       Header(31 downto 0) := Hd_Bus;

       Hd_fetch <= '1';                 -- Sag AHT1_In: Header abgeholt!

       WAIT UNTIL Clk_com'event and Clk_com = '1' and Hd_rdy = '0';

     ------------------------ 2. Takt ----------------------------------
     --  Adressiere die Routing-Tabelle VPI_2, VCI_2
       Address := Header(27 downto 24) & Header(15 downto 4); -- Get VCI/VPI
       -- Hier kann man die Adressrechnung einsetzen Addr:=F1(Addr);
       -- Adressiere die RT und hole die neuen VPI und VCI's
       RT_Addr <= Address; -- Setze die Adresse
       rt_read <= '1';
       Hd_fetch <= '0';
     ------------------------ 3. Takt ----------------------------------
       -- Warte auf das RT-Data Ergebnis
       WAIT UNTIL Clk_com'event and Clk_com = '1' and rtdta_strobe = '1';
       rtdata := RT_data;
```

```
       rtag := rtdata(39 downto 24);      -- Hilfsvariablen wegen CADDY
       vpivci := rtdata(23 downto 0);   -- Hilfsvariablen wegen CADDY
       -- Header(31 DOWNTO 28) ist GFC, von Bit 3 bis 0: PTI und CLP
       Header(47 downto 4) := rtag & Header(31 DOWNTO 28) & vpivci;

       rt_read <= '0';

      ------------------------ 4. Takt ----------------------------------

      -- Speichere das Headerregister im Headerpuffer ab    ------------------
      WAIT UNTIL Clk_com'event and Clk_com = '1' and ffull = '0';
         fwrq <= '1';
         headout <= Header;

     ------------------------- 5. Takt -----------------------------------
     WAIT UNTIL Clk_com'event and Clk_com = '1' and head_taken = '1';
       fwrq <= '0';
       WAIT UNTIL Clk_com'event and Clk_com = '1';
   END LOOP;
 END Process;
END;           -- End architecture
```

7.2.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 2000 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, September 2000
--
-- Verhaltensbeschreibung des HT (Header Translator)
-- auf RT-Ebene mit FSM
-- ht_rtl/src/ht_rtl.vhd
-- Reset wird eingefuehrt. Sync-Signal wird eingefuehrt.
-- Autor: W. Lange. Letzte Aenderung: Sept.  2000

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


ENTITY ht IS
  PORT ( Clk_com     : IN  STD_LOGIC ;       -- Clock common
Hd_Bus       : IN  STD_LOGIC_Vector(31 DOWNTO 0); -- Header - 4 Bytes
Hd_rdy       : IN  STD_LOGIC;   -- AHT1 has Header for you
Hd_fetch     : OUT STD_LOGIC;   -- Header arrived in HT: Header fechted
RT_Addr      : OUT STD_LOGIC_Vector(15 downto 0);
rt_read      : OUT STD_LOGIC;
RT_data      : IN  STD_LOGIC_Vector(39 downto 0);
        rtdta_strobe : IN  STD_LOGIC;
fwrq         : OUT STD_LOGIC;                       -- Fifo write re.
  headout      : Out STD_LOGIC_Vector(47 downto 0);   -- Fifo input
        head_taken   : IN  STD_LOGIC;
ffull        : IN  STD_LOGIC;                        -- fifo ist voll
        Reset        : IN  STD_LOGIC);
END ht;
---------------------------------

ARCHITECTURE rtl OF ht IS
  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4);
  SIGNAL state_0, next_state_0 : state_type;
  signal sync_sig : STD_LOGIC;          -- Sync - Signal

BEGIN
 htproc : PROCESS(sync_sig, Reset)

  VARIABLE  Address    : STD_LOGIC_Vector(0 TO 15);
  VARIABLE  Header     : STD_LOGIC_Vector(47 DOWNTO 0);  -- Header w. RTag
  VARIABLE  rtag       : STD_LOGIC_Vector(15 DOWNTO 0);  -- Hilfsvar. f.rtag
  VARIABLE  vpivci     : STD_LOGIC_Vector(23 DOWNTO 0);  -- Hilfsvar. f. vpivci
  VARIABLE  rtdata     : STD_LOGIC_Vector(39 downto 0);  -- Hilfsvar.
  VARIABLE  rdwrt      : STD_LOGIC;

 BEGIN -- Process -----------------------------------------------------

   IF (Reset = '1') THEN
     Hd_fetch <= '0';  -- Setze die Signale auf null..
     fwrq     <= '0';
     rt_read  <= '0';
     next_state_0 <= ST0;

   ELSE

     CASE state_0 IS
      WHEN ST0 =>
      -- Hole einen Header vom AHT_In ab   -----------------
         IF (Hd_rdy = '1') THEN
```

```
            Header(31 downto 0) := Hd_Bus;
            Hd_fetch <= '1';          -- Sag AHT1_In: Header abgeholt!
            next_state_0 <= ST1;
         ELSE
            next_state_0 <= ST0;
         END IF;

      WHEN ST1 =>
        IF (Hd_rdy = '0') THEN
          --  Adressiere die Routing-Tabelle VPI_2, VCI_2
          Address := Header(27 downto 24) & Header(15 downto 4); -- Get VCI/VPI
          -- Hier kann man die Adressrechnung einsetzen Addr:=F1(Addr);
          -- Adressiere die RT und hole die neuen VPI und VCI's
          RT_Addr <= Address; -- Setze die Adresse
          rt_read <= '1';
          Hd_fetch <= '0';
          next_state_0 <= ST2;
        ELSE
          next_state_0 <= ST1;
        END IF;
      WHEN ST2 =>
          -- Warte auf das RT-Data Ergebnis
         IF (rtdta_strobe = '1') THEN
           rtdata := RT_data;
           rtag := rtdata(39 downto 24);  -- Hilfsvariablen wegen CADDY
           vpivci := rtdata(23 downto 0);   -- Hilfsvariablen wegen CADDY
           -- Header(31 DOWNTO 28) ist GFC, von Bit 3 bis 0: PTI und CLP
           Header(47 downto 4) := rtag & Header(31 DOWNTO 28) & vpivci;
           rt_read <= '0';
          next_state_0 <= ST3;
        ELSE
          next_state_0 <= ST2;
        END IF;

      WHEN ST3 =>
        IF (ffull = '0') THEN
           -- Speichere das Headerregister im Headerpuffer ab    -------
           fwrq <= '1';
           headout <= Header;
           next_state_0 <= ST4;
        ELSE
           next_state_0 <= ST3;
        END IF;

      WHEN ST4 =>
         IF (head_taken = '1') THEN
            fwrq <= '0';
            next_state_0 <= ST0;
         ELSE
            next_state_0 <= ST4;
         END IF;
      END CASE;
   END IF;
 END Process;

 Sync_Proc: PROCESS
  BEGIN
    IF (Reset = '1') THEN
      state_0 <= ST0;
      sync_sig <= '0';
      Wait until Clk_com'EVENT AND Clk_com = '1';
    ELSE
      if sync_sig = '0' then
         sync_sig <= '1';
      else
         sync_sig <= '0';
      end if;
      state_0 <= next_state_0;
      Wait until Clk_com'EVENT AND Clk_com = '1';
   END IF;
  END Process;

END;           -- End architecture
```

7.3 Routing-Steuerung (RC)

7.3.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Science,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Oct. 1998 and April 1999
```

```
--
-- Verhaltensbeschreibung des RC fuer
-- Synthese
-- rc.vhd hat ein two-way handshaking IF auch zum Header_FIFO
-- und hat Interfaces zu Head_FIfo, CC und SRC_Out,
-- in dieser Version wurde die Abfrage von 'free' umgestellt.
-- dadurch ist die Simulation stabiler.
-- moegliche Verbesserungen: siehe notes.
-- Auf der Basis der Monet-source rc_mon.vhd.
-- 2 Prozesse werden verwendet RCIN, RCOUT.
-- RCIN nimmt header und Routing Tag auf, untersucht den Routing Tag
-- und kommuniziert mit dem Connection Controller (CC)
-- RCOUT gibt den header an SRC weiter und kommuniziert mit SRC
-- multicast muss sein, damit der CC weiss, wann "configure" gegeben wird
-- Das Signal free gibt an, wann xmit_bsy nach einer Uebertragung
-- wieder aktiv werden darf, d.h. wann die Belegung eines Ausgangs
-- aufgehoben ist.
-- use IEEE.std_logic_unsigned.all; auskommentiert
-- Attribute: dont_unroll fuer die LOOPS.
-- Autor: W. Lange.  Last change: 4. Juli 2000
-- Global reset eingefhrt
-- start_xmit eingefuehrt 21. 4. 99 (4/21/99)
-- transm_ok eingefhrt am 26. 7. 99
-- Achtung: Architekturname ist geaendert: "behavior"

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY rc IS
  PORT (
       Clk_com      : IN STD_LOGIC;
       Reset        : IN STD_LOGIC;
       Hd_read      : OUT STD_LOGIC;            -- HEADER_FIFO
       Hd_write     : IN  STD_LOGIC;
       Hd_Data      : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
       Hd_loaded    : OUT STD_LOGIC; -- for tw-handshake
       busy         : IN  STD_LOGIC;            -- SRC_OUT
       transmit     : OUT STD_LOGIC;
       load_hd      : OUT STD_LOGIC;
       header       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
       load_ok      : IN  STD_LOGIC;
       multicast    : OUT STD_LOGIC;   -- CC
       RR           : OUT STD_LOGIC;
       OA           : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       WAIT_CC      : IN  STD_LOGIC;
       start_xmit   : IN  STD_LOGIC;  -- 4/21/99
       transm_ok    : IN  STD_LOGIC;  -- 7/26/99
       free         : IN STD_LOGIC;
       xmit_bsy     : OUT STD_LOGIC);
 END rc;
--------------------------------------------------------------------------

ARCHITECTURE behavior OF rc IS
  SIGNAL connected    : STD_LOGIC;
  SIGNAL head_reg     : STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL hd_reg_bsy   : STD_LOGIC;
  SIGNAL hd_r_b       : STD_LOGIC;    -- sagt RCOUT proc, head_reg busy
  SIGNAL head_ready   : STD_LOGIC;
  SIGNAL mcast        : STD_LOGIC;  -- 4/21/99

BEGIN

 RCIN: PROCESS -- Der RCIN Process nimmt den Header und den R-tag auf
   VARIABLE r_tag        :  STD_LOGIC_VECTOR(15 DOWNTO 0);
   VARIABLE r_tag_old    :  STD_LOGIC_VECTOR(15 DOWNTO 0);
   VARIABLE first        :  STD_LOGIC;



  BEGIN -- Process
  --  Reset: Setze alle Variablen und Indices zurueck ----------------------
    RR <= '0';
    OA <= "0000";
    hd_r_b <= '0';
    Hd_read <= '0';
    Hd_loaded <= '0';
    head_ready <= '0';
    connected       <= '0';
    r_tag_old := "0000000000000000";
    r_tag     := "0000000000000000";
    multicast <= '0';
    xmit_bsy <= '0';
    mcast    <= '0';    -- 4/21/99
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

  main_loop: loop    -- main loop --------------------------
    first := '1';

    -- Warte bis das Header Reg. frei ist.
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and hd_reg_bsy = '0';

    -- Setze Hd_read: Lies den Zellkopf ein  ----------------
    Hd_read <= '1';
```

```
  -- Warte bis ein Header da ist.
  Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and Hd_write = '1';

  r_tag := Hd_Data(15 downto 0);
  Hd_loaded <= '1'; -- two way handshake
                                     --
  Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and Hd_write = '0';
  Hd_loaded <= '0';
  -- hd_r_b <= '1';  -- kuendige den Header an (spaeter)

  Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and Hd_write = '1';
  head_reg <= Hd_Data;
  Hd_read <= '0';
  Hd_loaded <= '1';
  head_ready <= '1'; -- bedeutet: der Header ist geladen

  Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and Hd_write = '0';
  Hd_loaded <= '0';

  Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and  busy = '0';
  hd_r_b <= '1';    -- kuendige den Header an (hierher geschoben)

  IF (r_tag_old /= r_tag) THEN
    r_tag_old := r_tag;
    xmit_bsy  <= '0';  -- CC: Gib die Verbindungen frei
    connected <= '0';

    IF r_tag(15) = '1' THEN
      multicast <= '1';
      mcast     <= '1';  -- 4/21/99
    END IF;

    Wait UNTIL Clk_com'EVENT AND Clk_com = '1'; -- versetzt fuer monet

    -- Die loop L1 prueft 14 bits im rtag nach requests fuer
    -- Verbindungen. (Multicast) Fuer jedes aktive Bit:
    -- Generierung einer Ausgangsadresse

    L1:  FOR i IN 13 DOWNTO  0 loop  -- Wir haben nur 14 Kanaele
      IF r_tag(i) = '1' THEN

         IF (first = '1')  THEN  -- Tue's nur das erste mal

           -- Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and free = '1';

           Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and free = '1';
           xmit_bsy <= '1';  -- Wenn die Verbindung geloest ist ..
           first := '0';
           -- OA <= convert(i);
           OA  <= std_logic_vector(conv_unsigned(i, 4)); -- library fct..
           RR <= '1';
         ELSE
           -- OA <= convert(i);
   OA <= std_logic_vector(conv_unsigned(i, 4)); -- library fct..
           RR <= '1';
   Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
         END IF;  -- IF first

         Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and WAIT_CC = '1';
         -- WAIT_CC = '0' heisst: die Verbindung steht, es geht weiter
         Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and WAIT_CC = '0';
         RR <= '0';
         Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
      END IF;    -- IF r_tag(i) = '1'

     END loop;
  END IF;  -- If r_tag-old /= r_tag ..

  connected <= '1';
  Wait UNTIL Clk_com'EVENT AND Clk_com = '1';  -- 4/21/99

  multicast <= '0';  -- 4/21/99
  mcast <= '0';     -- 4/21/99

  Wait UNTIL Clk_com'EVENT AND Clk_com = '1';   --  4/21/99

  Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and hd_reg_bsy = '0';
  hd_r_b <= '0';
  head_ready <= '0';
  Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
 end loop; -- main loop
END process;


RCOUT: PROCESS
 BEGIN -- Process
    --  Reset: Setze alle Variablen und Indices zurueck ---------------
    transmit  <=  '0';
    load_hd <= '0';
    hd_reg_bsy <= '0';
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

  main_loop: loop    -- main loop --------------------------
```

```
      -- Warte bis der header kommt   -----------------------

      Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and hd_r_b = '1';
      hd_reg_bsy <= '1';

      Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and head_ready = '1' ;
      -- Warte bis SRC fertig ist
      Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and busy = '0';
      header <= head_reg;
      load_hd <= '1';

      Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and load_ok = '1';
      -- Der header ist abgegeben, Das Register ist wieder frei
      hd_reg_bsy <= '0';
      load_hd <= '0';

      -- Warte bis die Verbindung steht, dann kann uebertragen werden  ---

      Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and connected = '1';
      IF mcast = '1' THEN   -- 4/21/99  -- Bei multicast, warte
        Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and start_xmit = '1';
        transmit <= '1';

      ELSE                  -- 4/21/99
        transmit <= '1';
      END IF;               -- 4/21/99

      wait until Clk_com'event and Clk_com = '1'; -- 4/99 for sim R_39

      -- SRC uebertraegt jetzt
      Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and transm_ok = '1';
      transmit <= '0';
      wait until Clk_com'event and Clk_com = '1';
  END loop; -- main loop
 END Process; -- Prozess rcout..

