ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Fachgruppe Technische Informatik
Wintersemester 2002/2003

Diplomarbeit DA-2003.15

Inter-Task-Communication in Reconfigurable Operating Systems

10. März 2003

Andres Erni
Stefan Reichmuth

Betreuer: Herbert Walder
Co-Betreuer: Matthias Dyer

Supervisor: Prof. Dr. Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Inhaltsverzeichnis

1 Aufgabenstellung **3**

2 Einleitung **9**

2.1 Entwicklungsplattform 9

2.1.1 Xilinx Virtex XCV800 10

2.1.2 XESS-Board XSV800 10

2.1.3 Entwicklungswerkzeuge 10

2.2 Modular Design Flow 4.1i 12

2.2.1 Einleitung 12

2.2.2 Überblick Design-Flow 12

2.3 Ausgangslage 14

3 Systemkonzeption **19**

3.1 Datenfluss 19

3.2 Systemüberblick 21

3.3 Partitionierung 21

3.3.1 Services der OS-Hardware-Komponente 22

3.3.2 Tasks 22

3.4 Standard-Task-Interface 22

3.5 Task-Template 24

3.6 Steuerkommandos 25

4 OS-Frame **29**
4.1 Überblick 29
4.2 OS-Left 31
4.2.1 audioplay 35
4.2.1.1 playstream 35
4.2.1.2 Audio-Puffer 36
4.2.2 ethaccept 36
4.2.3 arpreply 36
4.3 OS-Middle 37
4.3.1 pc_interface 39
4.3.1.1 pc_write 39
4.3.1.2 pc_read 39
4.4 OS-Right 41
4.5 FIFO-Puffer 43
4.5.1 Einführung und Evaluation von Lösungen 43
4.5.2 Implementation 45

5 Tasks **47**
5.1 AES $^{-1}$ 47
5.1.1 Einführung 47
5.1.2 Grundlagen 48
5.1.3 Funktionsweise 48
5.1.4 Implementation 48
5.1.4.1 Überblick 48
5.1.4.2 AESDecTask 49
5.1.4.3 decround 50
5.1.4.4 decroundcontroller 50
5.1.4.5 keygenerator 54
5.1.4.6 keycontroller 54
5.1.4.7 keyregister 55
5.1.4.8 conv8to32 57

5.1.4.9 conv32to8 57
5.1.4.10 fifoadapter 57
5.2 Audio 58
5.3 FIFO to LED 59
5.4 IP / UDP 59
5.4.1 Akzeptiertes Paketformat 59
5.4.2 Innerer Aufbau 60
5.5 Knightrider 62
5.6 RS232 Hex 62
5.6.1 Aufbau 62
5.6.2 Anpassen der Baudrate 63
5.7 RS232 ASCII 63
5.8 Dummy 63
5.9 Grössenvergleich 64

6 PC-Software 65
6.1 OS-Software-Komponente : HWOS V0.3 65
6.1.1 Scheduling 65
6.1.2 Task-Download 69
6.2 Send Data 69
6.2.1 Datentypen 70
6.2.2 Zieladresse und AES-Key 71

7 Probleme / Bugs 73
7.1 BlockRAM-Bug 73
7.2 Bitbreiten in Xilinx Foundation 75
7.3 Ethernet 76
7.3.1 Konfiguration 10 MBps Vollduplex 76
7.3.2 Transmitter 77
7.3.3 CRC 78
7.4 Pullups / Pulldowns 78

7.5 Rauschen / Echtzeitanwendung unter Windows 79
7.6 Störungen während der Ausführung 79
7.7 Clocknetzaktivierung 80
7.8 Unterbrechungen während des Bitstream-Downloads 80

8 Ausblick / Erweiterungen 83
8.1 Zugriff auf Ethernet-Sender 83
8.2 IFCC-Hard-Macro 84
8.3 Memory-Management durch OS / Flexibler Scheduler 84
8.4 Debug-Output zum PC 84
8.5 OS-Builder 84
8.6 Eigenes Board CPU / FPGA 85

9 Zusammenfassung / Schlussbemerkungen 87

A Inbetriebnahme 89
A.1 XESS-Board vorbereiten 89
A.2 RS232-Terminal 92
A.3 Ethernet-Sender 92
A.4 Software-OS 92
A.5 Erzeugen der Audio-Daten aus einer MP3-Datei 92
A.6 Tests 93

B Synthese-Informationen 95
B.1 Constraints-File `top.ucf` 95
B.2 Tabellen der Verbindungen in Top.vhd 99
B.2.1 Inter-Frame-Communication Channels 99
B.2.2 Standard-Task-Interfaces 102

C Modular Design How-To **105**
C.1 Design Entry / Synthesis Process 105
C.2 Initial Budgeting Phase 107
C.3 Active Module Phase 107
C.4 Final Assembly 108

D Präsentation **109**

E CD-Inhalt **115**

Abbildungsverzeichnis

2.1 XESS-Board 11
2.2 Initial Budgeting Phase 13
2.3 Active Module Phase 14
2.4 Final Assembly 15
2.5 Hardware des *Reconfigurable OS Prototype* 15
2.6 Task-Kategorien 16
2.7 Task, Task-Template, STI, IFCCs 17

3.1 Datenfluss der Applikation 20
3.2 Systemüberblick 21
3.3 Partitionierung: OS-HW-Komponente 23
3.4 Partitionierung: Tasks 24
3.5 VHDL-Code des linken Task-Templates 26

4.1 Überblick `top` 30
4.2 Datenfluss `OS-Left` 32
4.3 Hierarchie `OS-Left` 32
4.4 Schema `OS-Left` 33
4.5 Datenfluss `OS-Middle` 37
4.6 Hierarchie `OS-Middle` 38
4.7 Zustandsmaschine `pc_write` 40
4.8 Datenfluss `OS-Right` 41
4.9 Schema `OS-Right` 42

5.1 Hierarchischer Aufbau des AES^{-1}-Tasks 49
5.2 Schema “AES Decryption”-Task 51
5.3 RTL-Schema `decround` 52
5.4 Grössenvergleich der 128- mit der 32-Bit-Variante von `decround` . . 53
5.5 RTL-Schema Keygenerator 55
5.6 Vergleich der Schlüsselspeicher-Varianten 56
5.7 Vereinfachte fifoadapter FSM 58
5.8 IP-Header 60
5.9 UDP-Header 61

6.1 HW-OS V0.3 66
6.2 `HWOS.log` 67
6.3 Task-Download 69
6.4 Send Data 70

7.1 Ende eines partiellen Bitstreams 74
7.2 Abgeänderter Bitstream. 75
7.3 Unterschiedliches IFCC-Routing 81

A.1 XESS-Board- und Patch-Board-Verbindungen 90
A.2 Patch-Board-Verbindungen mit Zusatzboard 91

C.1 Verzeichnisstruktur 105

E.1 Verzeichnisstruktur der CD 115

Tabellenverzeichnis

3.1 Steuerkommandos 25
3.2 Zuweisung der Eingangs-FIFO-Queues (dfifo1) 27

4.1 Steuersignale `OS-Left` 34
4.2 Steuersignale der ersten Multiplexstufe 44
4.3 Vergleich FIFO-Speicher-Varianten 45

5.1 Zuordnung der Ports zum *t_fifo2_sel*-Signal 62
5.2 Grössenvergleich der Task-Bitstream in Bytes 64

6.1 Task Prioritäten 68

7.1 Konfigurationen des Ethernet-Chips 76
7.2 Vergleich von Ethernet-Frames 77

B.1 IFCCs Richtung rechts 100
B.2 IFCCs Richtung links 101
B.3 STI-Verbindungen Slot 0, Richtung rechts 102
B.4 STI-Verbindungen Slot 1, Richtung rechts 102
B.5 STI-Verbindungen Slot 0, Richtung links 103
B.6 STI-Verbindungen Slot 1, Richtung links 104

Kapitel 1

Aufgabenstellung

Diplomarbeit 2003.15

Inter-Task-Communication in Reconfigurable Operating Systems

Andres Erni & Stefan Reichmuth

Betreuung:

Herbert Walder
Matthias Dyer

Prof. Dr. Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich

Computer Engineering
and Networks Laboratory (TIK)

1 Einleitung

In *Embedded Systems (ES)* werden immer häufiger rekonfigurierbare Komponenten (CPLDs, FPGAs, etc.) eingesetzt, um besonders performante Teilaufgaben effizient abzuarbeiten.

Die Kapazität heutiger SRAM-basierter FPGAs (Field Programmable Gate Arrays) übersteigt um ein Vielfaches die Grösse von einzelnen Schaltungsblöcken, wie z.B. Krypto-Algorithmen, Filter (FIR/IIR), Multimedia Encoder/Decoder, sogar ganzer CPU-Soft-Cores. Zusätzlich lassen sich moderne FPGAs partiell rekonfigurieren, d.h. einzelne Teilbereiche können zur Laufzeit verändert werden, während die restlichen FPGA-Ressourcen parallel dazu weiterarbeiten *(Multitasking)*.

Dies eröffnet neue Einsatzmöglichkeiten für FPGAs: Mehrere voneinander unabhängige Schaltungsblöcke *(Hardware Tasks)* können zur Laufzeit auf FPGAs geladen, abgearbeitet und wieder entfernt werden. Somit wird ein FPGA zu einer dynamisch allozierbaren Ressource, die von einem Betriebssystem *(Reconfigurable Operating System)* verwaltet werden kann.

Einige Funktionen bzw. Services, die ein solches Betriebssystem den Applikationen zur Verfügung stellen soll, sind vergleichbar mit denjenigen eines herkömmlichen *Software RTOS (Real Time Operating System)* für Embedded Systems, wie z.B. Virtuoso oder DSP/BIOS.

Die klassischen Aufgaben eines RTOS sind:

- Ressource Management
- Task Management (load / unload Tasks, Scheduling)
- Task Synchronisation / -Kommunikation
- I/O Management
- Time Management

Applikationen, die unter einem RTOS ablaufen, werden typischerweise als *kooperierende Tasks* realisiert, wobei das Betriebssystem verschiedene Verbindungselemente bereitstellt, z.B.

- FIFO-Buffers
- Message-Boxes
- Shared Memory
- Semaphoren
- etc.

Diese OS-Elemente ermöglichen den Tasks miteinander zu kommunizieren bzw. sich gegenseitig zu synchronisieren. In SW-Betriebssystemen können diese relativ einfach implementiert werden, in Reconfigurable OS hingegen ist der gesamte Themenbereich „Inter-Task-Communication" noch Gegenstand der Forschung. Bis dato sind nur wenige realisierbare Lösungsansätze vorhanden, nicht zuletzt deshalb, weil die bisherigen Design-Tools einen taskorientierten Design-Flow nicht unterstützten.

Mit dem neuen *Modular-Design-Package* von *Xilinx* stehen nun Elemente und Funktionen zur Verfügung, mit denen sich modulare Designs implementieren und die dazugehörigen partiellen Bitstrems generieren lassen.

Als Resultat einer früher durchgeführten Diplomarbeit [1] steht ein rudimentärer Prototyp eines Reconfigurable OS in HW und SW zur Verfügung, mit dem dynamisch HW-Tasks geladen und ausgeführt werden können. Diese erste Version ermöglicht den Tasks mittels eines standardisierten Interfaces den Zugriff auf externe Komponenten; das Problem der Inter-Task-Kommunikation ist jedoch noch nicht gelöst.

2 Ziele

In dieser Arbeit soll

1) ein detailliertes Lösungskonzept für *Inter-Task-Kommunikation* im Umfeld eines Reconfigurable OS erarbeitet werden, das zeigt, welche Services bzw. Interfaces ein OS zur Verfügung stellen muss, damit Tasks untereinander kommunizieren können;
2) das Lösungskonzept mit heute vorhandenen Design-Tools implementiert werden;
3) eine Case-Study-Applikation aus dem Bereich Multimedia und/oder Networking implemeniert werden, die aus kommunizierenden und dynamisch rekonfigurierten Tasks besteht.

Die Arbeit soll sich auf die in [1] gewonnenen Erkenntnisse stützen und der bereits bestehende OS-Prototyp soll ensprechend erweitert werden.

3 „Take off" / Aufgaben / Vorgehen

Einstieg:

- Kennenlernen der Xilinx Virtex Architektur und Konfigurationsmodi (full, partial) anhand der Xilinx XAPP138, XAPP151;
- Installieren und kennenlernen der Xilinx Design Tools (Xilinx Foundation), Modular Design
- Inbetriebnahme des XESS-Boards; Implementieren und testen einfacher Schaltungen, z.B. LED-Blinker, Counter, etc.;
- Studium von Literatur zum Thema „Reconfigurable OS" (Forschungspublikationen) bzw. Berichte früherer Diplomarbeiten (siehe Literaturverzeichnis);
- Inbetriebnahme / Tests / Analyse von bereits existierenden Tools

Bearbeiten der folgenden Teilaufgaben bzw. Fragestellungen:

- Welche Applikation soll implementiert werden? Spezifikation?
- Wie soll die Applikation in kooperierende Tasks zerlegt werden?
- Welche Tasks haben welche Prioritäten bzw. verursachen Ressourcenkonflikte?
- Welche Tasks werden ins OS-Frame integriert (statisch)? Welche Tasks sollen zur Laufzeit in die Task-Slots geladen werden (dynamisch)?
- Welche Elemente bzw. Services zur Inter-Task-Kommunikation sollen vom OS bereitgestellt werden?
- Wie sollen Tasks auf die Elemente zugreifen bzw. die Services in Anspruch nehmen können?
- Wie muss das STI (Standard Task Interface) bzw. IFCC (Inter-Frame Communication Channels) abgeändert oder erweitert werden?
- Welchen Anforderungen muss der Task-Scheduler genügen?
- Welches Systemverhalten ist vorherzusehen? (aufgrund von Ressourcenkonflikten oder Rahmenbedingungen der Plattform bzw. Implementation)

Konzeptuelles Vorgehen

- Implementieren und Testen des Gesamtsystems (Case-Study) als statische (volle) Konfiguration.
- Erweitern des OS-Frames / Bereitstellen des entsprechenden Task-Templates
- Isolieren der Tasks / Migration in Task-Templates
- “Manuelles” Testen von einzelnen Tasks bzw. einzelnen Applikationsteilen

- Implementieren des Task-Schedulers (unter Berücksichtigung der Task-Prioritäten)
- Testen des Systems / Soll-/Istvergleich Spezifikation vs. Implementation
- Verbesserungsvorschläge / Ausblick

4 Deliverables

- Funktionierende Case-Study-Applikation
- Schlussbericht, der eine vollständige und nachvollziehbare Dokumentation der entwickelten Konzepte, Algorithmen, Implementationen, etc. enthält.
  (Text: LaTex, Illustrationen: Corel Draw)
- Präsentation der DA: ca. 20min.
- Source-Code (C++, VHDL), alles in elektronischer Form (CD-ROM); zusätzlich: Schlussbericht 3x als Hardcopy (gebunden)

5 Rahmenbedingungen

5.1 Technisch

- FPGA-Plattform: XESS XSV-800 Xilinx Virtex Rapid Prototyping Board
- Windows PC (Windows NT / 2000 / XP)
- PCI-basierter I/O-Karte; inkl. Erweiterungs-HW
- Design Tool: Xilinx ISE Foundation, Xilinx Modular Design, ModelSim
- Microsoft Visual C++ Version 6.0, MFC

5.2 Projektmanagement

- Projektmeetings in regelmässigen Abständen sorgen dafür, dass der Stand der Arbeit laufend beurteilt werden kann, der Projektplan angepasst bzw. neue Etappenziele (Milestones) definiert werden können.
  Die Projektmeetings werden vom Studenten angesetzt, vorbereitet und durchgeführt.
- Anpassungen der Aufgabenstellung erfolgen nach Rücksprache mit dem Diplomprofessor bzw. dem Betreuer der Arbeit.
- Arbeit in Etappen; Dokumentation von Erkenntnissen und Resultaten laufend; Gesamtsicherung und Dokumentation von Zwischenständen in sinnvollen Intervallen (Versioning).

5.3 Administrativ

- Zeitdauer der Diplomarbeit: 04. November 2002 – 07. März 2003 (inkl. Weihnachtsferien)
- Diplomprofessor: Prof. Dr. Lothar Thiele; Betreuer: Herbert Walder, Co-Betreuer: Matthias Dyer
- Das „Merkblatt für Studienarbeiten am TIK“ bildet die Grundlage für die Arbeit. Siehe: http://www.tik.ee.ethz.ch/~walder/Library/tik_merkblatt.pdf
- Die Informationen bzw. Weisungen auf der TIK-HomePage: http://www.tik.ee.ethz.ch/new_homepage/education/sadas/SADAInfo.html sind zu berücksichtigen bzw. einzuhalten.

6 Literaturverzeichnis / Links

[1] Michael Ruppen:
Reconfigurable OS Prototyoe

[2] Michael Lerjen, Chris Zbinden:
Reconfigurable Bluetooth / Ethernet Bridge

[3] Matthias Dyer, Marco Wirz:
Reconfigurable System on an FPGA

[4] Xess Inc.: XESS-Board Manual
http://www.xess.com

[5] Gordon Brebner:
A Virtual Hardware Operating System for the Xilinx XC6200

[6] Grant Wigley et al.:
Research Issues in Operating Systems for Reconfigurable Computing

[7] Grant Wigley et al.:
The Development of an Operating Systems for Reconfigurable Computing

[8] Herbert Walder et al.:
Non-preemptive Multitasking of FPGAs: Task Placement and Footprint-Transform

[9] Pedro Merino et al.:
A Hardware Operating System for Dynamic Reconfiguration of FPGAs

Kapitel 2

Einleitung

Zu Beginn soll in diesem Kapitel die Entwicklungsplattform, der *Modular Design Flow* und die von uns angetroffene Ausgangslage näher beschrieben werden. Im nächsten Kapitel wird unser Konzept zur Realisierung eines Systems mit Inter-Task-Kommunikation erläutert. In den Kapiteln 4 und 5 wird die entwickelte Hardware näher erklärt. Im ersten der beiden Kapitel betrifft dies den Hardware-Teil des Betriebssystems, während im anderen die Tasks das Thema sind. Im folgenden Kapitel 6 werden dann der Software-Teil des Betriebssystems und eine Hilfsapplikation zum Versenden von Daten über Ethernet vorgestellt. Sehr wichtig für jemanden, der sich weiter mit unserer Arbeit beschäftigt (und mit der gleichen Entwicklungsplattform arbeitet), ist Kapitel 7. In diesem Kapitel sind die aufgetauchten Probleme und Bugs zusammengestellt. In Kapitel 8 folgen schliesslich einige Überlegungen zu einer Weiterentwicklung unserer Arbeit, bevor Kapitel 9 den Bericht mit einer Zusammenfassung abschliesst. In den Anhängen befinden sich zudem eine Anleitung zur Inbetriebssetzung unserer Applikation (Anhang A), sowie nützliche Information für Entwickler (Anhänge B und C).

2.1 Entwicklungsplattform

Für die Software-Komponenten wurde ein PC mit einem 1000 MHz Pentium III und 512 MB Hauptspeicher verwendet. Als Betriebssystem diente Windows XP. Die Verbindung zwischen PC-Software und FPGA wurde durch eine *ADLINK PCI-7200*-Karte hergestellt. Diese besitzt je 32 digitale Ein- und Ausgänge. In den folgenden beiden Unterabschnitten werden das FPGA und das Prototypen-Board kurz vorgestellt.

2.1.1 Xilinx Virtex XCV800

Die FPGAs der XCV800-Baureihe verfügen über eine Kapazität von ca. 800'000 GE[1]. Die konfigurierbaren Blöcke (CLBs) sind in 84 Spalten und 56 Reihen angeordnet. Auf der linken und rechten Seite sind je 14 BlockRAMs platziert. Ein BlockRAM kann 4096 Bits speichern, was eine totale Kapazität von 14 KB ergibt. Bei einer partiellen Rekonfiguration werden immer ganze Teil-Spalten (sogenannte *Frames*) beschrieben. Eine ausführliche Beschreibung der Architektur und deren (partiellen) Rekonfiguration ist in [22] und [23] zu finden.

2.1.2 XESS-Board XSV800

Das XSV-Board stellt neben dem FPGA mehrere Ressourcen für Prototypenentwicklungen zur Verfügung. Es sind dies u.a. :

- RS232-Schnittstelle
- Ethernet-Interface (Physical-Layer)
- USB-Interface
- LED-Bar
- Zwei 7-Segment-Anzeigen
- Video-Decoder
- Audio-Codec
- 512K x 16 SRAM

Auf der XESS-Homepage (www.xess.com) stehen viele Beispiel-Anwendungen, die diese Ressourcen benützen, zur freien Verfügung. Die Abbildung 2.1 zeigt ein Schema des Boards.

2.1.3 Entwicklungswerkzeuge

Für die Software-Entwicklung wurde *Microsoft Visual C++ 6.0* eingesetzt. Zur Hardware-Implementierung wurden die *Xilinx Foundation Tools* in der Version 5.1.03i verwendet. Während der Arbeit zeigten sich einige unangenehme Eigenschaften bei den letztgenannten Tools (vgl. Kapitel 7).

[1] Gate Equivalents

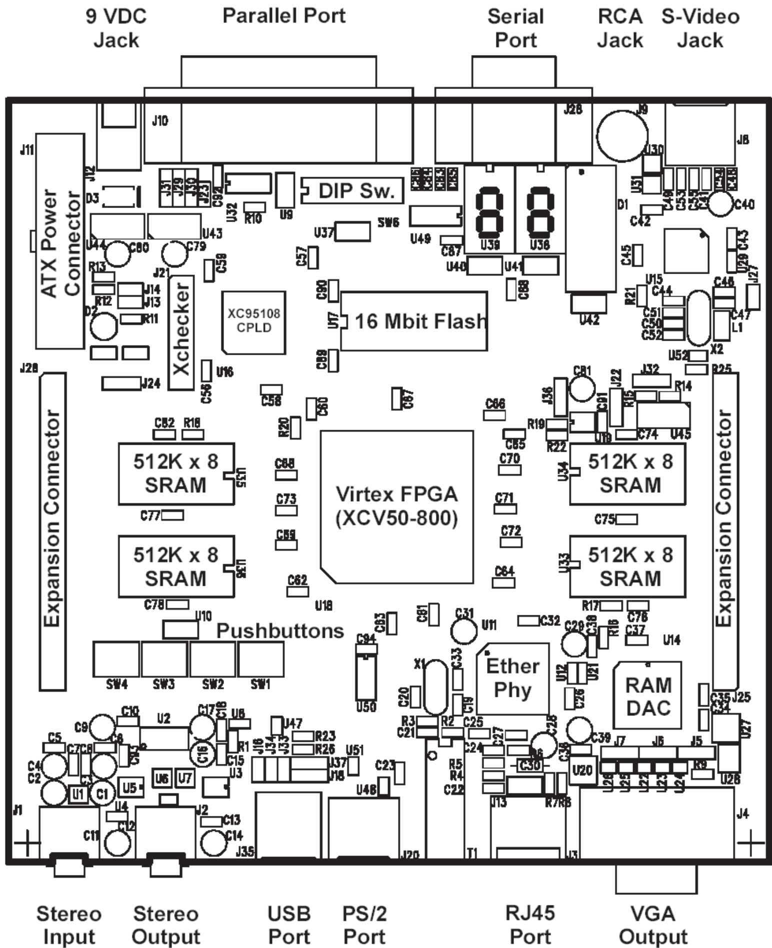


Abbildung 2.1: XESS-Board

2.2 Modular Design Flow 4.1i

2.2.1 Einleitung

Um ein Virtex FPGA teilweise rekonfigurieren zu können, müssen partielle Bitstreams erzeugt werden. Mit dem Standard-Design-Flow von Xilinx ist dies jedoch nicht möglich.

In früheren Arbeiten wurden die Bitstreams mit *JBits* manipuliert. Dies ist aber sehr umständlich und für grössere Designs kaum anwendbar (vgl. mit [8]). Ausserdem konnten keine Routing-Constraints gesetzt werden, was beim Austauschen von Modulen zu grossen Problemen führt. Vor allem konnte man aber nicht garantieren, dass die Leitungen das "eigene" Gebiet nicht verlassen.

In dieser Diplomarbeit haben wir den Modular-Design-Flow von Xilinx verwendet, welcher das Generieren von partiellen Bitstreams deutlich vereinfacht. Dazu sei noch erwähnt, dass dieses Konzept ursprünglich nicht für die dynamische Rekonfiguration entwickelt wurde. Es diente vielmehr einem Design-Team dazu, verschiedene Teile der Schaltung unabhängig voneinander zu entwickeln (Synthese und Simulation). Man vergleiche dazu [25],[26] und [27]. Eine Beschreibung, wie man den Modular-Design-Flow für die partielle Rekonfiguration einsetzen kann, kann in [24] nachgelesen werden.

Eine kompakte Anleitung zu diesem Design Flow befindet sich im Anhang C.

2.2.2 Überblick Design-Flow

Der Modular-Design-Flow besteht aus drei Phasen:

1. **Initial Budgeting Phase:** Zu Beginn wird der Floorplan der Schaltung erstellt. Dabei werden allen fixen und rekonfigurierbaren Teilen (auch Module genannt) je eine Fläche auf dem FPGA zugewiesen. Die Gebiete müssen rechteckig sein und die ganze Höhe des FPGAs ausnutzen. Diese Einschränkung beruht auf der Tatsache, dass die kleinste Konfigurationseinheit auf dem Virtex FPGA ein sogenanntes *Frame* ist, welches jeweils die gesamte Höhe des FPGAs ausfüllt.

   Module, die miteinander kommunizieren möchten, müssen eine gemeinsame Gebietsgrenze haben. Zusätzlich werden sogenannte *Bus-Macros* als *Anschlusspunkte* für die Leitungen benötigt, welche von mehreren Modulen benutzt werden (Schnittstelle zwischen den Modulen). Die Kommunikation kann nur über solche, mit Constraints genau platzierten, Macros ablaufen.

Eine weitere Einschränkung besteht darin, dass die Module nur Zugriff auf Pins haben, welche direkt an ihr zugewiesenes Gebiet grenzen.

Neben den Area-Constraints können auch noch Timing-Constraints gesetzt werden. Auf jeden Fall muss aber für jedes Modul die Routing-Beschränkung `DISALLOW_BOUNDARY_CROSSING` eingefügt werden, damit das Tool die Grenzen der verschiedenen Gebiete auch wirklich respektiert.

Neben einem `.ucf`-File, resultiert aus der ersten Phase eine VHDL-Datei, in welcher das Top-Level-Design definiert ist. Dieses sollte nur aus den verschiedenen instantierten Modulen und den Bus-Macros bestehen. Zur Illustration betrachte man Abbildung 2.2.

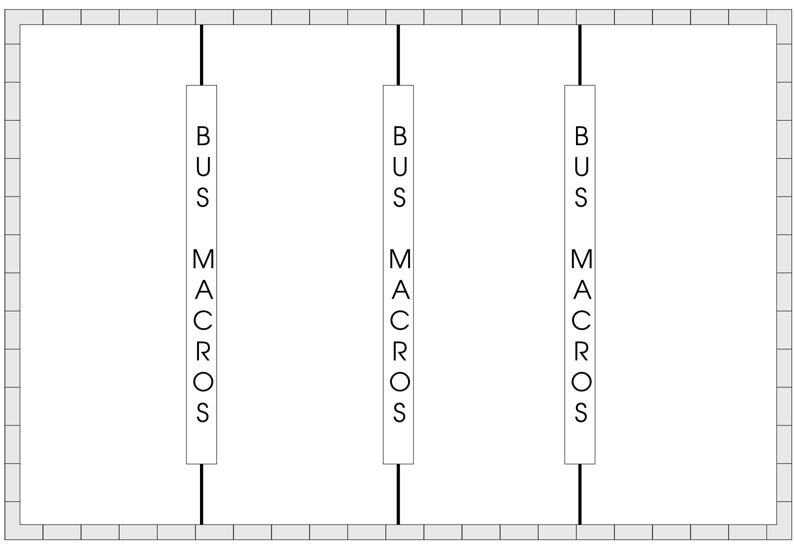


Abbildung 2.2: Initial Budgeting Phase

2. **Active Module Phase:** Nun können die einzelnen Module getrennt voneinander entwickelt werden. Die Bedingungen, die bei der Umsetzungen berücksichtigt werden müssen, sind im `.ucf`-File aus der ersten Phase enthalten.

   Für die Gebiete, welche rekonfiguriert werden, können mehrere Einheiten erstellt werden. Dabei ist zu beachten, dass die Entitäten jeweils den selben Namen tragen (Wie die Instanzen im VHDL-File der Initial Budgeting Phase).

   Das Resultat sind partielle Bitstreams, von jedem der Module (fix und rekonfigurierbar). Abbildung 2.3 stellt dies graphisch dar.