END behavior;              -- Architekturname ist behavior
```

7.3.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 2000 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Science,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Sept. 2000
--
-- Verhaltens-RTL-Beschreibung des RC fuer
-- Synthese mit DC
-- rc_rtl.vhd hat ein two-way handshaking IF auch zum Header_FIFO
-- rc_rtl.vhd hat Interfaces zu Head_FIfo, CC und SRC_Out,
-- in dieser Version wurde die Abfrage von 'free' umgestellt.
-- dadurch ist die Simulation stabiler.
-- moegliche Verbesserungen: siehe notes.
-- 2 Prozesse werden verwendet RCIN, RCOUT.
-- RCIN nimmt header und Routing Tag auf, untersucht den Routing Tag
-- und kommuniziert mit dem Connection Controller (CC)
-- RCOUT gibt den header an SRC weiter und kommuniziert mit SRC
-- multicast muss sein, damit der CC weiss, wann "configure" gegeben wird
-- Das Signal free gibt an, wann xmit_bsy nach einer Uebertragung
-- wieder aktiv werden darf, d.h. wann die Belegung eines Ausgangs
-- aufgehoben ist.
-- Package ersetzt durch std. library function
-- Autor: W. Lange.   14. 9. 98
-- start_xmit und transm_ok uebernommen von rc_mon.vhd
-- Last update: 2000/09/29
-- state removed from sensitvity list
-- sync_sig introduced

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY rc IS
  PORT (
       Clk_com      : IN STD_LOGIC;
       Reset        : IN STD_LOGIC;
       Hd_read      : OUT STD_LOGIC;            -- HEADER_FIFO
       Hd_write     : IN  STD_LOGIC;
       Hd_Data      : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
       Hd_loaded    : OUT STD_LOGIC; -- for tw-handshake
       busy         : IN  STD_LOGIC;            -- SRC_OUT
       transmit     : OUT STD_LOGIC;
       load_hd      : OUT STD_LOGIC;
       header       : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
```

```
      load_ok      : IN  STD_LOGIC;
      multicast    : OUT STD_LOGIC;   -- CC
      RR           : OUT STD_LOGIC;
      OA           : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
      WAIT_CC      : IN  STD_LOGIC;
      start_xmit   : IN  STD_LOGIC;  -- 4/21/99
      transm_ok    : IN  STD_LOGIC;  -- 7/26/99
      free         : IN STD_LOGIC;
      xmit_bsy     : OUT STD_LOGIC);
 END rc;
---------------------------------
-- Use work.utilsSRC.all; ersetzt durch Standardfunktionen

ARCHITECTURE rtl OF rc IS
  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4,
ST5,ST6,ST7,ST8,ST9,ST10,
                        ST_A1,ST_A2,ST_A3,ST_A4,  -- Verzweigungs-States
                        ST_B1,ST_B2,ST_B3,ST_B4,
                        ST_C1,ST_C2);

  SIGNAL state_0, state_1, next_state_0, next_state_1 : state_type;
  SIGNAL connected    : STD_LOGIC;
  SIGNAL head_reg     : STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL hd_reg_bsy   : STD_LOGIC;
  SIGNAL hd_r_b       : STD_LOGIC;    -- sagt RCOUT proc, head_reg busy
  SIGNAL head_ready   : STD_LOGIC;
  SIGNAL mcast        : STD_LOGIC;  -- 4/21/99
  signal sync_sig     : STD_LOGIC;          -- Sync Signal

BEGIN
 -- Der SRCIN Process nimmt den Header und den R-tag auf
 RCIN: PROCESS(sync_sig, Reset)
   -- TYPE countertype  IS RANGE 0 TO 13;
   -- VARIABLE count      :  countertype;
   variable i : integer;       -- Laufvariable
   VARIABLE r_tag        :  STD_LOGIC_VECTOR(15 DOWNTO 0);
   VARIABLE r_tag_old    :  STD_LOGIC_VECTOR(15 DOWNTO 0);
   VARIABLE hilf         :  STD_LOGIC_VECTOR(31 DOWNTO 0);
   VARIABLE first        :  STD_LOGIC;
--    VARIABLE Statehilf   :  state_type;

 BEGIN -- Process
  --  Reset: Setze alle Variablen und Indices zurueck ---------------------

   IF (Reset = '1') THEN
     RR <= '0';
     OA <= "0000";
     hd_r_b <= '0' ;
     Hd_read <= '0';
     Hd_loaded <= '0';
     head_ready <= '0' ;
     connected       <= '0' ;
     r_tag_old := "0000000000000000";
     r_tag     := "0000000000000000";
     multicast <= '0';
     xmit_bsy <= '0';
     next_state_0 <= ST0;
     mcast    <= '0';    -- 4/21/99
     i := 13;

   ELSE

     CASE state_0 IS
       WHEN ST0 =>
         first := '1';
         -- count := 0;
         -- Warte bis das Header Reg. frei ist.
         -- Setze Hd_read: Lies den Zellkopf ein  ----------------
         if (hd_reg_bsy = '0') then
           next_state_0 <= ST1;
         else
           next_state_0 <= ST0;
         end if;

       WHEN ST1 =>
        Hd_read <= '1';
        next_state_0 <= ST2;

       WHEN ST2 =>
       -- Warte bis ein Header da ist.
         if (Hd_write = '1') then
           r_tag := Hd_Data(15 downto 0);
           Hd_loaded <= '1'; -- two way handshake
           next_state_0 <= ST3;
         else
           next_state_0 <= ST2;
         end if;

       WHEN ST3 =>
         if (Hd_write = '0') then
           Hd_loaded <= '0';            -- two way hs
       --    hd_r_b <= '1'; -- kuendige den Header an
           next_state_0 <= ST4;
```

```
    else
      next_state_0 <= ST3;
    end if;

  WHEN ST4 =>
    if (Hd_write = '1') then
      head_reg <= Hd_Data;
      Hd_read <= '0';
      Hd_loaded <= '1';
      head_ready <= '1'; -- bedeutet: der Header ist geladen
      next_state_0 <= ST5;
    else
      next_state_0 <= ST4;
    end if;

  WHEN ST5 =>
    if (Hd_write = '0') then
     Hd_loaded <= '0';   -- two way
     next_state_0 <= ST6;
    else
     next_state_0 <= ST5;
    end if;

  WHEN ST6 =>
     if busy = '0' then   -- Problem, kann hier haengen
--    if hd_reg_bsy = '0'  then -- wenn's haengt
      hd_r_b <= '1'; -- kuendige den Header an (hierher geschoben)
      next_state_0 <= ST7;
    else
      next_state_0 <= ST6;
    end if;

  WHEN ST7 =>
     IF (r_tag_old /= r_tag) then      -- Verzweigung, neuer r_tag
      next_state_0 <= ST_A1;
     else
       next_state_0 <= ST8;         -- Zweig 2
     end if;

    WHEN ST_A1 =>
      r_tag_old := r_tag;
      xmit_bsy  <= '0';  -- CC: Gib die Verbindungen frei
      connected <= '0' ;
      next_state_0 <= ST_A2;

    when ST_A2 =>
      IF r_tag(15) = '1' THEN
        multicast <= '1';
      end IF;
        next_state_0 <= ST_A3;

    -- Generierung der Ausgangsadresse, (oder Adressen bei Multicast)
    when ST_A3 =>
    -- L1:  FOR i IN 13 DOWNTO  0 loop  -- Wir haben 14 Kanaele
      IF r_tag(i) = '1' then    -- adressiere einen Kanal
         next_state_0 <= ST_B1;
      else
         next_state_0 <= ST_A4;
      end IF;

        when ST_B1 =>
         IF first = '1'  THEN  -- Tue's nur das erste mal
            next_state_0 <= ST_C1;
         else
            next_state_0 <= ST_B2;
        end if;

          when ST_C1 =>
            if free = '1'  then
             next_state_0 <= ST_C2;
            else
             next_state_0 <= ST_C1;
            end if;

           when ST_C2 =>
             xmit_bsy <= '1';  -- Wenn die Verbindung geloest ist ..
             first := '0' ;
--      OA  <= std_logic_vector(conv_unsigned(i, 4));
         --     RR <= '1';
         --     next_state_0 <= ST_B3;  muss ev. aus Zeitgruenden sein
             next_state_0 <= ST_B2;   -- Versuch!

         when ST_B2  =>
           --  ELSE
   OA  <= std_logic_vector(conv_unsigned(i, 4)); -- library fct..
           RR <= '1';
           -- END IF;  -- IF first
           next_state_0 <= ST_B3;

         when ST_B3 =>
          if WAIT_CC = '1' then
           next_state_0 <= ST_B4;
          else
```

```
               next_state_0 <= ST_B3;
              end if;

             when ST_B4 =>
               if WAIT_CC = '0' then
                next_state_0 <= ST_A4;
                RR <= '0';
               else
                next_state_0 <= ST_B4;
               end if;

           when ST_A4 =>
            IF i  > 0 THEN
              next_state_0 <= ST_A3;
              i := i - 1;
            ELSE
              i := 13;
              next_state_0 <= ST8;
            END IF;

     when ST8 =>
      multicast <= '0';
      mcast <= '0';      -- 4/21/99
      connected <= '1';
      next_state_0 <= ST9;

     when ST9 =>
       if hd_reg_bsy = '0' then
         next_state_0 <= ST10;
       else
          next_state_0 <= ST9;
       end if;

     when ST10 =>
       hd_r_b <= '0' ;
       head_ready <= '0' ;
       next_state_0 <= ST0;

     when others =>
    end case;
  end if;
 END process;

RCOUT: PROCESS(sync_sig, Reset)
 BEGIN -- Process
 --  Reset: Setze alle Variablen und Indices zurueck ----------------
 IF (Reset = '1') THEN
   transmit  <=  '0';
   load_hd <= '0';
   hd_reg_bsy <= '0' ;

 ELSE
 -- main loop ---------------------------
 CASE state_1 IS

   WHEN ST0 =>
     IF (hd_r_b = '1') THEN
       hd_reg_bsy <= '1';
       next_state_1 <= ST1;
     ELSE
       next_state_1 <= ST0;
     END IF;

    WHEN ST1 =>
     IF (head_ready = '1') THEN
       next_state_1 <= ST2;
     ELSE
       next_state_1 <= ST1;
     END IF;

    WHEN ST2 =>
     IF (busy = '0') THEN
       header <= head_reg;
       load_hd <= '1';
       next_state_1 <= ST3;
     ELSE
       next_state_1 <= ST2;
     END IF;

  -- Der header ist abgegeben, Das Register ist wieder frei
    WHEN ST3 =>
     IF (load_ok = '1') THEN
       hd_reg_bsy <= '0';
       load_hd <= '0';
       next_state_1 <= ST4;
     ELSE
       next_state_1 <= ST3;
     END IF;

  -- Warte bis die Verbindung steht, dann kann uebertragen werden  ---
    WHEN ST4 =>
     IF (connected = '1') THEN
     --  transmit <= '1';
```

```
        next_state_1 <= ST5;
      ELSE
        next_state_1 <= ST4;
      END IF;

     WHEN ST5 =>
        IF mcast = '1' THEN
           IF start_xmit  = '1' THEN
             next_state_1 <= ST6;
           ELSE
             next_state_1 <= ST5;
           end IF;
        ELSE
          next_state_1 <= ST6;
        end IF;

     WHEN ST6 =>
      transmit <= '1';
      next_state_1 <= ST7;

     when ST7 =>
      IF (transm_ok  = '1') THEN
        transmit <= '0';
        next_state_1 <= ST0;
      ELSE
        next_state_1 <= ST7;
      END IF;

     WHEN OTHERS =>
    END Case;
   END IF;
 END Process;

  Sync_Proc: PROCESS
  BEGIN
    if Reset = '1' then
       state_0 <= ST0;
       state_1 <= ST0;
       sync_sig <= '0';
       Wait until Clk_com'EVENT AND Clk_com = '1';
    ELSE
       if sync_sig = '0' then
         sync_sig <= '1';
       else
         sync_sig <= '0';
       end if;
       state_0 <= next_state_0;
       state_1 <= next_state_1;
       Wait until Clk_com'EVENT AND Clk_com = '1';
     end if;
  END Process;

END rtl;
```

7.4 Schieberegister-Steuerung (SRC)

7.4.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Science,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Autor:  Walter Lange, Oct. 1998 and April 1999
--
-- Verhaltensbeschreibung des SRC
-- src.vhd hat einen Prozess
-- Prozeduren shift und encode wieder inline eingearbeitet.
-- Reset eingefuehrt
-- Das Schieberegister wird ausgelagert, da BC einen
-- eine Taktzyclus hinzufuegt.
-- Letzte Version von srcout.vhd mit 4B/5B encoder
-- Am Anfang vor jeder Uebertragung einer Header-Zelle
-- steht das Header-Startzeichen: "11001" (X19)
-- Am Anfang vor jeder Uebertragung eines 32 - bit - Reg
-- steht das Payload-Register-Startzeichen: "10001" (X11)
-- Autor: W. Lange 1.9.98
-- dont_unroll fuer loop L1 eingefuehrt
-- transm_ok eingefhrt 27. 7. 99

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY srcout IS
  PORT (Clk_xmit    : IN STD_LOGIC;
```

```
       reset        : IN  STD_LOGIC;            -- Fuer BC und Monet
       busy         : OUT STD_LOGIC;            -- von und zu SRC_IN
       transmit     : IN  STD_LOGIC;
       load_hd      : IN  STD_LOGIC;
       header       : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
       load_ok      : OUT STD_LOGIC;
       Cell_read    : OUT STD_LOGIC;           -- Cell_FIFO
       Cell_write   : IN  STD_LOGIC;
       Cell_word    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
       Cell_loaded  : OUT  STD_LOGIC;
--     Cell_data    : OUT  STD_LOGIC   -- Datenleitung entfernt
       sr_ready     : IN   STD_LOGIC;  -- neu fuer SR
       sr_strobe    : OUT  STD_LOGIC;  -- neu fuer SR - Date
       transm_ok    : OUT  STD_LOGIC;  -- 27. 7. 99
       shiftreg     : OUT  STD_LOGIC_VECTOR(47 DOWNTO 0)); -- neuer Ausgang
END srcout;
----------------------------------------------------------------------
Use work.utils_src.all;

ARCHITECTURE srcoutar OF srcout IS

BEGIN

  Shift_Proc: PROCESS

  VARIABLE shift_reg : STD_LOGIC_VECTOR(47 DOWNTO 0);
  VARIABLE aux_reg : STD_LOGIC_VECTOR(31 DOWNTO 0);

-- Bitlokationen des shift_reg
--              47  43 42 38 37 33 32 28 27 23 22 18 17 13 12  8 7   3 2 0
-- shift_reg := "11001 00000 00000 00000 00000 00000 00000 00000 00000 000";


  Begin  -- Process
   -- reset_loop: loop
   --  Reset: Setze alle Variablen und Indices zurueck ---------------------
      busy <= '0';
      load_ok <= '0';
      Cell_loaded  <=  '0';
      Cell_read  <=  '0';
      sr_strobe  <= '0';
      transm_ok <= '0';
      shift_reg := "000000000000000000000000000000000000000000000000";
      shiftreg  <= "000000000000000000000000000000000000000000000000";
      aux_reg :=   "00000000000000000000000000000000";
      Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1';

    main_loop: loop
     -- Load Header ---------------------------------------------

    Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and load_hd = '1';
       load_ok <= '1';
       busy <= '1';

     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1';

     -- Lade Schieberegister und encodiere den Header
     --  load_reg_encode; encode wandelt 4 Bit in 5 Bit um
        shift_reg(42 downto 38) := encd(header(31 downto 28));
  shift_reg(37 downto 33) := encd(header(27 downto 24));
        shift_reg(32 downto 28) := encd(header(23 downto 20));
        shift_reg(27 downto 23) := encd(header(19 downto 16));
        shift_reg(22 downto 18) := encd(header(15 downto 12));
        shift_reg(17 downto 13) := encd(header(11 downto 8));
shift_reg(12 downto 8) := encd(header(7 downto 4));
  shift_reg(7 downto 3) := encd(header(3 downto 0));

     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and load_hd = '0';
     load_ok <= '0';

     -- Warte auf transmit und sr_ready   ---------------------

     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and transmit = '1';
     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and sr_ready = '1';

     -- das und-Verknuepfen (transmit und sr_ready)  geht hier nicht

     transm_ok <= '1';

     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and transmit = '0';
     transm_ok <= '0';

     shift_reg(47 downto 43) := "11001";   -- lade das "Start_Cell" Zeichen
     -- shift;-- call Procedure shift - Schiebe den Header durch den Switch
     -- entfernt
     sr_strobe <= '1';
     shiftreg  <= shift_reg;     -- lade das Schieberegister

     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and sr_ready ='0';

     sr_strobe <= '0';

     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1';
```

```
      -- Schiebe die Payload durch den Switch    ---------------------
  L1: FOR k IN 1 TO 12 loop
Cell_read <= '1';

        Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and Cell_write = '1';
        -- Lade Schieberegister und encodiere die Daten (load_reg_encode;)

            shift_reg(42 downto 38) := encd(Cell_word(31 downto 28));
            shift_reg(37 downto 33) := encd(Cell_word(27 downto 24));
            shift_reg(32 downto 28) := encd(Cell_word(23 downto 20));
            shift_reg(27 downto 23) := encd(Cell_word(19 downto 16));
            shift_reg(22 downto 18) := encd(Cell_word(15 downto 12));
            shift_reg(17 downto 13) := encd(Cell_word(11 downto 8));
    shift_reg(12 downto 8) := encd(Cell_word(7 downto 4));
      shift_reg(7 downto 3) := encd(Cell_word(3 downto 0));

        Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1';

        Cell_loaded <= '1';
        Cell_read   <= '0';

        Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and Cell_write = '0';
Cell_loaded <= '0';

        Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and sr_ready  = '1';

        shift_reg(47 downto 43) := "10001";   -- lade das "Start_Reg" Zeichen

        -- Schiebe ein Payload-Wort durch den Switch (call proc shift) -----
        -- entfernt

        sr_strobe <= '1';
        shiftreg  <= shift_reg;     -- lade das Schieberegister

        Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1' and sr_ready  = '0';
        sr_strobe <= '0';

        Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1';

     END loop;  -- For i = 1 to 12

     busy <= '0';
     Wait UNTIL Clk_xmit'EVENT AND Clk_xmit = '1';

    END loop;    -- main_loop
    -- end loop;     -- reset loop
   END Process;

  END srcoutar;
```

SRC-Package

```
---------------------------------------------------
Library IEEE;
USE IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;

package utils_src is
  subtype slv5 is STD_LOGIC_VECTOR(4 downto 0);
  function encd(v: STD_LOGIC_VECTOR(3 downto 0)) return slv5;

end;

---------------------------------------------------

package body utils_src is


function encd(v : STD_LOGIC_VECTOR(3 downto 0)) return slv5 IS
--
-- Encapsulation of logic below according to sold BC users guide
-- Optimizing Timing and Area
-- return type has changed from std_logic_vector to slv5
--
-- synopsys preserve_function
--
  VARIABLE temp   : STD_LOGIC_VECTOR(4 downto 0);
  VARIABLE vector : STD_LOGIC_VECTOR(3 downto 0);

  begin
    vector := v;

    CASE vector IS
WHEN  "0000"  => temp := "11110";
WHEN  "0001"  => temp := "01001";
        WHEN  "0010"  => temp := "10100";
WHEN  "0011"  => temp := "10101";
WHEN  "0100"  => temp := "01010";
        WHEN  "0101"  => temp := "01011";
        WHEN  "0110"  => temp := "01110";
WHEN  "0111"  => temp := "01111";
```

```
WHEN  "1000"  => temp := "10010";
WHEN  "1001"  => temp := "10011";
WHEN  "1010"  => temp := "10110";
WHEN  "1011"  => temp := "10111";
WHEN  "1100"  => temp := "11010";
WHEN  "1101"  => temp := "11011";
WHEN  "1110"  => temp := "11100";
WHEN  "1111"  => temp := "11101";
WHEN OTHERS => temp := "00000"; -- Probleme mit v_parser
     END CASE;

   return temp;
  end encd;
end utils_src;
---------------------------------------------------
```

7.4.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, May 1999 and Sept 2000
--
-- Verhaltensbeschreibung des SRC fuer
-- Simulation mit Synopsys
-- src_rtl.vhd hat einen Prozess
-- Prozeduren shift und encode wieder inline eingearbeitet.
-- Reset eingefuehrt
-- Das Schieberegister wird ausgelagert, da BC einen
-- eine Taktzyclus hinzufuegt.
-- srctl.vhd mit 4B/5B encoder
-- Am Anfang vor jeder Uebertragung einer Header-Zelle
-- steht das Header-Startzeichen: "11001" (X19)
-- Am Anfang vor jeder Uebertragung eines 32 - bit - Reg
-- steht das Payload-Register-Startzeichen: "10001" (X11)
-- Clk_xmit changed to Clk_com
-- Author: W. Lange Sept-26-200
-- sync_sig introduced

Library IEEE;
USE IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;

ENTITY srcout IS
  PORT (Clk_com     : IN STD_LOGIC;
       Reset        : IN  STD_LOGIC;              -- Fuer BC
       busy         : OUT STD_LOGIC;              -- von und zu SRC_IN
       transmit     : IN  STD_LOGIC;
       load_hd      : IN  STD_LOGIC;
       header       : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
       load_ok      : OUT STD_LOGIC;
       Cell_read    : OUT STD_LOGIC;             -- Cell_FIFO
       Cell_write   : IN  STD_LOGIC;
       Cell_word    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
       Cell_loaded  : OUT  STD_LOGIC;
--     Cell_data    : OUT  STD_LOGIC   -- Datenleitung entfernt
       sr_ready     : IN   STD_LOGIC;  -- neu fuer SR
       sr_strobe    : OUT  STD_LOGIC;  -- neu fuer SR - Date
       shiftreg     : OUT  STD_LOGIC_VECTOR(47 DOWNTO 0)); -- neuer Ausgang
END srcout;
----------------------------------------------------------------------
Use work.utils_src.all;

ARCHITECTURE rtl OF srcout IS
  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4,ST5,ST6,
                        ST_A1,ST_A2,ST_A3,ST_A4,ST_A5,ST_A6
                        );
  SIGNAL state, next_state : state_type;
  signal sync_sig : STD_LOGIC;          -- Sync Signal

BEGIN

  Shift_Proc: PROCESS(sync_sig, Reset)

    VARIABLE shift_reg : STD_LOGIC_VECTOR(47 DOWNTO 0);
    TYPE countertype  IS RANGE 0 TO 12;
    VARIABLE count      :  countertype;
    VARIABLE Statehilf  :  State_type;


  Begin  -- Process
   IF (Reset = '1') THEN
      busy <= '0';
      load_ok <= '0';
      Cell_loaded  <=  '0';
      Cell_read  <=  '0';
```

```
      sr_strobe  <= '0';
      shift_reg := "000000000000000000000000000000000000000000000000";
      shiftreg  <= "000000000000000000000000000000000000000000000000";
      count     := 1;

   ELSE

    CASE state IS
      WHEN ST0 =>

     -- Load Header -----------------------------------------------
         IF (load_hd = '1') THEN
           load_ok <= '1';
           busy <= '1';
           next_state <= ST1;
         ELSE
            next_state <= ST0;
         END IF;

     WHEN ST1 =>
        -- Lade Schieberegister und encodiere den Header
        --  load_reg_encode; encode wandelt 4 Bit in 5 Bit um

        shift_reg(42 downto 38) := encd(header(31 downto 28));
  shift_reg(37 downto 33) := encd(header(27 downto 24));
        shift_reg(32 downto 28) := encd(header(23 downto 20));
        shift_reg(27 downto 23) := encd(header(19 downto 16));
        shift_reg(22 downto 18) := encd(header(15 downto 12));
        shift_reg(17 downto 13) := encd(header(11 downto 8));
shift_reg(12 downto 8) := encd(header(7 downto 4));
  shift_reg(7 downto 3) := encd(header(3 downto 0));

        next_state <= ST2;

     WHEN ST2 =>
         IF (load_hd = '0') THEN
           load_ok <= '0';
           next_state <= ST3;
         ELSE
            next_state <= ST2;
         END IF;

     WHEN ST3 =>
     -- Warte auf transmit und sr_ready   ---------------------
         IF (transmit = '1' and sr_ready ='1') THEN
           shift_reg(47 downto 43) := "11001";   -- "Start_Cell" Zeichen
           sr_strobe <= '1';
           shiftreg  <= shift_reg;     -- lade das Schieberegister
           next_state <= ST4;
         ELSE
           next_state <= ST3;
         END IF;

     WHEN ST4 =>
         IF (sr_ready = '0') THEN
           sr_strobe <= '0';
           next_state <= ST5;
         ELSE
            next_state <= ST4;
         END IF;

     WHEN ST5 =>
      -- Schiebe die Payload durch den Switch    ---------------------
      -- Start state of the loop
             next_state <= ST_A1;

WHEN ST_A1 =>
    Cell_read <= '1';
          -- Warte auf das Zell-Wort
          IF (Cell_write = '1') THEN
            -- Lade Schieberegister und encodiere die Daten
            shift_reg(42 downto 38) := encd(Cell_word(31 downto 28));
            shift_reg(37 downto 33) := encd(Cell_word(27 downto 24));
            shift_reg(32 downto 28) := encd(Cell_word(23 downto 20));
            shift_reg(27 downto 23) := encd(Cell_word(19 downto 16));
            shift_reg(22 downto 18) := encd(Cell_word(15 downto 12));
            shift_reg(17 downto 13) := encd(Cell_word(11 downto 8));
    shift_reg(12 downto 8) := encd(Cell_word(7 downto 4));
      shift_reg(7 downto 3) := encd(Cell_word(3 downto 0));

            next_state <= ST_A2;
         ELSE
            next_state <= ST_A1;
         END IF;

WHEN ST_A2 =>
          Cell_loaded <= '1';
          Cell_read   <= '0';
          next_state <= ST_A3;

WHEN ST_A3 =>
         IF (Cell_write  = '0') THEN
   Cell_loaded <= '0';
```

```
            next_state <= ST_A4;
          ELSE
             next_state <= ST_A3;
          END IF;

WHEN ST_A4 =>
          IF (sr_ready  = '1') THEN

            shift_reg(47 downto 43) := "10001";   -- "Start_Reg" character
            sr_strobe <= '1';
            shiftreg  <= shift_reg;     -- lade das Schieberegister
            next_state <= ST_A5;
          ELSE
             next_state <= ST_A4;
          END IF;

WHEN ST_A5 =>
          IF (sr_ready  = '0') THEN
            sr_strobe <= '0';
            next_state <= ST_A6;
          ELSE
             next_state <= ST_A5;
          END IF;

WHEN ST_A6 =>
            IF count < 12 THEN
              count := count + 1;
              next_state <= ST_A1;
            ELSE
              count := 1;
              next_state <= ST6;
            END IF;


        WHEN ST6 =>
           busy <= '0';
           next_state <= ST0;

      END Case;
     END IF;
   END Process;

  Sync_Proc: PROCESS
  BEGIN
        if Reset = '1' then
          state <= ST0;
          sync_sig <= '0';
  Wait until Clk_com'EVENT AND Clk_com = '1';
        else
         if sync_sig = '0' then
           sync_sig <= '1';
         else
           sync_sig <= '0';
         end if;
  state <= next_state;
          Wait until Clk_com'EVENT AND Clk_com = '1';
        end if;
  END Process;

END rtl;
```

7.5 Verbindungs-Steuerung (CC)

7.5.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Oct. 1998 and April 1999
--
-- Verhaltensbeschreibung des CC mit Unicast
-- Broadcast und Multicast
-- cc_mp3.vhd nimmt die Routing Requests der RC's auf,
-- und veranlasst den Switch, eine Verbindung zwischen
-- Eingangs - und Ausgangsadresse zu schalten (load und configure)
-- Bis die Verbindung geschaltet ist, wird das Signal
-- Wait_CC aktiviert.
-- cc_new hat folgende Prozesse:
-- 14 Eingangsprozesse: (ccin_x prozesse), fr jeden Kanal einen Prozess.
-- fuer den Test reduziert auf 3 Eingangsprozesse (mo3)
-- Braodcast-Prozess, Multicast-Prozess, Switch-Prozess
-- switch-process: Interface zum switch
```

```
-- und Freigabe des belegt-Buffer und der verbind-Matrix
-- zu Beginn wird der switch auf Passthru gesetzt.
-- Autor: W. Lange,
-- Neu geschrieben fuer Monet: 14. April 1999
-- stark geaendert: 4. 7. 2000

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
Use work.utilsCC.all;


ENTITY cc IS
  PORT (
    rst        : IN STD_LOGIC;
    Clk_com    : IN STD_LOGIC;
    RR         : IN STD_LOGIC_VECTOR(0 to 13); -- Routing Request
    multicast  : IN STD_LOGIC_VECTOR(0 to 13);
    OA         : IN Address;
    xmit_bsy   : IN STD_LOGIC_VECTOR(0 to 13);
    Wait_CC    : OUT STD_LOGIC_VECTOR(0 to 13);
    free       : OUT  STD_LOGIC_VECTOR(0 to 13);
    start_xmit : OUT  STD_LOGIC_VECTOR(1 to 13);
    S_IA       : OUT STD_LOGIC_VECTOR(3 downto 0);      -- zum switch
    S_OA       : OUT STD_LOGIC_VECTOR(3 downto 0);
    load       : OUT STD_LOGIC;
    configure  : OUT STD_LOGIC;
    reset      : OUT STD_LOGIC);
  END cc;

ARCHITECTURE behavior OF cc IS

  SIGNAL  active            : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  active_uc         : STD_LOGIC_VECTOR(0 to 13);  -- unicast
  SIGNAL  belegt_reg        : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  act_ok            : STD_LOGIC_VECTOR(1 to 13);
  SIGNAL  act_ok_uc         : STD_LOGIC_VECTOR(1 to 13);  -- unicast
--  SIGNAL  cancel            : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  cancel            : STD_LOGIC_VECTOR(0 to 2);  -- auf (mp) 3 gesetzt
  SIGNAL  cancel_ok         : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  unicast_lock      : STD_LOGIC;
  SIGNAL  multicast_lock    : STD_LOGIC;                -- multicast
  SIGNAL  set_conf          : STD_LOGIC;                -- multicast
  SIGNAL  set_load          : STD_LOGIC;
  SIGNAL  set_res,set_ldconf       : STD_LOGIC;
  SIGNAL  setres_ok,setldconf_ok   : STD_LOGIC;
  SIGNAL  setconf_ok        : STD_LOGIC;
  SIGNAL  set_load_ok       : STD_LOGIC;
  SIGNAL  inpaddr_sig,outaddr_sig  : STD_LOGIC_VECTOR(3 downto 0);
  SIGNAL  inpaddr_sig_uc    : STD_LOGIC_VECTOR(3 downto 0);  -- unicast
  SIGNAL  outaddr_sig_uc    : STD_LOGIC_VECTOR(3 downto 0);  -- unicast
  CONSTANT zeros            :  STD_LOGIC_VECTOR(0 to 13) := "00000000000000";
 --  CONSTANT three_zeros            :  STD_LOGIC_VECTOR(0 to 2) := "000"; -- auf 3 gesetzt

BEGIN  -- architecture

ccin_0: PROCESS -- nimmt Routing Request von Channel 0 auf
                -- one of 14 processes
BEGIN -- process
    --  Reset: Setze alle Signale zurueck ----------------------
    Wait_CC(0)     <=  '0';  -- 1 Wait auf '0'
    free(0)        <=  '1';  -- 1 free auf '1'
    cancel(0)      <=  '0';
    unicast_lock          <= '0';
    set_res               <= '0';

    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

    main_loop: loop    -- main loop ---------------------------
    -- Warte bis  ein RR kommt
      wait until Clk_com'event and Clk_com = '1' and RR(0)  = '1';
      Wait_CC(0)   <= '1';
      unicast_lock <= '1';
      wait until Clk_com'event and Clk_com = '1' and belegt_reg  = zeros;
      set_res        <= '1';   -- set switch to broadcast
      wait until Clk_com'event and Clk_com = '1' and setres_ok  = '1';
      set_res  <= '0';
      Wait_CC(0) <= '0';
      free(0)     <= '0';
      wait until Clk_com'event and Clk_com = '1' and xmit_bsy(0)  = '1';
      wait until Clk_com'event and Clk_com = '1' and xmit_bsy(0)  = '0';
      cancel(0) <= '1';
      unicast_lock <= '0';
      wait until Clk_com'event and Clk_com = '1' and cancel_ok(0)  = '1';
      cancel(0) <= '0';
      free(0)  <= '1';
      wait until Clk_com'event and Clk_com = '1';
    end loop;  -- mainloop
END process;  -- ccin_0

ccin_1: PROCESS -- nimmt Routing Request von Channel 1auf
                -- one of 14 processes
BEGIN -- process
```

```
    --  Reset: Setze alle Signale zurueck ----------------------
    Wait_CC(1)    <=  '0';  -- 1 Wait auf '0'
    free(1)       <=  '1';  -- 1 free auf '1'
    active(1)     <=  '0';
    active_uc(1)  <=  '0';
    cancel(1)     <=  '0';
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

    main_loop: loop    -- main loop ---------------------------
      -- Warte bis  ein RR kommt
      wait until Clk_com'event and Clk_com = '1' and RR(1) = '1';
      -- multicast?
      IF multicast(1) =  '1' THEN

        WHILE multicast(1) = '1' loop  -- multicast loop
          Wait_CC(1) <= '1';
          wait until Clk_com'event and Clk_com = '1'and act_ok(1) = '0';
          active(1)  <= '1';
          wait until Clk_com'event and Clk_com = '1'and act_ok(1) = '1';
          Wait_CC(1) <= '0';
          active(1)  <= '0';
          free(1)    <= '0';
          wait until Clk_com'event and Clk_com = '1' and RR(1) = '0';
        END loop; -- while loop