3. **Final Assembly:** Wenn ein FPGA partiell rekonfiguriert werden soll, muss zu Beginn ein vollständiger Bitstream geladen werden. Um die *Startkonfiguration*

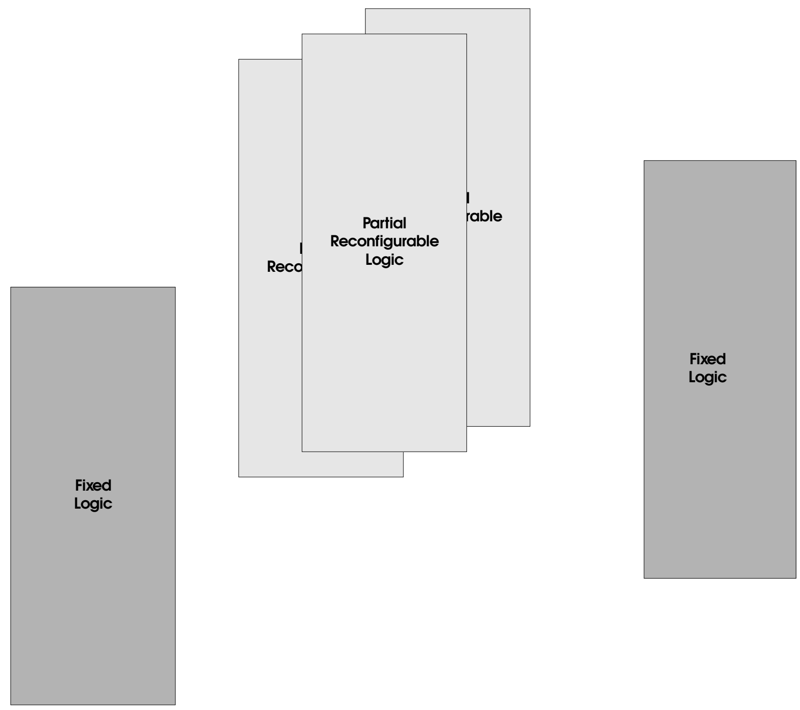


Abbildung 2.3: Active Module Phase

zu erhalten, werden in diesem Schritt die fixen Module, sowie für jedes Gebiet eines der rekonfigurierbaren zu einer "vollen" Konfiguration zusammengefügt.

Abbildung 2.4 zeigt das Resultat dieses Schrittes. Dabei sind mögliche Kommunikationsverbindungen mit schwarzen Pfeilen gekennzeichnet. Verbotene hingegen sind jeweils rot vermerkt.

2.3 Ausgangslage

In der Diplomarbeit *Reconfigurable OS Prototype* [17] von Michael Ruppen, die sich teilweise mit unserer überlappte, wurde eine einfache Implementierung eines Hardware-Betriebssystems entwickelt. Das Betriebssystem besteht dabei aus einer Software-Komponente, die auf einem PC läuft, sowie aus einer Hardware-Komponente auf dem FPGA. Letztere wird als *OS-Frame* bezeichnet. Das OS-Frame wiederum besteht aus den drei Teilen OS-Left, OS-Middle und OS-Right. Zwischen diesen Modulen befinden sich zwei Task-Slots, die einige einfache Tasks aufnehmen können. Die Tasks kommunizieren dabei über verschiedene, auf dem XESS-Board vorhandene, Ressourcen mit der Aussenwelt. Die Abbildung 2.5 gibt einen kleinen Überblick zum Hardware-Teil dieses Systems.

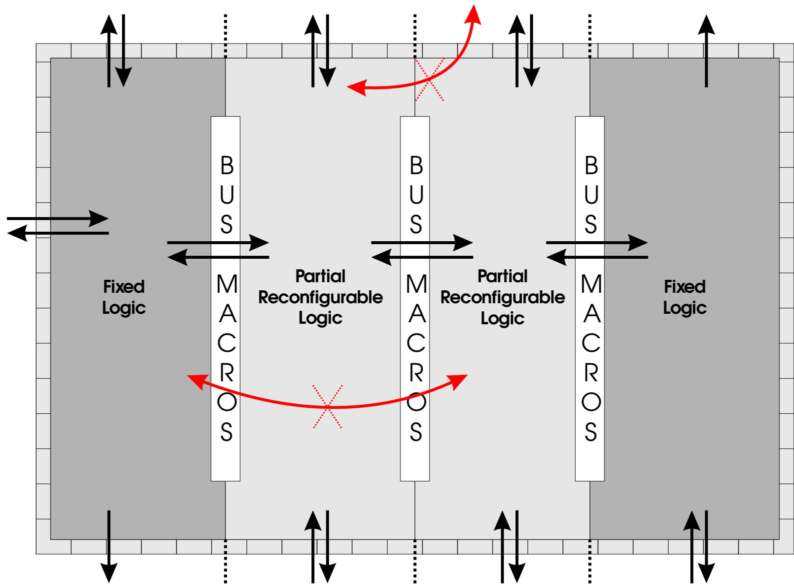


Abbildung 2.4: Final Assembly

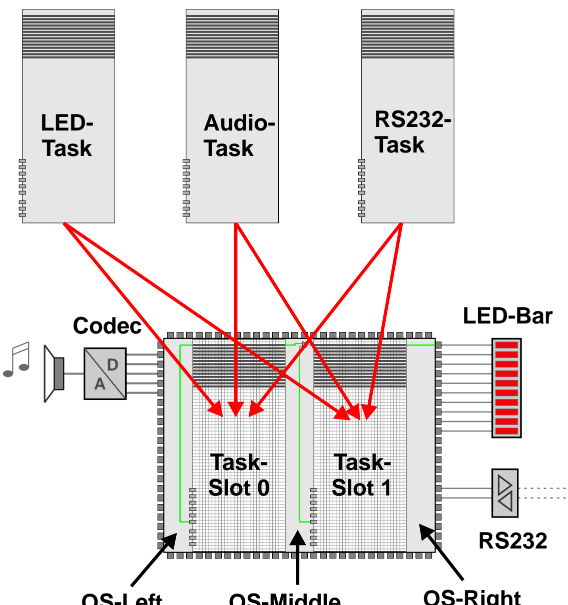
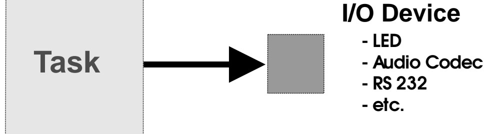


Abbildung 2.5: Hardware des *Reconfigurable OS Prototype*

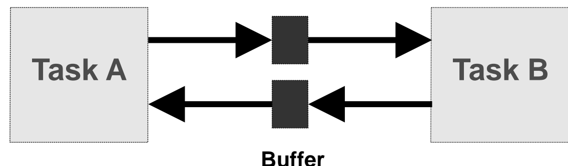
Zur Charakterisierung von Hardware-Tasks kann man z.B. folgende Task-Kategorien definieren (zur Illustration betrachte man Abbildung 2.6):

- *Stand-Alone-Task* : Ein Stand-Alone-Task kommuniziert nur mit dem OS und Ressourcen ausserhalb des FPGAs. Die in [17] verwendeten Tasks gehören zu dieser Kategorie.
- *Non-Time-Critical-Task* : Diese Tasks kommunizieren auch untereinander, was für ein komplexe Schaltung unabdingbar ist. Der Datenaustausch erfolgt dabei über Pufferspeicher, die eine gewisse Latenz mit sich bringen. In dieser Arbeit wurden solche Tasks implementiert.
- *Time-Critical-Task* : Die Tasks dieser Kategorie sind auf eine verzögerungsfreie Kommunikation angewiesen. Sie müssen ihre Daten direkt miteinander austauschen. Aufgrund von Einschränkungen des verwendeten Modular Design Flows, sowie einer erheblich grösseren Komplexität, konnten solche Tasks noch nicht implementiert werden.

Stand-Alone-Task



Non-Time-Critical-Tasks



Time-Critical-Tasks

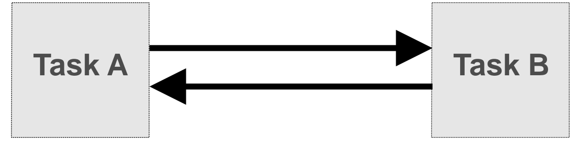
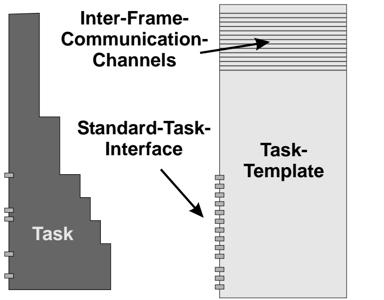


Abbildung 2.6: Task-Kategorien

Damit, trotz eines völlig unterschiedlichen inneren Aufbaus, verschiedene Tasks in das OS-Frame geladen werden können, wurde ein sogenanntes Task-Template definiert (vgl. Abbildung 2.7). Dieses legt eine genormte Schnittstelle zwischen Task und Betriebssystem fest, die als *Standard-Task-Interface (STI)* bezeichnet wird. Aufgrund der Einschränkung des Modular Design Flows, dass ein Modul jeweils die ganze Höhe des FPGAs bedecken muss, müssen die Verbindungen zwischen den drei OS-Gebieten die Task-Fläche durchqueren. Das Task-Template muss deshalb auch diese, *Inter-Frame-Communication-Channels (IFCCs)* genannten, Leitungen beinhalten.



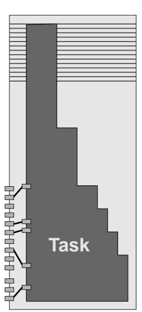


Abbildung 2.7: Task und Task-Template mit Standard-Task-Interface (STI) und Inter-Frame-Communication-Channels (IFCCs)

Kapitel 3

Systemkonzeption

Um die angestrebte Inter-Task-Kommunikation auszutesten wurde eine “Case-Study-Application” entwickelt. Dabei soll einerseits ein Realtime-Flow (Audio-Streaming) und andererseits ein Non-Realtime-Flow (Text-Ausgabe, z.T. verschlüsselt) existieren. Zusätzlich sollen auch durch User-Interaktionen bestimmte Tasks geladen werden.

3.1 Datenfluss

Zuerst wurde ein Datenfluss einer solchen Applikation erstellt. Die Abbildung 3.1 zeigt denjenigen der realisiert wurde. Als Eingang für die Audio-Streaming-Daten wurde die Ethernet-Schnittstelle gewählt, da nur so eine genügend grosse Datenmenge in einer relativ kurzen Zeit auf das FPGA gelangen kann. Text-Nachrichten werden ebenfalls via Ethernet übermittelt. Als Transport-Protokoll kommt das sehr einfach aufgebaute UDP [14] zum Einsatz. Zur Unterscheidung, welchen Weg im Datenfluss ein ankommendes Paket nehmen soll, wird die UDP-Port-Nummer verwendet.

Weil UDP auf IP aufgesetzt ist, muss zu Beginn einer Übertragung die Ziel-IP-Adresse in eine Ethernet-Adresse übersetzt werden [15]. Dies geschieht entweder durch Setzen einer statischen Route mittels des `ARP`-Befehls auf dem sendenden PC (z.B. : `arp -s 10.0.0.4 01-02-03-04-05-06`) oder automatisch durch ARP-Requests und Replies. Für die Implementierung der zweiten Möglichkeit müssen ARP-Pakete von IP-Paketen unterschieden werden. Zudem ist natürlich auch ein Ethernet-Sender notwendig.

Der Datenfluss funktioniert nun folgendermassen : Kommt ein Ethernet-Frame an, wird es vom Ethernet-Empfänger (ETH Rx) überprüft. Wenn es korrekt ist

(CRC-Überprüfung), wird der Inhalt entweder in den IP-Puffer-Speicher (Datenpakete) oder in denjenigen für ARP-Requests geschrieben. Für ARP-Requests wird eine Antwort generiert (ARP) und in die FIFO-Queue des Ethernet-Senders (ETH Tx) geschrieben, der ein entsprechendes Frame erzeugt und sendet. Bei Datenpaketen werden IP- und UDP-Header überprüft und wenn diese in Ordnung sind, in den Speicher des entsprechenden Ports geschrieben (IP / UDP). Daten, die an den Port 6000 gesendet werden, werden in Hexadezimaldarstellung auf der RS232-Schnittstelle ausgegeben (Hex Dump). Beim Port 6002 erfolgt ebenfalls eine Ausgabe über die serielle Schnittstelle, jedoch als Klartext (Text Output). Daten aus der Queue des Ports 6001 werden mit dem AES-Algorithmus entschlüsselt (AES^{-1}) und zum Port 6002 geschrieben, wo sie ausgegeben werden. Beim Audio-Streaming erfolgt eine Zwischenspeicherung im Puffer des Ports 7000, von wo die Daten im korrekten Timing auf dem Audio-Codec ausgegeben werden müssen (CoDec Driver). Ebenfalls in diesen Puffer-Speicher gelangen Audio-Daten, die von einem Sägezahngenerator (Wave Gen.) erzeugt werden, wenn ein Benutzer einen Switch (SW 3) des XESS-Boards betätigt und damit ein Trigger-Signal auslöst (TR3). Durch einen zweiten Switch (SW 1) kann analog dazu ein Muster auf dem LED-Bar angefordert werden (LED Pattern).

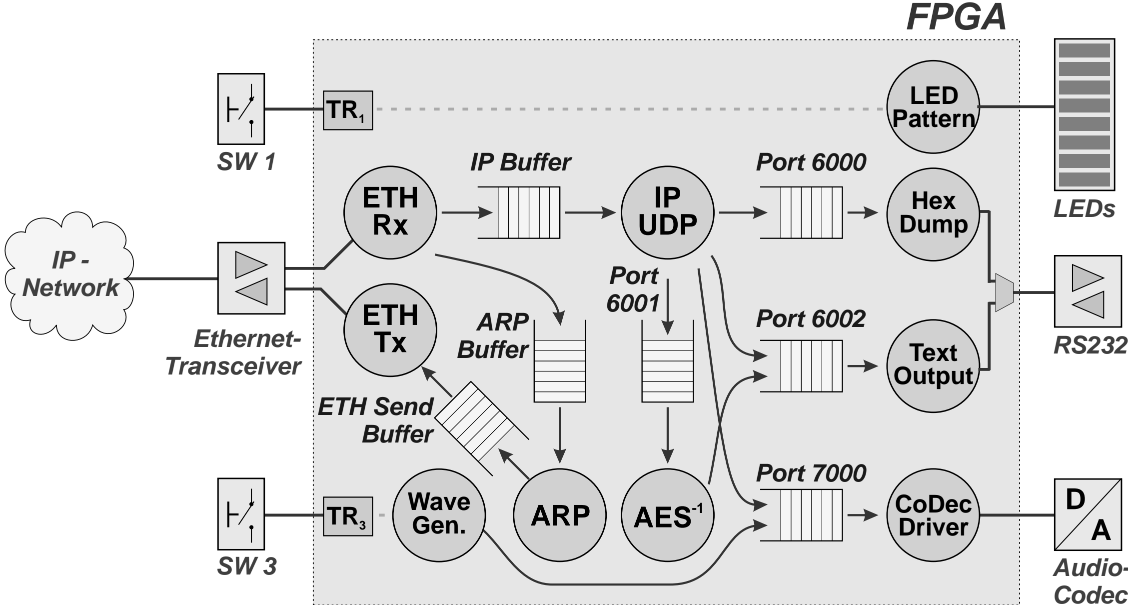


Abbildung 3.1: Datenfluss der Applikation

3.2 Systemüberblick

Die Abbildung 3.2 zeigt die Gesamtansicht des Systems. Um die Tasks laden zu können ist der PC mit der Software-Komponente des Betriebssystems an die SelectMAP-Konfigurationsschnittstelle angeschlossen. Ein zweiter PC sendet Audio- und/oder Text-Daten über den Ethernet-Anschluss zum XESS-Board. Die Ausgaben der RS232-Schnittstelle werden schliesslich auf einem dritten Computer angezeigt, der als RS232-Terminal fungiert[1].

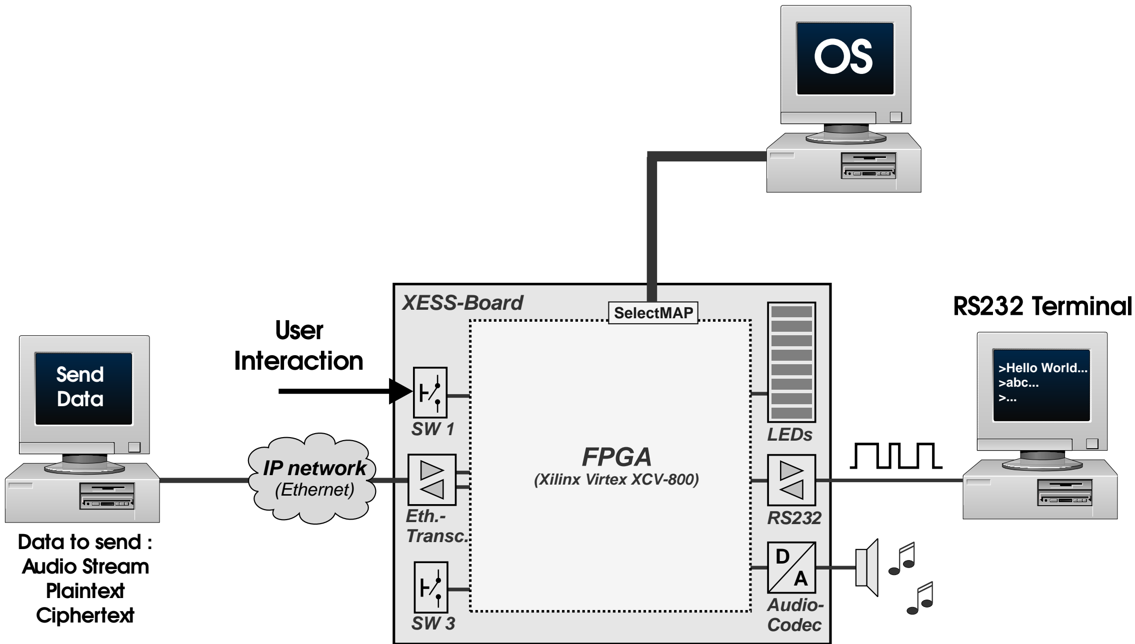


Abbildung 3.2: Systemüberblick

3.3 Partitionierung

Nach der Festlegung des Datenflusses mussten nun die darin vorhandenen Elemente der Hardware-Komponente des Betriebssystems einerseits, sowie den Tasks andererseits zugeordnet werden.

[1]Natürlich könnten auch alle drei Schnittstellen an den gleichen PC angeschlossen werden. Die Darstellung mit drei Computern entspricht aber mehr einer realistischen Situation. Vor allem sollte der Software-Teil des OS getrennt vom Datenfluss sein.

3.3.1 Services der OS-Hardware-Komponente

In Abbildung 3.3 sind die als OS-Services implementierten Elemente entsprechend markiert. Die Gründe für diese Zuteilung sind dabei :

- ETH Rx : Der Ethernet-Empfänger muss ständig präsent sein, da sonst ankommende Pakete verloren gehen können.
- ARP und ETH Tx : Ein ARP-Reply muss innerhalb einer relativ kurzen Zeit nach dem ARP-Request gesendet werden. Wenn diese Komponeneten erst geladen werden müssten, würde dies viel zu lange dauern (vgl. Abschnitt 6.1).
- CoDec Driver : Die genaue Ansteuerung von Ressourcen ist eine Aufgabe des Betriebssystems. Ein Task sollte sich nicht darum kümmern müssen.
- Puffer-Speicher : Auch die Speicher zum Datenaustausch und deren Verwaltung ist eine Betriebssystemaufgabe. Die Tasks sollten nicht die Kontrolle darüber haben.
- TR1, TR3 : User-Interaktionen haben Einfluss auf das Scheduling und gehören damit zum OS.

3.3.2 Tasks

Als Tasks wurden diejenigen Elemente implementiert, die nicht ständig vorhanden sein müssen. Wenn ein solches Element momentan nicht benötigt wird, kann die Chip-Fläche für ein anderes Element benutzt werden. Diese bessere Ausnutzung der FPGA-Fläche stellt einen der Hauptvorteile eines Hardware-Betriebssystems dar. In Abbildung 3.4 sind die Tasks im Datenfluss entsprechend markiert.

3.4 Standard-Task-Interface

Nach der Partitionierung wurde das Standard-Task-Interface (STI) festgelegt. Das STI umfasst die folgenden Signale :

- Clock
- Reset
- Enable

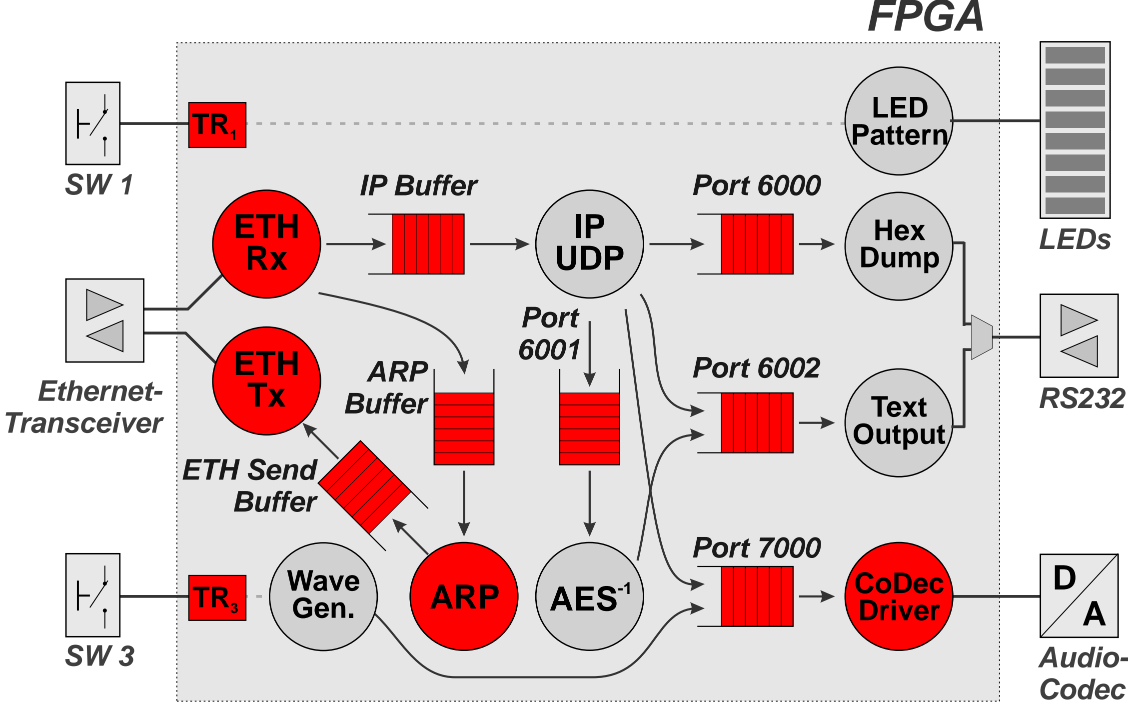


Abbildung 3.3: Partitionierung: OS-HW-Komponente

- Finished
- LED (10 Bits)
- RS232 (TX und RX)
- Input (RE, 8 Bit DataIn, Empty)
- Output (WE, 8 Bit DataOut, Full, 3 Bit FIFO-Select)
- Inter-Frame-Communication-Channels (IFCCs), davon
  - 44 Bit für die Kommunikation von links nach rechts und
  - 32 Bit für diejenige von rechts nach links

Gegenüber dem in [17] verwendeten Interface ist die wesentliche Neuerung der Zugriff auf die Puffer-Speicher. Jeder Task erhält jeweils einen Port für Eingangs- und Ausgangsdaten. Das Signal RE (Read Enable) dient zur Anforderung des nächsten Daten-Bytes. Die Datenbreite von 8 Bits entstand als Kompromiss zwischen einem möglichst hohen Durchsatz und einem geringen Routing-Overhead. Durch

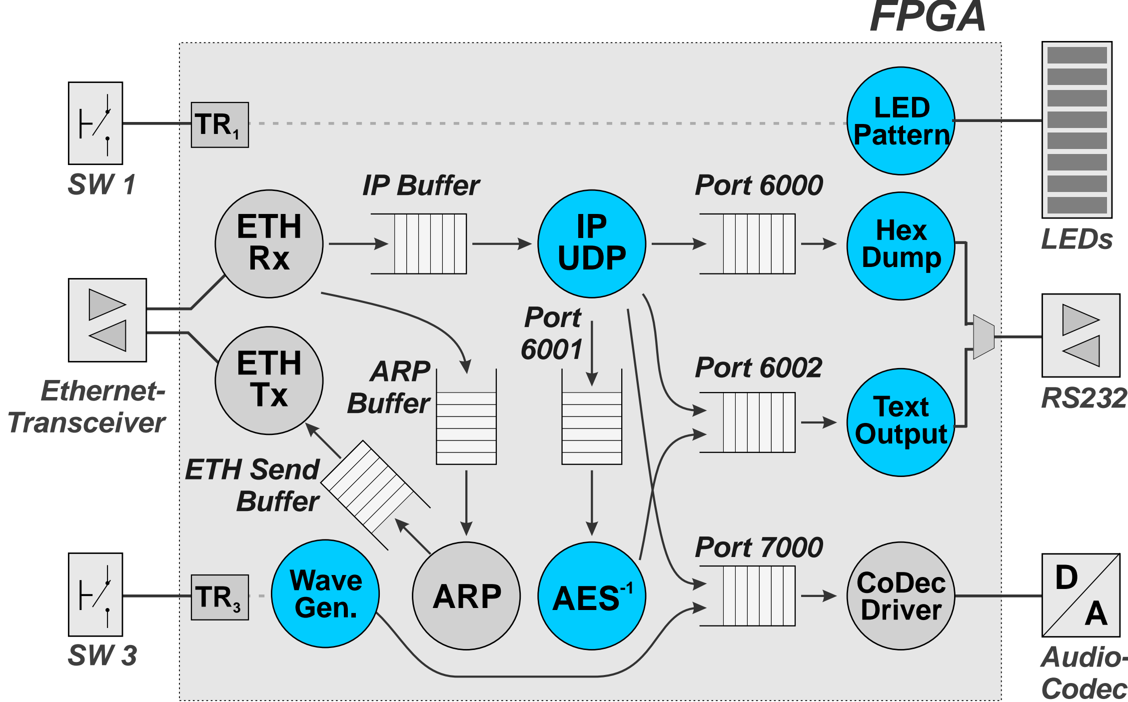


Abbildung 3.4: Partitionierung: Tasks

das Empty-Signal wird der Task informiert, wenn keine Daten mehr im Eingangs-Speicher vorhanden sind.

Durch das Setzen des WE-Signals (Write Enable) wird ein Byte in die Ausgangs-Queue hineingeschrieben. Das Full-Signal zeigt an, wenn der Puffer vollständig gefüllt ist. Mittels FIFO-Select muss beim Ausgang zudem der gewünschte Speicher ausgewählt werden. Bei den Eingagsdaten wird diese Selektion durch das Betriebssystem vorgenommen.

Auf eine exakte Füllstandsangabe der FIFO-Queues wurde verzichtet, da dies eine erhebliche Vergrösserung des Routing-Aufwands mit sich gebracht hätte.

Die verwendeten Signalnamen im VHDL-Code sind in der Entity-Deklaration in Abbildung 3.5 ersichtlich.

3.5 Task-Template

Das Task-Template besteht aus je einer VHDL-Datei für den linken und den rechten Task-Slot. Der einzige Unterschied ist dabei der Name der Entity. Im `architecture`-Teil sind allen Ausgängen des STIs sinnvolle Vorgabe-Werte zugewiesen.

Normalerweise ist dies '0' mit den folgenden Ausnahmen :

- *t_rs232_tx*, weil nach dem RS232-Standard während der Idle-Zeit eine '1' anliegt.
- *t_fifo2_sel*, weil aus Konvention der Wert "111" gewählt wurde, wenn kein Output-Puffer selektiert ist.[2]

Diese Werte sollten nicht geändert werden, wenn die entsprechenden Ausgänge bei der Task-Implementierung nicht benutzt werden. Ausserdem sind die IFCCs bereits miteinander verbunden. Die Abbildung 3.5 zeigt den VHDL-Code des linken Task-Templates.

3.6 Steuerkommandos

Der Software-Teil des Betriebssystems steuert den in Hardware implementierten Teil durch sogenannte Kommandos. Ein Kommando besteht aus einer Kommandonummer und optional mehreren Konfigurationsbits. Die Tabelle 3.1 zeigt die vier implementierten Kommandos.

	ID	0	1	2	3	4	5	6	7
OS Config	00000001	Global Reset	OS Reset	FIFO Disable	Audio Sel	res	res	res	res
Reset Reader	00000010								
T0 Config	00000011	T0 Reset	T0 Enable	IP or Data	DIn 0	DIn 1	res	res	res
T1 Config	00000100	T1 Reset	T1 Enable	IP or Data	DIn 0	DIn 1	res	res	res

Tabelle 3.1: Steuerkommandos

Wenn Zustands-Daten vom FPGA zum PC übertragen werden sollen, wird dafür vom PC eine spezielle Schaltung im Hardware-Teil des OS getaktet (vgl. Abschnitt 4.3.1). Nach dem Lesevorgang muss diese mit dem `Reset Reader`-Kommando zurückgesetzt werden. Dieses Kommando besitzt als einziges *keine* Konfigurationsbits. Die anderen Kommandos `OS Config`, `T0 Config` und `T1 Config` sind mit je acht Konfigurations-Bits definiert. Obwohl nicht alle benützt werden, müssen diese trotzdem zum FPGA übertragen werden. Zur Erläuterung der einzelnen Konfigurations-Bits :

- `OS Config` :

[2] Beim Herunterladen eines partiellen Bitstreams werden die unterbrochenen Leitungen auf '1' gesetzt, weshalb dies durchaus eine Begründung hat.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- task template for left task
entity task_left is
  Port (
    t_clk :  in  std_logic;
    t_rst :  in  std_logic;
    t_finished :  out std_logic;
    t_enable :  in  std_logic;

    -- ledbar interface
    t_ledbar :  out std_logic_vector(9 downto 0);

    -- RS232 interface
    t_rs232_tx :  out std_logic;
    t_rs232_rx :  in  std_logic;

    -- FIFO interface
    t_fifo1_re :  out std_logic;
    t_fifo1_empty :  in  std_logic;
    t_fifo1_din :  in  std_logic_vector(7 downto 0);
    t_fifo2_we :  out std_logic;
    t_fifo2_full :  in  std_logic;
    t_fifo2_dout :  out std_logic_vector(7 downto 0);
    t_fifo2_sel :  out std_logic_vector(2 downto 0);

    -- IFCC signals
    data_left2right_in : in  std_logic_vector(43 downto 0);
    data_left2right_out : out std_logic_vector(43 downto 0);

    data_right2left_in : in  std_logic_vector(31 downto 0);
    data_right2left_out : out std_logic_vector(31 downto 0)
  );
end task_left;


architecture Template of task_left is

begin
  -- enter your code here ...



  -- default assignments
  -- leave these values if you don't use the ports
  t_finished <= '0';
  t_ledbar <= (others => '0');
  t_rs232_tx <= '1';
  t_fifo1_re <= '0';
  t_fifo2_we <= '0';
  t_fifo2_dout  <= (others => '0');
  t_fifo2_sel <= (others => '1');

   -- IFCCs : DON'T CHANGE !
  data_left2right_out <= data_left2right_in;
  data_right2left_out <= data_right2left_in;

end Template;
```

Abbildung 3.5: VHDL-Code des linken Task-Templates

  - *Global Reset* : '1' bedeutet Reset active. Dies betrifft sowohl OS-Frame, als auch beide Tasks. Für alle Resets gilt, dass das entprechende Reset-Signal im Chip seinen Wert nach dem Kommando behält. Will man also eine Schaltung initialisieren und danach in Betrieb setzen, muss zuerst ein Kommando mit dem Reset-Wert '1' und daraufhin eines mit dem Wert '0' gesendet werden.
  - *OS Reset* : '1' aktiviert den Reset für die Elemente des OS-Frames (Ethernet-Empfänger / -Sender, Puffer-Speicher).
  - *FIFO Disable* : '1' "friert" die kritischen Signale ein, die bei einem Task-Download unterbrochen werden (vgl. Abschnitt 7.8).
  - *Audio Sel* : '0' gibt dem linken Task-Slot den exklusiven Zugriff auf den Audio-Puffer-Speicher (Port 7000). Bei '1' gilt dies für den rechten Slot. Dadurch wird verhindert, dass die Audio-Daten, des durch den Benutzer initierten Audio-Tasks, mit denjenigen, via Ethernet empfangenen, gemischt werden.

- `T0 / T1 Config` (T0 betrifft den linken Slot, T1 analog den rechten):
  - *T0 / T1 Reset* : '1' aktiviert den Reset des Tasks.
  - *T0 / T1 Enable* : Setzt das *t_enable*-Signal des STIs auf den entsprechenden Wert.
  - *IP or Data*, *DIn 0* und *DIn 1* : Diese Bits steuern die Zuteilung des Eingang-Puffer-Speichers zum jeweiligen Task. Den Zusammenhang zwischen den Bit-Werten und dem ausgewählten Speicher zeigt Tabelle 3.2.

	dfifo1=ip	dfifo1=6000	dfifo1=6001	dfifo1=6002	dfifo1=not used
IP or Data	0	1	1	1	1
DIn 0	X	0	0	1	1
DIn 1	X	0	1	0	1

Tabelle 3.2: Zuweisung der Eingangs-FIFO-Queues (dfifo1)

Kapitel 4

OS-Frame

4.1 Überblick

Das OS-Frame bildet den Hardwareteil des Betriebssystems. Das Toplevel-Design auf dem FPGA heisst `top`. Der Kern des OS-Frames besteht aus drei Bereichen, die sinngemäss `OS-Left`, `OS-Middle` bzw. `OS-Right` heissen. Diese drei Teile werden in den folgenden Abschnitten beschrieben. Getrennt werden die Gebiete von zwei Task-Slots, welche die dynamisch rekonfigurierbare Fläche bilden. Der Aufbau der Entität `top` ist in Abbildung 4.1 dargestellt. In dieser Abbildung sind die Namen der Komponenten, wie sie im VHDL-File `top.vhd` gebraucht werden, *kursiv* gedruckt. Die **fett** dargestellten Bezeichnungen werden in diesem Bericht verwendet.

Die Einheit `top` bildet die Grundlage für den Modular Design Flow. Sie instantiert fünf Komponenten, welche den drei festen (`OS-Left`, `-Middle` und `-Right`) und den beiden dynamisch rekonfigurierbaren (Task 0 und Task 1) Modulen entsprechen. Die entsprechenden Area-Constraints sind im UCF-File (Anhang B) festgehalten.

Eine weitere wichtige Aufgabe von `top` ist das Verbinden der fünf Teile untereinander (STIs und IFFCs). Zu diesem Zweck wurden insgesamt 98 Bus-Macros instantiert. Aufgrund der grossen Anzahl von Leitungen, musste der strukturelle VHDL-Code im File `top.vhd` nach einem gut skalierbaren Muster erzeugt werden. Im Anhang B befinden sich nützliche Tabellen, die aufzeigen, wie die IFCC-Signale mit den Ports des OS-Frames und den Bus-Macros verbunden sind. Im gleichen Anhang sind analoge Tabellen für die STIs zu finden.

Die Instanzen der Bus-Macros werden jeweils mit `generate`-Statements erzeugt. Deshalb können ganz einfach weitere Leitungen zu den STIs bzw. IFCCs hinzugefügt werden. Man muss lediglich darauf achten, ob das Signal von rechts nach links, oder

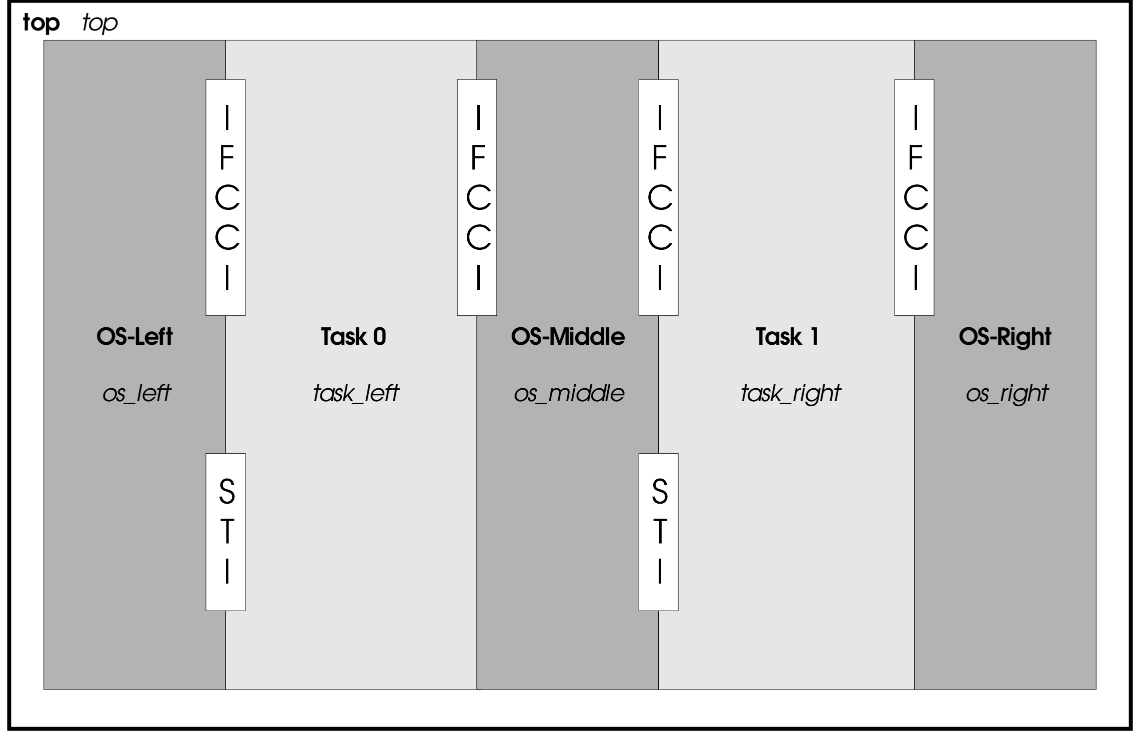


Abbildung 4.1: Überblick `top`

von links nach rechts läuft. Momentan sind noch einige Leitungen unbenutzt (Siehe Anhang B).

4.2 OS-Left

Was sich im linken Teil des OS-Frames befindet, hängt stark mit der Pin-Belegung auf dem XESS-Board zusammen [19]. Auf dieser Seite des FPGAs ist ein Audio-Codec [1] und ein Ethernet-Transceiver (Physical-Layer) [5] angeschlossen. Dementsprechend sind in der Entität `OS-Left` ein Ethernet-Empfänger (MAC-Layer), die Komponente "ARP Reply", ein Audio-Codec-Treiber, sowie die dazugehörigen Puffer-Speicher implementiert. Ein Ethernet-Sender ist ebenfalls eingebaut. Er wird bis jetzt nur von der Entität "ARP Reply" genutzt und ist deshalb auch noch logisch dieser Einheit zugeordnet.

Ausserdem stellt `OS-Left` die Schnittstelle zum linken Task-Slot, in Form des Standard-Task-Interfaces, zur Verfügung. Die Kommunikation zu den anderen Teilen des OS-Frames und zum rechten Task-Slot wird durch die Inter-Frame-Connection-Channels (IFCCs) ermöglicht. Abbildung 4.2 gibt einen Überblick über den Datenfluss in `OS-Left`.

Nach dieser Einführung betrachten wir `OS-Left` etwas genauer. Der hierarchische Aufbau der Entität zeigt Abbildung 4.3. Ein Schema der obersten Hierarchiestufe ist in Abbildung 4.4 dargestellt. Die drei instantierten Komponenten `ethaccept`, `arpreply` und `audioplay` werden in den nächsten Unterabschnitten beschrieben. Die aus Multiplexern und Flip-Flops bestehende Logik dient der Zuordnung der Tasks zu einem Ein- bzw. Ausgangs-Puffer. Ausserdem braucht es auch Logik, welche das "Einfrieren" der kritischen Signale während des Herunterladens eines Tasks ermöglicht (vgl. Abschnitt 7.8). Eine genauere Betrachtung der einzelnen Signale findet man in der folgenden Aufzählung. Die Signale sind zur besseren Übersicht entsprechend der Abbildung 4.2 gruppiert.

- **Standard-Task-Interface (STI):** Alle Signale dessen Namen mit *t0_* beginnen und nicht mit *_mid* aufhören gehören zum STI des Task-Slots 0. Ihre funktionale Bedeutung kann in Abschnitt 2.3 nachgelesen werden. Speziell ist in `OS-Left` die Rolle von *t0_fifo2_sel*. Dieses Signal bestimmt den Ausgangs-Puffer des Tasks 0. In `OS-Left` muss aufgrund des Wertes von *t0_fifo2_sel* entschieden werden, ob die Daten in den Audio-Puffer oder in einen der Daten-Puffer (in `OS-Right`) geschrieben werden. Deshalb wird ein Signal generiert, welches die Multiplexer steuert, die das Ausgangs-Puffer-Interface des STIs entweder mit dem Audio-Puffer oder mit den Daten-FIFO-Speichern verbindet (in Abbildung 4.4 violett markiert).

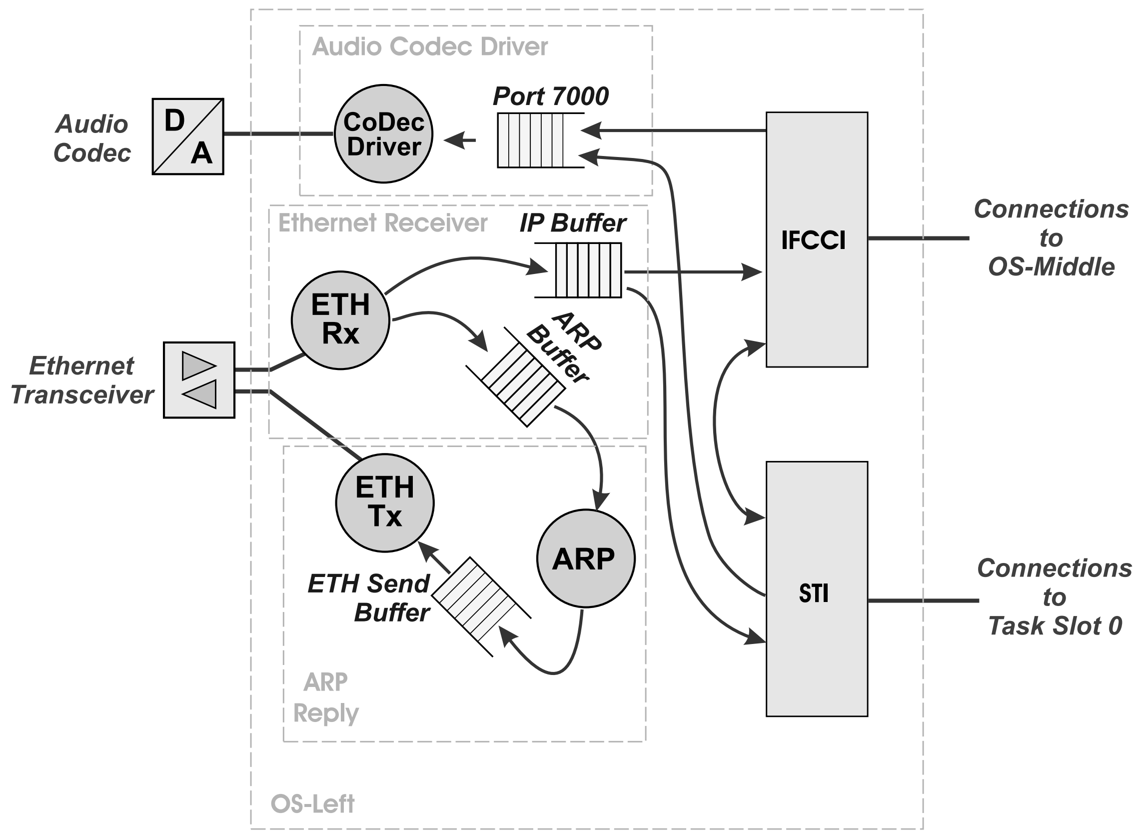


Abbildung 4.2: Datenfluss `OS-Left`

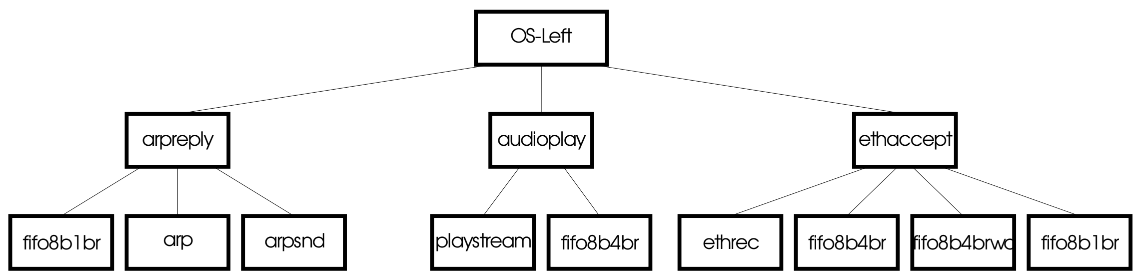


Abbildung 4.3: Hierarchie `OS-Left`

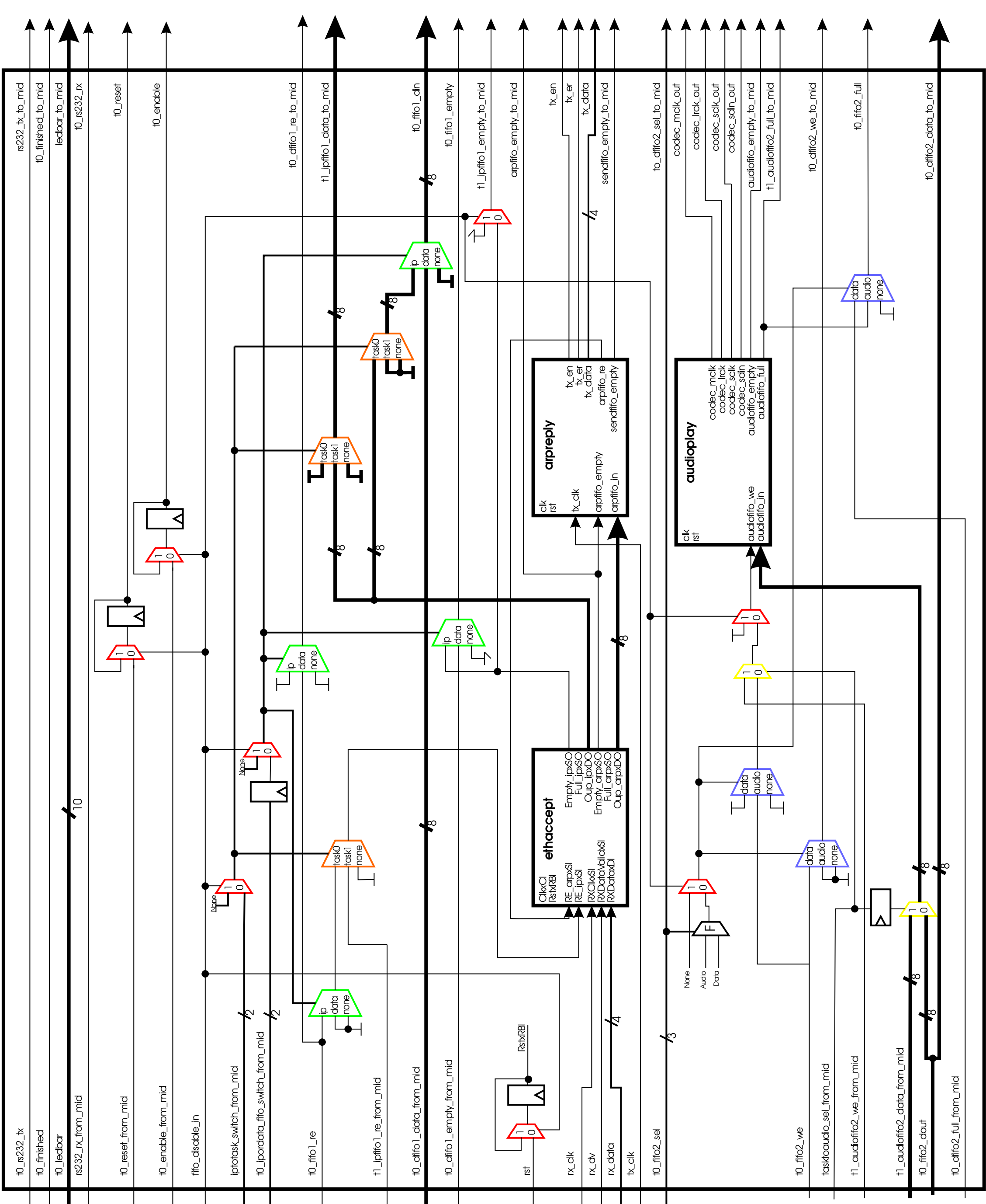


Abbildung 4.4: Schema `OS-Left`

- **Inter-Frame-Communication-Channel-Interface (IFCCI):** Die Endung *_mid* eines Signalnamens zeigt an, dass die Signale zum IFCCI gehören. Sinngemäss haben die Signale der Form **_from_mid* die Richtung `OS-Middle` → `OS-Left` und diejenigen der Form **_to_mid* die Richtung `OS-Left` → `OS-Middle`. Bei dieser Signalgruppe ist zu beachten, dass die Leitungen während des Herunterladens eines Tasks unterbrochen werden, da sie einen oder mehrere Task-Slots durchqueren (Abschnitt 7.8). Um die negativen Auswirkungen dieser Tatsache abzuschwächen, wurden verschiedene Gegenmassnahmen getroffen. Die beiden kritischen Signale zum Task-Interface *t0_reset* und *t0_enable* werden während eines Task-Downloads zwischengespeichert. Die drei Signale *iptask_switch_from_mid*, *t0_ipordata_fifo_switch_from_mid* und *tasktoaudio_sel_from_mid* steuern die Multiplexer, welche in der Abbildung 4.4 orange, grün bzw. gelb eingefärbt sind. Tabelle 4.1 zeigt die möglichen Werte der Signale und deren Auswirkungen auf den Datenfluss. Während des Herun-

Steuersignal	Wert	Auswirkung
iptotask_switch_from_mid	task0	Verbindet den IP-Puffer mit Task 0
	task1	Verbindet den IP-Puffer mit Task 1
	none	Verbindet den IP-Puffer mit den Default-Signalen [1]
t0_ipordata_fifoswitchfrom_mid	ip	Verbindet das STI des Tasks 0 mit dem IP-Puffer
	data	Verbindet das STI des Tasks 0 mit einem Daten-Puffer
	none	Verbindet das STI des Tasks 0 mit den Default-Werten [1]
tasktoaudio_sel_from_mid	0	Verbindet den Audio-Puffer mit Task 0
	1	Verbindet den Audio-Puffer mit Task 1

Tabelle 4.1: Steuersignale `OS-Left`

terladens eines Tasks werden die Steuersignale *iptotask_switch_from_mid* und *t0_ipordata_fifo_switch_from_mid* auf den Wert *none* gesetzt. Dies hat zwei Auswirkungen: Einerseits wird das "Read Enable"-Signal des IP-Puffers, sowie die Datenleitungen zu den beiden Task-Slots auf Null gesetzt. Anderseits wird auch das Empty Signal zum Task-Slot 0 auf '1' gesetzt.

Das zusätzliche Register vor einem der beiden Multiplexer, die vom Signal *tasktoaudio_sel_from_mid* gesteuert werden, berücksichtigt bei einem Umschalt-

[1] "Read Enable"-Signal des IP-Puffers = '0', Datenleitungen zu den beiden Tasks = '0' und Empty-Signal zum Task 0 = '1'

prozess die Latenz, welche die Daten gegenüber dem “Write Enable”-Signal haben.

- **Audio-Codec:** Alle Signale, die mit *codec_* beginnen gehören zum Audio-Codec und sind direkt mit der Komponente `audioplay` verbunden.

- **Ethernet-Transceiver:** Signalnamen, welche mit *tx_* beginnen, gehören zu den Signalen, welche der Ethernet-Sender benötigt und sind dementsprechend mit der Komponente `arpreply` verbunden. Analog sind Signale der Form *rx_** mit der Komponente `ethaccept` verbunden, da diese die Kommunikation mit dem Ethernet-Empfänger herstellen.

- **Clock und OS-Reset:** In Abbildung 4.4 sind die Eingänge *ClkxCI* bzw. *clk* und *RstxRBI* bzw. *rst* der instantierten Komponenten aus Gründen der Übersichtlichkeit nicht verbunden. Sie sind in Wahrheit an den Systemclock bzw. an den OS-Reset angeschlossen. Das Reset-Signal (*rst*) wird während des Herunterladens eines Tasks ebenfalls zwischengespeichert, um ein ungewolltes Zurücksetzen des Systems zu verhindern.

- ***fifo_disable_in***: Dieses Signal kommt direkt über einen Pin in die Einheit `OS-Left`. Es wird während des Downloadprozesses auf ‘1’ gesetzt, um die kritischen Signale “einzufrieren”. Die Multiplexer, welche dies bewirken sind in Abbildung 4.4 rot eingefärbt.

4.2.1 audioplay

Die Einheit `audioplay` besteht aus zwei Komponenten. Einerseits aus dem Treiber für den Audio-Codec selbst, hier `playstream` genannt, und andererseits aus dem Puffer, in welchem die Audiodaten zwischengespeichert werden (Audio-Puffer).

4.2.1.1 playstream

Die Komponente `playstream` ermöglicht das Abspielen von PCM-Audiodaten, die fogende Eigenschaften haben:

- Abtastfrequenz 12.207 kHz
- 16-Bit-Samples
- mono
- linear codiert, ohne Vorzeichen

Wie solche Daten aus einem mp3-File generiert werden, ist in Anhang A beschrieben. Die 16-Bit-Daten werden zu zwei Teilen aus dem Audio-Puffer ausgelesen und in einem 16-Bit-Eingangsregister zwischgespeichert. Danach werden die Daten seriell über *SOupxDO* an den Audio-Codec ausgegeben. Eigentlich erwartet der Audio-Codec 20 Bit breite, vorzeichenbehaftete (zweierkomplement) PCM-Samples. Deshalb wird das MSB (Vorzeichenbit) der 20 Bits konstant auf '0' gesetzt. Die Bits 19 bis 4 entsprechen dann den 16 Bits der Daten. Die drei Clocks (*MclkxSO*, *LrckxSO* und *SclkxSO*), welche der Audio-Codec benötigt, werden von einem Zähler generiert. Das Auslesen der Samples wird durch das Signal *EmptyxSO* des Audio-Puffers gesteuert.

4.2.1.2 Audio-Puffer

Der Audio-Puffer ist eine FIFO-Queue, die aus vier BlockRAMs besteht. Der Datenausgang wird vom `audioplay` genutzt. Wahlweise können Task 0 und Task 1 in diesen Puffer hineinschreiben.

4.2.2 ethaccept

Die Komponente `ethaccept` implementiert den Ethernet-Empfänger, den IP-Puffer, sowie den ARP-Puffer. Ein weiterer Zwischenspeicher ist nötig, damit die CRC-Checksumme auf ihre Richtigkeit überprüft werden kann. Diese FIFO-Queue ist mit einem synchronen "Clear"-Signal ausgerüstet, welches das Leeren des Zwischenspeichers erlaubt. So kann ein fehlerhaftes Paket verworfen werden. Korrekt empfangene IP-Pakete werden in den IP-Puffer geschrieben. Entsprechend werden korrekt empfangene ARP-Pakete in den ARP-Puffer geschrieben. Der IP-FIFO-Speicher fasst 2047 Bytes, was vier BlockRAMs entspricht. Aus Gründen der Häufigkeit und Grösse der Pakete kann die Speichergrösse der ARP-FIFO-Queue geringer ausfallen (nur ein BlockRAM).

4.2.3 arpreply

`arpreply` besteht aus den Komponenten `arp`, `arpsnd` und einem Sende-Puffer. Liegt ein ARP-Request an, das heisst, das Empty-Signal des ARP-Puffers wird auf '0' gesetzt, dann liest die Komponente `arp` das entsprechende Paket aus und generiert ein ARP-Reply-Paket. Ausserdem wird die 32-Bit-Ethernet-Adresse des Gerätes, von welchem die Anfrage kommt, aus dem Paket extrahiert und vor dem eigentlichen ARP-Reply-Paket in den Sende-Puffer geschrieben. Anschliessend liest `arpsnd` die

Ethernet-Adresse und das ARP-Reply-Paket aus. Daraus wird dann ein Ethernet-Frame generiert, welches über den Ethernet-Transceiver verschickt wird.

4.3 OS-Middle

Abbildung 4.5 gibt einen Überblick über den Datenfluss in `OS-Middle`. Der hier-

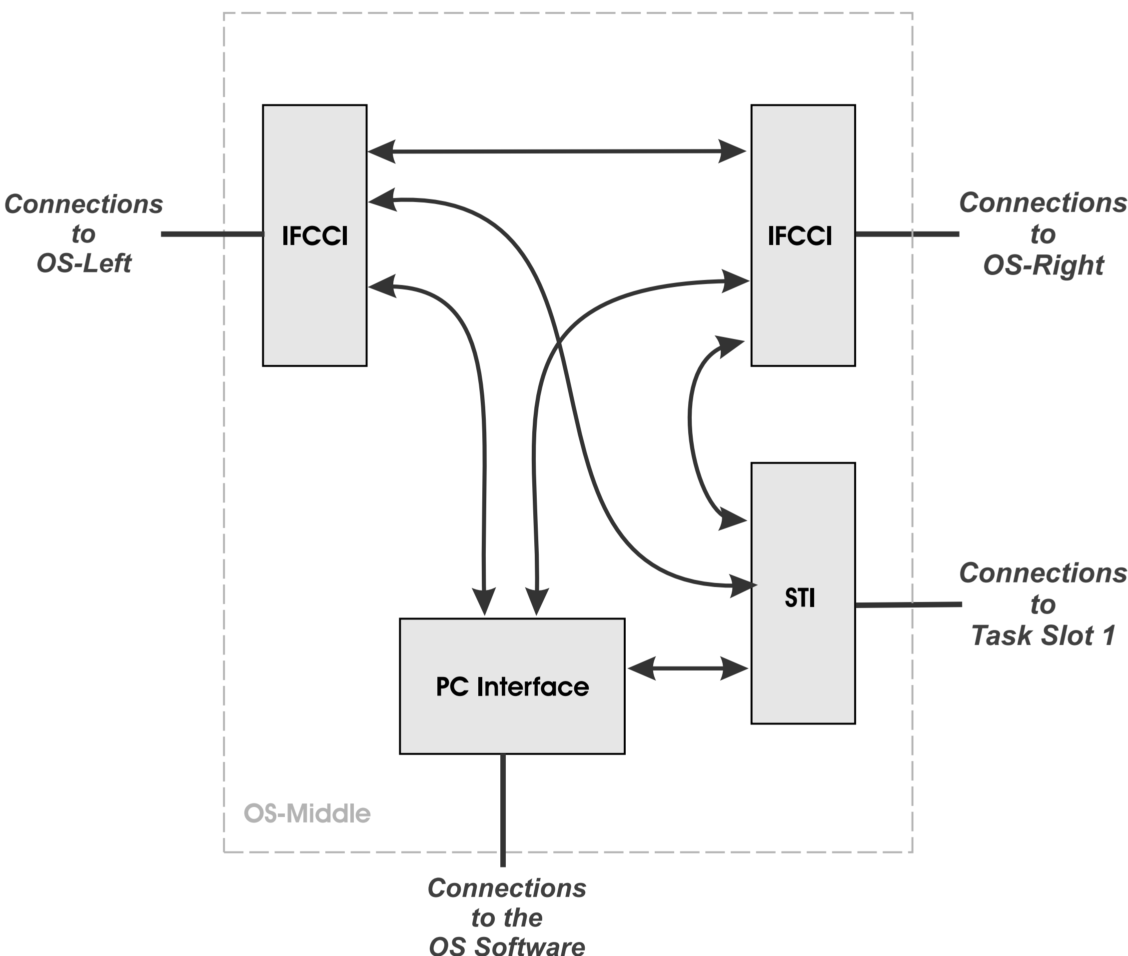


Abbildung 4.5: Datenfluss `OS-Middle`

archische Aufbau der Entität wird aus Abbildung 4.6 ersichtlich. Weil die meisten Pins der "Expansion Connectors" auf dem XESS-Board an dieses Modul angrenzen, wurde hier das Interface zum PC, auf welchem der Softwareteil des OS läuft, implementiert. Dieses wird im nächsten Unterabschnitt näher betrachtet. `OS-Middle` stellt auch das Standard-Task-Interface für den rechten Task-Slot zur Verfügung.

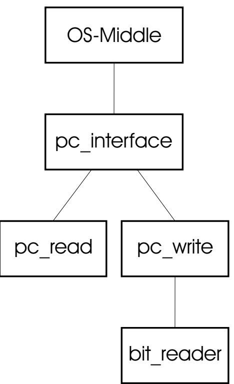


Abbildung 4.6: Hierarchie `OS-Middle`

Die Verbindungen zu den beiden anderen Teilen des OS-Frames werden wiederum von den Inter-Frame-Communication-Channels bereitgestellt. Auch in `OS-Middle` ist zusätzliche Logik nötig, um die FIFO-Puffer-Interfaces des STIs an die richtigen Puffer zu hängen. Dazu sind zwei Signale nötig:

- *t1_ipordata_fifo_switch*: Dieses Signal entspricht dem Steuersignal *t0_ipordata_fifo_switch*, welches in OS-Left die Multiplexer steuert (Siehe Abschnitt 4.2 und Abbildung 4.4). Es bestimmt, ob es sich beim Eingangs-Puffer des Tasks 1 um den IP- oder einen Daten-Puffer handelt. Demzufolge müssen die entsprechenden Signale über das jeweilige Inter-Frame-Communication-Channel-Interface (IFFCI) mit dem Modul `OS-Left` (IP-Puffer) bzw. `OS-Right` (Daten-Puffer) verbunden werden.

- *t1_audioordata_fifo_switch*: Ähnliche Überlegungen gelten auch für dieses Steuersignal. Es wird ein Daten-Puffer (Portnummer 6000, 6001 und 6002) oder der Audio-Puffer (Portnummer 7000) als Ausgangs-Puffer ausgewählt. Das Signal entspricht eigentlich dem *t1_fifo2_sel*-Signal, welches vom Task 1 generiert wird, da in erster Linie der Task selbst und nicht das OS den Ausgangs-Puffer bestimmt.

Während des Herunterladens eines Tasks werden die Multiplexer so gesteuert, dass am Eingangs-Puffer-Interface des STIs Default-Werte[2] anliegen. Diese verhindern parasitäre Effekte, die durch die kurzzeitige Unterbrechung der Leitungen zustandekommen.

[2]Die Datenbits werden auf '0' und das Empty-Signal wird auf '1' gesetzt

4.3.1 pc_interface

Im PC-Interface sind zwei Komponenten instantiert: `pc_write` und `pc_read`. Beide Komponenten werden nicht vom FPGA-Systemclock getaktet, sondern vom Signal *pc_clk*, welches vom Softwareteil des OS generiert wird. Diese beiden Clocks sind asynchron zueinander. Deshalb müssen die Daten, welche vom PC an das FPGA geschickt werden, zuerst synchronisiert werden [7]. Diese Aufgabe wird im Toplevel-Design der Einheit `pc_interface` bewältigt. Die Synchronisationsregister werden auch verwendet, um die Steuersignale zu speichern. Denn es werden nur jene Signale erneuert, die durch ein entprechendes Kommando neu gesetzt werden. Die anderen Signale behalten ihren Wert.

4.3.1.1 pc_write

Die Komponente `pc_write` generiert aufgrund der Kommandos, welche vom Softwareteil des OS gechickt werden, die Steuersignale für das OS-Frame und den Task-Slot 0 bzw. 1. Das eigentliche Lesen der 8-Bit-Kommandos übernimmt die Komponente *bit_reader*. Diese Einheit besteht im wesentlichen aus einem Schieberegister.