      ELSE
        Wait_CC(1) <= '1';
        active_uc(1)  <= '1';
        wait until Clk_com'event and Clk_com = '1' and act_ok_uc(1) = '1';
        Wait_CC(1) <= '0';
        active_uc(1)  <= '0';
        free(1)    <= '0';
      end IF;
      -- wait for src busy,
      wait until Clk_com'event and Clk_com = '1' and xmit_bsy(1) = '1';
      -- wait for src copleted
      wait until Clk_com'event and Clk_com = '1' and xmit_bsy(1) = '0';

      cancel(1) <= '1';
      wait until Clk_com'event and Clk_com = '1' and cancel_ok(1) = '1';
      cancel(1) <= '0';
      free(1) <= '1';
      wait until Clk_com'event and Clk_com = '1';
    end loop;  -- mainloop
END process;

ccin_2: PROCESS -- nimmt Routing Request von Channel lauf
                -- one of 14 processes
BEGIN -- process
    --  Reset: Setze alle Signale zurueck ----------------------
    Wait_CC(2)    <=  '0';  -- 1 Wait auf '0'
    free(2)       <=  '1';  -- 1 free auf '1'
    active(2)     <=  '0';
    cancel(2)     <=  '0';
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

    main_loop: loop    -- main loop ---------------------------
    -- Warte bis  ein RR kommt
      wait until Clk_com'event and Clk_com = '1' and RR(2) = '1';
      -- multicast?
      IF multicast(2) =  '1' THEN

        WHILE multicast(2) = '1' loop  -- multicast loop
          Wait_CC(2) <= '1';
          wait until Clk_com'event and Clk_com = '1' and act_ok(2) = '0';
          active(2)  <= '1';
          wait until Clk_com'event and Clk_com = '1' and act_ok(2) = '1';
          Wait_CC(2) <= '0';
          active(2)  <= '0';
          free(2)    <= '0';
          wait until Clk_com'event and Clk_com = '1' and RR(2) = '0';
        END loop; -- while loop

      ELSE
        Wait_CC(2) <= '1';
        active_uc(2)  <= '1';
        wait until Clk_com'event and Clk_com = '1' and act_ok_uc(2) = '1';
        Wait_CC(2) <= '0';
        active_uc(2)  <= '0';
        free(2)    <= '0';
      end IF;

      wait until Clk_com'event and Clk_com = '1' and xmit_bsy(2) = '1';
      wait until Clk_com'event and Clk_com = '1' and xmit_bsy(2) = '0';

      cancel(2) <= '1';
      wait until Clk_com'event and Clk_com = '1' and cancel_ok(2) = '1';
      cancel(2) <= '0';
      free(2) <= '1';
      wait until Clk_com'event and Clk_com = '1';
    end loop;  -- main loop
END process;
```

```
unicast_proc: Process

   VARIABLE inp_addr, out_addr     :  STD_LOGIC_VECTOR(3 DOWNTO 0);
   VARIABLE OAd      :  NATURAL;
   variable i : natural := 0;           -- laufvariable nur fuer debug

BEGIN -- Process
    --  Reset: Setze alle Signale zurueck ----------------------
    inp_addr         := "0000";
    out_addr         := "0000";
    OAd            := 0;
    set_ldconf      <= '0';
    inpaddr_sig_uc  <= "0000";
    outaddr_sig_uc  <= "0000";
    act_ok_uc      <=  "0000000000000";
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

    main_loop: loop

      L1: FOR i in 1 to 13 loop    -- 13 Kanaele-loop
        wait until Clk_com'event and Clk_com = '1' and unicast_lock  = '0';
        wait until Clk_com'event and Clk_com = '1' and multicast_lock  = '0';
        inp_addr := std_logic_vector(conv_unsigned(i, 4)); -- library fct..

        IF active_uc(i) = '1'  THEN
          out_addr    := OA(i);
          OAd  := natural(CONV_INTEGER(out_addr));

          wait until Clk_com'event and Clk_com = '1' and belegt_reg(OAd)  = '0';

          inpaddr_sig_uc  <= inp_addr;
          outaddr_sig_uc <= out_addr;
        set_ldconf  <= '1';  -- an switch process;

          wait until Clk_com'event and Clk_com = '1' and setldconf_ok  = '1';
          act_ok_uc(i)    <= '1';
          set_ldconf <= '0';

          inp_addr    := "0000";
          out_addr    := "0000";
          inpaddr_sig_uc    <= "0000";
          outaddr_sig_uc    <= "0000";

          wait until Clk_com'event and Clk_com = '1' and active_uc(i)  = '0';
          act_ok_uc(i) <= '0';

        END IF;
      END loop;  -- 1 to 13 loop

    End loop; -- main loop
END process;

multicast_proc: Process

   VARIABLE inp_addr, out_addr     :  STD_LOGIC_VECTOR(3 DOWNTO 0);
   VARIABLE OAd      :  NATURAL;
   variable i  : natural := 0;          -- laufvariable, nur fuer debug

BEGIN -- Process
    --  Reset: Setze alle Signale zurueck ----------------------
    inp_addr       := "0000";
    out_addr       := "0000";
    OAd            := 0;
    set_load       <= '0';
    inpaddr_sig    <= "0000";
    outaddr_sig    <= "0000";
    act_ok         <=  "0000000000000";
    set_conf       <=  '0';
    start_xmit     <=  "0000000000000"; -- for mcast, tells RC to begin xmit
    multicast_lock  <= '0';
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

    main_loop: loop

      L1: FOR i in 1 to 13 loop    -- 13 Kanaele-loop
        wait until Clk_com'event and Clk_com = '1' and unicast_lock  = '0';

        inp_addr := std_logic_vector(conv_unsigned(i, 4)); -- library fct..
IF active(i)  = '1' THEN
          multicast_lock  <= '1';
          While multicast(i) = '1' loop
            IF active(i)  = '1' then    -- wird wiederholt, es muss sein

             -- OA(i) wird immer neu gestezt von RC
             out_addr    := OA(i);         -- conv to std_logic_vector
             OAd  := natural(CONV_INTEGER(out_addr));

             wait until Clk_com'event and Clk_com = '1' and belegt_reg(OAd) = '0';
             inpaddr_sig  <= inp_addr;
             outaddr_sig  <= out_addr;
             set_load  <= '1';  -- an switch process;
--              Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
             wait until Clk_com'event and Clk_com = '1' and set_load_ok = '1';
```

```
            set_load  <= '0';  -- an switch process;
            act_ok(i) <= '1';

            wait until Clk_com'event and Clk_com = '1' and active(i)  = '0';
            act_ok(i) <= '0';
            ELSE
             wait until Clk_com'event and Clk_com = '1';
           END IF; -- if active(i) = '1';
         End loop; -- while loop
       multicast_lock <= '0';
       set_conf  <= '1';  -- an switch process;
       wait until Clk_com'event and Clk_com = '1' and setconf_ok  = '1';
       set_conf <= '0';

       start_xmit(i) <= '1';
       wait until Clk_com'event and Clk_com = '1' and xmit_bsy(i) = '1';
       start_xmit(i) <= '0';
       -- Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
       END IF;  -- if active(i) = '1';
     END loop;  -- 1 to 13 loop

--      Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
    End loop; -- main loop
END process;


switch_proc: process -- interface zum switch

   VARIABLE  verbind           : matrix;
   VARIABLE  vector_14         : STD_LOGIC_VECTOR(0 to 13) := "00000000000000";
   VARIABLE i,OAd,m: NATURAL;
   VARIABLE inp_addr, out_addr     :  STD_LOGIC_VECTOR(3 DOWNTO 0);

BEGIN
    -- reset first
    load         <= '0';
    configure    <= '1';
    reset        <= '1';
    setres_ok    <= '0';
    setldconf_ok <= '0';
    setconf_ok   <= '0';
    set_load_ok   <= '0';
    cancel_ok    <= "00000000000000";
    belegt_reg   <= "00000000000000";
    i            := 0;
    m            := 0;
    OAd          := 0;
    inp_addr     := "0000";
    out_addr     := "0000";
    S_IA           <=  "0000";
    S_OA           <=  "0000";
    vector_14 := "00000000000000";

    -- Set Switch auf Passthru
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
    -- Warte einen Taktzyklus (mind. 7 ns)
    configure  <= '0';
    reset      <= '0';
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
    -- Setze Verbindungsmatrix gesamt auf '0'
    For i in 0 to 13 loop  verbind(i) := "00000000000000";  end loop;
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

    main_loop: loop
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
     -- unicast
     IF (set_ldconf = '1') then         -- kommt von unicast process
       load <= '1';
       -- Write belegt_reg und verbind-matrix
       out_addr := outaddr_sig_uc;
       inp_addr := inpaddr_sig_uc;
       S_IA      <= inp_addr;
       S_OA      <=  out_addr;
       i    :=  natural(CONV_INTEGER(inp_addr));
       OAd  :=  natural(CONV_INTEGER(out_addr));
       configure <= '1';
       vector_14 := zeros;
       vector_14(OAd) := '1';
       verbind(i) :=  verbind(i) or vector_14;
       belegt_reg(OAd) <= '1';
       wait until Clk_com'event and Clk_com = '1';
       -- Warte einen Takt lang, mind. 7 ns
       load <= '0';
       configure <= '0';
 --       reset <= '0';
       setldconf_ok <= '1';
       wait until Clk_com'event and Clk_com = '1' and set_ldconf = '0';
       setldconf_ok <= '0';
       wait until Clk_com'event and Clk_com = '1';
     END IF;

     -- multicast: set load
     IF (set_load = '1') then         -- kommt von unicast process
```

```
        load <= '1';
        -- Write belegt_reg und verbind-matrix
        out_addr := outaddr_sig;
        inp_addr := inpaddr_sig;
        S_IA      <= inp_addr;
        S_OA      <=  out_addr;
        i    :=  natural(CONV_INTEGER(inp_addr));
        OAd  :=  natural(CONV_INTEGER(out_addr));

        vector_14 := zeros;
        vector_14(OAd) := '1';
        verbind(i) :=  verbind(i) or vector_14;
        belegt_reg(OAd) <= '1';
        wait until Clk_com'event and Clk_com = '1';
        -- Warte einen Takt lang, mind. 7 ns
        load <= '0';
        set_load_ok <= '1';
        wait until Clk_com'event and Clk_com = '1' and set_load = '0';
        set_load_ok <= '0';
      END IF;

      -- multicast: set_configure
      IF (set_conf = '1') THEN
        configure <= '1';
        wait until Clk_com'event and Clk_com = '1';
        configure <= '0';
        setconf_ok <= '1';
        wait until Clk_com'event and Clk_com = '1' and set_conf = '0';
        setconf_ok <= '0';
      END IF;

      -- broadcast
      IF (set_res = '1') THEN
        reset <= '1';
        belegt_reg <=  "11111111111111";
        wait until Clk_com'event and Clk_com = '1';
        -- Warte einen Takt lang, mind. 7 ns
        reset <= '0';
        setres_ok <= '1';
        wait until Clk_com'event and Clk_com = '1' and set_res = '0';
        setres_ok <= '0';
      END IF;

       -- clear busy-reg (belegt_reg) and verbind(i) matrix
      IF (cancel /= zeros) THEN
         IF  cancel(0) = '1' THEN   -- broadcast - Abfrage
                belegt_reg  <= "00000000000000";
                cancel_ok(0) <= '1';
                wait until Clk_com'event and Clk_com = '1' and cancel(0) = '0';
                cancel_ok(0) <= '0';
                Wait UNTIL Clk_com'EVENT AND Clk_com = '1';
         END IF;  --

    --     L2: FOR m in 1 to 13 loop  -- kanaele loop
          L2: FOR m in 1 to 2 loop  -- kanaele loop auf 3 gesetzt
           IF  cancel(m) = '1' THEN
               -- Loesche die Belegungenn im belegt-reg
               belegt_reg <= belegt_reg and not verbind(m);
               -- Loesche Belegung in der Verbind-matrix
               verbind(m) := zeros;
               cancel_ok(m) <= '1';
               wait until Clk_com'event and Clk_com = '1' and cancel(m) = '0';
               cancel_ok(m) <= '0';
            END IF;    -- cancel(m)
         END loop; -- kanaele-loop

      END IF; -- if cancel /= zeros

  end loop; -- main loop
end process;

END behavior; -- End architecture
```

CC-Package

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Oct. 1998 and May 1999
--
-- function conv_integer(ARG..) uebernommen von std_logic_unsigned
-- sonst gibt es Probleme im cc_bc.vhd source code mit
-- std_logic_arith
-- damit werden die Funktionen convert und convert_to nat nicht mehr gebraucht.
-- Sept 15. '98 W. L.


Library IEEE;
USE IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;
```

```
package utilsCC is

 TYPE Address IS ARRAY (Integer Range 0 to 13) OF STD_LOGIC_VECTOR(3 downto 0);
 TYPE matrix  IS ARRAY (Integer Range 0 to 13) OF STD_LOGIC_VECTOR(0 to 13);

  function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

end;

---------------------------------------------------

package body utilsCC is

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER is
        variable result    : UNSIGNED(ARG'range);
    begin
        result    := UNSIGNED(ARG);
        return   CONV_INTEGER(result);
    end;

end utilsCC;

---------------------------------------------------
```

7.5.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 2000 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, May 2000
--
-- Verhaltensbeschreibung des CC mit Unicast
-- Broadcast und Multicast
-- cc_mp3-rtl.vhd nimmt die Routing Requests der RC's auf,
-- und veranlasst den Switch, eine Verbindung zwischen
-- Eingangs - und Ausgangsadresse zu schalten (load und configure)
-- Bis die Verbindung geschaltet ist, wird das Signal
-- Wait_CC aktiviert.
-- cc_ hat folgende Prozesse:
-- 14 Eingangsprozesse: (ccin_x prozesse), fr jeden Kanal einen Prozess.
-- reduziert auf 3, fuer Testzwecke.
-- Braodcast-Prozess, Multicast-Prozess, Switch-Prozess
-- switch-process: Interface zum switch
-- und Freigabe des belegt-Buffer und der verbind-Matrix
-- zu Beginn wird der switch auf Passthru gesetzt.
-- Autor: W. Lange, 27. 9. 2000
-- Neu geschrieben fuer Monet: 14. April 1999
-- cc_mp3-rtl.vhd hat 3 Eingangsprozesse, um das Testen etc. zu vereinfachen
-- Umgeschrieben zur RTL-Beschreibung: 26. 5. 00
-- als Ausgangsdatei wird cc_mon-mp3.vhd genommen.
-- sync_sig eingefuehrt

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

Use work.utilscc.all;

ENTITY cc IS
  PORT (
    rst        : IN STD_LOGIC;
    Clk_com    : IN STD_LOGIC;
    RR         : IN STD_LOGIC_VECTOR(0 to 13); -- Routing Request
    multicast  : IN STD_LOGIC_VECTOR(0 to 13);
    OA         : IN Address;
    xmit_bsy   : IN STD_LOGIC_VECTOR(0 to 13);
    Wait_CC    : OUT STD_LOGIC_VECTOR(0 to 13);
    free       : OUT  STD_LOGIC_VECTOR(0 to 13);
    start_xmit : OUT  STD_LOGIC_VECTOR(1 to 13);
    S_IA       : OUT STD_LOGIC_VECTOR(3 downto 0);      -- zum switch
    S_OA       : OUT STD_LOGIC_VECTOR(3 downto 0);
    load       : OUT STD_LOGIC;
    configure  : OUT STD_LOGIC;
    reset      : OUT STD_LOGIC);
  END cc;

ARCHITECTURE rtl OF cc IS
  TYPE   state_type IS (ST00,
                        ST0,ST1,ST2,ST3,ST4,ST5,ST6,
                        ST_A1,ST_A2,ST_A3,ST_A4,ST_A5,ST_A6,
                        ST_B1,ST_B2,ST_B3,ST_B4,
                        ST_C1,ST_C2,ST_C3,
                        ST_D1,ST_D2,ST_D3,
                        ST_E1,ST_E2,ST_E3,ST_E4
                        );
```

```
  SIGNAL state_0, state_1, state_2, state_3, state_4, state_5,
    next_state_0, next_state_1, next_state_2, next_state_3,
    next_state_4, next_state_5     : state_type;