Zur Funktionsweise: Sobald das Signal *pc_write_en* vom PC gesetzt wird, wird das Kommando seriell ins Register eingelesen. Wenn ein Kommando vollständig empfangen wurde, wird es an den Ausgang *pc_data_read* gelegt. Um der höheren Hierarchiestufe (`pc_write`) den Empfang eines neuen Paketes zu signalisieren, wird das Signal *full_bit_read* während eines Taktzyklus auf ‘1’ gesetzt. Wenn das erste Kommando empfangen wurde, wird die Zustandsmaschine in `pc_write` gestartet. Der Wert dieses Kommandos entscheidet, ob `pc_write` in eines der drei möglichen Konfigurationsmodi (OS-, Task0- bzw. Task1-Configuration) übergeht, oder die Komponente `pc_read` mit dem entsprechenden Signal zurückgesetzt werden soll. Wenn sich `pc_write` in einem Konfigurationsmodus befindet, wird das nächste Kommando abgewartet, um die entsprechenden Signale zu setzen. Danach geht die Zustandsmaschine wieder in den Idle-Zustand über. Zur Illustration ist in Abbildung 4.7 die besprochene Zustandsmaschine vereinfacht (ohne Ausgänge) dargestellt.

4.3.1.2 pc_read

Die Zustände der Puffer, der beiden Tasks und der zwei Switches werden über diese Komponente dem PC mitgeteilt. Sobald der PC das Signal *pc_read_en* gesetzt hat, werden nacheinander folgende Signale angelegt:

1. Die Finished-Signale von Task 0 und Task 1

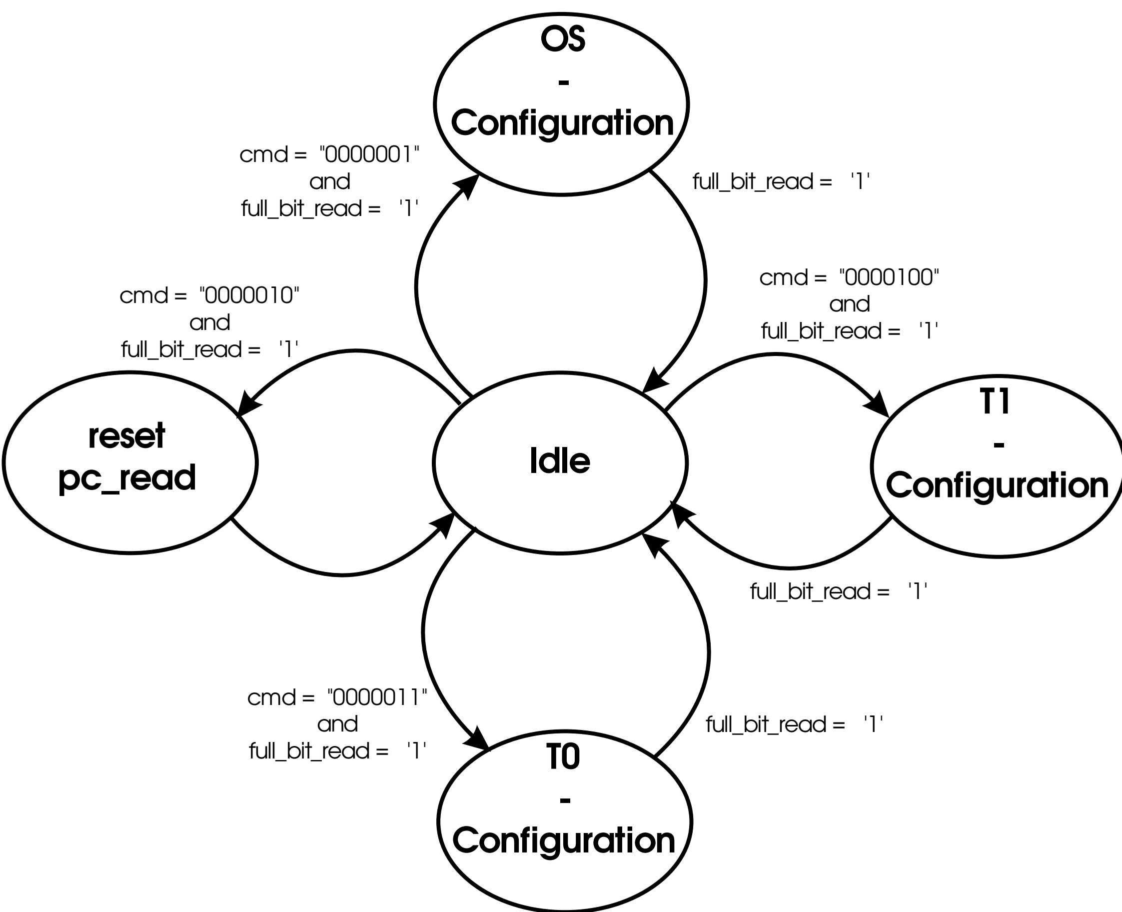


Abbildung 4.7: Zustandsmaschine `pc_write`

2. Die Empty-Signale der Puffer (IP, 6000, 6001, 6002, Audio, ARP, Send)
3. Die Signale von Switch 1 und Switch 3

4.4 OS-Right

Die beiden Switches SW1 und SW3, der LED-Bar und das RS232-Interface sind auf dem XESS-Board an Pins angeschlossen, die an dieses Modul grenzen. Da diese Ressourcen nicht von `OS-Right` gebraucht werden, sind sie direkt mit dem IFCCI verbunden.

Aufgrund der Position der BlockRAMs befinden sich die Daten-Puffer in dieser Einheit. Es wurden drei Daten-FIFO-Queues implementiert, die jeweils aus vier BlockRAMs bestehen. Ihnen wurden die Portnummern 6000, 6001 bzw. 6002 zugeordnet.

Eine Übersicht über den Datenfluss findet man in Abbildung 4.8. Ein detailliertes

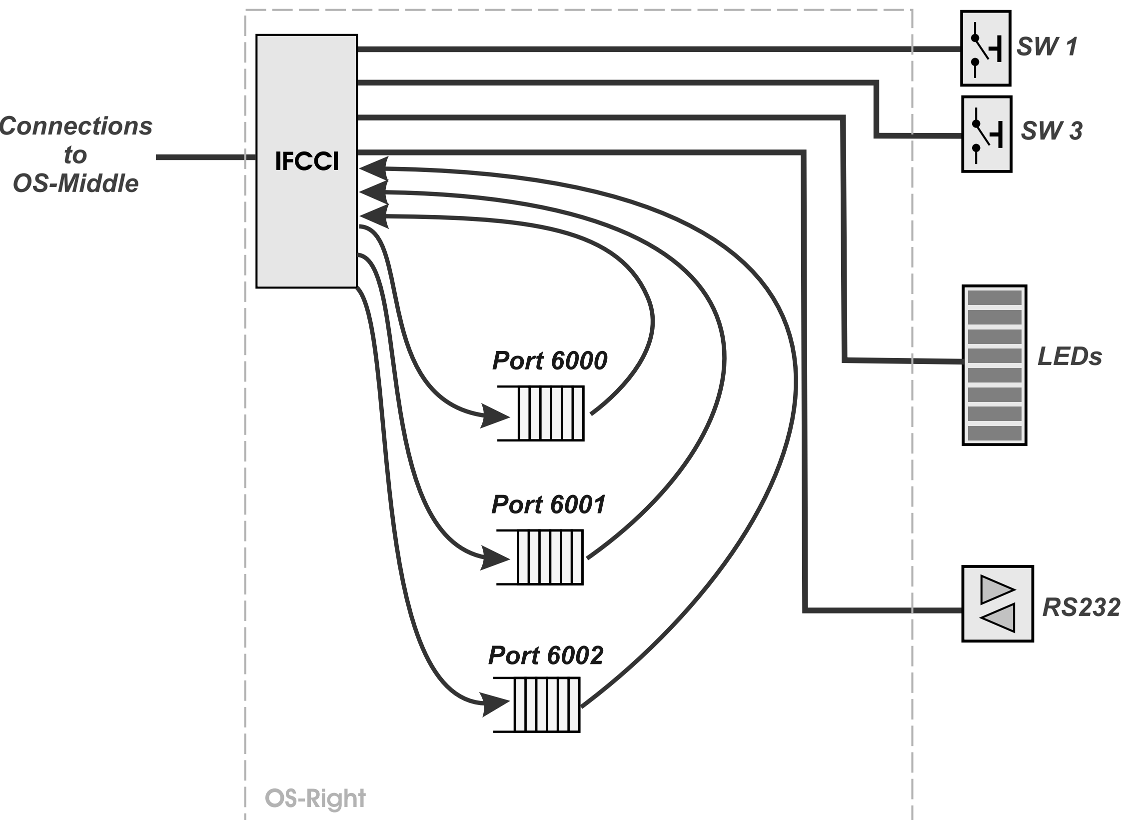


Abbildung 4.8: Datenfluss `OS-Right`

Schema ist in Abbildung 4.9 dargestellt. Daraus wird ersichtlich, dass man drei

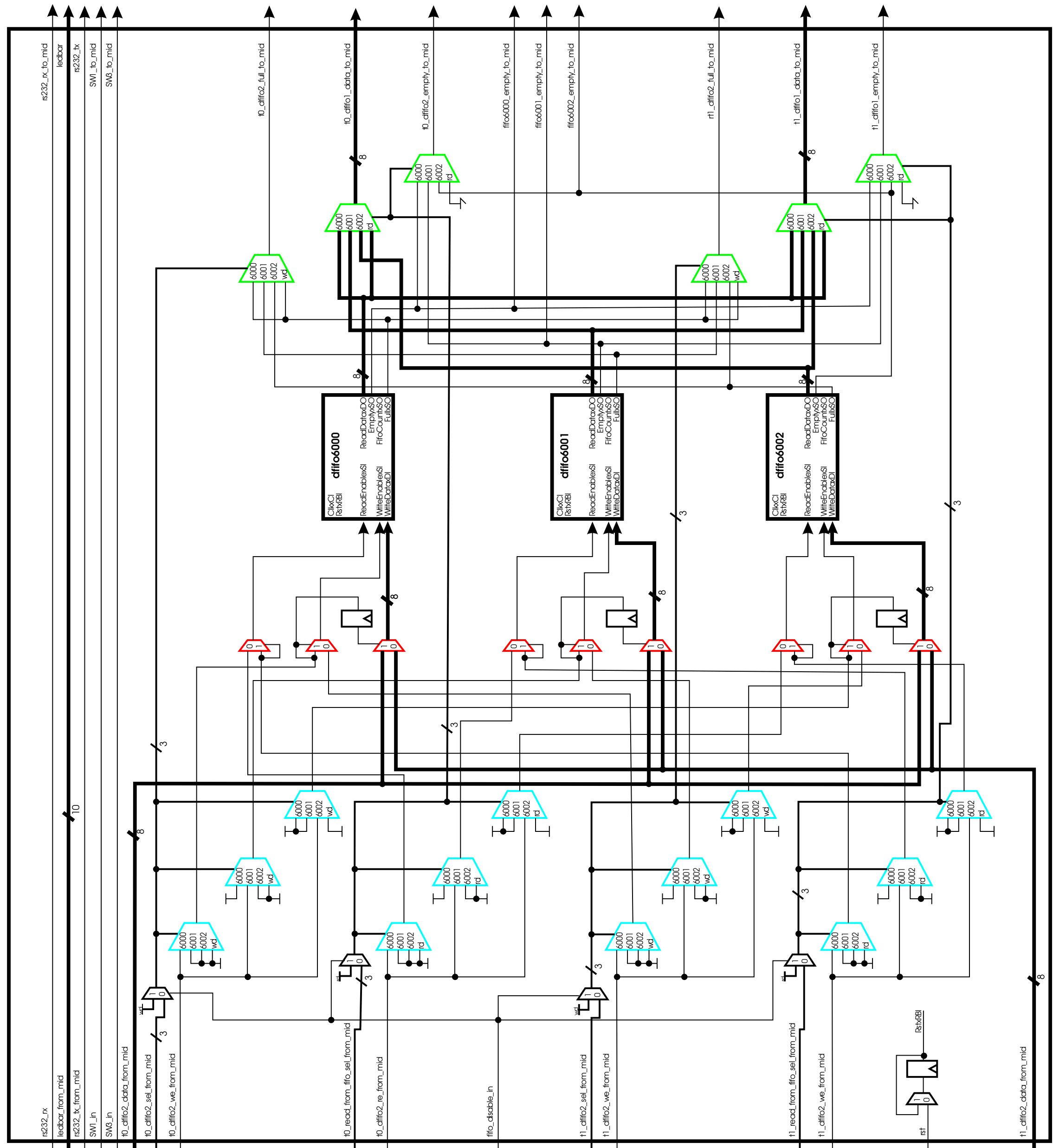


Abbildung 4.9: Schema `OS-Right`

Multiplexerstufen unterscheiden kann:

1. In einer ersten Stufe (blau eingefärbt) wird bestimmt, welcher FIFO-Speicher als Eingangs- bzw. Ausgangs-Puffer eines Tasks benutzt wird. Dazu werden

die “Read Enable”- und “Write Enable”-Signale der beiden Task-Slots zum benötigten FIFO-Speicher durchgeschaltet. Es ist auch möglich, dass ein Task keinen dieser Datenspeicher als Ein- bzw. Ausgangs-Puffer verwendet. In diesem Fall werden die entsprechenden Signale zu keinem der drei Queues durchgeschaltet. Tabelle 4.2 gibt eine Übersicht über die Steuersignale der ersten Multiplexerstufe.

2. Die Stufe 2 (rot markiert) bestimmt, wie Schreib- bzw. Lesekonflikte der beiden Tasks gelöst werden. Dabei hat der Task 0 Priorität beim Schreiben und der Task 1 hat Vorrang beim Lesen aus einem der drei FIFO-Speicher. Die Register berücksichtigen bei einem Umschaltprozess die Latenz der Daten gegenüber dem “Write Enable”-Signal.

3. Die Multiplexer nach den Puffern (grün eingefärbt) weisen den Tasks die richtigen Statussignale und den richtigen Datenausgang, der in Stufe 1 ausgewählten Eingangs- bzw. Ausgangs-Puffer, zu. Für diese Multiplexer werden die gleichen Steuersignale verwendet, wie für diejenigen in Stufe 1. Die Auswirkungen auf die Multiplexer aufgrund dieser Signalwerte können ebenfalls aus Tabelle 4.2 abgeleitet werden.

Die restlichen Multiplexer verhindern einen unkontrollierten Zugriff auf die FIFO-Speicher durch parasitäre Transitionen, während des Task-Downloads.

4.5 FIFO-Puffer

4.5.1 Einführung und Evaluation von Lösungen

In unserem System kommunizieren alle funktionellen Einheiten (Tasks, aber auch Komponenten im OS-Frame) über FIFO-Puffer miteinander. Damit die Daten während eines Task-Downloads nicht verloren gehen, müssen die Speicher in fixen Modulen (`OS-Left`, `-Middle` oder `-Right`) implementiert werden. Es gibt grundsätzlich vier Möglichkeiten, wie diese Puffer platziert werden können. Tabelle 4.3 enthält einen Vergleich dieser vier Speichervarianten.
Als wichtige Eigenschaft, welche nicht in der Tabelle aufgeführt ist, muss die Position der Speicher betrachtet werden. Dazu ein paar Überlegungen:

- Register-Arrays und Distributed RAMs können auf der ganzen Fläche des FPGAs implementiert werden.
- Die BlockRAMs sind auf dem Virtex-FPGA jeweils am rechten und linken Rand angebracht.

Steuersignal	Wert	Auswirkung
t0_dfifo2_data_from_mid	6000	Wählt dfifo6000 als Ausgangs-Puffer für Task 0 aus
	6001	Wählt dfifo6001 als Ausgangs-Puffer für Task 0 aus
	6002	Wählt dfifo6002 als Ausgangs-Puffer für Task 0 aus
	wd	”write disable”: verbietet das Schreiben von Task 0 in einen der Puffer
t0_read_from_fifo_sel_from_mid	6000	Wählt dfifo6000 als Eingangs-Puffer für Task 0 aus
	6001	Wählt dfifo6001 als Eingangs-Puffer für Task 0 aus
	6002	Wählt dfifo6002 als Eingangs-Puffer für Task 0 aus
	rd	”read disable”: verbietet das Lesen von Task 0 aus einem der Puffer
t1_dfifo2_data_from_mid	6000	Wählt dfifo6000 als Ausgangs-Puffer für Task 1 aus
	6001	Wählt dfifo6001 als Ausgangs-Puffer für Task 1 aus
	6002	Wählt dfifo6002 als Ausgangs-Puffer für Task 1 aus
	wd	”write disable”: verbietet das Schreiben von Task 1 in einen der Puffer
t1_read_from_fifo_sel_from_mid	6000	Wählt dfifo6000 als Eingangs-Puffer für Task 1 aus
	6001	Wählt dfifo6001 als Eingangs-Puffer für Task 1 aus
	6002	Wählt dfifo6002 als Eingangs-Puffer für Task 1 aus
	rd	”read disable”: verbietet das Lesen von Task 1 aus einem der Puffer

Tabelle 4.2: Steuersignale der ersten Multiplexstufe

- Ein externes RAM muss mit dem OS-Frame über angrenzende Pins kommunizieren können.

Speichervariante	Flächenbedarf pro Speichervolumen auf FPGA	Geschwindigkeit
Register-Array	sehr gross	sehr schnell
externes SRAM	extern	langsam
Distributed RAM	klein	schnell
BlockRAM	sehr klein	schnell

Tabelle 4.3: Vergleich FIFO-Speicher-Varianten

Aufgrund dieser Überlegungen sind wir zu folgenden Schlussfolgerungen gelangt:

1. Die FIFO-Queues in einem externem SRAM zu implementieren ist nicht umsetzbar, da die Pins, welche zu den RAMs auf dem XESS-Board führen, zum Teil schon belegt sind und nicht an das OS-Frame angrenzen. Dadurch sind die kritischen Leitungen zu den RAMs während eines Task-Downloads nicht kontrollierbar. Weiter würde die geringe Geschwindigkeit einen beträchtlichen Performance-Nachteil mitsichbringen.

2. Register-Arrays könnten nur für sehr geringe Datenmengen eingesetzt werden, da sonst der Flächenbedarf sehr gross wäre.

3. Distributed RAMs als FIFO-Speicher zu verwenden ist grundsätzlich keine schlechte Lösung. Da aber nicht unnötig Fläche im OS-Frame verbraucht werden sollte, und die BlockRAMs sowieso schon vorhanden sind, haben wir uns für letztere Variante entschieden.

4.5.2 Implementation

Wie in Abschnitt 4.5.1 erläutert, haben wir uns für die Variante mit den BlockRAMs entschieden. Das Design basiert auf demjenigen für synchrone FIFO-Speicher von Xilinx. Einzelheiten dazu findet man in [21]. Gegenüber dem Original wurden folgende Änderungen vorgenommen:

1. Wir stellten fest, dass sich der FIFO-Speicher nicht ganz leeren liess. Dieses Problem wurde behoben.

2. Die Variante von Xilinx enthielt nur ein BlockRAM. Wir haben eine weitere FIFO-Queue implementiert, die aus vier BlockRAMs besteht. Demzufolge stehen FIFO-Puffer mit 511 und 2047 Bytes Speicherkapazität zur Verfügung. Weshalb nicht die volle Kapazität der BlockRAMs (512 bzw. 2048 Bytes) ausgeschöpft wird, kann ebenfalls in [21] nachgelesen werden.

3. Für den Zwischenspeicher im Ethernet-Empfänger musste noch ein synchrones Clear-Signal eingebaut werden (Siehe Abschnitt 4.2.2). Deshalb wurde noch eine dritte Variante implementiert, welche ebenfalls 511 Bytes Speicherkapazität aufweist.

Das Interface der FIFO-Queues sieht folgendermassen aus:

- Eingänge:
  - Clock
  - Reset
  - Write Enable
  - Read Enable
  - 8-Bit-Dateneingang
  - Clear (nur in einer Variante)
- Ausgänge:
  - 8-Bit-Datenausgang
  - Statussignal Full
  - Statussignal Empty
  - Füllstand des Speichers (wird in unserem System nicht benutzt)

Noch zwei Bemerkungen zum Schreib- bzw. Leseprozess:

- Schreibprozess: Das "Write Enable"-Signal muss einen Taktzyklus vor den Daten an den FIFO-Speicher-Eingang angelegt werden.
- Leseprozess: Einen Taktzyklus, nachdem die FIFO-Queue das setzen des "Read Enable"-Signals erkannt hat (Während einer aktiven Clock-Flanke ist das "Read Enable" Signal am Eingang des FIFO-Speichers auf '1'), liegen die Daten am Ausgang an.

Kapitel 5

Tasks

5.1 AES $^{-1}$

5.1.1 Einführung

Der Advanced Encryption Standard (AES) ist der Nachfolger des Data Encryption Standards (DES). Er ging im Herbst 2000 als Sieger eines Wettbewerbs hervor, der vom National Institute of Standards and Technology (NIST) international ausgeschrieben wurde. Es handelt sich dabei um einen symmetrischen Blockalgorithmus der 128-Bit-Datenblöcke verschlüsselt. Im Standard sind Schlüssellängen von 128, 192 bzw. 256 vorgesehen. Je nach Wahl der Schlüssellänge wird der Algorithmus auch als *AES-128*, *AES-192* und *AES-256* bezeichnet. Nähere Informationen zur Entstehung und zur Funktionsweise können in [10] gefunden werden.

In einer früheren Semesterarbeit haben wir den *AES-128* Algorithmus als ASIC umgesetzt [4].[1] Da wir deshalb sehr gute Kenntnisse dieses Algorithmus hatten, entschieden wir uns, einen Entschlüsselungs-Task zu implementieren. Dadurch war es möglich in relativ kurzer Zeit einen komplexen Task herzustellen. Den VHDL-Code konnten wir jedoch nicht eins zu eins übernehmen, da das Design über 80% der gesamten FPGA-Fläche in Anspruch genommen hätte. Aus Optimierungs-Gründen sind der Ent- und Verschlüsselungspfad im ASIC-Design nicht getrennt. Demzufolge mussten diese beiden Komponenten voneinander getrennt und stark verkleinert werden.

[1] In einer anderen Semesterarbeit wurde der zweitplatzierte Serpent-Algorithmus implementiert. Ein Vergleich der beiden Chips ist in [9] zu finden.

5.1.2 Grundlagen

In diesem Abschnitt wird kurz der Ablauf einer Entschlüsselung mit dem AES im ECB-Mode[2] erläutert. Dabei soll nur ein kurzer Überblick über den komplexen Algorithmus gegeben werden, da eine genaue Beschreibung den Rahmen dieses Berichts sprengen würde. Für ein vertieftes Verständnis muss hier auf weitere Literatur verwiesen werden [2]. Insbesondere sollte der Bericht zu unserer Semesterarbeit, auf welcher diese Implementation beruht, beigezogen werden [4].

Am Eingang liegen jeweils 128 Bit lange, verschlüsselte Datenblöcke an. Diese durchlaufen eine Reihe von Transformationen, die zusammen als "Runde" bezeichnet werden. Ein Datenblock muss eine solche Runde zehnmal absolvieren. In jedem Durchgang, sowie am Anfang der Entschlüsselung, werden die Daten, mit Hilfe einer XOR-Operation, mit einem 128 Bit breiten Teilschlüssel verknüpft. Dieser Teilschlüssel ist jeweils unterschiedlich. Die benötigten elf Teilschlüssel werden aus einem 128-Bit-Schlüssel berechnet.

5.1.3 Funktionsweise

Nach dem Herunterladen wartet der AES-Decryption-Task (AES^{-1}), bis alle elf Teilschlüssel, aus einem konstanten 128-Bit-Schlüssel, berechnet wurden. Anschliessend liest er jeweils 128-Bit-Datenblöcke aus dem FIFO-Puffer mit der Portnummer 6001 und schreibt den entschlüsselten Klartext in den Puffer mit der Portnummer 6002. Sobald der Eingangs-Puffer leer ist (Empty Signal wird auf '1' gesetzt), geht der Task in den Finished-Zustand über und das Ausgangssignal *t_finished* wird auf '1' gesetzt. Der Task kann durch Anlegen des Reset-Signals wieder in den Anfangszustand gebracht werden, in welchem er neue Daten aus dem Puffer auslesen kann.

5.1.4 Implementation

5.1.4.1 Überblick

Der hierarchische Aufbau der Entitäten wird aus Abbildung 5.1 ersichtlich. Das Top Level Design heisst `AESDecTask`. Die Komponenten `keycontroller` und `keygenerator` sind für die Berechnung der 128-Bit-Teilschlüssel aus dem 128-Bit-Schlüssel zuständig. Alle elf Teilschlüssel werden im `keyregister` gespeichert. Das eigentliche Herzstück des Tasks bildet die Komponente `decround`. Sie implementiert eine "Runde". Dieser Datenpfad wird von der Einheit `decroundcontroller` gesteuert. Die Komponenten `conv32to8`, `conv8to32` und `fifoadapter` passen den Task an

[2]Der Electronic-Code-Book-Mode (ECB) ist der einfachste aller möglichen Modi [18].

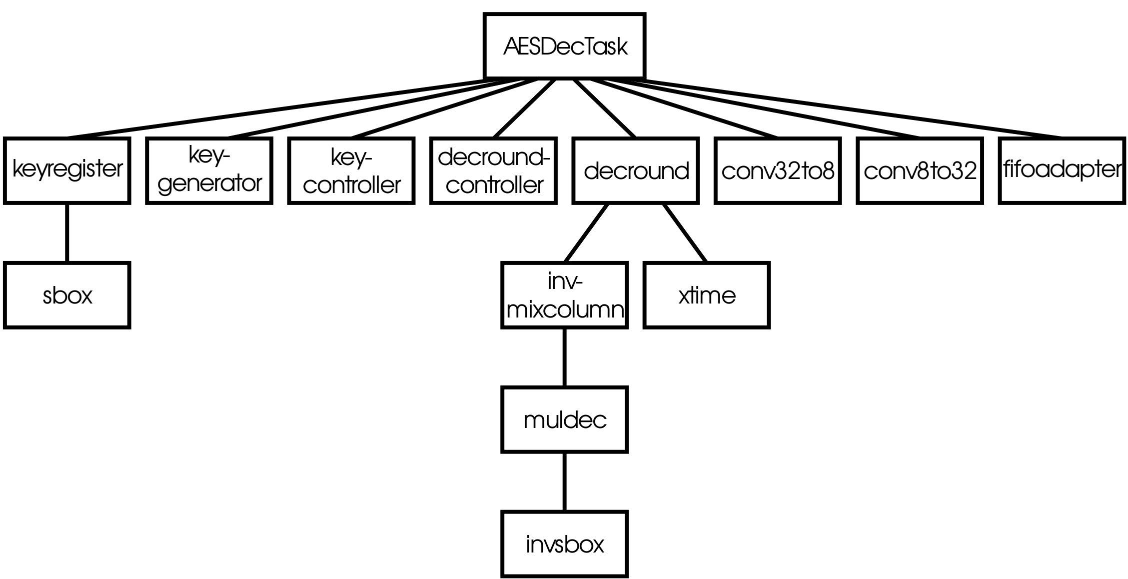


Abbildung 5.1: Hierarchischer Aufbau des AES^{-1}-Tasks

das Standard-Task-Interface (STI) an. Dazu gehört eine 8-Bit-Datenbreite, sowie das richtige Timing um die FIFO-Puffer anzusteuern. Die genaue Funktionsweise der einzelnen Komponenten wird in den folgenden Abschnitten erläutert.

5.1.4.2 AESDecTask

Das Schema der Entität `AESDecTask` ist in Abbildung 5.2 aufgezeichnet. In dieser obersten Hierarchiestufe des Designs werden acht Komponenten miteinander verbunden. Das Ausgangssignal *t_fifo2_sel* wird konstant auf den Wert "010" gesetzt, was der Portnummer 6002 entspricht. Um Timingproblemen vorzubeugen, sind an den Ausgängen kritischer Signalpfade Register angebracht. Am Datenausgang befinden sich zwei Registerblöcke, um den Daten *t_fifo2_dout* einen zusätzlichen Taktzyklus Latenz hinzuzufügen. Das ist notwendig, weil das Steuersignals *t_fifo2_we* aufgrund der FIFO-Puffer-Interfaces einen Taktzyklus vor den Daten am Ausgang erscheinen muss. Zusätzliche Logik, hier *Key Input Logic* (KIL) genannt, musste eingebaut werden, da die Schlüsselberechnung direkt im Task eingebaut ist. Die KIL initialisiert nach dem lösen des Resets die Schlüsselberechnung, indem sie das *NewKeyxSI* Signal des `keycontrollers` für einen Taktzyklus auf '1' setzt. Anschliessend werden die vier 32-Bit-Teile des 128-Bit-Schlüssels zu den richtigen Zeitpunkten an den Eingang der Komponente `keygenerator` angelegt. Alle Bits des Schlüssels sind dabei konstant auf den Wert '0' gesetzt. Ein Multiplexer weist der Komponente `keygenerator` die aktuelle Teilschlüssel-Nummer zu. Diese wird

während der Schlüsselgenerierung (Schlüssel ins `registerarray` einlesen) vom `key-controller` und während der Entschlüsselung (Schlüssel aus dem `registerarray` auslesen) vom `decroundcontroller` erzeugt.

5.1.4.3 decround

Das Schema der Einheit `decround` ist in Abbildung 5.3 dargestellt. Der interne Datenpfad wurde auf 32 Bit festgelegt. Dies entspricht einer möglichst kleinen, aber aufgrund des Algorithmus natürlichen und deshalb sinnvollen Datenbreite. Vor allem weil der Dateneingang bzw. -ausgang der Transformation `InvMixColumn` 32 Bit breit ist. Die Reduktion der internen Datenbreite von 128 Bit auf 32 Bit bringt eine Flächeneinsparung von etwa 57%. Man vergleiche dazu die Auszüge aus den beiden “Synthesis Reports” des Xilinx Synthese Tools in Abbildung 5.4. Dieser Designentscheid war nötig, um die Flächenanforderungen zu erfüllen, die von einem Taskslot vorgegeben werden. Den Preis, den man dafür zahlt, ist eine vierfache Latenz im Vergleich zur 128-Bit-Variante.

Auf Pipelining wurde verzichtet, da die Timinganforderungen, welche durch die 12.5 MHz Taktfrequenz des FPGAs vorgegeben werden, problemlos eingehalten werden können. Beschreibungen zu den Teilblöcken `InvMixColumn` und `InvSBox` können in [4] gefunden werden. Es werden zwei 128-Bit-Register verwendet, da die Operation `InvShiftRows` die Reihenfolge der Bits innerhalb eines Datenblocks ändert. Weil nicht alle 128 Bit gleichzeitig bearbeitet werden, würde dies zu einem Überschreiben von Bits führen. Deshalb werden die Resultate nach jeder Runde abwechslungsweise in eines der beiden Register geschrieben.

5.1.4.4 decroundcontroller

Die Komponente `decroundcontroller` ist eine Zustandsmaschine, die aus 46 Zuständen besteht. Die Bedeutung der Ausgangssignale wird im Folgenden beschrieben (Man betrachte dazu auch Abbildung 5.2):

- *DataReadyxSO*: Dieses Signal ist genau dann ‘1’, wenn die ersten 32 Bit (127 downto 96) eines vollständig entschlüsselten Datenblocks am Ausgang der Einheit `decround` anliegen. Die restlichen 3 Teile des Datenblocks folgen entsprechend in den nächsten 3 Taktzyklen, jedoch ohne, dass das Signal *DataReadyxSO* gesetzt ist.

- *InpSelxSO*: Dient zum Einlesen der verschlüsselten Daten. Es steuert die Multiplexer in `decround`, welche den Eingang zum ersten XOR durchschaltet. *InpSelxSO* wird für einen Taktzyklus auf ‘1’ gesetzt, wenn neue 32 Bit breite

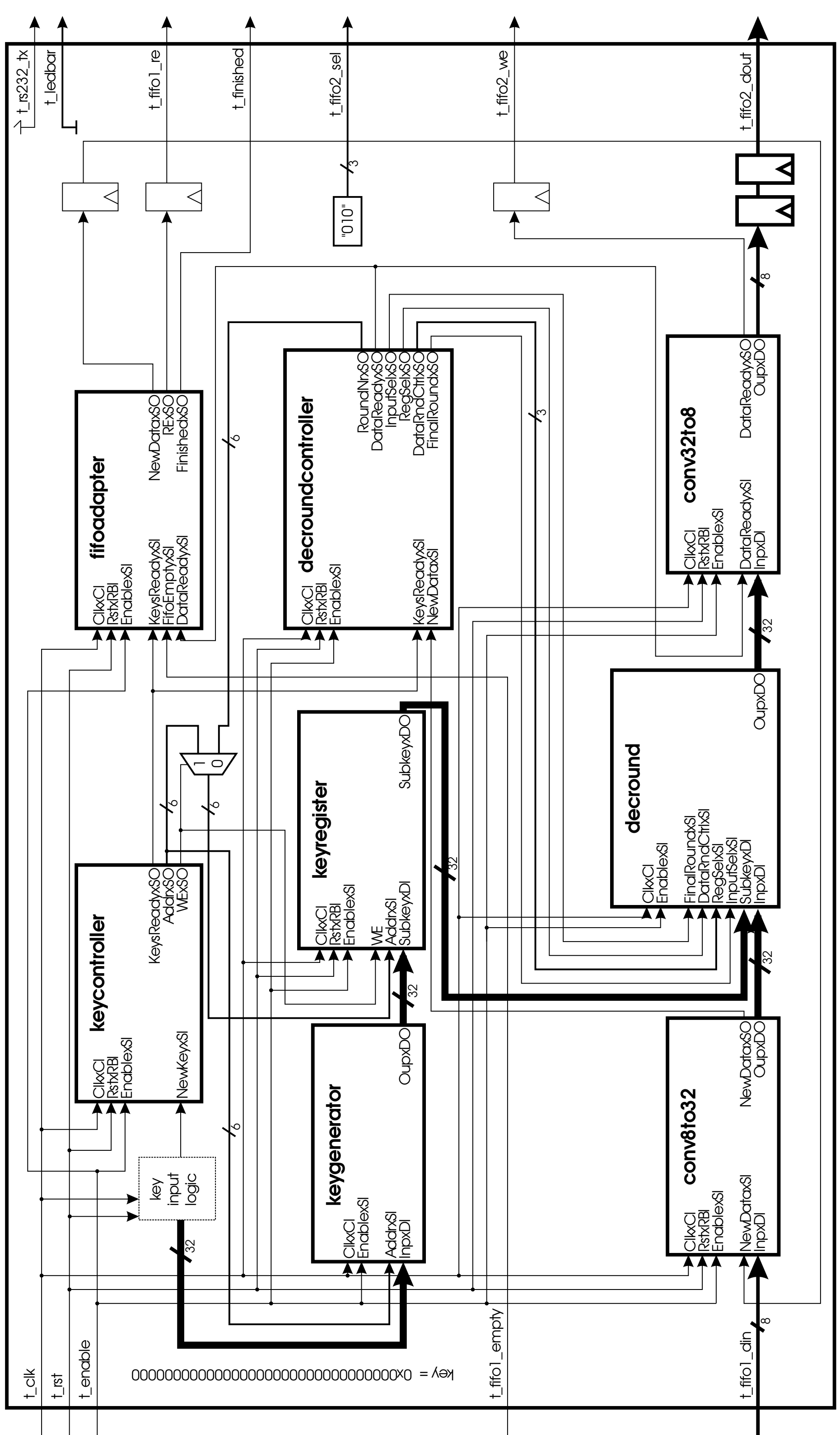


Abbildung 5.2: Schema “AES Decryption”-Task

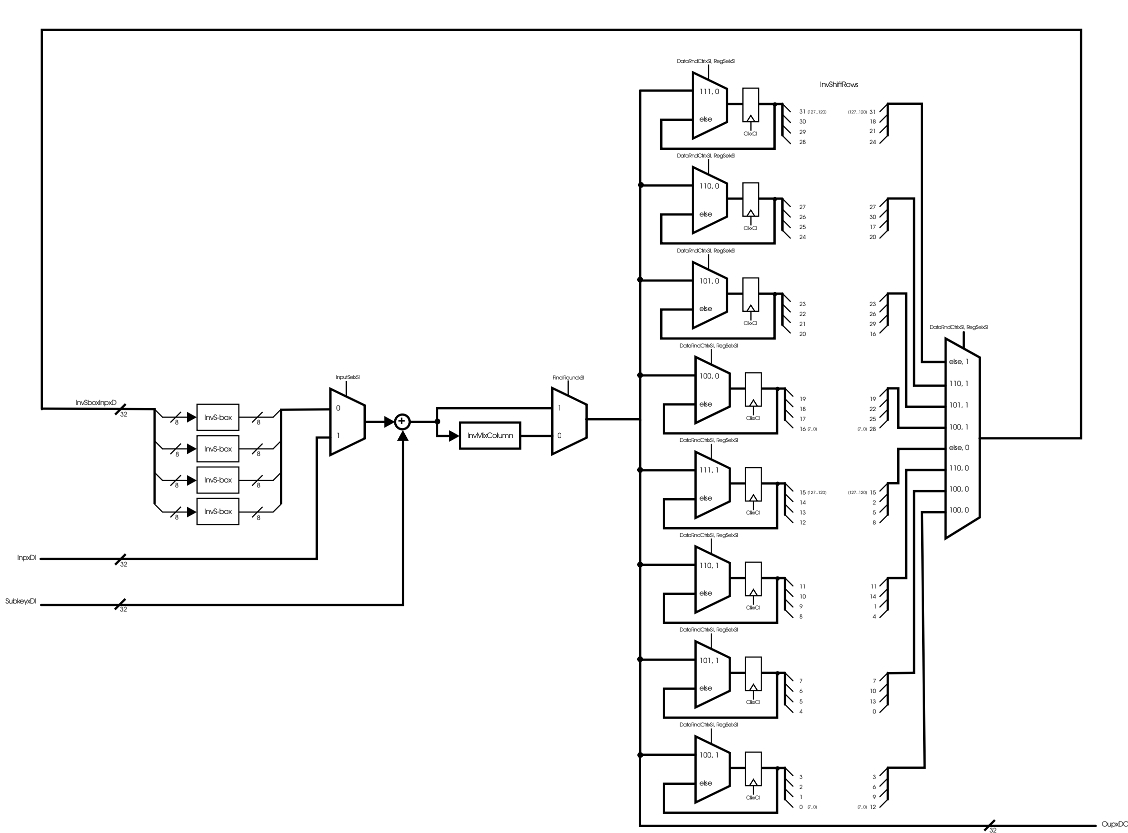


Abbildung 5.3: RTL-Schema `decround`

`decround` (128 Bit, Optimization Goal: Area)