  SIGNAL  active            : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  active_uc         : STD_LOGIC_VECTOR(0 to 13);  -- unicast
  SIGNAL  belegt_reg        : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  act_ok            : STD_LOGIC_VECTOR(1 to 13);
  SIGNAL  act_ok_uc         : STD_LOGIC_VECTOR(1 to 13);  -- unicast
  SIGNAL  cancel            : STD_LOGIC_VECTOR(0 to 2);  -- auf (mp) 3 gesetzt
  -- SIGNAL  cancel            : STD_LOGIC_VECTOR(0 to 13);
  SIGNAL  cancel_ok         : STD_LOGIC_VECTOR(0 to 13);
  -- SIGNAL  act_bcast         : STD_LOGIC;
  SIGNAL  act_bcast_ok      : STD_LOGIC;
  SIGNAL  broadcast_lock     : STD_LOGIC;
  SIGNAL  multicast_lock    : STD_LOGIC;                -- multicast
  SIGNAL  set_conf          : STD_LOGIC;
  SIGNAL  set_load          : STD_LOGIC;
  SIGNAL  set_res, set_ldconf       : STD_LOGIC;
  SIGNAL  setres_ok, setldconf_ok   : STD_LOGIC;
  SIGNAL  set_load_ok          : STD_LOGIC;
  SIGNAL  setconf_ok        : STD_LOGIC;
  SIGNAL  inpaddr_sig,outaddr_sig  : STD_LOGIC_VECTOR(3 downto 0);
  SIGNAL  inpaddr_sig_uc    : STD_LOGIC_VECTOR(3 downto 0);  -- unicast
  SIGNAL  outaddr_sig_uc    : STD_LOGIC_VECTOR(3 downto 0);  -- unicast
  signal sync_sig : STD_LOGIC;          -- Sync Signal

  CONSTANT zeros        :  STD_LOGIC_VECTOR(0 to 13) := "00000000000000";
  CONSTANT three_zeros  :  STD_LOGIC_VECTOR(0 to 2) := "000";

BEGIN  -- architecture

-- Prozess ccin_0 nimmt Routing Request von Channel 0 auf
ccin_0: PROCESS(sync_sig, rst)  --  state_0)

BEGIN -- process
    --  Reset: Setze alle Signale zurueck ----------------------
  IF (rst = '1') THEN
    Wait_CC(0)     <=  '0';  -- 1 Wait auf '0'
    free(0)        <=  '1';  -- 1 free auf '1'
    cancel(0)      <=  '0';
    next_state_0 <= ST0;
    broadcast_lock          <= '0';
    set_res               <= '0';

  ELSE
      -- main loop ---------------------------
    -- Warte bis  ein RR kommt
    CASE state_0 IS
     WHEN ST0 =>
       if RR(0) = '1'   then
         Wait_CC(0)   <= '1';
         -- act_bcast    <= '1';
         broadcast_lock <= '1';
         next_state_0 <= ST1;
       else
           next_state_0 <= ST0;
       end if;

      WHEN ST1 =>
       if belegt_reg = zeros then
           set_res        <= '1';   -- set switch to broadcast
           next_state_0 <= ST2;
         else
           next_state_0 <= ST1;
       end if;

      WHEN ST2 =>
       if setres_ok  = '1' then
           set_res  <= '0';
           Wait_CC(0) <= '0';
           free(0)     <= '0';
           next_state_0 <= ST3;
         else
           next_state_0 <= ST2;
       end if;

      WHEN ST3 =>
       if xmit_bsy(0)  = '1'   then -- src is busy transmitting
         next_state_0 <= ST4;
       else
         next_state_0 <= ST3;
       end if;

     WHEN ST4 =>
       if xmit_bsy(0)  = '0'   then
         cancel(0) <= '1';
         broadcast_lock <= '0';
         next_state_0 <= ST5;
       else
           next_state_0 <= ST4;
       end if;
```

```
     WHEN ST5 =>
       if cancel_ok(0)  = '1'   then
         cancel(0) <= '0';
         free(0)  <= '1';
         next_state_0 <= ST0;
       else
           next_state_0 <= ST5;
       end if;

     when others =>

    -- end loop;  -- mainloop
    end CASE;
  END IF;
END process;  -- ccin_0

 -- ccin_1: PROCESS nimmt Routing Request von Channel lauf
ccin_1: PROCESS(sync_sig, rst) --  state_1)
BEGIN -- process
    --  Reset: Setze alle Signale zurueck ----------------------
  IF (rst = '1') THEN
   Wait_CC(1)    <=  '0';  -- 1 Wait auf '0'
    free(1)       <=  '1';  -- 1 free auf '1'
    active(1)     <=  '0';
    active_uc(1)  <=  '0';
    cancel(1)     <=  '0';
    next_state_1 <= ST0;

  ELSE
    -- main loop ---------------------------
    -- Warte bis  ein RR kommt
    CASE state_1 IS
     WHEN ST0 =>
       if RR(1) = '1'   then
           next_state_1 <= ST1;
       else
         next_state_1 <= ST0;
       end if;

     WHEN ST1 =>
        -- multicast?
        -- WHILE multicast(1) = '1' loop  -- multicast loop
        IF multicast(1) =  '1' THEN
          next_state_1 <= ST_A1;
        else
          next_state_1 <= ST_B1;        -- else Zweig
        end if;

        when ST_A1 =>                   -- while loop body
           Wait_CC(1) <= '1';
           if act_ok(1) = '0' then
             next_state_1 <= ST_A2;
           else
             next_state_1 <= ST_A1;
           end if;

         when ST_A2 =>
           active(1)  <= '1';
           next_state_1 <= ST_A3;

         when ST_A3 =>
           if act_ok(1) = '1' then
             Wait_CC(1) <= '0';
             active(1)  <= '0';
             free(1)    <= '0';
             next_state_1 <= ST_A4;
            else
             next_state_1 <= ST_A3;
            end if;

         when ST_A4 =>
           if RR(1) = '0' then
             next_state_1 <= ST_A5;     -- end while loop
           else
             next_state_1 <= ST_A4;
           end if;

         when ST_A5 =>
           IF multicast(1) =  '1' THEN
             next_state_1 <= ST_A1;
           else
             next_state_1 <= ST2;       -- end of if - else
           end if;
--         END loop; -- while loop

     WHEN ST_B1 =>                      -- else Zweig
        Wait_CC(1) <= '1';
        active_uc(1)  <= '1';
        next_state_1 <= ST_B2;

     WHEN ST_B2 =>
        if act_ok_uc(1) = '1' then
          Wait_CC(1) <= '0';
```

```
          active_uc(1)  <= '0';
          free(1)    <= '0';
          next_state_1 <= ST2;
        else
          next_state_1 <= ST_B2;
        end if;                         -- end if - else  multicast

     WHEN ST2 =>    --  jump here from multicast tree
        if xmit_bsy(1) = '1' then      -- src busy,
          next_state_1 <= ST3;
        else
          next_state_1 <= ST2;
        end if;

     WHEN ST3 =>
        if xmit_bsy(1) = '0' then      -- src completed
          next_state_1 <= ST4;
        else
          next_state_1 <= ST3;
        end if;

     WHEN ST4 =>
      cancel(1) <= '1';
      next_state_1 <= ST5;

     WHEN ST5 =>
       if cancel_ok(1) = '1' then
          cancel(1) <= '0';
          free(1) <= '1';
          next_state_1 <= ST6;
        else
          next_state_1 <= ST5;
        end if;

     WHEN ST6 =>
       next_state_1 <= ST0;

     when others =>
   end CASE;
  END IF;
-- end loop;  -- mainloop
END process;                             -- ccin_1

 -- ccin_2: PROCESS nimmt Routing Request von Channel 2 auf
ccin_2: PROCESS(sync_sig, rst) --  state_2)
BEGIN -- process
    --  Reset: Setze alle Signale zurueck ----------------------
  IF (rst = '1') THEN

    Wait_CC(2)    <=  '0';  -- 1 Wait auf '0'
    free(2)       <=  '1';  -- 1 free auf '1'
    active(2)     <=  '0';
    active_uc(2)  <=  '0';
    cancel(2)     <=  '0';
    next_state_2 <= ST0;

  ELSE
    -- main loop ---------------------------
    -- Warte bis  ein RR kommt
    CASE state_2 IS
     WHEN ST0 =>
       if RR(2) = '1'   then
           next_state_2 <= ST1;
       else
         next_state_2 <= ST0;
       end if;

     WHEN ST1 =>
        -- multicast?
        -- WHILE multicast(2) = '1' loop  -- multicast loop
        IF multicast(2) =  '1' THEN
          next_state_2 <= ST_A1;
        else
          next_state_2 <= ST_B1;        -- else Zweig
        end if;

        when ST_A1 =>                   -- while loop body
           Wait_CC(2) <= '1';
           if act_ok(2) = '0' then
             next_state_2 <= ST_A2;
           else
             next_state_2 <= ST_A1;
           end if;

         when ST_A2 =>
           active(2)  <= '1';
           next_state_2 <= ST_A3;

         when ST_A3 =>
           if act_ok(2) = '1' then
             Wait_CC(2) <= '0';
             active(2)  <= '0';
             free(2)    <= '0';
```

```
            next_state_2 <= ST_A4;
           else
            next_state_2 <= ST_A3;
           end if;

        when ST_A4 =>
          if RR(2) = '0' then
            next_state_2 <= ST_A5;     -- end while loop
          else
            next_state_2 <= ST_A4;
          end if;

        when ST_A5 =>
          IF multicast(2) =  '1' THEN
            next_state_2 <= ST_A1;
          else
            next_state_2 <= ST2;       -- end of if - else
          end if;
--        END loop; -- while loop

     WHEN ST_B1 =>                     -- else Zweig
        Wait_CC(2) <= '1';
        active_uc(2)  <= '1';
        next_state_2 <= ST_B2;

     WHEN ST_B2 =>
        if act_ok_uc(2) = '1' then
          Wait_CC(2) <= '0';
          active_uc(2)  <= '0';
          free(2)    <= '0';
          next_state_2 <= ST2;
        else
          next_state_2 <= ST_B2;
        end if;                        -- end if - else  multicast

     WHEN ST2 =>    --  jump here from multicast tree
        if xmit_bsy(2) = '1' then      -- src busy,
          next_state_2 <= ST3;
        else
          next_state_2 <= ST2;
        end if;

     WHEN ST3 =>
        if xmit_bsy(2) = '0' then      -- src completed
          next_state_2 <= ST4;
        else
          next_state_2 <= ST3;
        end if;

     WHEN ST4 =>
      cancel(2) <= '1';
      next_state_2 <= ST5;

     WHEN ST5 =>
       if cancel_ok(2) = '1' then
          cancel(2) <= '0';
          free(2) <= '1';
          next_state_2 <= ST6;
        else
          next_state_2 <= ST5;
        end if;

     WHEN ST6 =>
       next_state_2 <= ST0;

     when others =>
    end CASE;
  END IF;
-- end loop;  -- mainloop
END process;                          -- ccin_2


-- Prozess No.3 fuer unicast requests
unicast_proc: Process(sync_sig, rst) --  state_3)

   VARIABLE inp_addr, out_addr     :  STD_LOGIC_VECTOR(3 DOWNTO 0);
   VARIABLE OAd      :  NATURAL;
   variable i : integer;        -- laufvariable u. Index fuer Vektoren

BEGIN -- Process
    --  Reset: Setze alle Signale zurueck ----------------------
  IF (rst = '1') THEN
    inp_addr         := "0000";
    out_addr         := "0000";
    OAd              := 0;
    set_ldconf       <= '0';
    inpaddr_sig_uc   <= "0000";
    outaddr_sig_uc   <= "0000";
    act_ok_uc        <=  "0000000000000";
    next_state_3     <= ST0;
    i                := 1;

  ELSE
```

```
  --  main_loop: loop
  CASE state_3 IS
   WHEN ST0 =>
     if ((broadcast_lock = '0') and (multicast_lock ='0')) then
       next_state_3 <= ST_A1;
     else
       next_state_3 <= ST0;
     end if;

     --      L1: FOR i in 1 to 13 loop    -- 13 Kanaele-loop
       WHEN ST_A1 =>
          IF active_uc(i)  = '1' THEN
            next_state_3 <= ST_B1;
          else
            next_state_3 <= ST_A2;      -- loop end, inc. i
          end IF;

          WHEN ST_B1 =>
            out_addr    := OA(i);
            OAd  := natural(CONV_INTEGER(out_addr));
            if belegt_reg(OAd)  = '0' then
              inp_addr := std_logic_vector(conv_unsigned(i, 4)); -- library fct
              inpaddr_sig_uc  <= inp_addr;
              outaddr_sig_uc  <= out_addr;
              set_ldconf  <= '1';  -- an switch process;
              next_state_3 <= ST_B2;
            else
              next_state_3 <= ST_B1;
            end if;

           WHEN ST_B2 =>
             if setldconf_ok  = '1' then
               act_ok_uc(i)    <= '1';
               set_ldconf  <= '0';  -- an switch process;
               inp_addr    := "0000";
               out_addr    := "0000";
               inpaddr_sig_uc    <= "0000";
               outaddr_sig_uc    <= "0000";
               next_state_3 <= ST_B3;
             else
               next_state_3 <= ST_B2;
             end if;

           WHEN ST_B3 =>
             if active_uc(i)  = '0' then
               act_ok_uc(i) <= '0';   -- verzoegere das Zuruecksetzten
               next_state_3 <= ST_A2;
             else
               next_state_3 <= ST_B3;
             end if;

        WHEN ST_A2 =>
           if i < 13  then
             i := i + 1;
           else
               i := 1;
           end if;
              next_state_3 <= ST0;

         when others =>
    end CASE;
  END IF;
END process;                              -- end unicast

multicast_proc: Process(sync_sig, rst) --  state_4)
   --  TYPE countertype  IS RANGE 0 TO 13;
   VARIABLE inp_addr, out_addr     :  STD_LOGIC_VECTOR(3 DOWNTO 0);
   VARIABLE OAd      :  NATURAL;
   variable i : integer := 0;
    --  variable count : countertype := 0;  -- Zaehler fuer Schleife

BEGIN -- Process
    --  Reset: Setze alle Signale zurueck ----------------------
   IF (rst = '1') THEN
    inp_addr       := "0000";
    out_addr       := "0000";
    OAd            := 0;
    -- broadcast_lock <= '0';
    inpaddr_sig    <= "0000";
    outaddr_sig    <= "0000";
    act_ok         <=  "0000000000000";
    set_conf       <=  '0';
    start_xmit     <=  "0000000000000"; -- for mcast, tells RC to begin xmit
    next_state_4   <= ST0;
    i := 1;
    multicast_lock <= '0';

  ELSE
  --  main_loop: loop
  CASE state_4 IS

   WHEN ST0 =>
    if  broadcast_lock  = '0' then
```

```
        next_state_4 <= ST_A1;
      else
        next_state_4 <= ST0;
    end if;

      WHEN ST_A1 =>
     --      L1: FOR i in 1 to 13 loop  ST_A gehoert zur For-loop
       IF active(i)  = '1' THEN
         multicast_lock  <= '1';
         inp_addr := std_logic_vector(conv_unsigned(i, 4)); -- library fct..
         next_state_4 <= ST_B1;
       else
          next_state_4 <= ST_A2;
       end if;

         WHEN ST_B1 =>
           IF multicast(i) = '1' then        --  While multicast(i) = '1' loop
             IF active(i)  = '1' THEN        --  If active-loop
               next_state_4 <= ST_C1;
             end if;
           else
             next_state_4 <= ST_B2;   -- check!
           end if;

          WHEN ST_C1 =>     -- While loop & act(i)-loop
            out_addr    := OA(i);
            OAd  := natural(CONV_INTEGER(out_addr));
            if belegt_reg(OAd)  = '0' then
              -- OA(i) wird immer neu gestezt von RC
              inpaddr_sig  <= inp_addr;
              outaddr_sig  <= out_addr;
              set_load  <= '1';  -- an switch process;
              next_state_4 <= ST_C2;
            else
              next_state_4 <= ST_C1;
            end if;

          WHEN ST_C2 =>
            if set_load_ok  = '1' then
              set_load  <= '0';  -- an switch process;
              act_ok(i)    <= '1';
              next_state_4 <= ST_C3;
            else
              next_state_4 <= ST_C2;
            end if;

          WHEN ST_C3 =>
            if active(i)  = '0' then
              act_ok(i) <= '0';
              next_state_4 <= ST_B1;
            else
              next_state_4 <= ST_C3;
            end if;                 -- end if active(i) = '1';