```
Device utilization summary:
---------------------------

Selected Device : v800hq240-5

 Number of Slices:                     1642  out of   9408    17%
 Number of Slice Flip Flops:            128  out of  18816     0%
 Number of 4 input LUTs:               3171  out of  18816    16%
```

`decround` (32 Bit, Optimization Goal: Area)

```
Device utilization summary:
---------------------------

Selected Device : v800hq240-5

 Number of Slices:                      706  out of   9408     7%
 Number of Slice Flip Flops:            256  out of  18816     1%
 Number of 4 input LUTs:               1328  out of  18816     7%
```

Abbildung 5.4: Grössenvergleich der 128- mit der 32-Bit-Variante von `decround`

Daten am Eingang von `decround` anliegen. Pro Datenblock wird dieses Signal also insgesamt während vier (nicht aufeinanderfolgenden) Taktzyklen auf '1' gesetzt.

- *FinalRoundxSO*: Steuert den Multiplexer, welcher entscheidet, ob die Daten die Transformation `InvMixColumn` durchlaufen oder nicht. Der Name Final-Round kommt daher, weil beim Verschlüsselungsalgorithmus die Daten in der letzten der zehn Runden nicht mit der Umkehrfunktion `MixColumn` transformiert werden. Das heisst umgekehrt, beim Entschlüsseln wird das Signal für jeden 32-Bit-Teilblock jeweils bei der ersten Runde auf '1' gesetzt. Zusätzlich wird das Signal gesetzt, wenn neue Daten eingelesen werden, da diese zuerst unverändert in eines der beiden 128-Bit-Register eingelesen werden müssen.

- *RegSelxSO*: Das Signal wählt das 128-Bit-Register aus, in welches die Daten nach einer abgeschlossenen Runde zwischengespeichert werden sollen. Dabei wechseln sich die beiden Register aus oben beschriebenen Gründen nach jeder Runde ab.

- *DataRndCtrlxSO*: Gibt an, in welche 32-Bit-Teilregister innerhalb eines 128-Bit-Registers die soeben berechneten Daten geschrieben werden sollen. Dabei wird mit den höchstwertigsten 32 Bit begonnen.

- *RoundNrxSO*: Wählt den Teilschlüssel aus, der einen Taktzyklus später am Eingang *SubkeyxDI* von `decround` anliegen soll.

5.1.4.5 keygenerator

Der `keygenerator` erzeugt aus dem 128-Bit-Schlüssel zehn weitere Teilschlüssel. Das Schema wird aus Abbildung 5.5 ersichtlich. Die Datenbreite wurde auf 32 Bit reduziert, um sie derjenigen von `decround` anzupassen.

5.1.4.6 keycontroller

Der `keycontroller` steuert den Ablauf der Berechnung der elf Teilschlüssel. Dabei bedeuten die Steuersignale folgendes:

- *KeysReadyxSO*: Wird auf '1' gesetzt, sobald sämtliche elf Schlüssel im `keyregister` bereitstehen. Es dient als Startsignal, welches dem `decroundcontroller` erlaubt mit dem Entschlüsseln der Daten zu beginnen.

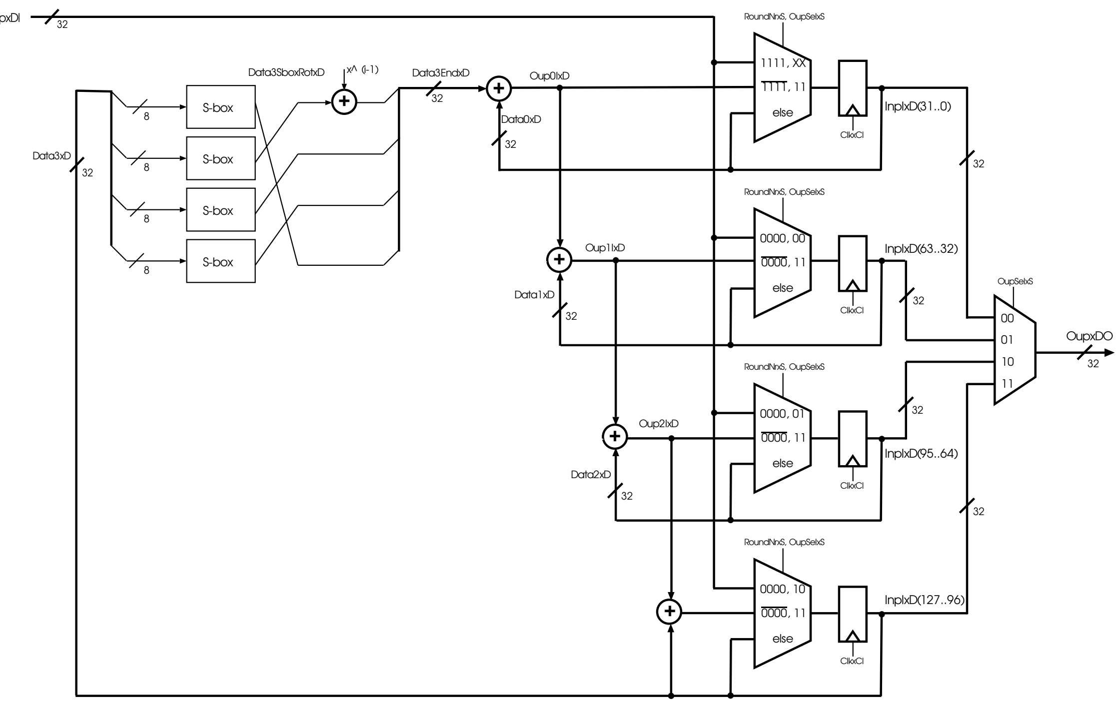


Abbildung 5.5: RTL-Schema Keygenerator

- *WExSO* und *AddrxSO*:Dies Signale dienen zum Speichern der Teilschlüssel ins `keyregister`. Das Interface wurde so gemacht, dass die Schlüssel später ohne grossen Aufwand auch in einem speziellen Schlüsselspeicher, z.B. im OS-Frame, gespeichert werden können. Ausserdem steuert das Signal *AddrxSO* die Multiplexer im `keygenerator`.

5.1.4.7 keyregister

Alle elf Teilschlüssel werden abgespeichert. Theoretisch wäre es auch möglich die Schlüssel "on the fly" zu berechnen. Im Design, auf welchem diese Implementation basiert, wurde ebenfalls die erste Variante gewählt (vgl. mit der Begründung in [4]). Bezüglich Designzeit war es deshalb sinnvoll diese Lösung zu bevorzugen.

Ein weiterer Vorteil des Abspeicherns der Schlüssel ist, dass man die Berechnung der Schlüssel problemlos von der Entschlüsselungslogik trennen kann, da mit dem Abspeichern der Schlüssel bereits eine geeignete Schnittstelle geschaffen wurde. Grundsätzlich bietet das Virtex FPGA drei Möglichkeiten (Registerarray, Distributed RAM, BlockRAM) die 11*128-Bit-Schlüssel abzuspeichern. Alle drei Varianten wurden untersucht. Die Ergebnisse sind in Abbildung 5.6 zusammengetragen. Gedanken dazu folgen in den nächsten Unterabschnitten:

```
Design Summary - BlockRAM
-------------------------
   Number of Slices:              1,174 out of  9,408   12%
   Number of Slice Flip Flops:      413 out of 18,816    2%
   Total Number 4 input LUTs:     2,262 out of 18,816   12%
      Number used as LUTs:                     2,253
      Number used as a route-thru:                 9
   Number of bonded IOBs:             8 out of    166    4%
      IOB Flip Flops:                              7
   Number of Block RAMs:              5 out of     28   17%
   Number of GCLKs:                   1 out of      4   25%
   Number of GCLKIOBs:                1 out of      4   25%
Total equivalent gate count for design:  101,852
Additional JTAG gate count for IOBs:  432
Peak Memory Usage:  88 MB

Design Summary - Register
-------------------------
   Number of Slices:              2,425 out of  9,408   25%
   Total Number Slice Registers:  1,885 out of 18,816   10%
      Number used as Flip Flops:               1,853
      Number used as Latches:                     32
   Total Number 4 input LUTs:     4,713 out of 18,816   25%
      Number used as LUTs:                     4,704
      Number used as a route-thru:                 9
   Number of bonded IOBs:             8 out of    166    4%
      IOB Flip Flops:                              7
   Number of Block RAMs:              4 out of     28   14%
   Number of GCLKs:                   1 out of      4   25%
   Number of GCLKIOBs:                1 out of      4   25%
Total equivalent gate count for design:  111,854
Additional JTAG gate count for IOBs:  432
Peak Memory Usage:  100 MB