        WHEN ST_B2 =>
          set_conf  <= '1';  -- an switch process, set configure
          next_state_4 <= ST_B3;

        WHEN ST_B3 =>
           if setconf_ok  = '1'  then
              multicast_lock <= '0';
              set_conf <= '0';
              start_xmit(i) <= '1';
              next_state_4 <= ST_B4;
            else
              next_state_4 <= ST_B3;
            end if;

        WHEN ST_B4 =>
          if xmit_bsy(i) = '1'  then
            start_xmit(i) <= '0';
            next_state_4 <= ST_A2;
          else
            next_state_4 <= ST_B4;
          end if;

     WHEN ST_A2 =>
       if i < 13  then
         i := i + 1;
       else
         i := 1;
       end if;
         next_state_4 <= ST0;

         when others =>
    end CASE;
  END IF;
END process;


-- interface zum switch: zugeschnitten auf den TRIQUINT-switch
switch_proc: process(sync_sig, rst)  --  state_5)
```

```
   VARIABLE  verbind           : matrix;
   VARIABLE  vector_14         : STD_LOGIC_VECTOR(0 to 13) := "00000000000000";
   variable  i,m,k : integer range 0 to 13 := 0;  -- Laufvariable
   VARIABLE  OAd : NATURAL;
   VARIABLE  inp_addr, out_addr     :  STD_LOGIC_VECTOR(3 DOWNTO 0);

   CONSTANT verbind_zero : matrix := (zeros,zeros,zeros,zeros,zeros,
                                   zeros,zeros,zeros,zeros,zeros,
                                   zeros,zeros,zeros,zeros
                                  );

BEGIN
    -- reset first: reset hat Wait-statements.
    -- dafuer sollten die Zustaende ST00, ST01 ..  genommen werden

  IF (rst = '1') THEN
    load         <= '0';
    configure    <= '1';
    reset        <= '1';
    setres_ok    <= '0';
    setldconf_ok <= '0';
    setconf_ok   <= '0';
    set_load_ok   <= '0';
    cancel_ok    <= "00000000000000";
    belegt_reg   <= "00000000000000";
    i            := 1;
    m            := 0;
    OAd          := 0;
    k            := 1;
    inp_addr     := "0000";
    out_addr     := "0000";
    next_state_5 <= ST00;
    S_IA           <=  "0000";
    S_OA           <=  "0000";
    vector_14 := "00000000000000";

    CASE  state_5 IS
    -- Set Switch auf Passthru
    -- Reset Sequenz
    -- Warte einen Taktzyklus (mind. 7 ns)
     WHEN ST00 =>
      configure  <= '0';
      reset      <= '0';
      verbind := verbind_zero;    -- setzte die matrix insgesamt auf 0
      next_state_5 <= ST0;

     when others =>
    end CASE;

  ELSE

    CASE state_5 IS
--    main_loop: loop
     -- unicast handling
     WHEN ST0 =>                        -- anstatt ST2
      IF (set_ldconf = '1') then        -- unicast
        next_state_5 <= ST_A1;
      end if;

      IF (set_load = '1') then          -- multicast
        next_state_5 <= ST_B1;
      end if;

      IF (set_conf = '1') then          -- multicast
        next_state_5 <= ST_C1;
      end if;

      IF (set_res = '1') then           -- broadcast
        next_state_5 <= ST_D1;
      end if;

     -- IF (cancel /= zeros) then         -- clear belegt matrix & reg
     IF (cancel /= three_zeros) then    -- temp. auf 3 reduziert
        next_state_5 <= ST_E1;
      end if;

    WHEN ST_A1 =>
         out_addr := outaddr_sig_uc;
         inp_addr := inpaddr_sig_uc;
         m    :=  natural(CONV_INTEGER(inp_addr));
         OAd  :=  natural(CONV_INTEGER(out_addr));
         S_IA      <= inp_addr;
         S_OA      <=  out_addr;
         load <= '1';
         configure <= '1';
         vector_14 := zeros;
         vector_14(OAd) := '1';
         verbind(m) :=  verbind(m) or vector_14;
         belegt_reg(OAd) <= '1';
         next_state_5 <= ST_A2;

       -- Warte einen Takt lang, mind. 7 ns
    WHEN ST_A2 =>
```

```
        load <= '0';
        configure <= '0';
       --  reset <= '0';
        setldconf_ok <= '1';
        next_state_5 <= ST_A3;

    WHEN ST_A3 =>
        if set_ldconf = '0' then
          setldconf_ok <= '0';
          next_state_5 <= ST0;          -- war ST2
        else
          next_state_5 <= ST_A3;
        end if;

--      IF (set_load = '1')             -- multicast load command
   when ST_B1 =>
        load <= '1';
        -- Write belegt_reg und verbind-matrix
        out_addr := outaddr_sig;
        inp_addr := inpaddr_sig;
        S_IA      <= inp_addr;
        S_OA      <=  out_addr;
        m    :=  natural(CONV_INTEGER(inp_addr));
        OAd  :=  natural(CONV_INTEGER(out_addr));

        vector_14 := zeros;
        vector_14(OAd) := '1';
        verbind(m) :=  verbind(m) or vector_14;
        belegt_reg(OAd) <= '1';
        next_state_5 <= ST_B2;

    when ST_B2 =>
       load <= '0';
       set_load_ok <= '1';
       next_state_5 <= ST_B3;

    when ST_B3 =>
       if set_load = '0' then
         set_load_ok <= '0';
         next_state_5 <= ST0;
         else
           next_state_5 <= ST_B3;
       end if;

--      IF (set_conf = '1')    -- multicast configure command
    when ST_C1 =>
        configure <= '1';
        next_state_5 <= ST_C2;

    when ST_C2 =>
        configure <= '0';
        setconf_ok <= '1';
        next_state_5 <= ST_C3;

    when ST_C3 =>
       if set_conf = '0' then
         setconf_ok <= '0';
         next_state_5 <= ST0;           -- war ST2
       else
         next_state_5 <= ST_C3;
       end if;

--      IF (set_res = '1')  -- broadcast configure command
    when ST_D1 =>
         reset <= '1';
         belegt_reg <=  "11111111111111";
         next_state_5 <= ST_D2;

    when ST_D2 =>
       -- Warte einen Takt lang, mind. 7 ns
         reset <= '0';
         setres_ok <= '1';
         next_state_5 <= ST_D3;

       --  wait until Clk_com'event and Clk_com = '1';

    when ST_D3 =>
         if set_res = '0' then
           setres_ok <= '0';
           next_state_5 <= ST0;           -- war ST2
         else
           next_state_5 <= ST_D3;
         end if;

---------------------------------------------------------

    when ST_E1 =>
       -- clear busy-reg (belegt_reg) and verbind(i) matrix
       --      IF (cancel /= zeros) THEN
        IF  cancel(0) = '1' THEN   -- broadcast - Abfrage
           belegt_reg  <= "00000000000000";
           cancel_ok(0) <= '1';
           next_state_5 <= ST_E2;
```

```
          else
           next_state_5 <= ST_E3;
        end if;

    when ST_E2 =>
           if cancel(0) = '0' then
             cancel_ok(0) <= '0';
             next_state_5 <= ST_E3;       -- war ST2
             else
              next_state_5 <= ST_E2;
           end if;

    when ST_E3 =>                   -- war ST3
       --     L2: FOR k in 1 to 13 loop  -- kanaele loop
            L2: FOR i in 1 to 2 loop  -- kanaele loop reduziert auf 2
             IF  cancel(i) = '1' THEN
                -- Loesche die Belegungenn im belegt-reg
                belegt_reg <= belegt_reg and not verbind(i);
                -- Loesche Belegung in der Verbind-matrix
                verbind(i) := zeros;
                cancel_ok(i) <= '1';
                next_state_5 <= ST_E4;  -- war E3
                k := i;
                exit;
              else
                next_state_5 <= ST0;
              end if;
           end loop;

     when ST_E4 =>
            if cancel(k) = '0' then
                 cancel_ok(k) <= '0';
                 next_state_5 <= ST0;    -- war ST_E2
               else
                 next_state_5 <= ST_E4;
               end if;

    when others =>
   end CASE;
  END IF;
 end process;


 Sync_Proc: PROCESS
  BEGIN
     if rst = '1'  then
         state_0 <= ST0;
         state_1 <= ST0;
         state_2 <= ST0;
         state_3 <= ST0;
         state_4 <= ST0;
         state_5 <= ST00;
         sync_sig <= '0';
         Wait until Clk_com'EVENT AND Clk_com = '1';

       else
 if sync_sig = '0' then
           sync_sig <= '1';
         else
           sync_sig <= '0';
         end if;
    state_0 <= next_state_0;
         state_1 <= next_state_1;
         state_2 <= next_state_2;
         state_3 <= next_state_3;
         state_4 <= next_state_4;
         state_5 <= next_state_5;
        Wait until Clk_com'EVENT AND Clk_com = '1';
      end if;
  END Process;

END rtl; -- End architecture
```

7.6 Datenaufnahme-Modul RD

7.6.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Science,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Autor:  Walter Lange, Oct. 1998 and April 1999
--
```

```
-- Verhaltensbeschreibung des rd (receive data) fuer
-- Simulation mit quicksim.
-- Synthese mit Monet
-- rd_mon.vhd nimmt zunaechst in einem 5-Bit Schieberegister
-- die Daten aus dem Switch auf.
-- Das Interface muss gegenueber rd.vhd geaendert werden,
-- da wegen des BC auch hier das SR ausgelagert wird.
-- das SR liefert 48 bit Daten parallel.
-- Die ersten 5 bit tragen das Startzeichen, die dekodiert werden.
-- Der decoder ist ein 5B/4B  Decoder.
-- Jede Zelle beginnt mit einem Start-Of-Cell Zeichen ('11001')
-- Jedes der 12 folgenden Zellworte (mit 48 Bit) der Payload
-- beginnt mit einem Start-Of-Payload Zeichen ('10001') (reg-start)
-- Die letzten 3 bits des Zellworts werden ignoriert.
-- Der 5B/4B decoder prueft auf Fehler: Wenn ein unerlaubtes
-- Zeichen kommt, wird das Signal ``error'' ungleich Null
-- In diesem Fall wird das Schreiben in den FIFO-OUT Buffer untrdrueckt..
-- Nach jeder Zelluebertragung muessen wenigstens 2 Leertakte kommen
-- rd.vhd  ist in 2 Prozesse aufgeteilt:
-- RDIN nimmt die Daten (rcell_data) auf, im Unterschied zu rd.vhd
-- aus einem 5 bit Schieberegister (hilf5),
-- dekodiert sie und setzt sie in ein 32 Bit Reg (out_reg).
-- RDOUT schreibt das out_reg in das out_fifo mit rcell_rdy, rcell_bus
-- Der Takt wird geaendert zu Clk_com (statt Clk_transmit)
-- Monet doesn't like BOOLEAN Signals
-- Autor: W. Lange 7. 8. 98

Library IEEE;
USE IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;


ENTITY rd IS
  PORT (Clk_com       : IN  STD_LOGIC;
       Reset       : IN  STD_LOGIC;         -- fuer BC eingefuehrt
       sr_strobe      : IN  STD_LOGIC;
       sr_data        : STD_LOGIC_VECTOR(47 DOWNTO 0);
       srdta_taken    : OUT STD_LOGIC;
       rcell_rdy      : OUT STD_LOGIC;
       rcell_fetch    : IN  STD_LOGIC;
       rbuffer_full   : IN  STD_LOGIC;
       rcell_bus      : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END rd;
---------------------------------
Use WORK.utilsrd.all;

ARCHITECTURE rtl OF rd IS
  SIGNAL out_reg      :  STD_LOGIC_VECTOR(31 downto 0);
--  SIGNAL write_fifo   :  BOOLEAN;
  SIGNAL write_fifo   :  STD_LOGIC;
  SIGNAL error_h      :  STD_LOGIC_VECTOR(4 downto 0);
  SIGNAL error_pl     :  STD_LOGIC_VECTOR(4 downto 0);
-- SIGNAL wr_ack       :  BOOLEAN;
  SIGNAL wr_ack       :  STD_LOGIC;

BEGIN

  RDIN: PROCESS
  -- Der RDIN Process nimmt die Zelle vom Switch auf, dekodiert sie,
  -- und gibt sie an den FIFO-OUT weiter.

   VARIABLE hilf_reg     :  STD_LOGIC_VECTOR(47 downto 0);
   VARIABLE hilf5        :  STD_LOGIC_VECTOR(4 downto 0);
   VARIABLE err,err_hold :  STD_LOGIC_VECTOR(4 downto 0);
   VARIABLE hilf4        :  STD_LOGIC_VECTOR(3 downto 0);
   VARIABLE out_reg_h    :  STD_LOGIC_VECTOR(31 downto 0);
   CONSTANT start_cell   :  STD_LOGIC_VECTOR(4 downto 0):= "11001";
   CONSTANT start_payl   :  STD_LOGIC_VECTOR(4 downto 0):= "10001";
   CONSTANT header_err   :  STD_LOGIC_VECTOR(4 downto 0):= "00001";
   CONSTANT payl_err     :  STD_LOGIC_VECTOR(4 downto 0):= "00010";

-- Bitlokationen von out_reg und Zellwort
--                    31  28  27 24 23 20 19 16 15 12 11 8  7  4  3  0
-- out_reg :=          "0000  0000  0000  0000  0000  0000  0000  0000";
--              47  43 42 38 37 33 32 28 27 23 22 18 17 13 12  8 7   3 2 0
-- Zellwort  := "11001 00000 00000 00000 00000 00000 00000 00000 00000 000";


  BEGIN -- Process
  --  Reset: Setze alle Variablen und Signale zurueck ----------------------
    srdta_taken <= '0';
    hilf_reg    := "000000000000000000000000000000000000000000000000";
    err         := "00000";
    err_hold    := "00000";
    write_fifo  <= '0';
    error_h     <= "00000";
    error_pl    <= "00000";
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1'; -- noetig, sonst ist
-- moeglicherweise eine loop ohne wait's
  --  eine neue Zelle kommt an...

  main:  loop    -- main loop ---------------------------
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and rbuffer_full = '0';
```

```
    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and sr_strobe  = '1';
    -- Der Header ist da..     ------------------------------

    hilf_reg := sr_data;
    srdta_taken <= '1';

    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and sr_strobe  = '0';

    srdta_taken <= '0';

    IF hilf_reg(47 downto 43) = start_cell THEN
    -- Das Startzeichen ist da, sammle den Header auf..
         hilf5 := hilf_reg(42 downto 38);
         out_reg_h(31 downto 28) := decode(hilf5);
         hilf5 := hilf_reg(37 downto 33);
         out_reg_h(27 downto 24) := decode(hilf5);
         hilf5 := hilf_reg(32 downto 28);
         out_reg_h(23 downto 20) := decode(hilf5);
         hilf5 := hilf_reg(27 downto 23);
         out_reg_h(19 downto 16) := decode(hilf5);
         hilf5 := hilf_reg(22 downto 18);
         out_reg_h(15  downto 12) := decode(hilf5);
         hilf5 := hilf_reg(17 downto 13);
         out_reg_h(11 downto 8) := decode(hilf5);
         hilf5 := hilf_reg(12 downto 8);
         out_reg_h(7 downto 4) := decode(hilf5);
         hilf5 := hilf_reg(7 downto 3);
         out_reg_h(3 downto 0) := decode(hilf5);

-- IF err = "00000" THEN write_fifo <= '1'; END IF; -- fuer Fehlerbehandlung
      write_fifo <= '1';
      error_h <= err_hold;
      out_reg <=  out_reg_h;

    ELSE error_h <= header_err;

    END IF;

    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and (wr_ack  = '1');
    write_fifo <= '0';
    -- Wait UNTIL Clk_com'EVENT AND Clk_com = '1';

   -- Sammle die Payload auf

   L3 : FOR k in 1 to 12 loop

    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and (sr_strobe = '1');
    -- Das Zellwort ist da..   ----------------------

    hilf_reg := sr_data;
    srdta_taken <= '1';

    Wait UNTIL Clk_com'EVENT AND Clk_com = '1' and (sr_strobe = '0');
    srdta_taken <= '0';

     IF  hilf_reg(47 downto 43)  = start_payl THEN
    -- Das Startzeichen ist da, sammle das Zellwort auf..
         hilf5 := hilf_reg(42 downto 38);
         out_reg_h(31 downto 28) := decode(hilf5);
         hilf5 := hilf_reg(37 downto 33);
         out_reg_h(27 downto 24) := decode(hilf5);
         hilf5 := hilf_reg(32 downto 28);
         out_reg_h(23 downto 20) := decode(hilf5);
         hilf5 := hilf_reg(27 downto 23);
         out_reg_h(19 downto 16) := decode(hilf5);
         hilf5 := hilf_reg(22 downto 18);
         out_reg_h(15  downto 12) := decode(hilf5);
         hilf5 := hilf_reg(17 downto 13);
         out_reg_h(11 downto 8) := decode(hilf5);
         hilf5 := hilf_reg(12 downto 8);
         out_reg_h(7 downto 4) := decode(hilf5);
         hilf5 := hilf_reg(7 downto 3);
         out_reg_h(3 downto 0) := decode(hilf5);

-- IF err = "00000" THEN write_fifo <= TRUE; END IF; -- fuer Fehlerbehandlung


       write_fifo <= '1';
       error_pl <= err_hold;
       out_reg <=  out_reg_h;

    ELSE error_pl <= payl_err;

    END IF;

    Wait UNTIL Clk_com'EVENT AND Clk_com  = '1' and (wr_ack = '1');
    write_fifo <= '0';

   End loop;     -- 1 to 12 loop
   -- WAIT UNTIL  Clk_com'EVENT and Clk_com = '1';
   End loop;     -- Main Loop
  End Process;
```

```
  RDOUT: PROCESS
  -- Der RDOUT Process laedt das out_register in den rd-FIFO
  -- und kommuniziert mit dem rd-FIFO
  -- Achtung, Kommunikation zwischen RDIN und RDOUT
  -- ueber write_fifo ist synchron. Das kann Probleme geben, wenn
  -- rcell_fetch zu lange ausbleibt.
  -- rcell_fetch
  -- sollte innerhalb von 4 Takten nach rcell_rdy kommen.

  VARIABLE Cell_reg    : UNSIGNED(31 downto 0);  -- Reg fuer Cell Bytes

  BEGIN -- Process
  --  Reset: Setze alle Variablen und Signale zurueck ----------------------
    rcell_rdy  <= '0';
    rcell_bus <= "00000000000000000000000000000000";
    wr_ack <= '0';
    Wait UNTIL Clk_com'EVENT AND Clk_com  = '1';

   main: loop     -- main loop ---------------------------
     WAIT UNTIL  Clk_com'EVENT and Clk_com = '1' and (rbuffer_full = '0');

     WAIT UNTIL  Clk_com'EVENT and Clk_com = '1' and (write_fifo  = '1');
     rcell_rdy <= '1';
     rcell_bus <= out_reg;
     wr_ack    <= '1';

     WAIT UNTIL  Clk_com'EVENT and Clk_com = '1' and (write_fifo  = '0');
     wr_ack <= '0';

     WAIT UNTIL  Clk_com'EVENT and Clk_com = '1' and (rcell_fetch  = '1');
     rcell_rdy <= '0';
     Wait UNTIL Clk_com'EVENT AND Clk_com  = '1';

   END loop; -- main loop
  END Process;
 END rtl;
```

RD-Package

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Oct. 1998 and May 1999
--
-- Geaendert: procedure zu function und
-- subtype slv4 eingefuehrt (17. 9. 98)
-- preserve_function eingefuehrt
---------------------------------------------------
Library IEEE;
USE IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;

package utilsrd is

  subtype slv4 is STD_LOGIC_VECTOR(3 downto 0);
  function decode(v: STD_LOGIC_VECTOR(4 downto 0)) return slv4;

end;

---------------------------------------------------

package body utilsrd is

  function decode(v: STD_LOGIC_VECTOR(4 downto 0)) return slv4 IS
--
-- Encapsulation of logic below according to sold BC users guide
-- Optimizing Timing and Area
-- return type has changed from std_logic_vector to slv5
--
-- synopsys preserve_function
--

  VARIABLE vector : STD_LOGIC_VECTOR(4 downto 0);
--  VARIABLE err :    STD_LOGIC_VECTOR(4 downto 0);
  VARIABLE ret :    STD_LOGIC_VECTOR(3 downto 0);

  begin
    vector := v;
--    err := "00000";
    CASE vector IS
WHEN  "11110" => ret :=  "0000";
WHEN  "01001" => ret :=  "0001";
        WHEN  "10100" => ret :=  "0010";
WHEN  "10101" => ret :=  "0011";
WHEN  "01010" => ret :=  "0100";
        WHEN  "01011" => ret :=  "0101";
        WHEN  "01110" => ret :=  "0110";
WHEN  "01111" => ret :=  "0111";
WHEN  "10010" => ret :=  "1000";
WHEN  "10011" => ret :=  "1001";
```

```
WHEN  "10110" => ret :=  "1010";
WHEN  "10111" => ret :=  "1011";
WHEN  "11010" => ret :=  "1100";
WHEN  "11011" => ret :=  "1101";
WHEN  "11100" => ret :=  "1110" ;
WHEN  "11101" => ret :=  "1111";
  --  WHEN OTHERS   => err :=    v; -- Probleme mit v_parser
        WHEN OTHERS   =>
     END CASE;

    return ret;
  end decode;
end utilsrd;

---------------------------------------------------
```

7.6.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Oct. 1998 and May 1999
--
-- Verhaltensbeschreibung des rd (receive data) fuer
-- Simulation mit Synopsys.
-- rd_rtl.vhd nimmt zunaechst in einem 5-Bit Schieberegister
-- die Daten aus dem Switch auf.
-- Das Interface muss gegenueber rd.vhd geaendert werden,
-- da wegen des BC auch hier das SR ausgelagert wird.
-- das SR liefert 48 bit Daten parallel.
-- Die ersten 5 bit tragen das Startzeichen, die dekodiert werden.
-- Der decoder ist ein 5B/4B  Decoder.
-- Jede Zelle beginnt mit einem Start-Of-Cell Zeichen ('11001')
-- Jedes der 12 folgenden Zellworte (mit 48 Bit) der Payload
-- beginnt mit einem Start-Of-Payload Zeichen ('10001') (reg-start)
-- Die letzten 3 bits des Zellworts werden ignoriert.
-- Der 5B/4B decoder prueft auf Fehler: Wenn ein unerlaubtes
-- Zeichen kommt, wird das Signal ``error'' ungleich Null
-- In diesem Fall wird das Schreiben in den FIFO-OUT Buffer untrdrueckt..
-- Nach jeder Zelluebertragung muessen wenigstens 2 Leertakte kommen
-- rd.vhd  ist in 2 Prozesse aufgeteilt:
-- RDIN nimmt die Daten (rcell_data) auf, im Unterschied zu rd.vhd
-- aus einem 5 bit Schieberegister (hilf5),
-- dekodiert sie und setzt sie in ein 32 Bit Reg (out_reg).
-- RDOUT schreibt das out_reg in das out_fifo mit rcell_rdy, rcell_bus
-- Der Takt wird geaendert zu Clk_com
-- Autor: W. Lange 27. 9. 2000
-- 17. 9. 98: Loops (L1,L2) entfernt
-- Umgeschrieben zur rtl-Version: 4. Mai 1999
-- sync_sig eingefuehrt

Library IEEE;
USE IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;

ENTITY rd IS
  PORT (Clk_com       : IN  STD_LOGIC;
       Reset       : IN  STD_LOGIC;         -- fuer BC eingefuehrt
       sr_strobe      : IN  STD_LOGIC;
       sr_data        : STD_LOGIC_VECTOR(47 DOWNTO 0);
       srdta_taken    : OUT STD_LOGIC;
       rcell_rdy      : OUT STD_LOGIC;
       rcell_fetch    : IN  STD_LOGIC;
       rbuffer_full   : IN  STD_LOGIC;
       rcell_bus      : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END rd;
---------------------------------
Use WORK.utilsrd.all;

ARCHITECTURE rtl OF rd IS
  SIGNAL out_reg      :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL write_fifo   :  STD_LOGIC;
  SIGNAL error_h      :  STD_LOGIC_VECTOR(4 downto 0);
  SIGNAL error_pl     :  STD_LOGIC_VECTOR(4 downto 0);
  SIGNAL wr_ack       :  STD_LOGIC;

  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4,
                        ST_A1,ST_A2,ST_A3,ST_A4,ST_A5
                        );
  SIGNAL state_0, state_1, next_state_0, next_state_1 : state_type;
  signal sync_sig : STD_LOGIC;          -- Sync signal

BEGIN

  RDIN: PROCESS(sync_sig, Reset)
  -- Der RDIN Process nimmt die Zelle vom Switch auf, dekodiert sie,
```

```
  -- und gibt sie an den FIFO-OUT weiter.

   VARIABLE hilf_reg     :  STD_LOGIC_VECTOR(47 downto 0);
   VARIABLE hilf5        :  STD_LOGIC_VECTOR(4 downto 0);
   VARIABLE err,err_hold :  STD_LOGIC_VECTOR(4 downto 0);
   VARIABLE out_reg_h    :  STD_LOGIC_VECTOR(31 downto 0);
   CONSTANT start_cell   :  STD_LOGIC_VECTOR(4 downto 0):= "11001";
   CONSTANT start_payl   :  STD_LOGIC_VECTOR(4 downto 0):= "10001";
   CONSTANT header_err   :  STD_LOGIC_VECTOR(4 downto 0):= "00001";
   CONSTANT payl_err     :  STD_LOGIC_VECTOR(4 downto 0):= "00010";

   TYPE countertype  IS RANGE 0 TO 12;
   VARIABLE count      :  countertype;

-- Bitlokationen von out_reg und Zellwort
--                    31  28  27 24 23 20 19 16 15 12 11 8  7  4  3  0
-- out_reg :=          "0000  0000  0000  0000  0000  0000  0000  0000";
--              47  43 42 38 37 33 32 28 27 23 22 18 17 13 12  8 7   3 2 0
-- Zellwort  := "11001 00000 00000 00000 00000 00000 00000 00000 00000 000";


  BEGIN -- Process
  --  Reset: Setze alle Variablen und Signale zurueck ----------------------

    IF (Reset = '1') THEN
      srdta_taken <= '0';
      hilf_reg    := "000000000000000000000000000000000000000000000000";
      err         := "00000";
      err_hold    := "00000";
      write_fifo  <= '0';
      error_h     <= "00000";
      error_pl    <= "00000";
      count       := 1;

     ELSE

     --  eine neue Zelle kommt an...
     CASE state_0 IS
WHEN ST0 =>
          IF (rbuffer_full = '0') THEN
            next_state_0 <= ST1;
        ELSE
            next_state_0 <= ST0;
        END IF;


        WHEN ST1 =>
  IF (sr_strobe = '1') THEN
            -- Der Header ist da..
            hilf_reg := sr_data;
            srdta_taken <= '1';
    next_state_0 <= ST2;
  ELSE
            next_state_0 <= ST1;
          END IF;

WHEN ST2 =>
  IF (sr_strobe = '0') THEN
            srdta_taken <= '0';
    next_state_0 <= ST3;
  ELSE
            next_state_0 <= ST2;
          END IF;

  WHEN ST3 =>
          IF hilf_reg(47 downto 43) = start_cell THEN
            -- Das Startzeichen ist da, sammle den Header auf..
            hilf5 := hilf_reg(42 downto 38);
            out_reg_h(31 downto 28) := decode(hilf5);
            hilf5 := hilf_reg(37 downto 33);
    out_reg_h(27 downto 24) := decode(hilf5);
    hilf5 := hilf_reg(32 downto 28);
    out_reg_h(23 downto 20) := decode(hilf5);
       hilf5 := hilf_reg(27 downto 23);
    out_reg_h(19 downto 16) := decode(hilf5);
    hilf5 := hilf_reg(22 downto 18);
    out_reg_h(15  downto 12) := decode(hilf5);
    hilf5 := hilf_reg(17 downto 13);
    out_reg_h(11 downto 8) := decode(hilf5);
    hilf5 := hilf_reg(12 downto 8);
    out_reg_h(7 downto 4) := decode(hilf5);
    hilf5 := hilf_reg(7 downto 3);
    out_reg_h(3 downto 0) := decode(hilf5);

            -- IF err /= "00000" THEN err_hold := err; END IF;
            -- IF err = "00000" THEN write_fifo <= '1'; END IF; --Err.handlg.
            write_fifo <= '1';
            error_h <= err_hold;
            out_reg <=  out_reg_h;

        ELSE
            error_h <= header_err;
        END IF;
```

```
            next_state_0 <= ST4;

WHEN ST4 =>
  IF wr_ack = '1'  THEN
            write_fifo <= '0';
    next_state_0 <= ST_A1;
  ELSE
            next_state_0 <= ST4;
          END IF;

         -- Sammle die Payload auf
         -- FOR k in 1 to 12 loop
         WHEN ST_A1 =>
  IF (sr_strobe = '1')  THEN
            -- Das Zellwort ist da..
            hilf_reg := sr_data;
            srdta_taken <= '1';
    next_state_0 <= ST_A2;
  ELSE
            next_state_0 <= ST_A1;
          END IF;

         WHEN ST_A2 =>
  IF (sr_strobe = '0')  THEN
            srdta_taken <= '0';
    next_state_0 <= ST_A3;
  ELSE
            next_state_0 <= ST_A2;
          END IF;

          WHEN ST_A3 =>
              IF  hilf_reg(47 downto 43)  = start_payl THEN
                -- Das Startzeichen ist da, sammle das Zellwort auf..
                hilf5 := hilf_reg(42 downto 38);
                out_reg_h(31 downto 28) := decode(hilf5);
                hilf5 := hilf_reg(37 downto 33);
                out_reg_h(27 downto 24) := decode(hilf5);
                hilf5 := hilf_reg(32 downto 28);
          out_reg_h(23 downto 20) := decode(hilf5);
                hilf5 := hilf_reg(27 downto 23);
                out_reg_h(19 downto 16) := decode(hilf5);
                hilf5 := hilf_reg(22 downto 18);
                out_reg_h(15  downto 12) := decode(hilf5);
                hilf5 := hilf_reg(17 downto 13);
                out_reg_h(11 downto 8) := decode(hilf5);
                hilf5 := hilf_reg(12 downto 8);
                out_reg_h(7 downto 4) := decode(hilf5);
                hilf5 := hilf_reg(7 downto 3);
                out_reg_h(3 downto 0) := decode(hilf5);
        -- IF err = "00000" THEN write_fifo <= '1'; END IF; --Err.hdlg..
        -- IF err /= "00000" THEN err_hold := err; END IF;
                write_fifo <= '1';
                error_pl <= err_hold;
                out_reg <=  out_reg_h;
              ELSE
                error_pl <= payl_err;
              END IF;
              next_state_0 <= ST_A4;

          WHEN ST_A4 =>
    IF wr_ack = '1'  THEN
              write_fifo <= '0';
      next_state_0 <= ST_A5;
    ELSE
              next_state_0 <= ST_A4;
            END IF;

           WHEN ST_A5 =>
             IF count < 12 THEN
               count := count + 1;
               next_state_0 <= ST_A1;
             ELSE
               next_state_0 <= ST0;
               count := 1;
             END IF;
     END Case;
    END IF;
  End Process;


  RDOUT: PROCESS(sync_sig, Reset)
  -- Der RDOUT Process laedt das out_register in den rd-FIFO
  -- und kommuniziert mit dem rd-FIFO
  -- Achtung, Kommunikation zwischen RDIN und RDOUT
  -- ueber write_fifo ist synchron. Das kann Probleme geben, wenn
  -- rcell_fetch zu lange ausbleibt.
  -- rcell_fetch
  -- sollte innerhalb von 4 Takten nach rcell_rdy kommen.

  VARIABLE Cell_reg    : UNSIGNED(31 downto 0);  -- Reg fuer Cell Bytes

  BEGIN -- Process
      --  Reset: Setze alle Variablen und Signale zurueck ------------
```

```
     if Reset = '1' then
      rcell_rdy  <= '0';
      rcell_bus <= "00000000000000000000000000000000";
      wr_ack <= '0';

     else

     CASE state_1 IS
WHEN ST0 =>
        -- Start state
           IF (rbuffer_full = '0') THEN
             next_state_1 <= ST1;
           ELSE
             next_state_1 <= ST0;
           END IF;

WHEN ST1 =>
           IF (write_fifo = '1') THEN
             rcell_rdy <= '1';
             rcell_bus <= out_reg;
             wr_ack    <= '1';
             next_state_1 <= ST2;
           ELSE
             next_state_1 <= ST1;
           END IF;

WHEN ST2 =>
           IF (write_fifo = '0') THEN
             wr_ack    <= '0';
             next_state_1 <= ST3;
           ELSE
             next_state_1 <= ST2;
           END IF;

WHEN ST3 =>
           IF (rcell_fetch = '1') THEN
             rcell_rdy <= '0';
             next_state_1 <= ST0;
           ELSE
             next_state_1 <= ST3;
           END IF;

        WHEN OTHERS =>
     END Case;
    end if;
   END Process;

  Sync_Proc: PROCESS
  BEGIN
        if Reset = '1' then
          state_0 <= ST0;
          state_1 <= ST0;
          sync_sig <= '0';
          Wait until Clk_com'EVENT AND Clk_com = '1';
else
          if sync_sig = '0' then
            sync_sig <= '1';
          else
            sync_sig <= '0';
          end if;
  state_0 <= next_state_0;
          state_1 <= next_state_1;
          Wait until Clk_com'EVENT AND Clk_com = '1';
        end if;
  END Process;

 END rtl;
```