Design Summary - Distributed RAM
--------------------------------
   Number of Slices:              1,223 out of  9,408   12%
   Number of Slice Flip Flops:      413 out of 18,816    2%
   Total Number 4 input LUTs:     2,360 out of 18,816   12%
      Number used as LUTs:                     2,287
      Number used as a route-thru:                 9
      Number used as 16x1 RAMs:                   64
   Number of bonded IOBs:             8 out of    166    4%
      IOB Flip Flops:                              7
   Number of Block RAMs:              4 out of     28   14%
   Number of GCLKs:                   1 out of      4   25%
   Number of GCLKIOBs:                1 out of      4   25%
Total equivalent gate count for design:  93,864
Additional JTAG gate count for IOBs:  432
Peak Memory Usage:  88 MB
```

Abbildung 5.6: Vergleich der Schlüsselspeicher-Varianten

Registerray Auf den ersten Blick erscheint diese Variante als nicht sehr sinnvoll, da der Platzbedarf sehr gross ist. Für das Testen des Systems kann es jedoch notwendig sein, einen Task zu haben, der einen Task-Slot praktisch vollständig ausfüllt. Deshalb wurde diese Variante implementiert und im Task verwendet.

Distributed RAM Das Virtex FPGA bietet diese platzsparende Möglichkeit an, ein RAM zu implementieren. Der Flächenbedarf ist nur wenig grösser als bei der BlockRAM Variante. Da die BlockRAMs zur Inter-Task-Kommunikation gebraucht werden, stellen die Distributed RAMs eine echte Alternative dar. Auch diese Lösung wurde implementiert. Wenn diese Variante verwendet werden soll, muss im Task die Komponente `keyregister` durch `keydistram` ersetzt werden.

BlockRAM Da sich die BlockRAMs im Gebiet des OS-Frames befinden und nur unwesentliche Flächenvorteile gegenüber den Distributed RAMs aufweisen, wurde auf die Umsetzung verzichtet. Interessant wird diese Möglichkeit, wenn die Schlüsselberechnung getrennt von der Entschlüsselung als Task implementiert wird (Wobei auch hier die Distributed RAM Variante nicht ausser Acht gelassen werden sollte).

5.1.4.8 conv8to32

Diese Komponente passt den internen 32-Bit-Datenbus an das 8-Bit-FIFO-Daten-Interface an. Dazu wird auch das *NewDataxS* Signal verzögert, welches das Entschlüsseln eines neuen Datenblocks auslöst.

5.1.4.9 conv32to8

Verzögert das *DataReadyxS* Signal und reduziert die Datenbreite am Ausgang von 32 auf 8 Bit.

5.1.4.10 fifoadapter

In dieser Einheit ist die Zustandsmaschine implementiert, welche das Auslesen von neuen Daten aus dem FIFO-Puffer handhabt. Nach dem Verarbeiten der Daten verweilt die Zustandmaschine in einem Finished-Zustand, wie dies in Abschnitt 5.1.3 beschrieben wurde. Eine vereinfachte Darstellung der Zustandsmaschine (ohne Ausgänge) ist in Abbildung 5.7 dargestellt. Die Ausgänge werden in der folgenden Auflistung beschrieben:

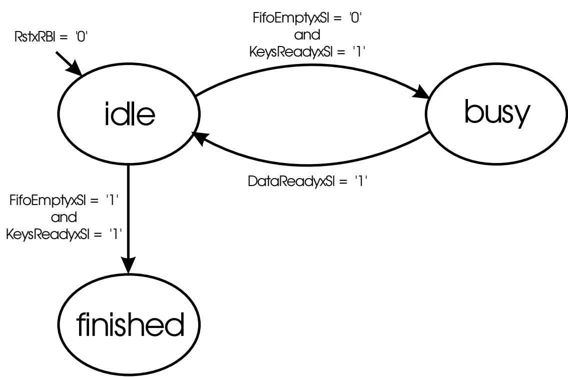


Abbildung 5.7: Vereinfachte fifoadapter FSM

- *NewDataxSO*: Dieses Signal wird einen Taktzyklus bevor die ersten 8 Bit eines 128-Bit-Blocks am Eingang des Tasks anliegen auf '1' gesetzt. Es dient als Startsignal für die Einheit `conv8to32`, die dann viermal hintereinander 32 Bit in einem Register zwischenspeichert und an die Komponente `decround` weiterleitet.

- *RExSO*: Entspricht dem "Read Enable" des FIFO-Puffers, aus dem gelesen wird. Das Signal wird so gesetzt, dass am Dateneingang von `conv8to32` die 8-Bit-Daten des Eingangs-Puffers zum richtigen Zeitpunkt anliegen. Um diese Funktionalität zu erhalten wird ein Hilfszähler eingesetzt.

5.2 Audio

Der Audio-Task erzeugt einen gleichmässigen Ton, sobald er gestartet wird. Er besteht aus einem Sägezahngenerator und einem 16-Bit-Ausgangsregister, welches das richtige Timing für den 16-Bit-mono-Codec-Treiber (ca. 12.207 kHz) herstellt. Die Daten werden, zu jeweils 8-Bit-Teilen, in den Audio-Puffer geschrieben (*t_fifo2_sel*= "100"). Der Sägezahngenerator ist als Komponente implementiert, damit er einfach durch einen anderen Generator, z.B. mit einer anderen Tonhöhe, ersetzt werden kann. Je nach Bitbreite des Zählers im Sägezahngenerator variert die Tonhöhe. Die oberen und unteren 8 Bit des 16-Bit-Ausgangs sind identisch, damit eine Vertauschung der beiden Bytes keine Rolle spielt. Da der Task durch eine User-Interaktion gestartet und auch wieder gestoppt wird, kann er nicht selbst wissen, wann er die Ausführung beenden soll. Deshalb wird das Finished-Signal nie auf '1' gesetzt.

5.3 FIFO to LED

Der “FIFO to LED”-Task liest byteweise den ihm zugewiesenen Input-Puffer aus und zeigt den Inhalt auf den LEDs 9 bis 2 an. Ein Zähler dient zur Verzögerung zwischen zwei Lesevorgängen. Ist der Puffer leer wird das Finished-Signal auf ‘1’ gesetzt und der Task verharrrt in einem Ruhezustand. Dieser Task wird vom Scheduler in der OS-Software-Komponente nicht verwendet (vgl. Abschnitt 6.1). Er kann aber für Versuchszwecke problemlos eingefügt werden.

5.4 IP / UDP

Die Aufgabe des “IP / UDP”-Tasks ist es, die vom Ethernet-Receiver empfangenen IP-Pakete zuerst auf ihre Korrektheit zu überprüfen. Danach werden die UDP-Daten in den FIFO-Puffer mit der entsprechenden Portnummer geschrieben. Da vom Ethernet-Empfänger auch Padding- und CRC-Bytes in den IP-Puffer geschrieben werden, müssen diese Daten ebenfalls ausgelesen werden.

5.4.1 Akzeptiertes Paketformat

Aufgrund der begrenzten Ressourcen können nicht alle dem IP-Standard [12] entsprechenden Pakete akzeptiert werden. Die Einschränkungen sind dabei wie folgt :

1. **Maximale Grösse :** Nach Standard beträgt diese 64 KB. Da die FIFO-Queues aber nur jeweils 2047 Bytes an Daten fassen, ist eine Beschränkung notwendig. In unserer Version beträgt die Limite 512 Bytes für die Daten (ohne IP- und UDP-Header, d.h. Total 540 Bytes IP-Paketgrösse). Die akzeptierte Maximal-Grösse kann aber mit sehr geringen Aufwand angepasst werden. Allerdings darf die gesamte IP-Paketgrösse nicht mehr als 1500 Bytes betragen, da dann durch die Längen-Beschränkung der Ethernet-Frames eine Fragmentierung notwendig wäre (s.a. nächsten Punkt).

2. **Fragmentierung :** Das Zusammensetzen von fragmentierten Paketen würde eine erhebliche Komplexität mit sich bringen. Eine Fragmentierung sollte deshalb vermieden werden. Pakete die das “More Fragments”-Flag gesetzt haben, werden verworfen.

3. **Header-Options :** Auch diese werden zur Vereinfachung ignoriert. Pakete welche eine Headerlänge von mehr als fünf Wörtern (= 20 Bytes) aufweisen, werden ebenfalls verworfen.

Abbildung 5.8 veranschaulicht welche Felder des IP-Headers überprüft und welche ignoriert werden.

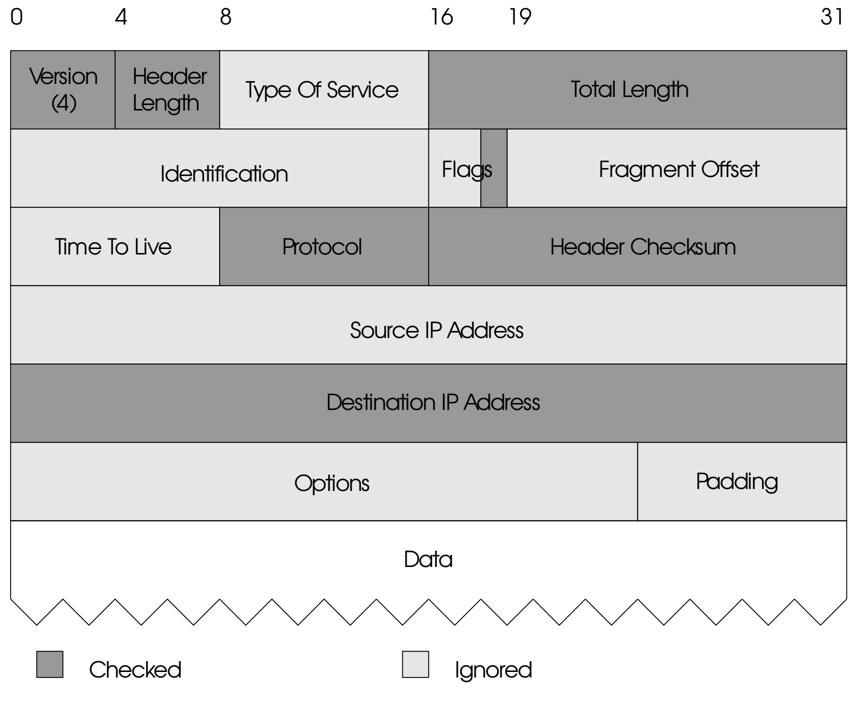


Abbildung 5.8: IP-Header

Für den UDP-Header ist die einzige Einschränkung, dass man auf einen der vier möglichen Ports (6000-6002 und 7000) schreiben sollte. Bei einer falschen Portnummer werden die Daten zwar aus dem IP-Puffer ausgelesen, aber nirgends gespeichert. Da das ICMP-Protokoll [13] nicht implementiert ist, erhält der sendende Host auch keine Rückmeldung (Port Unreachable). Die Daten werden also quasi ins "Leere" gesendet. Source-Port und Längenangabe des Headers werden ignoriert. Die optionale Checksumme wird ebenfalls nicht überprüft. Das UDP-Header-Format zeigt Abbildung 5.9.

5.4.2 Innerer Aufbau

Der Task stellt einen Moore-Zustandsautomaten dar. Durch ein Reset gelangt er in einen Idle-Zustand, in dem er solange verweilt, bis das Empty-Signal des Input-Puffers (Normalerweise der IP-Puffer) den Wert '0' annimmt. Danach wird in jedem Taktzyklus ein Byte ausgelesen und, wenn nötig, überprüft. Entsprechen die Angaben im Header den akzeptierten Werten, werden die Daten direkt aus dem Input-Puffer in den zugehörigen Output-Puffer geschrieben. Wie schon erwähnt,

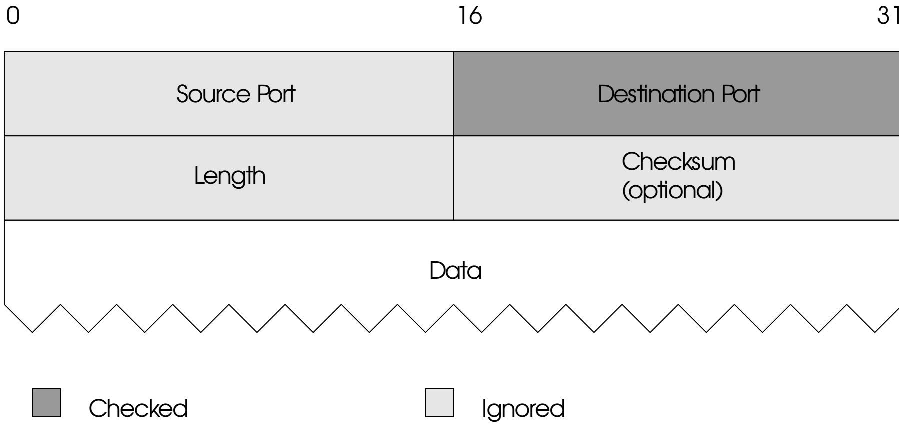


Abbildung 5.9: UDP-Header

werden danach noch die vier CRC-Bytes und bei einer Paketlänge von weniger als 46 Bytes die Padding-Daten ausgelesen. Befinden sich jetzt immer noch Daten im Input-Puffer (d.h. *t_fifo1_empty* = '0') wird in den Idle-Zustand zurückgekehrt und das nächste Paket analysiert. Andernfalls erfolgt der Übergang in den Finished-Zustand. Hier wird das Finished-Signal auf '1' gesetzt und das Empty-Signal wird nicht mehr beachtet, d.h. auch wenn jetzt nochmals Daten in den Input-Puffer geschrieben werden, werden diese nicht mehr ausgelesen.

Wird bei der Analyse des Headers ein Fehler festgestellt, erfolgt der Übergang in einen Error-Zustand. Hier wird der Input-Puffer komplett ausgelesen. Falls also im Input-Puffer nach dem fehlerhaften Paket ein eigentlich korrektes Paket folgen würde, wird dieses auch entfernt.[3] Danach wird in den Finished-Zustand übergegangen.

Das *t_fifo2_sel*-Signal welches den Output-Puffer auswählt, wird gesetzt, sobald die Portnummer im Header vollständig gelesen wurde. Grundsätzlich könnte man natürlich auch gleich diese Nummer als Steuersignal zur Puffer-Auswahl benützen. Da Ports aber 16-bit-breite Zahlen sind, geschähe dies zum Nachteil des Routings. Auch liegt die Anzahl von 65'536 möglichen Ports ausserhalb der Realisierbarkeit mit den BlockRAMs des FPGAs. Die 16-Bit-Portnummer wird deshalb in ein 3-Bit-Signal codiert. Die korrespondierenden Werte zeigt Tabelle 5.1.

[3]Ein Möglichkeit um solche Pakete nicht zu verwerfen, wäre nur mit grossem Aufwand zu realiseren und würde in einem schlechten Verhältnis zum Nutzen (besonders beim Audio-Streaming) stehen.

Port	t_fifo2_sel
6000	000
6001	001
6002	010
7000	100
sonstige	111

Tabelle 5.1: Zuordnung der Ports zum *t_fifo2_sel*-Signal

5.5 Knightrider

Der Knightrider-Task erzeugt auf dem LED-Bar ein zwischen LED0 und LED9 pendelndes Lauflicht. Wie der Audio-Task befindet er sich immer im Zustand *Running*, d.h. das Finished-Signal hat konstant den Wert '0'. Die Geschwindigkeit kann durch Änderung der Bitbreite des implementierten Zählers angepasst werden.

5.6 RS232 Hex

Dieser Task schreibt die Daten des Input-Puffers in Hexadezimaldarstellung auf die RS232-Schnittstelle. Wenn also z.B. *Hello World !* im Puffer steht, erscheint als Ausgabe *48 65 6C 6C 6F 20 57 6F 72 6C 64 20 21*. Dadurch, dass so auch die "non-printable characters" des ASCII-Zeichensatzes angezeigt werden, kann dieser Task sehr gut zu Debugging-Zwecken eingesetzt werden.

5.6.1 Aufbau

Der Task besteht aus zwei Hierarchiestufen. Die Entity `RS232` beinhaltet die grundlegende Funktion ein 8-bit-parallel übergebenes Byte RS232-konform zu senden. Über das Signal *ReadyxSO* wird mitgeteilt, ob die Einheit gerade mit einem Sendevorgang beschäftigt ist, oder ob sie bereit ist, das nächste Byte zu übernehmen. Will man also ein Byte senden, muss zuerst überprüft werden, ob *ReadyxSO* auf '1' gesetzt ist. Wenn dies der Fall ist, muss gleichzeitig (im gleichen Taktzyklus) das Signal *WritexSI* für einen Takt auf '1' gesetzt und das zu sendende Byte an den Dateneingang *InpxDI* gelegt werden. Für das nächste Byte muss wieder gewartet werden, bis *ReadyxSO* erneut den Wert '1' annimmt.

Die zweite, übergeordnete Entity in den Dateien `RS232HexTask_left.vhd` bzw. `RS232HexTask_right.vhd` hat die Aufgabe die Bytes aus dem Input-Puffer auszule-

sen und die zwei entsprechenden Bytes der Hexdezimaldarstellung, sowie zusätzlich ein Leerzeichen (0x20), der RS232-Komponente zeitlich korrekt zuzuführen.

5.6.2 Anpassen der Baudrate

Die Senderate wird durch Teilung des Systemclocks erzeugt. Will man die Baudrate ändern, muss der Quotient angepasst werden. Insbesondere muss der Teiler aber bei einer Änderung des Systemclocks mitangepasst werden, da sonst keine korrekte Ausgabe mehr möglich ist ! Bei der hier beschriebenen Version hat der Quotient den Wert 217 (= 12.5 MHz Systemclock / 57'600 Bps). Sonst sind jedoch keine weiteren Anpassungen nötig.

5.7 RS232 ASCII

Mit diesem Task kann in einem Puffer gespeicherter Text über die RS232-Schnittstelle ausgegeben werden.

Wie beim, im vorherigen Abschnitt beschriebenen, "RS232 Hex"-Task wird die Entity RS232 als Low-Level-Interface benützt. Die eigentliche Task-Entity (`RS232HexTask_left.vhd` bzw. `RS232HexTask_right.vhd`) ist aber noch einfacher aufgebaut, da die Daten aus der FIFO-Queue gelesen und ohne weitere Bearbeitung im korrekten Timing an die RS232-Einheit weitergereicht werden.

5.8 Dummy

Da ein Task-Slot nie "leer" sein kann, wird ein Task benötigt, der den Slot belegt, wenn sich kein "normaler" Task darin befinden soll. Dies ist vor allem bei der Initialiserung der Fall, wenn ein vollständiger Bitstream geschrieben werden muss. Zu einem späteren Zeitpunkt kann man einen Task durch Überschreiben mit dem Dummy-Task auch explizit "entfernen". Dies kann nötig sein, wenn man einen Task, der sich immer im Running-Zustand befindet "beenden" möchte und sonst aber keinen sinnvollen Nachfolge-Task hat (z.B. kann man so die Geräusch-Emission des Audio-Tasks unterbinden). Weitere Überlegungen, die *gegen* ein solches Überschreiben sprechen, findet man in Abschnitt 6.1.1.

Die "Funktion" des Task ist es, die Ports der Inter-Frame-Communication-Channels miteinander zu verbinden und wo nötig, den Ausgängen des Standard-Task-Interfaces sinnvolle Werte zuzuweisen. Damit ist der Task fast identisch mit dem, in Abschnitt 3.5 beschriebenen, Task-Template. Die einzigen Unterschiede sind die folgenden :

- Dem STI-Signal *t_finished* wird der Wert '1' zugewiesen. Weil der Task keine Arbeit verrichtet, sollte er so dem Scheduler signalisieren, dass er jederzeit problemlos ausgetauscht werden kann.
- Damit der VHDL-Code synthetisierbar ist, muss eine minimale Logik vorhanden sein. Deshalb wird ein beliebig gewählter Eingang in einem "Dummy"-`process` "benutzt".

5.9 Grössenvergleich

Zum Abschluss dieses Kapitels soll hier noch ein Grössenvergleich zwischen den einzelnen Tasks gegeben werden (Tabelle 5.2). Man beachte dabei, dass nur der AES^{-1}-Task für beide Task-Slots dieselbe Grösse aufweist. Dies dürfte darauf zurückzuführen sein, dass dieser Task auch als einziger einen Task-Slot vollständig belegt.

Task	Slot 0	Slot 1
AES^{-1}	187'224	187'224
Audio	89'604	96'796
FIFO to LED	85'924	94'836
IP / UDP	112'204	179'100
Knightrider	85'628	100'068
RS232 Hex	88'952	129'196
RS232 ASCII	108'848	93'992
Dummy	82'316	89'968

Tabelle 5.2: Grössenvergleich der Task-Bitstream in Bytes

Kapitel 6

PC-Software

6.1 OS-Software-Komponente : HWOS V0.3

Diese Applikation basiert auf dem in der Diplomarbeit [17] entwickelten Prototyp. Die wichtigsten Änderungen betreffen dabei die Kommandos, mit denen das OS-Frame gesteuert wird, sowie das Scheduling.

Wie in Abbildung 6.1 ersichtlich ist, werden zudem die Zustände der FIFO-Puffer, sowie diejenigen der Switches SW1 und SW3 angezeigt. Die, mit den Tasknamen beschrifteten, Buttons erlauben die einzelnen Tasks manuell in einen Slot zu laden. Hierfür sollte die Option *Enable automatic Scheduling* deaktiviert sein. Die Option *OS Reset* dient zum Reset der FIFO-Puffer und des Ethernet-Receiver/Transmitters. Mit *Global Reset* werden gleichzeitig die Task-Resets und der OS-Reset aktiviert.

Bei Programmstart wird ein Log-File (`HWOSDemo.log`) angelegt in dem chronologisch die geladenen Tasks und die dafür benötigte Download-Zeit festgehalten werden. Abbildung 6.2 zeigt ein Beispiel mit den für unsere Entwicklungs-Umgebung typischen Zeiten.

6.1.1 Scheduling

Während das Scheduling der in [17] verwendeten Stand-Alone-Tasks relativ einfach zu bewältigen ist, stellen miteinander, z.T. in Echtzeit, kommunizierende Tasks wesentlich grössere Anforderungen. Das Ziel wäre es, dass dem Hardware-Betriebssystem lediglich eine Art Datenabhängigkeitsgraph der einzelnen Tasks übergeben wird und dieses dann daraus automatisch das Scheduling bestimmt. Um dies zu realisieren, dürfte noch ein erheblicher Aufwand nötig sein. Die dafür benötigte Zeit stand uns

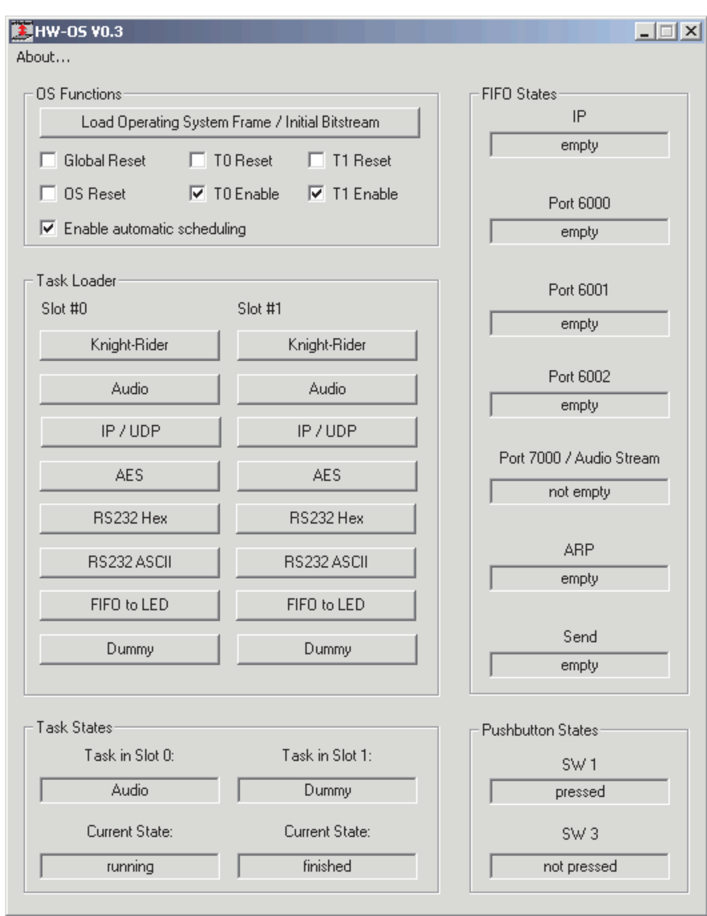


Abbildung 6.1: HW-OS V0.3

```
# HW-OS log (Resolution : 0.000279 ms)
Time to load Task  Knightrider in Slot #0 : 292.752850 ms
Time to load Task    Audio Low in Slot #0 : 292.748659 ms
Time to load Task     IP / UDP in Slot #0 : 450.155816 ms
Time to load Task    RS232 Hex in Slot #0 : 395.970996 ms
Time to load Task  RS232 ASCII in Slot #0 : 370.029710 ms
Time to load Task  FIFO to LED in Slot #0 : 294.742209 ms
Time to load Task          AES in Slot #0 : 509.085931 ms
Time to load Task        Dummy in Slot #0 : 284.916658 ms
Time to load Task  Knightrider in Slot #1 : 311.472268 ms
Time to load Task    Audio Low in Slot #1 : 307.755874 ms
Time to load Task     IP / UDP in Slot #1 : 464.966916 ms
Time to load Task    RS232 Hex in Slot #1 : 383.405153 ms
Time to load Task  RS232 ASCII in Slot #1 : 378.830829 ms
Time to load Task  FIFO to LED in Slot #1 : 312.806516 ms
Time to load Task          AES in Slot #1 : 508.206490 ms
Time to load Task        Dummy in Slot #1 : 303.894210 ms
```

Abbildung 6.2: `HWOS.log`

aber leider nicht mehr zu Verfügung. Auch ist es unserer Auffassung nach sinnvoll, zuerst dem OS die alleinige Kontrolle über die FIFO-Puffer zu geben, so dass diese den Tasks flexibel zugewiesen werden können. Aus diesen Gründen ist der Scheduling-Algorithmus in unserer Anwendung fest einprogrammiert.

Die Grundlage für den Schedulingentscheid bildet eine Anfrage vom PC an das OS-Frame. Dieses gibt den Zustand der Empty-Signale der einzelnen Queues und der Finished-Signale der beiden Tasks zurück. Anstelle solch eines Polling-Verfahrens, wäre es sicher besser, bei einer Änderung dieser Zustände einen Interrupt zu generieren. Aufgrund der verwendeten Hardware war dies aber nicht möglich. Der Lese-Event wird durch einen Multimedia-Timer ausgelöst. Die Wartezeit zwischen zwei Schedulingentscheiden sollte dadurch auf eine minimal mögliche Dauer von einer Millisekunde begrenzt werden. Der Zeitaufwand für den Entscheid selbst, einen allfälligen Bitstream-Download, sowie die Bearbeitung, der von der Windows-Umgebung generierten Events, übersteigen allerdings diese Wartezeit um ein Vielfaches. Durch diese Ausführungen wird deutlich, dass die PC-Software den eigentlichen Flaschenhals der Anwendung bildet.

Grundsätzlich gibt es zwei Ereignisse, die einen Task-Download veranlassen können. Zum einen, die Betätigung eines Switches (SW1, SW3), zum anderen, ein FIFO-Puffer (IP, Port 6000, Port 6001, Port 6002,) der vom Ethernet-Empänger oder einem anderen Task, Daten erhalten hat. Bei mehreren Möglichkeiten ent-

scheiden Prioritäten welcher Task geladen wird. Die Zuordnung der auslösenden Events und der Prioritäten, zu den einzelnen Tasks ist in Tabelle 6.1 ersichtlich.

Während die Tasks mit der Priorität 0 auch geladen werden, wenn beide Slots von laufenden Tasks besetzt sind, können die übrigen Tasks nur in einen Slot geladen werden, bei dem das Finished-Signal das Ende der Ausführung anzeigt. Wenn beide Slots frei sind, wird der linke bevorzugt. Muss ein Task von einem anderen Task der Priorität 0 überschrieben werden, erfolgt die Auswahl ebenfalls aufgrund der Prioritäten (in umgekehrter Reihenfolge).

Priorität	Event	Task
0	Switch 1	Audio
0	Switch 3	Knightrider
1	IP-Puffer	IP / UDP
2	Port 6000 Puffer	RS232 Hex
3	Port 6001 Puffer	AES
4	Port 6002 Puffer	RS232 ASCII

Tabelle 6.1: Task Prioritäten

Als zweite Bedingung bei der Auswahl des nächsten Tasks, gilt es zu beachten, ob die vom neuen Task benötigten Ressourcen noch zur Verfügung stehen, oder ob sie schon vom zweiten, im anderen Slot befindlichen Task, benutzt werden. Die kritischen Ressorcen hierbei sind die Lese- und Schreib-FIFO-Queue, sowie das RS232-Interface. Eine gleichzeitige Benutzung des LED-Bars wird nicht ausgeschlossen.

Es stellt sich nun die Frage, ob ein Task, der seine Arbeit beendet hat mit dem Dummy-Task überschrieben oder im OS-Frame belassen werden soll, wenn nicht unmittelbar ein neuer Task geladen werden muss. Für die erste Möglichkeit spricht, dass so die Ressourcen sicher freigegeben werden. Sie hat allerdings einen gravierenden Nachteil, der am folgenden Beispiel sichtbar wird: Wenn Audio-Daten in Echtzeit übertragen werden, erfolgt dies mit 192 kBps. Angenommen der "IP / UDP"-Task hat gerade eben die Daten im IP-Puffer ausgelesen und in denjenigen des Ports 7000 geschrieben. Der IP-Puffer ist leer und deshalb wird nun das Finished-Signal auf '1' gesetzt. Nun liest die OS-Applikation die Zustände aus und beschliesst den Dummy-Task zu laden. Beim nächsten Auslesen ist der IP-Puffer wieder mit Daten gefüllt und der "IP / UDP"-Task muss erneut geladen werden. Die Zeit für das Laden des Dummy-Tasks beträgt etwa 300 ms, diejenige für den "IP / UDP"-Task etwa 450 ms. Daraus folgt, dass der Audio-Puffer mindestens 17.5 KB ($192kBps \cdot (300ms + 450ms)$) an Daten enthalten muss, um diese Zeit unterbruchsfrei überstehen zu können. Die Gesamtkapazität aller BlockRAMs auf dem FPGA beträgt aber nur 14 KB, woraus sofort klar wird, dass dies so nicht funktionieren

kann. Auch bei einer starken Verringerung der Audioqualität wird dieses Problem nur wenig vereinfacht. Neben dem Task, der an der Audio-Übertragung beteiligt ist, muss ja manchmal auch noch ein anderer geladen werden. Dadurch können sich die Zeiten noch deutlich vergrössern.

Die bessere Lösung ist also einen Task, der seine Arbeit beendet hat, im Slot zu belassen und ihn, wenn er später wieder benötigt wird, durch einen Task-Reset zu reaktivieren.

6.1.2 Task-Download

Wenn ein Task geladen werden soll, müssen vorher und nachher zwei Kommandos gesendet werden. Zuerst wird der zu ersetzende Task mit dem entsprechenden Task-Kommando deaktiviert, dann werden die kritischen Signale mittels eines OS-Kommandos "eingefroren". Nun folgt der eigentliche Bitstream-Download. Anschliessend werden die eingefrorenen Signale durch ein erneutes OS-Kommando wieder aktiviert. Daraufhin wird der neue Task durch ein Task-Kommando gestartet. Diesen Ablauf, mit den typischen auf dem PC gemessenen Zeiten, veranschaulicht Abbildung 6.3.

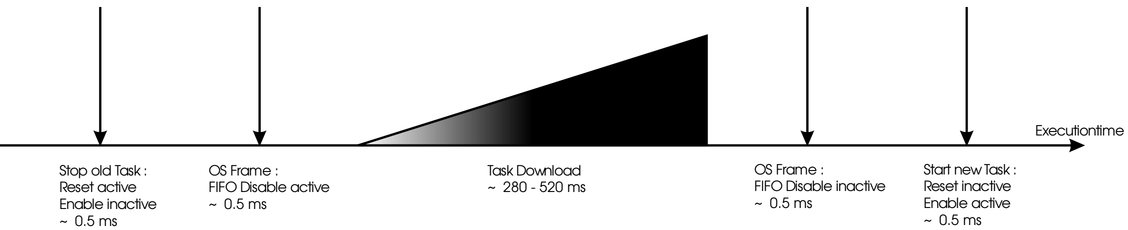


Abbildung 6.3: Task-Download

6.2 Send Data

Das Programm *Send Data* dient zum einfachen Versenden von Audio- und Text-Daten an die Demo-Applikation auf dem FPGA. Die Abbildung 6.4 zeigt die Benutzeroberfläche während eines Sendevorgangs.

Mittels des *File ...*-Buttons kann eine beliebige Datei mit den gewünschten Daten gewählt werden. Es wird dabei nicht überprüft, ob es sich um ein korrektes Audio- bzw. Text-Format handelt. Bei den *Destination Ports* stehen die Nummern 6000 bis 6002 für die Text-Nachrichten zur Verfügung, während der Port 7000 nur für Audiostreaming verwendet werden sollte. Die Auswahl der AES-Verschlüsselungoption

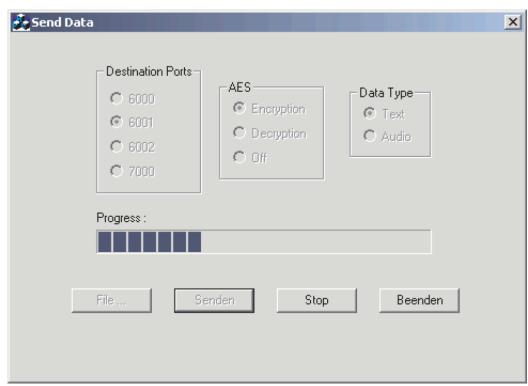


Abbildung 6.4: Send Data

hat nur Einfluss, wenn die Daten als Text verschickt werden. Audio-Daten werden immer unverschlüsselt übertragen. Weiter ist zu beachten, dass Nachrichten auch entschlüsselt (Decryption) versendet werden können, allerdings ist der für die Umkehrfunktion auf dem FPGA zuständige Verschlüsselungs-Task nicht implementiert.

Für den Sendevorgang wird jeweils ein eigener Thread gestartet, der durch Betätigung des Stop-Buttons abgebrochen wird.

6.2.1 Datentypen

Die Auswahl des Datentyps hat Auswirkung auf die Paketgrösse und die zeitlichen Intervalle zwischen den Paketen. Wird der Typ *Text* gewählt, werden jeweils 16 Byte in einem Paket versendet. Wenn die Dateigrösse kein Vielfaches von 16 beträgt, wird das letzte Paket mit Leerzeichen (0x20) auf 16 Byte aufgefüllt. Der Grund für diese Paketgrösse liegt bei der optionalen Verschlüsselung. Der AES verlangt immer 128 Bit (= 16 Byte) grosse Blöcke. Die einzelnen Pakete werden in diesem Modus so schnell wie möglich, d.h. ohne zusätzliche Verzögerung gesendet.

Wird als Datentyp *Audio* angegeben, muss den speziellen Anforderungen des Streamings Rechnung getragen werden. Der im OS-Frame eingebaute Codec-Treiber verarbeitet mit 12'207 Hz abgetastete 16 Bit grosse Mono-Samples. Daraus folgt, dass im Mittel eine Übertragungsrate von 195'312 Bps erzielt werden muss. Da keine Flusskontrolle implementiert ist, muss versucht werden, möglichst genau diese Senderate zu erreichen. Die Paketgrösse sollte einen Kompromiss zwischen einer guten Auslastung des "IP / UDP"-Tasks (Wie in Abschnitt 6.1 erwähnt, reagiert der Scheduler ziemlich träge) und einer kleinen zeitlichen Lücke beim Verlust des Pakets bilden. Die zweite Einschränkung bildet die Granularität, des zur zeitlichen Staffelung verwendeten Multimedia-Timers. Bei Windows XP beträgt die Auflösung

1 ms. Unter Berücksichtung dieser Bedingungen wurde eine Paketgrösse von 488 Bytes und ein Abstand von 20 ms gewählt. Damit wird eine Senderate von 195'200 Bps erreicht. Zu Beginn der Übertragung wird jeweils ein "Burst" von vier Paketen gesendet. Dies ist die maximale Anzahl von vollständigen Paketen, die ein Puffer-Speicher (2047 Bytes) fassen kann. Die Qualität der so erzielten Audio-Ausgabe ist trotz einiger "Knackgeräusche" relativ gut. Natürlich könnte sie durch einen etwas ausgefeilteren Sende-Algorithmus noch gesteigert werden. Dies war allerdings kein Hauptziel dieser Diplomarbeit.

6.2.2 Zieladresse und AES-Key

Im Programm-Code kann bei Bedarf die IP-Zieladresse (10.0.0.4) und der vom AES-Algorithmus verwendete Schlüssel (alle 128 Bit haben den Wert '0') angepasst werden.

Kapitel 7

Probleme / Bugs

7.1 BlockRAM-Bug

Bei den ersten Versuchen mit der Inter-Task-Kommunikation zeigte sich ein merkwürdiges Verhalten. Wenn vom PC ein IP-Paket an das nur mit Dummy-Tasks besetzte OS-Frame gesendet und dann der "IP / UDP"-Task geladen wurde, wurde dieses erste Paket vom Task jeweils verworfen. Die nachfolgenden Pakete wurden jedoch akzeptiert und korrekt weitergeleitet. Wenn sich hingegen der "IP / UDP"-Task von Anfang an in einem Task-Slot befand, wurde auch das erste Paket problemlos angenommen. Eine Analyse des IP-Puffer-Inhalts zeigte, dass im ersten Fall die ersten acht Bytes des IP-Headers auf Null gesetzt waren. Da im zweiten beschriebenen Fall das Paket akzeptiert wurde, mussten diese Bytes irgendwie beim Download überschrieben worden sein.

Eine genauere Analyse ergab schliesslich, dass die ersten acht Bytes jedes Block-RAMs auf der linken FPGA-Seite bei jedem Task-Download mit Null überschrieben werden. Eine Untersuchung, des in der *Active Module Phase* des Modular Design Flows erzeugten, partiellen Bitstreams zeigte, dass immer am Ende die Schreibadresse auf ein Frame der linken BlockRAMs gesetzt wird. Sowohl nachher als auch vorher folgt ein kompletter Frame (34 Bytes) bei dem alle Bytes den Wert Null aufweisen (Abbildung 7.1 zeigt solch einen Bitstream, wie er im Hexadezimal-Editor erscheint). Gemäss der Beschreibung des partiellen Bitstream-Downloads in [23] sollte die Schreibadresse allerdings nicht mehr geändert werden. Es stellte sich zudem heraus, dass nicht die Daten *nach* dem Ändern der Adresse in die Block-RAMs geschrieben werden, sondern die Bytes *vorher*. Dies obwohl im vorangehenden Schreibkommando die Länge einschliesslich diesem Null-Frame angegeben ist und daher auch dorthin geschrieben werden sollten.

Zur Behebung dieses offensichtlichen Bugs kann nun leider nicht einfach das

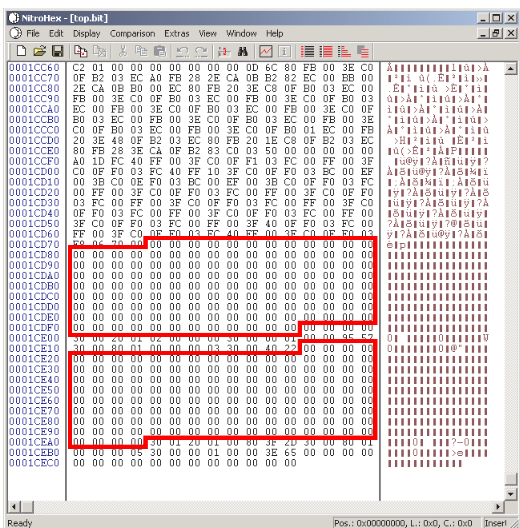


Abbildung 7.1: Das Ende eines partiellen Bitstreams, wie er während des Modular Design Flow erzeugt wird. Die beiden Frames, die Null-Werte enthalten sind markiert. Dazwischen liegt das Kommando, das die Schreibadresse auf ein Frame der linken BlockRAMs setzt.

Ende des Bitstreams abgeschnitten werden, da dort ein CRC-Check durchgeführt wird. Bei einem Fehler geht die Download-Logik des FPGA in einen Error-Zustand über und es können dann keine partiellen Konfigurationen mehr geschrieben werden.

Unser "Workaround" zur Lösung des Problems besteht aus einem Programm (`BRAMpatch.exe` [1]), welches im Bitstream nach der letzten normalen Frameadresse sucht und diese zusammen mit den Daten des ersten folgenden Frames anstelle der BlockRAM-Frameadresse und des Null-Frames einsetzt. Zusätzlich muss auch der CRC-Wert neu berechnet werden. Das Ergebnis dieser Manipulationen zeigt Abbildung 7.2. Die Absicht dabei ist, dass die gleichen Daten einfach zweimal geschrieben werden und darum kein Schaden entstehen sollte. Die so behandelten Dateien funktionierten nun problemlos. Da aber, wie schon erwähnt, normalerweise ja die Daten *vor* dem Setzen der Schreib-Adresse an den von der letzten Adresse bestimmten Ort geschrieben werden, kann trotzdem nicht ausgeschlossen werden, dass dadurch ein Frame falsch konfiguriert wird.

Anzufügen ist, dass wir die Version 4.1 des Modular Design Flows verwendet haben. Möglicherweise ist dieses Problem in der neuen Version 5.1 bereits behoben.

[1] Das Programm nimmt eine Bitstream-Datei als Argument und erzeugt daraus die korrigierte Datei *patch.bit*.

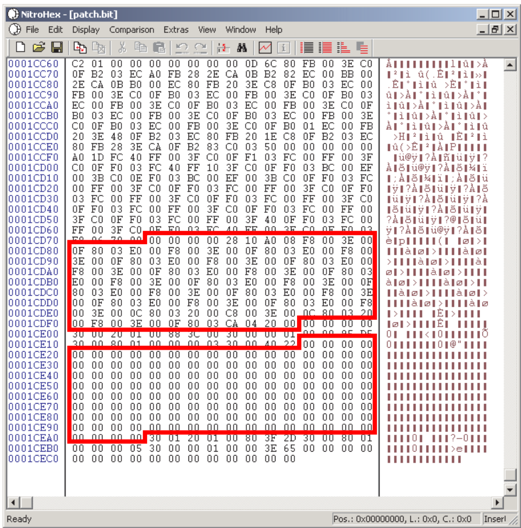


Abbildung 7.2: Der abgeänderte Bitstream: Im ersten markierten Frame stehen nun die Daten des letzten "normalen" Frames. Die Schreibadresse wurde ebenfalls geändert. Nach dem zweiten markierten Frame ist auch noch der neu berechnete CRC-Wert zu finden.

7.2 Bitbreiten in Xilinx Foundation

In der verwendeten Version 5.1 der Xilinx Foundation Tools werden die Bitbreiten bei Conditional Signal Assignements nicht überprüft. Folgende Anweisung führt weder zu einer Warning noch zu einem Error während der Synthese :