7.7 AHT-Ausgangs-Modul

7.7.1 Eingangsbeschreibung

VHDL-Verhaltensbeschreibung

```
--
-- Copyright University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Science,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Autor:  Walter Lange, August 2000
--
-- Verhaltensbeschreibung des ahtout fuer
-- Synthese
-- aht_out besteht aus 2 Prozessen:
-- AHTOUT holt Zell-Daten aus dem FIFO_out-Speicher und
-- und gibt sie an den PROCESS AHTOUT_Transmit weiter
```

```
-- AHTOUT_Transmit gibt die Daten in 8-Bit Paketen (an die PHYS-Schicht)
-- auf die Leitung.
-- Es werden  53 Byte uebertragen (5 Byte fuer den Header, HEC:
-- Wiederholung des vorhergehenden Bytes)
-- Es wird davon ausgegengen, dass der HEC in der PHYS-Schicht
-- eingefuegt wird.
-- Autor: W. Lange 29. 8. 2000
-- Reset: Da die reset Funktion automatisch generiert
-- wird, ist lediglich das Reset Signal in der entity aufgefuehrt

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

ENTITY ahtout IS
  PORT (Clk_aht_out    : IN  STD_LOGIC;
        Cell_Sync_out  : OUT STD_LOGIC;
        Cell_preSync   : OUT STD_LOGIC;
        Chan_bsy       : IN  STD_LOGIC;
        Reset          : IN STD_LOGIC;
        Data_out       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        rnew_cell      : OUT STD_LOGIC;
        rcell_data     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        rcell_read     : OUT STD_LOGIC;
        rcell_write    : IN  STD_LOGIC);
END ahtout;
--------------------------------

ARCHITECTURE ahtoutar OF ahtout IS
  SIGNAL transmit     : STD_LOGIC;
  SIGNAL cell_reg     : STD_LOGIC_VECTOR(31 downto 0);  --Reg fuer Cellwort
  SIGNAL cell_reg2    : STD_LOGIC_VECTOR(31 downto 0);  --Reg fuer Cellwort

BEGIN

  AHTOUT: PROCESS
  -- Der AHTOUT Process holt Zell-Daten aus dem FIFO_out-Speicher und
  -- und gibt sie an den Process AHTOUT_TRANSMIT weiter

  BEGIN -- Process
  --  Reset: Setze alle Variablen und Signale zurueck ----------------------
    rcell_read     <= '0';
    rnew_cell <= '0';
    transmit <= '0';
   Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';

   main_loop:  loop    -- main loop --------------------------

     Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1' and Chan_bsy = '0';
     -- Der Ausgangskanal ist frei, hole eine Zelle aus dem FIFO-Speicher.
     -- Uebertrage erst den Header
      rcell_read <= '1';
      rnew_cell  <= '1';

    Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1' and rcell_write='1';
      cell_reg <= rcell_data;  -- wird mit rcell_write uebernommen
      rcell_read <= '0';
      transmit <= '1';
      rnew_cell  <= '0';

    Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 1. Byte  (3)
    Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 2. Byte  (4)
      transmit <= '0';
     -- Wiederhole 12 mal fuer 32 - Bit-Register (48 Byte)

     L1: FOR i in 1 to 6 loop
       rcell_read <= '1';                       -- 3. Byte (1) rcell_read
       -- Warte auf die Antwort des Speicher-Mgmt's

       Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1' and rcell_write='1';
       cell_reg2 <= rcell_data;                   -- 4. Byte
       rcell_read <= '0';

       Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 1. Byte
       Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 2. Byte
--     Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 3. Byte
       rcell_read <= '1';
       Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1' and rcell_write='1';
       cell_reg <= rcell_data;                   -- 4. Byte
       rcell_read <= '0';
       Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 1. Byte
       Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1'; -- 2. Byte

     End loop;
  End loop;  -- main loop
  End Process;

  AHTOUT_Transmit: PROCESS
     VARIABLE hilf1         : STD_LOGIC_VECTOR(31 downto 0);
     VARIABLE hilf2         : STD_LOGIC_VECTOR(31 downto 0);

  BEGIN   -- Process
    -- Reset Part
```

```
    Cell_Sync_out  <= '0';
    Cell_preSync   <= '0';
    Data_out       <= "00000000";
    hilf1 := "00000000000000000000000000000000";
    hilf2 := "00000000000000000000000000000000";

    Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';

    main_loop: loop

        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1' and transmit = '1';
        Cell_preSync <= '1';
        hilf1 := cell_reg;
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
        Cell_Sync_out <= '1';
      -- Transmit Bytewise den Header
data_out <= hilf1(31 downto 24);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
data_out <= hilf1(23 downto 16);
        Cell_Sync_out <= '0';
        Cell_preSync <= '0';
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
data_out <= hilf1(15 downto 8);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
data_out <= hilf1(7 downto 0);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
       -- Das HEC Byte ist die Wiederholung der vorhergehenden Daten
        hilf2 := cell_reg2;
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
     -- Transmit Bytewise die Payload

     L2: FOR k IN 1 TO 6 loop  -- 12 * 32 Bit register

data_out <= hilf2(31 downto 24);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
data_out <= hilf2(23 downto 16);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
data_out <= hilf2(15 downto 8);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
data_out <= hilf2(7 downto 0);
        hilf1 := cell_reg;
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';

        data_out <= hilf1(31 downto 24);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
        data_out <= hilf1(23 downto 16);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
        data_out <= hilf1(15 downto 8);
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
        data_out <= hilf1(7 downto 0);
        hilf2 := cell_reg2;
        Wait UNTIL Clk_aht_out'EVENT AND Clk_aht_out = '1';
    end loop; -- 1 to 6 loop

   End loop; -- Main Loop
  End Process;

 END ahtoutar;
```

7.7.2 Ausgangsbeschreibung

RTL-Verhaltensbeschreibung

```
--
-- Copyright 1999 University of Tuebingen,
-- Wilhelm Schickard Institute for Computer Sciences,
-- Dept. of  Computer Engineering, Prof. Dr. W. Rosenstiel.
-- Author:  Walter Lange, Oct. 1998, May 1999, Sept 2000
--
-- Verhaltensbeschreibung des ahtout fuer
-- Simulation mit Synopsys. Compilierung mit Synopsys analyze/elaborate
-- Synthese mit DC
-- ahaout_rtl besteht aus 2 Prozessen (+ Sync-Prozess)
-- AHTOUT holt Zell-Daten aus dem FIFO_out-Speicher und
-- und gibt sie an den PROCESS AHTOUT_Transmit weiter
-- AHTOUT_Transmit gibt die Daten in 8-Bit Paketen (an die PHYS-Schicht)
-- auf die Leitung.
-- AHTOUT gibt einen Trigger (transmit_new) an AHTOUT_Transmit
-- sonst laufen die beiden Prozesse ohne Kommunikation, nur
-- synchronisiert durch den Takt.
-- Es werden nur 53 Byte uebertragen (5 Byte des Headers als HEC
-- aber als Wiederholung des 4. Bytes)
-- Es wird davon ausgegengen, dass der HEC in der PHYS-Schicht
-- eingefuegt wird.
-- Aenderungen: 1. reset wird eingefuegt
-- Autor: W. Lange Sept 2000
-- Fuer die Synthese. Arbeitet mit "Doppel-loops"
```

```
-- Endgueltige Beschreibung Sept 2000
-- sync_sig introduced

Library IEEE;
USE IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


ENTITY ahtout IS
  PORT (Clk_aht_out    : IN  STD_LOGIC;
        Cell_Sync_out  : OUT STD_LOGIC;
        Cell_preSync   : OUT STD_LOGIC;
        Chan_bsy       : IN  STD_LOGIC;
        Reset          : IN STD_LOGIC;
        Data_out       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        rnew_cell      : OUT STD_LOGIC;
        rcell_data     : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        rcell_read     : OUT STD_LOGIC;
        rcell_write    : IN  STD_LOGIC);
END ahtout;
---------------------------------

ARCHITECTURE rtl OF ahtout IS
  SIGNAL transmit     : STD_LOGIC;
  SIGNAL cell_reg      : STD_LOGIC_VECTOR(31 downto 0);  --Reg fuer Cellwort
  SIGNAL cell_reg2     : STD_LOGIC_VECTOR(31 downto 0);  --Reg fuer Cellwort

  TYPE   state_type IS (ST0,ST1,ST2,ST3,ST4,ST5,
                        ST_A1,ST_A2,ST_A3,ST_A4,ST_A5,ST_A6,
                        ST_A7,ST_A8,ST_A9);
  SIGNAL state_0, state_1, next_state_0, next_state_1 : state_type;
  signal sync_sig : STD_LOGIC;          -- Sync signal

BEGIN

  AHTOUT: PROCESS(sync_sig, Reset)
  -- Der AHTOUT Process holt Zell-Daten aus dem FIFO_out-Speicher und
  -- und gibt sie an den Process AHTOUT_TRANSMIT weiter

  TYPE countertype  IS RANGE 0 TO 12;
  VARIABLE count      :  countertype;

  BEGIN -- Process
  --  Reset: Setze alle Variablen und Signale zurueck --------------

    IF (Reset = '1') THEN
      rcell_read   <= '0';
      rnew_cell    <= '0';              -- zeigt nur "new cell" an
      transmit <= '0';
      count        := 1;

     ELSE

     CASE state_0 IS
WHEN ST0 =>
  IF (Chan_bsy = '0') THEN
     -- Der Ausgangskanal ist frei. Hole eine Zelle aus dem FIFO-Speicher.
            rnew_cell <= '1';
            rcell_read <= '1';          -- (1) rcell_read
    next_state_0 <= ST1;
  ELSE
            next_state_0 <= ST0;
          END IF;

WHEN ST1 =>
  IF (rcell_write='1') then     -- (2)
            cell_reg <= rcell_data; -- Der Header
            rcell_read <= '0';
            rnew_cell <= '0';
            transmit <= '1';
    next_state_0 <= ST2;
  ELSE
            next_state_0 <= ST1;
          END IF;

WHEN ST2 =>
    next_state_0 <= ST3;           -- 1. Byte cell_reg  (3)

        when ST3 =>                        -- 2. Byte (4)
             transmit <= '0';
             next_state_0 <= ST_A1;

      -- Wiederhole 6 mal fuer 32 - Bit-Register (48 Byte)

WHEN ST_A1 =>    -- First loop state
            rcell_read <= '1';          -- 3. Byte (1) rcell_read
    next_state_0 <= ST_A2;

WHEN ST_A2 =>
  IF (rcell_write = '1') then   -- 4. Byte
            cell_reg2 <= rcell_data;    -- cell_reg2 (2) rcell_read
            rcell_read <= '0';
            next_state_0 <= ST_A3;
```

```
  ELSE
            next_state_0 <= ST_A2;
          END IF;

        when ST_A3 =>
            next_state_0 <= ST_A4;     -- 1. Byte cell_reg2 (3)

        when ST_A4 =>
            next_state_0 <= ST_A5;     -- 2. Byte (4)

WHEN ST_A5 =>
            rcell_read <= '1';          -- 3. Byte (1)
            next_state_0 <= ST_A6;

   WHEN ST_A6 =>
    IF (rcell_write = '1') then   -- 4. Byte cell_reg2  (2)
             cell_reg <= rcell_data;
             rcell_read <= '0';
             next_state_0 <= ST_A7;
  ELSE
            next_state_0 <= ST_A6;
          END IF;

        when ST_A7 =>
            next_state_0 <= ST_A8;     -- 1. Byte cell_reg (3)

        when ST_A8 =>                   -- 2. Byte  (4)
             IF count < 6 then
                count := count + 1;
                next_state_0 <= ST_A1;
              ELSE
                next_state_0 <= ST0;
                count := 1;
              END IF;

        WHEN OTHERS =>
     END Case;
     END IF;
  End Process;


  AHTOUT_Transmit: PROCESS(sync_sig, Reset)
     VARIABLE hilf1         : STD_LOGIC_VECTOR(31 downto 0);
     VARIABLE hilf2         : STD_LOGIC_VECTOR(31 downto 0);
     TYPE countertype  IS RANGE 0 TO 12;
     VARIABLE count      :  countertype;

  BEGIN   -- Process
    -- Reset Part
    IF (Reset = '1') THEN
      Cell_Sync_out  <= '0';
      Cell_preSync   <= '0';
      Data_out       <= "00000000";
      count := 1;
      hilf1 := "00000000000000000000000000000000";
      hilf2 := "00000000000000000000000000000000";

    ELSE

     CASE state_1 IS
WHEN ST0 =>
  IF (transmit = '1') THEN
            Cell_preSync <= '1';
            hilf1 := cell_reg;
    next_state_1 <= ST1;
  ELSE
            next_state_1 <= ST0;
          END IF;

        -- Transmit Bytewise den Header
WHEN ST1 =>
          Cell_Sync_out <= '1';
          data_out <= hilf1(31 downto 24);     -- 1. Byte
  next_state_1 <= ST2;

WHEN ST2 =>
          data_out <= hilf1(23 downto 16);     -- 2. Byte
          Cell_Sync_out <= '0';
          Cell_preSync <= '0';
  next_state_1 <= ST3;

WHEN ST3 =>
          data_out <= hilf1(15 downto 8);      -- 3. Byte
  next_state_1 <= ST4;

WHEN ST4 =>
  data_out <= hilf1(7 downto 0);        -- 4. Byte
  next_state_1 <= ST5;


       -- Das HEC Byte ist die Wiederholung der vorhergehenden Daten
  WHEN ST5 =>
           next_state_1 <= ST_A1;
```

```
          hilf2 := cell_reg2;

    -- Transmit Bytewise die Payload in einer loop: 6 * 2 * 32 Bit, 6er loop
  WHEN ST_A1 =>                               -- 1. Byte
   data_out <= hilf2(31 downto 24);
          next_state_1 <= ST_A2;

   WHEN ST_A2 =>                               -- 2. Byte
         data_out <= hilf2(23 downto 16);
         next_state_1 <= ST_A3;

        WHEN ST_A3 =>                               -- 3. Byte
    data_out <= hilf2(15 downto 8);
         next_state_1 <= ST_A4;

        WHEN ST_A4 =>                               -- 4. Byte
    data_out <= hilf2(7 downto 0);
         hilf1 := cell_reg;
         next_state_1 <= ST_A5;

  WHEN ST_A5 =>                               -- 1. Byte
   data_out <= hilf1(31 downto 24);
          next_state_1 <= ST_A6;

    WHEN ST_A6 =>                               -- 2. Byte
         data_out <= hilf1(23 downto 16);
         next_state_1 <= ST_A7;

        WHEN ST_A7 =>                               -- 3. Byte
    data_out <= hilf1(15 downto 8);
         next_state_1 <= ST_A8;

        WHEN ST_A8 =>                               -- 4. Byte
    data_out <= hilf1(7 downto 0);
         hilf2 := cell_reg2;
            IF count < 6 THEN
              count := count + 1;
              next_state_1 <= ST_A1;
            ELSE
              next_state_1 <= ST0;
              count := 1;
            END IF;
        WHEN OTHERS =>
     END Case;
     END IF;
  End Process;


  Sync_Proc: PROCESS
  BEGIN
     IF (Reset = '1') THEN
state_0 <= ST0;
        state_1 <= ST0;
        sync_sig <= '0';
        Wait until Clk_aht_out'EVENT AND Clk_aht_out = '1';
     ELSE
        if sync_sig = '0' then
          sync_sig <= '1';
        else
          sync_sig <= '0';
        end if;
state_0 <= next_state_0;
        state_1 <= next_state_1;
        Wait until Clk_aht_out'EVENT AND Clk_aht_out = '1';
     end IF;
  END Process;

 END rtl;
```