```
signal sig_a : std_logic;
signal vector_b : std_logic_vector(2 downto 0); - 3 Bit Vector !

sig_a <= '1' when (vector_b = "01") else '0';
- anstatt sig_a <= '1' when (vector_b = "001") else '0';
```

Das Problem dabei ist insbesondere, dass die Simulation mit Modelsim zwar die korrekte Funktion zeigt. Wird nun aber die Schaltung auf das FPGA geladen, ergibt sich meistens ein unerklärliches Verhalten. Da besonders bei Vektoren mit vielen Stellen solche Tippfehler häufig auftreten können, sollte die korrekte Anzahl immer gründlich überprüft werden.

7.3 Ethernet

Erstaunlicherweise zeigten sich einige Probleme bei den aus [16] und [8] übernommenen Ethernet-Layer-Implementationen. Obwohl diese bei ihren Entwicklern scheinbar reibungslos funktionierten. Die Gründe dafür sind am wahrscheinlichsten bei der von uns verwendeten neueren Version der Xilinx Foundation Tools zu finden. Einige Beiträge in der FPGA-Newsgroup `comp.arch.fpga` berichteten nämlich von Designs die in der Version 4.1 entwickelt wurden und nun unter der neuen Version 5.1 Probleme zeigten.

7.3.1 Konfiguration 10 MBps Vollduplex

Die in [16] und [8] verwendeten Konfigurationen des Ethernet-Chips sollten gemäss Datenblatt [5] die Ethernetschnittstelle fest auf 10 MBps und Vollduplex-Betrieb einstellen. Die LED-Anzeigen des von uns benutzten Switchs zeigten jedoch, dass so nur der Halbduplex-Modus vom XESS-Board signalisiert wird. Da in beiden Designs keine Kollisionserkennung implementiert ist, können damit nicht gleichzeitig vom PC und vom XESS-Board aus Pakete gesendet werden. Eine Änderung der Konfiguration auf Auto-Negotiation mit der Beschränkung auf 10 MBps brachte schliesslich die gewünschte Einstellung. Die Tabelle 7.1 zeigt die Pin-Werte der Original-Konfiguration (links) und diejenigen der von uns benutzten Konfiguration (rechts).

Konfiguations-Pin	ohne Auto-Negotiation	mit Auto-Negotiation
mddis	HI	HI
fde	HI	HI
cfg(0)	LO	LO
cfg(1)	LO	HI
mf(0)	LO	HI
mf(1)	LO	LO
mf(2)	LO	LO
mf(3)	HI	HI
mf(4)	LO	LO

Tabelle 7.1: Zwei eigentlich identische Konfigurationen des Ethernet-Chips. Ohne Auto-Negotiation wird der Halbduplex-Modus gesetzt !

7.3.2 Transmitter

Nachdem das im vorherigen Abschnitt berschriebene Konfigurations-Problem gelöst war, war es nun möglich auch vom XESS-Board aus Pakete zu senden. Jedoch zeigten sich trotzdem noch Probleme. Zwischen einem Zehntel und der Hälfte der gesendeten Pakete wurden vom Switch verworfen. Da die Software auf einem PC keinen Zugang zu fehlerhaften Ethernet-Daten hat[2], konnte zur Analyse kein Sniffer-Programm (z.B. Shomiti Surveyor oder Ethereal) verwendet werden. Um Zugang zu den "rohen" Ethernet-daten zu bekommen, implementierten wir eine Schaltung, die alle vom Ethernet-Receiver empfangenen Nibbles in Hexadezimaldarstellung auf die RS232-Schnittstelle ausgibt. Die so gewonnene Debug-Ausgabe zeigte nun, dass der Sender bei den fehlerhaften Frames "Aussetzer" hat. Einzelne Nibbles werden vom XESS-Board gar nicht gesendet. Tabelle 7.2 zeigt den direkten Vergleich des korrekten Frames mit einem fehlerhaften. Zu beachten ist dabei, dass nach dem Fehlen

	SDF	Ethernet-Header
Korrekt	D5	00 50 DA 75 7B B6 01 02 03 04 05 06 08 00

	IP-Header, UDP-Header, Data, CRC
Korrekt	45 00 00 20 23 62 00 00 80 11 52 68 81 84 39 7E 0A 00 00 01 04 7E 13 89 00 0C 60 09 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 51 B9 87 54
Falsch	50 04 00 00 32 22 06 00 00 18 21 85 16 48 98 E3 A7 00 00 10 40 E0 37 91 08 C0 00 96 10 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 95 7B 48

Tabelle 7.2: Vergleich eines korrekt und eines fehlerhaft gesendeten Ethernet-Frames. Zu Beginn sind beide identisch (SDF und Ethernet-Header).

eines Nibbles die folgenden Nibbles paarweise vertauscht sind. Der Grund dafür ist, dass dem Ethernet-Chip jeweils das Low-Nibble vor dem High-Nibble übergeben werden muss. Dies und die Tatsache, dass nach dem zu kurzen Frame keine anderen Daten folgen, die auf die korrekte Länge auffüllen würden, zeigt, dass das Problem seine Ursache innerhalb des FPGAs zu haben scheint. Zudem zeigte sich auch eine Abhängigkeit der Fehlerhäufigkeit von den Syntheseoptionen, so erhöhte z.B. die Option *keep hierarchy* die Zahl der zu kurzen Frames deutlich. Dies würde eigentlich auf Timing-Probleme hindeuten, jedoch zeigte der Timingreport keine Verletzung der Timing-Constraints an. Die genauen Gründe konnten so nicht ermittelt werden. Sie sind aber sehr wahrscheinlich, wie schon eingangs erwähnt, auf ein geändertes Synthese-Verhalten der Xilinx-Tools zurückzuführen.

[2] Defekte Frames werden schon von der Hardware des Ethernetadapters verworfen. Die darüberliegenden Netzwerkschichten erhalten lediglich die Information, dass etwas verworfen wurde, aber nicht was genau.

Der einfache, von uns selber programmierte, Ethernet-Transmitter funktionierte jedoch ohne Probleme. Möglicherweise verursachen einige, für ein Hardware-Design nicht eindeutige, aber doch dem Standard entsprechende, VHDL-Anweisungen die Probleme in den anderen Designs[3] (vgl. auch Abschnitt 7.3.3).

7.3.3 CRC

Neben den fehlenden Nibbles war bei einigen Frames auch ein falscher CRC-Wert zu beobachten. Der aus [8] übernommene CRC-Generator verwendete für die CRC-Berechnung ein `procedure`. Da dies für eine Hardware-Synthese nicht eindeutig ist, haben wir die Funktion in strukturellen VHDL-Code umgeschrieben. Mit dem so abgeänderten CRC-Generator traten keine falschen CRC-Werte mehr auf. Es ist noch zu erwähnen, dass die Fehler beim Original-Generator wiederum von den Synthese-Optionen abhängig waren und die Simulation für beide Varianten immer die gleichen Werte lieferte.

7.4 Pullups / Pulldowns

Für die Instanzierung der Bus-Macros werden viele Signale mit den konstanten Werten '0' und '1' benötigt, um den Zustand der Tristate-Buffer und damit die "Richtung" der Verbindungen festzulegen. Gemäss Appendix B von [24] gibt es hierfür grundsätzlich zwei Methoden. Die Variante mit den "dummy"-LUTs ist allerdings nur bedingt geeignet, da wirklich für *jede* Konstante ein solcher LUT instanziert werden muss. [4] Bei unserem Design mit insgesamt 98 Bus-Macros ergäbe dies 784 LUTs, die instanziert werden müssten.

Bei der zweiten, in dieser Arbeit verwendeten Methode, muss in jeder Modul-Fläche ein Pin mit einem Pullup auf '1' und ein anderer mit einem Pulldown auf '0' gezogen werden. Die Signale von diesen Pins können dann problemlos mehreren Ports zugeordnet werden. Das Problem hierbei ist, dass auf dem XESS-Board die meisten Pins mehrfach belegt sind. Ohne Schwierigkeiten können nur Pins verwendet werden, die zu den "Expansion Headers" führen. Wenn aber keine dieser Pins

[3] Wie dem Kapitel *Modelling Hardware with VHDL* in [6] zu entnehmen ist, bedeutet ein dem VHDL-Standard IEEE 1076 und dem Logiksystem-Standard IEEE 1164 entsprechendes Design nur, dass eine Simulation möglich ist. Es gibt jedoch kein allgemein verbindliches Übereinkommen welche Teilmenge der Standards von den Hardware-Synthese-Tools umgesetzt werden müssen.

[4] Das Problem ist der in Kapitel 3 von [26] im Abschnitt *Modular Design Troubleshooting* beschriebene *Multiple Output Ports MAP Error*. Da im gleichen Abschnitt des Reference Guide zur neueren Version 5.1i [27], dieser Fehler nicht mehr vorkommt, sollte es in der entsprechenden Version des Modular Design Flows auch keine Probleme mehr bereiten.

zu Verfügung stehen, weil sie schon anderweitig benutzt werden (z.B. OS-Middle) oder auch gar nicht an die Modul-Fläche angrenzen (z.B. OS-Right), muss gründlich überprüft werden, welcher Pin sonst verwendet werden kann. So kann z.B. der Pin, der zu Switch 2 führt (Pin 175) nicht einfach als Pulldown benutzt werden, da, wenn der Switch nicht gedrückt ist, ein externer Pullup wirkt, der gegenüber dem internen Pulldown die Oberhand behält. Pulldowns sollten nur Pins zugewiesen werden, die ohnehin immer den Wert '0' aufweisen und für Pullups entsprechend den Wert '1'. Anstelle des oben erwähnten Pins 175 z.B. Pin 116, der zum, von uns nicht benutzten, Video-Decoder gehört. Sollte der Decoder aber einmal verwendet werden, muss *unbedingt* der Pulldown einem anderen Pin zugeordnet werden ! Bei einer falschen Beschaltung können die Inter-Frame-Communication-Channels und das Standard-Task-Interface unterbrochen sein.

7.5 Rauschen / Echtzeitanwendung unter Windows

Wenn ein Audio-Stream an das Board gesendet wird, kann zeitweilig ein kurzes Rauschen auftreten. Wenn auf dem sendenden PC z.B. häufig mit der Maus das Fenster der Sendeapplikation angeklickt wird, tritt dieses Rauschen vermehrt auf. Die Ursache scheint darin zu liegen, dass Windows kein Echtzeitbetriebssystem ist und sich die Timer-Events in der Event-Queue sozusagen "hinten anstellen" müssen. Da der aktuelle Füllstand des Audio-Puffers auf dem FPGA nicht zum PC übertragen wird, d.h. da keine Flow Control implementiert ist, kann es so vorkommen, dass der Puffer-Speicher unterläuft und somit dieses Rauschen auftritt.

7.6 Störungen während der Ausführung

Manchmal kommt es vor, dass ein Task "hängen" bleibt, z.B. kann bei einem gefüllten Puffer der "RS232 ASCII"-Task geladen werden. Der Task liest dann aber nur ein Zeichen aus und erst nach einem Reset werden die restlichen Daten ausgegeben. Ein zweiter Fall bei dem ein solches Problem auftritt, ist, wenn während einer Audio-Übertragung durch steten Druck auf Switch 3 der Knightrider-Task in den einen Slot geladen wird. Im zweiten Slot befindet sich dann der "IP / UDP"-Task, der Pakete aus dem IP-Puffer ausliest und in den Stream-Puffer schreibt. Wenn ein Paket ankommt wird der Input-Puffer vom Task komplett geleert, da er schneller arbeitet als dass die Ankunftsrate der Pakete. Danach geht er in den Finished-Zustand über, bis er vom OS bei angefülltem IP-Puffer durch Reset wieder zum Auslesen veranlasst wird. Nach einer nicht deterministischen Zeitspanne kann

nun aber folgendes vorkommen : Im Input-Puffer befinden sich Daten und der Task signalisiert, dass er sich im Running-Zustand befindet. Trotzdem werden die Daten aber *nicht* ausgelesen und zum Audio-Puffer transportiert.

Die Gründe für diese Verhalten konnten aufgrund der mangelnden Reproduzierbarkeit nicht eruiert werden.

7.7 Clocknetzaktivierung

Es zeigte sich, dass das Clocknetz nur für die CLBs aktiviert war, die in der Startkonfiguration benutzt wurden. Wenn danach also ein anderer Task geladen wurde, hatten Teile der Schaltung keinen Taktgeber. Zur Lösung diese Problems wird mit einem JBits-Programm am Ende der *Final Assembly*-Phase das gesamte Clocknetz aktiviert. Diese Problem wurde schon in [17] erkannt.

7.8 Unterbrechungen während des Bitstream-Downloads

Der gravierendste Nachteil bei der Verwendung des Modular Design Flows besteht darin, dass die IFCCs bei einem Task-Download unterbrochen werden. Der Grund dafür liegt in einem, von Task zu Task, unterschiedlichen Routing. Mit den Bus-Macros ist es zwar möglich, einen festen Verbindungspunkt zwischen zwei Modulen zu definieren. Darauf, wie das Routing innerhalb eines Moduls erfolgt, hat der Designer jedoch keinen Einfluss. Dies bedeutet, dass eine Verbindung, zwischen einem Macro an der linken Modulgrenze und einem auf der rechten Seite, bei jedem Task anders verlaufen kann. Diesen Sachverhalt veranschaulicht Abbildung 7.3. Während bei Tasks mit geringem Flächenbedarf meistens der Fall, des dargestellten Tasks A, auftritt, ist bei Tasks mit fast vollständiger Slot-Ausnutzung (z.B. AES^{-1}) auch ein Routing wie bei den Tasks B und C möglich. Zur Umgehung diese Problems mussten die kritischen Signale (Reset, Read Enable, Write Enable), die die IFCCs benutzen, während des Downloads "eingefroren" werden. Dies bedeutet, vor der betroffenen Komponente werden die Signale in einem Flip-Flop zwischengespeichert. Das Signal, das dieses "Einfrieren" steuert, darf natürlich auch nicht unterbrochen werden. Deshalb wird es im OS-Middle auf einen Pin hinausgeführt und durch einen externen Draht zu je einem Pin von OS-Left und OS-Middle verbunden. In Abbildung A.1 ist diese "nicht unterbrechbare" Leitung blau gekennzeichnet.

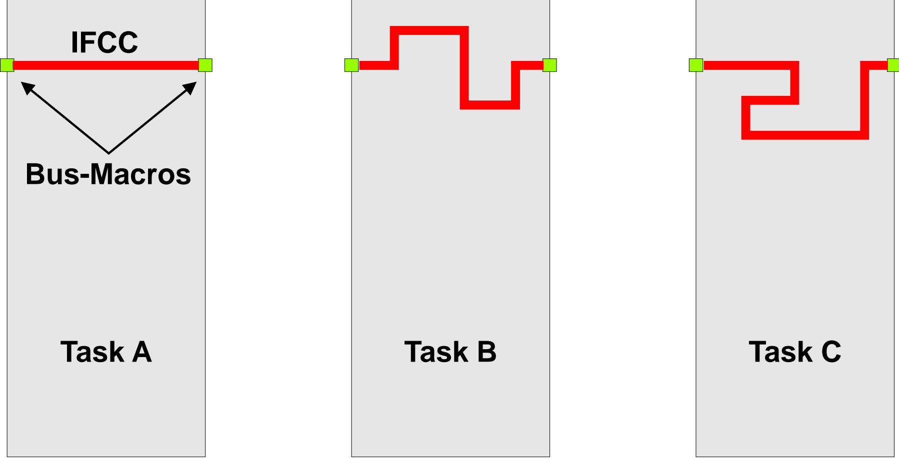


Abbildung 7.3: Unterschiedliches Routing für ein IFCC bei drei verschiedenen Tasks.

Kapitel 8

Ausblick / Erweiterungen

In diesem Kapitel sollen verschiedene Punkte für eine mögliche Weiterentwicklung unserer Arbeit kurz vorgestellt werden.

8.1 Zugriff auf Ethernet-Sender

Das entwickelte System benützt den Ethernet-Sender lediglich zur Beantwortung von ARP-Requests. Sinnvoll wäre aber, wenn auch die Tasks Pakete verschicken könnten. Das STI müsste dafür nicht geändert werden. Auch werden ausser ein oder zwei zusätzlichen Steuersignalen keine zusätzlichen IFCC-Verbindungen benötigt, da schon jetzt Daten von beiden Slots zum Audio-Puffer im OS-Left geführt werden. Mit dem Signal FIFO-Select wird dann einfach der Send-Puffer, der sich ebenfalls in OS-Left befindet, als Ausgangs-Speicher gewählt. Allerdings müsste ein Format definiert werden, in dem Daten an den Sender übergeben werden. Wie schon beim implementierten ARP-Reply, müsste zuerst die Ethernet-Adresse mitgeteilt werden. Wenn Pakete unterschiedlicher Länge gesendet werden sollen, müsste zusätzlich auch noch diese Länge angegeben werden. Schliesslich muss der Ethernet-Sender auch noch über den Pakettyp (ARP oder IP) informiert werden, da dieser auch im Frame-Header gesetzt werden muss.

Weitergehend könnte man anstatt der MAC-Adresse auch die IP-Adresse verwenden und das OS stellte dann, als zusätzlichen Dienst, die Adressübersetzung zur Verfügung. Zusätzlich könnten auch noch andere Protokolle implementiert werden, z.B. ICMP für Ping-Anfragen.

8.2 IFCC-Hard-Macro

Wie schon in Abschnitt 7.8 erklärt, sind die, während des Task-Downloads unterbrochenen, IFCCs eine grosse Einschränkung. Zur Lösung sollte versucht werden eine durchgehende Verbindung zwischen den Bus-Macros als Hard-Macro zu implementieren. Da so das Routing der IFCCs bei jedem Task genau gleich ist, sollten diese Verbindungen auch nicht mehr unterbrochen werden.

8.3 Memory-Management durch OS / Flexibler Scheduler

Um eine beliebige Applikation ausführen zu können, muss das Betriebssystem die absolute Kontrolle über die Puffer-Speicher haben. Es muss nach Bedarf einem Task einen Speicher zur Verfügung stellen und z.T. auch während der Laufzeit des Tasks diese Zuweisung ändern können. Wenn solch ein Memory-Management realisert ist, kann auch ein flexibler Scheduler implementiert werden, der selbständig den Ablauf der Tasks plant (vgl. 6.1.1).

8.4 Debug-Output zum PC

Weil jeder Task ein standardisiertes Interface zum OS-Frame besitzt und nicht direkt an FPGA-Pins angeschlossen ist, kann das Betriebssystem auch als Test-Bench benutzt werden. Eine Schaltung kann so getestet werden, ohne die Hardware um das FPGA herum ändern zu müssen. Der OS-PC würde Stimuli-Daten zum OS-Frame schicken und nachher die entsprechenden Antworten zurückerhalten. Auch könnte man den Inhalt aller Speicher ansehen und gegebenenfalls verändern.

8.5 OS-Builder

Der OS-Builder ist eine Applikation, die aufgrund von verschiedenen Parametern automatisch die VHDL-Dateien für Top, OS-Left, OS-Middle, OS-Right und die beiden Task-Templates erzeugt. Die Parameter sind z.B. die zur Verfügung stehenden Ressourcen und die gewünschte Anzahl Task-Slots.

8.6 Eigenes Board CPU / FPGA

Das Design eines dedizierten Boards würde viele Probleme beseitigen. So hätte man keine doppelt belegten Pins mehr und die gewünschten Ressourcen könnten gezielt an die Pins der OS-Frame-Aussengrenze angeschlossen werden. Für eine bedeutend schnellere Konfiguration wäre ein direkter Zugriff auf das SelectMAP-Interface vorteilhaft. Ausserdem könnte man durch den Einsatz eines Echtzeit-Betriebssystems, auf der ebenfalls auf diesem Board integrierten CPU, wesentlich bessere Ergebnisse erzielen.

Kapitel 9

Zusammenfassung / Schlussbemerkungen

Ausgehend von einem OS-Prototypen und einer selbstdefinierten Demo-Applikation wurde in dieser Diplomarbeit eine funktionierende Inter-Task-Kommunikation für ein rekonfigurierbares Hardware-Betriebssystem implementiert. Die Zielarchitekturen sind ein Xilinx Virtex XCV800 und ein PC mit Windows XP. Für den Datenaustausch werden FIFO-Pufferspeicher verwendet. Nachdem zuerst ein Datenfluss mit Realtime- und Non-Realtime-Daten, als auch Benutzerinteraktionen festgelegt war, wurden die entsprechenden Elemente in Komponenten des Hardware-Betriebssystems und in Tasks aufgeteilt. Durch Einschränkungen des verwendeten Design Flows, wird die Kommunikation gestört, wenn ein neuer Task geladen wird. Davon abgesehen konnte ein gelungenes Zusammenspiel von mehreren einfachen und einem komplex aufgebauten Task gezeigt werden. Für ein in der "realen" Welt einsetzbares System ist aber noch einige Arbeit zu leisten.

Zum Schluss möchten wir noch festhalten, dass dies für uns eine sehr interessante und lehrreiche Arbeit war. Wir möchten uns an dieser Stelle auch noch herzlich bei allen bedanken, die uns dabei geholfen haben. Insbesondere bei unserem Betreuer Herbert Walder, für die tatkräftige Unterstützung und bei Michael Ruppen, für die gute Zusammenarbeit.

Anhang A

Inbetriebnahme

Den Überblick über das gesamte System zeigt Abbildung 3.2. Die benötigten Dateien befinden sich auf der, zu dieser Diplomarbeit gehörigen, CD (siehe Anhang E). Alle für eine Demonstration der Anwendung notwendigen Files befinden sich im Verzeichnis `App`.

A.1 XESS-Board vorbereiten

1. Jumper J29 und J30 über die Pins 2 und 3 setzen.
2. Alle DIP-Switches auf OFF.
3. XESS-Board mit dem Parallel-Kabel zum OS-PC verbinden.
4. Strom einschalten
5. Systemclock mit dem GXSSETCLK-Programm auf 12.5 Mhz einstellen
6. Konfiguration des CPLD : Datei `selmap.svf` mit dem GXSLOAD-Tool laden.
7. Strom ausschalten.
8. Jumper J29 und J30 über die Pins 1 und 2 setzen.
9. DIP-Switches 1 und 9 auf ON.
10. XESS-Board mit dem Patch-Board verbinden (Abbildungen A.1 und A.2).
11. Die Pins A9 (links), D7 (links) und A9 (rechts) der Expansion-Headers miteinander verbinden (Abbildung A.1).

12. Die beiden Ausgänge der PCI-7200-Karte im OS-PC mit dem Patch-Board verbinden.

13. Lautsprecher an Codec-Ausgang des Boards anschliessen.

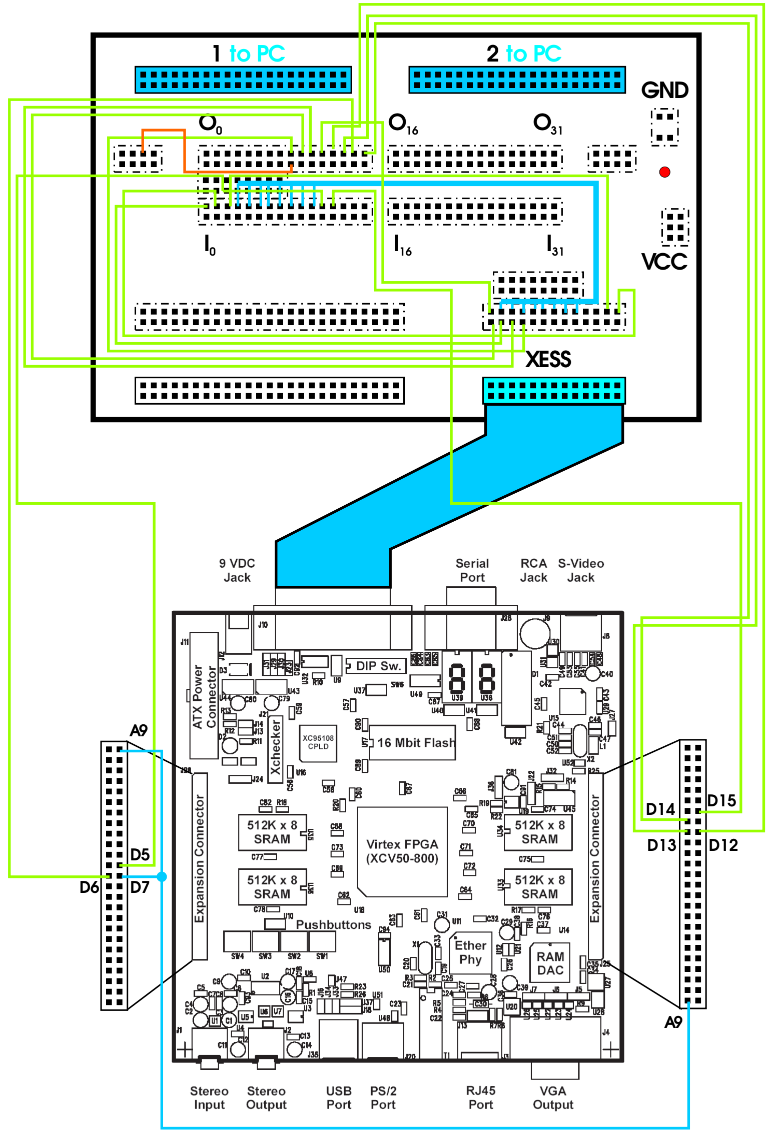


Abbildung A.1: XESS-Board- und Patch-Board-Verbindungen

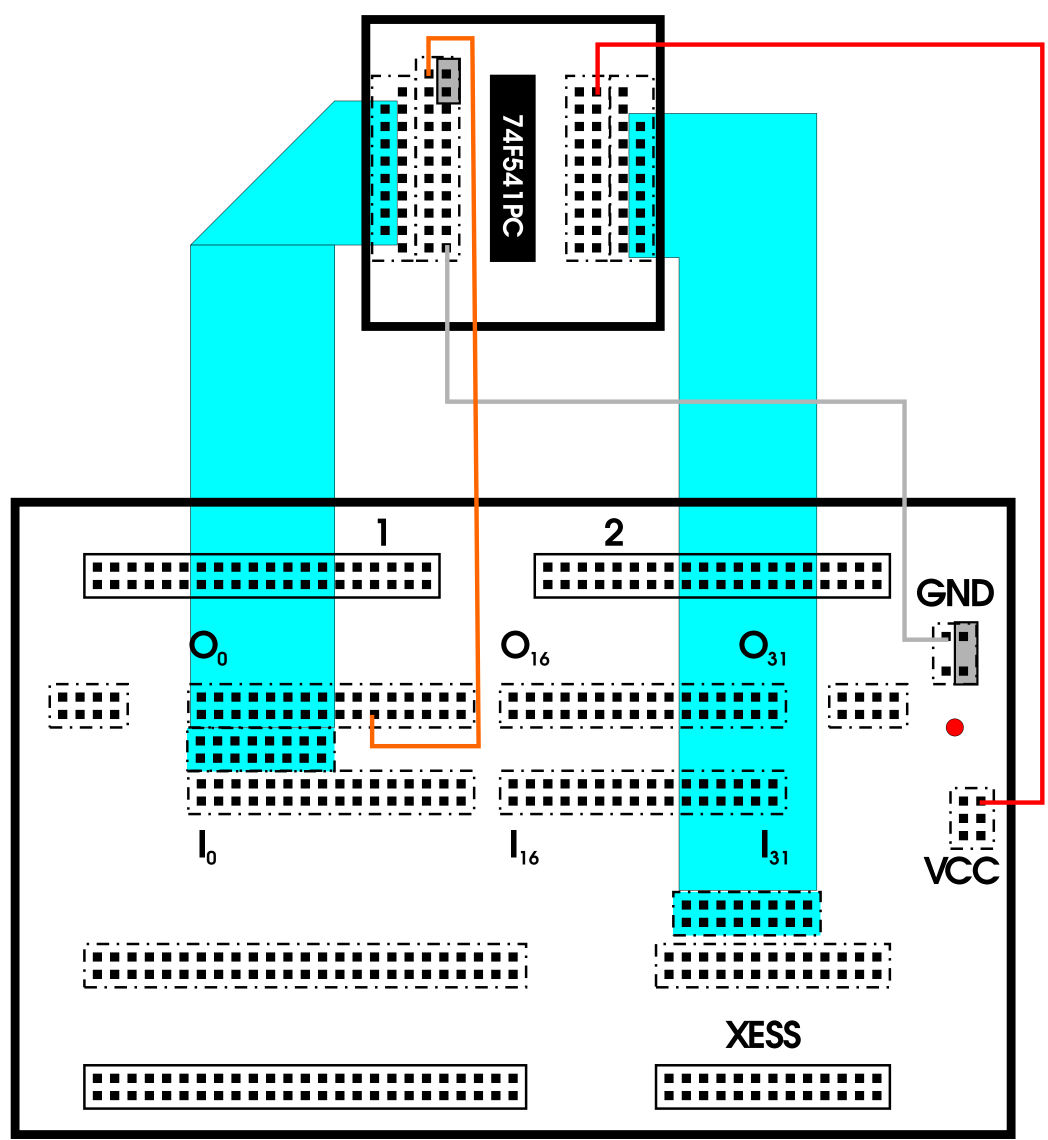


Abbildung A.2: Patch-Board-Verbindungen mit Zusatzboard

A.2 RS232-Terminal

1. RS232-Schnittstelle des XESS-Boards mit derjenigen des RS232-Terminal-PCs verbinden
2. HyperTerminal starten und in den Properties den richtigen (!) COM-Port auswählen, sowie unter Configure... die folgenden Werte einstellen :
   - Bits per second : 57600
   - Data bits : 8
   - Parity : None
   - Stop bits : 1
   - Flow control : None

A.3 Ethernet-Sender

1. IP-Adresse des sendenden PCs auf 10.0.0.1 setzen
2. Die Ethernet-Schnittstellen des sendenden PCs und des XESS-Boards über einen Switch oder direkt mit einem Cross-Connect-Kabel miteinander verbinden.
3. *Send Data*-Applikation starten.

A.4 Software-OS

1. *HW-OS V0.3* auf OS-PC starten.
2. *Global Reset* und *OS Reset* deaktivieren
3. Die Option *Enable automatic scheduling* aktivieren.

A.5 Erzeugen der Audio-Daten aus einer MP3-Datei

1. Liegen die Daten im mp3-Format vor, so müssen sie zuerst in eine WAV-Datei umgewandelt werden. Zum Beispiel mit *`Nullsoft Winamp Version 3.0`*:

- mit der rechten Maustaste auf die Datei im *Playlist Editor* klicken
- *convert* → *convert to PCM WAV*

2. Für die weiteren Schritte wird das Programm `sox` unter UNIX verwendet. Weil die Samplingrate reduziert wird, müssen die Daten zuerst mit einem Anti-Aliasing-Filter gefiltert werden. Hier wurde ein einpoliger Tiefpassfilter verwendet (`lowp`). Deshalb wurde auch die cutoff-Frequenz tief gewählt (3 KHz). Das entsprechende Komando lautet:

   ```
   sox -t wav Test.WAV -t wav Test3k.WAV lowp 3000
   ```

3. Mit dem folgenden Kommando wird aus den gefilterten Daten (`Test3k.WAV`) die Datei `Test.raw` im gewünschten Format erzeugt:

   ```
   sox -t wav Test3k.WAV -r 12207 -u -w -c 1 -t raw Test.raw
   ```

   Dabei bedeuten die Optionen für die Zieldatei folgendes:

   - `-r 12207`: Samplingfrequenz 12.207 kHz
   - `-u`: unsigned
   - `-w`: Die Grösse eines Samples beträgt 16 Bit (ein Wort)
   - `-c 1`: mono (1 Kanal)
   - `-t raw`: "rohe" PCM-Daten

A.6 Tests

- Switch SW1 drücken : Es sollte das Geräusch des Sägezahngenerators hörbar sein. Das Fenster der HW-OS-Applikation sollte im linken Slot den Audio-Task anzeigen. Zudem sollte der *Port 7000 / Audio Stream*-Puffer *not empty* und der Pushbutton SW1 *pressed* sein. Nach dem Loslassen des Switchs sollte wieder der ursprüngliche Zustand hergestellt sein.

- Switch SW3 drücken : Nun sollte auf dem LED-Bar das Knightrider-Lauflicht erscheinen. Entsprechend müsste nun auch die Anzeige des HW-OS-Femsters sein.

- Audiostreaming : Im *Send Data*-Programm eine, gemäss Abschnitt A.5 erzeugte, Audio-Datei auswählen. Den Datentyp auf *Audio* setzen und *Send*-Button drücken. Nach ev. anfänglichen Rauschen sollte die Musik (mit leichtem "Knacken") hörbar sein.

- Text-Nachricht : Im *Send Data*-Programm eine Text-Datei auswählen. Den Datentyp auf *Text* setzen. Je nach Wahl des Ports und der AES-Option sollte nun auf dem RS232-Terminal folgende Ausgaben erscheinen :
  - Port 6000 : Hex Dump
  - Port 6001 und *Encryption* oder Port 6002 und *Off* : Klartext
  - In allen anderen Fällen sollte eine "unleserliche" Ausgabe erscheinen.
- Abschliessend können nun die oben durchgeführten Tests miteinander kombiniert werden.

Anhang B

Synthese-Informationen

B.1 Constraints-File `top.ucf`

```
#this is the top level user constraints file
#it defines the module boundaries, the location of
#the bus macros and, finally, contains the pin
#assignement


# the area constraints for the modules
INST "os_left_inst" AREA_GROUP = "os_left";
AREA_GROUP "os_left" RANGE = "CLB_R1C1:CLB_R56C8";
AREA_GROUP "os_left" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE;

INST "task_left_inst" AREA_GROUP = "task_left";
AREA_GROUP "task_left" RANGE = "CLB_R1C9:CLB_R56C36";
AREA_GROUP "task_left" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE;

INST "os_middle_inst" AREA_GROUP = "os_middle";
AREA_GROUP "os_middle" RANGE = "CLB_R1C37:CLB_R56C48";
AREA_GROUP "os_middle" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE;

INST "task_right_inst" AREA_GROUP = "task_right";
AREA_GROUP "task_right" RANGE = "CLB_R1C49:CLB_R56C76";
AREA_GROUP "task_right" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE;

INST "os_right_inst" AREA_GROUP = "os_right";
AREA_GROUP "os_right" RANGE = "CLB_R1C77:CLB_R56C84";
AREA_GROUP "os_right" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE;


# the location constraints for the bus macros
INST "bm_task_left_inst0" LOC = "TBUF_R56C5.0";
INST "bm_task_left_inst1" LOC = "TBUF_R55C5.0";
INST "bm_task_left_inst2" LOC = "TBUF_R54C5.0";
INST "bm_task_left_inst3" LOC = "TBUF_R53C5.0";
INST "bm_task_left_inst4" LOC = "TBUF_R52C5.0";
INST "bm_task_left_inst5" LOC = "TBUF_R51C5.0";
INST "bm_task_left_inst6" LOC = "TBUF_R50C5.0";
INST "bm_task_left_inst7" LOC = "TBUF_R49C5.0";
```

```
INST "bm_task_left_inst8" LOC = "TBUF_R48C5.0";
INST "bm_task_left_inst9" LOC = "TBUF_R47C5.0";
INST "bm_task_left_inst10" LOC = "TBUF_R46C5.0";

INST "bm_left_left_inst0" LOC = "TBUF_R1C5.0";
INST "bm_left_left_inst1" LOC = "TBUF_R2C5.0";
INST "bm_left_left_inst2" LOC = "TBUF_R3C5.0";
INST "bm_left_left_inst3" LOC = "TBUF_R4C5.0";
INST "bm_left_left_inst4" LOC = "TBUF_R5C5.0";
INST "bm_left_left_inst5" LOC = "TBUF_R6C5.0";
INST "bm_left_left_inst6" LOC = "TBUF_R7C5.0";
INST "bm_left_left_inst7" LOC = "TBUF_R8C5.0";
INST "bm_left_left_inst8" LOC = "TBUF_R9C5.0";
INST "bm_left_left_inst9" LOC = "TBUF_R10C5.0";
INST "bm_left_left_inst10" LOC = "TBUF_R11C5.0";
INST "bm_left_left_inst11" LOC = "TBUF_R12C5.0";
INST "bm_left_left_inst12" LOC = "TBUF_R13C5.0";
INST "bm_left_left_inst13" LOC = "TBUF_R14C5.0";
INST "bm_left_left_inst14" LOC = "TBUF_R15C5.0";
INST "bm_left_left_inst15" LOC = "TBUF_R16C5.0";
INST "bm_left_left_inst16" LOC = "TBUF_R17C5.0";
INST "bm_left_left_inst17" LOC = "TBUF_R18C5.0";
INST "bm_left_left_inst18" LOC = "TBUF_R19C5.0";

INST "bm_left_middle_inst0" LOC = "TBUF_R1C33.0";
INST "bm_left_middle_inst1" LOC = "TBUF_R2C33.0";
INST "bm_left_middle_inst2" LOC = "TBUF_R3C33.0";
INST "bm_left_middle_inst3" LOC = "TBUF_R4C33.0";
INST "bm_left_middle_inst4" LOC = "TBUF_R5C33.0";
INST "bm_left_middle_inst5" LOC = "TBUF_R6C33.0";
INST "bm_left_middle_inst6" LOC = "TBUF_R7C33.0";
INST "bm_left_middle_inst7" LOC = "TBUF_R8C33.0";
INST "bm_left_middle_inst8" LOC = "TBUF_R9C33.0";
INST "bm_left_middle_inst9" LOC = "TBUF_R10C33.0";
INST "bm_left_middle_inst10" LOC = "TBUF_R11C33.0";
INST "bm_left_middle_inst11" LOC = "TBUF_R12C33.0";
INST "bm_left_middle_inst12" LOC = "TBUF_R13C33.0";
INST "bm_left_middle_inst13" LOC = "TBUF_R14C33.0";
INST "bm_left_middle_inst14" LOC = "TBUF_R15C33.0";
INST "bm_left_middle_inst15" LOC = "TBUF_R16C33.0";
INST "bm_left_middle_inst16" LOC = "TBUF_R17C33.0";
INST "bm_left_middle_inst17" LOC = "TBUF_R18C33.0";
INST "bm_left_middle_inst18" LOC = "TBUF_R19C33.0";

INST "bm_right_middle_inst0" LOC = "TBUF_R1C45.0";
INST "bm_right_middle_inst1" LOC = "TBUF_R2C45.0";
INST "bm_right_middle_inst2" LOC = "TBUF_R3C45.0";
INST "bm_right_middle_inst3" LOC = "TBUF_R4C45.0";
INST "bm_right_middle_inst4" LOC = "TBUF_R5C45.0";
INST "bm_right_middle_inst5" LOC = "TBUF_R6C45.0";
INST "bm_right_middle_inst6" LOC = "TBUF_R7C45.0";
INST "bm_right_middle_inst7" LOC = "TBUF_R8C45.0";
INST "bm_right_middle_inst8" LOC = "TBUF_R9C45.0";
INST "bm_right_middle_inst9" LOC = "TBUF_R10C45.0";
INST "bm_right_middle_inst10" LOC = "TBUF_R11C45.0";
INST "bm_right_middle_inst11" LOC = "TBUF_R12C45.0";
INST "bm_right_middle_inst12" LOC = "TBUF_R13C45.0";
INST "bm_right_middle_inst13" LOC = "TBUF_R14C45.0";
INST "bm_right_middle_inst14" LOC = "TBUF_R15C45.0";
INST "bm_right_middle_inst15" LOC = "TBUF_R16C45.0";
INST "bm_right_middle_inst16" LOC = "TBUF_R17C45.0";
INST "bm_right_middle_inst17" LOC = "TBUF_R18C45.0";
```

```
INST "bm_right_middle_inst18" LOC = "TBUF_R19C45.0";

INST "bm_task_right_inst0" LOC = "TBUF_R56C45.0";
INST "bm_task_right_inst1" LOC = "TBUF_R55C45.0";
INST "bm_task_right_inst2" LOC = "TBUF_R54C45.0";
INST "bm_task_right_inst3" LOC = "TBUF_R53C45.0";
INST "bm_task_right_inst4" LOC = "TBUF_R52C45.0";
INST "bm_task_right_inst5" LOC = "TBUF_R51C45.0";
INST "bm_task_right_inst6" LOC = "TBUF_R50C45.0";
INST "bm_task_right_inst7" LOC = "TBUF_R49C45.0";
INST "bm_task_right_inst8" LOC = "TBUF_R48C45.0";
INST "bm_task_right_inst9" LOC = "TBUF_R47C45.0";
INST "bm_task_right_inst10" LOC = "TBUF_R46C45.0";

INST "bm_right_right_inst0" LOC = "TBUF_R1C73.0";
INST "bm_right_right_inst1" LOC = "TBUF_R2C73.0";
INST "bm_right_right_inst2" LOC = "TBUF_R3C73.0";
INST "bm_right_right_inst3" LOC = "TBUF_R4C73.0";
INST "bm_right_right_inst4" LOC = "TBUF_R5C73.0";
INST "bm_right_right_inst5" LOC = "TBUF_R6C73.0";
INST "bm_right_right_inst6" LOC = "TBUF_R7C73.0";
INST "bm_right_right_inst7" LOC = "TBUF_R8C73.0";
INST "bm_right_right_inst8" LOC = "TBUF_R9C73.0";
INST "bm_right_right_inst9" LOC = "TBUF_R10C73.0";
INST "bm_right_right_inst10" LOC = "TBUF_R11C73.0";
INST "bm_right_right_inst11" LOC = "TBUF_R12C73.0";
INST "bm_right_right_inst12" LOC = "TBUF_R13C73.0";
INST "bm_right_right_inst13" LOC = "TBUF_R14C73.0";
INST "bm_right_right_inst14" LOC = "TBUF_R15C73.0";
INST "bm_right_right_inst15" LOC = "TBUF_R16C73.0";
INST "bm_right_right_inst16" LOC = "TBUF_R17C73.0";
INST "bm_right_right_inst17" LOC = "TBUF_R18C73.0";
INST "bm_right_right_inst18" LOC = "TBUF_R19C73.0";

# pin assignement
NET "clock" LOC = "P89";

NET "rs232_tx" LOC = "P144";
NET "rs232_rx" LOC = "P141";

NET "codec_mclk" LOC = "P3";
NET "codec_lrck" LOC = "P4";
NET "codec_sclk" LOC = "P5";
NET "codec_sdin" LOC = "P6";
NET "codec_sdout" LOC = "P7";

NET "ledbar(9)" LOC = "P152";
NET "ledbar(8)" LOC = "P154";
NET "ledbar(7)" LOC = "P157";
NET "ledbar(6)" LOC = "P160";
NET "ledbar(5)" LOC = "P162";
NET "ledbar(4)" LOC = "P169";
NET "ledbar(3)" LOC = "P168";
NET "ledbar(2)" LOC = "P173";
NET "ledbar(1)" LOC = "P131";
NET "ledbar(0)" LOC = "P171";

NET "pc_clk_in" LOC = "P86";
NET "pc_we_in" LOC = "P87";
NET "pc_re_in" LOC = "P209";

NET "pc_data_in" LOC = "P93";
```

```
NET "pc_data_out" LOC = "P94";
NET "pc_event" LOC = "P208";

NET "vcc_os_left0" PULLUP;
NET "vcc_os_left0" LOC = "P238";
NET "vcc_os_middle0" PULLUP;
NET "vcc_os_middle0" LOC = "P216";
NET "vcc_os_right0" PULLUP;
NET "vcc_os_right0" LOC = "P175";
NET "vcc_task_left0" PULLUP;
NET "vcc_task_left0" LOC = "P218";
NET "vcc_task_right0" PULLUP;
NET "vcc_task_right0" LOC = "P205";

NET "gnd_os_left0" PULLDOWN;
NET "gnd_os_left0" LOC = "P237";
NET "gnd_os_middle0" PULLDOWN;
NET "gnd_os_middle0" LOC = "P95";
NET "gnd_os_right0" PULLDOWN;
NET "gnd_os_right0" LOC = "P116";
NET "gnd_task_left0" PULLDOWN;
NET "gnd_task_left0" LOC = "P217";
NET "gnd_task_right0" PULLDOWN;
NET "gnd_task_right0" LOC = "P203";

NET "fifo_disable_left_in" LOC = "P53";
NET "fifo_disable_right_in" LOC = "P187";
NET "fifo_disable_out" LOC = "P215";

NET "sw1_in" LOC = "P174";
NET "sw3_in" LOC = "P176";


#disable the ram chips on the xess board
NET "ram_left_dis" LOC = "P186";
NET "ram_right_dis" LOC = "P109";

NET "rx_clk" LOC = "P213";
NET "rx_dv" LOC = "P26";
NET "rx_data(3)" LOC = "P34";
NET "rx_data(2)" LOC = "P35";
NET "rx_data(1)" LOC = "P36";
NET "rx_data(0)" LOC = "P38";
NET "tx_clk" LOC = "P210";
NET "tx_en" LOC = "P25";
NET "tx_data(3)" LOC = "P39";
NET "tx_data(2)" LOC = "P40";
NET "tx_data(1)" LOC = "P41";
NET "tx_data(0)" LOC = "P42";
NET "tx_er" LOC = "P27";
NET "mdc" LOC = "P19";
NET "mdio" LOC = "P20";
NET "trste" LOC = "P24";
```

B.2 Tabellen der Verbindungen in Top.vhd

B.2.1 Inter-Frame-Communication Channels

Die Tabellen B.1 und B.2 zeigen, wie die Ports der OS-Module mit den IFCC-Signalen zum nächsten Bus-Macro verbunden sind. Die Signale zwischen diesen Bus-Macros sind jeweils direkt miteinander verbunden, d.h. :

- IFCC_osleft_to_bm(43..0) ⇒ IFCC_bmleft_to_taskleft(43..0) ⇒ IFCC_taskleft_to_bmright(43..0) ⇒ IFCC_bmleft_to_osmiddle(43..0)
- IFCC_bm_to_osleft(31..0) ⇐ IFCC_taskleft_to_bmleft(31..0) ⇐ IFCC_bmright_to_taskleft(31..0) ⇐ IFCC_osmiddle_to_bmleft(31..0)
- IFCC_osmiddle_to_bmright(43..0) ⇒ IFCC_bmleft_to_taskright(43..0) ⇒ IFCC_taskright_to_bmright(43..0) ⇒ IFCC_bm_to_osright(43..0)
- IFCC_bmright_to_osmiddle(31..0) ⇐ IFCC_taskright_to_bmleft(31..0) ⇐ IFCC_bmright_to_taskright(31..0) ⇐ IFCC_osright_to_bm(31..0)

OS Left			OS Middle	OS Middle			OS Right
ledbar_to_mid(9)	osleft_to_bm(43)	bmleft_to_osmiddle(43)	ledbar_from_left(9)	ledbar_to_right(9)	osmiddle_to_bmright(43)	bm_to_osright(43)	ledbar_from_mid(9)
ledbar_to_mid(8)	osleft_to_bm(42)	bmleft_to_osmiddle(42)	ledbar_from_left(8)	ledbar_to_right(8)	osmiddle_to_bmright(42)	bm_to_osright(42)	ledbar_from_mid(8)
ledbar_to_mid(7)	osleft_to_bm(41)	bmleft_to_osmiddle(41)	ledbar_from_left(7)	ledbar_to_right(7)	osmiddle_to_bmright(41)	bm_to_osright(41)	ledbar_from_mid(7)
ledbar_to_mid(6)	osleft_to_bm(40)	bmleft_to_osmiddle(40)	ledbar_from_left(6)	ledbar_to_right(6)	osmiddle_to_bmright(40)	bm_to_osright(40)	ledbar_from_mid(6)
ledbar_to_mid(5)	osleft_to_bm(39)	bmleft_to_osmiddle(39)	ledbar_from_left(5)	ledbar_to_right(5)	osmiddle_to_bmright(39)	bm_to_osright(39)	ledbar_from_mid(5)
ledbar_to_mid(4)	osleft_to_bm(38)	bmleft_to_osmiddle(38)	ledbar_from_left(4)	ledbar_to_right(4)	osmiddle_to_bmright(38)	bm_to_osright(38)	ledbar_from_mid(4)
ledbar_to_mid(3)	osleft_to_bm(37)	bmleft_to_osmiddle(37)	ledbar_from_left(3)	ledbar_to_right(3)	osmiddle_to_bmright(37)	bm_to_osright(37)	ledbar_from_mid(3)
ledbar_to_mid(2)	osleft_to_bm(36)	bmleft_to_osmiddle(36)	ledbar_from_left(2)	ledbar_to_right(2)	osmiddle_to_bmright(36)	bm_to_osright(36)	ledbar_from_mid(2)
ledbar_to_mid(1)	osleft_to_bm(35)	bmleft_to_osmiddle(35)	ledbar_from_left(1)	ledbar_to_right(1)	osmiddle_to_bmright(35)	bm_to_osright(35)	ledbar_from_mid(1)
ledbar_to_mid(0)	osleft_to_bm(34)	bmleft_to_osmiddle(34)	ledbar_from_left(0)	ledbar_to_right(0)	osmiddle_to_bmright(34)	bm_to_osright(34)	ledbar_from_mid(0)
	osleft_to_bm(33)	bmleft_to_osmiddle(33)		t1_read_from_fifo_sel_to_right(2)	osmiddle_to_bmright(33)	bm_to_osright(33)	t1_read_from_fifo_sel_from_mid(2)
	osleft_to_bm(32)	bmleft_to_osmiddle(32)		t1_read_from_fifo_sel_to_right(1)	osmiddle_to_bmright(32)	bm_to_osright(32)	t1_read_from_fifo_sel_from_mid(1)
	osleft_to_bm(31)	bmleft_to_osmiddle(31)	dummy_in(3)	t1_read_from_fifo_sel_to_right(0)	osmiddle_to_bmright(31)	bm_to_osright(31)	t1_read_from_fifo_sel_from_mid(0)
	osleft_to_bm(30)	bmleft_to_osmiddle(30)	dummy_in(2)	t0_read_from_fifo_sel_to_right(2)	osmiddle_to_bmright(30)	bm_to_osright(30)	t0_read_from_fifo_sel_from_mid(2)
	osleft_to_bm(29)	bmleft_to_osmiddle(29)	dummy_in(1)	t0_read_from_fifo_sel_to_right(1)	osmiddle_to_bmright(29)	bm_to_osright(29)	t0_read_from_fifo_sel_from_mid(1)
	osleft_to_bm(28)	bmleft_to_osmiddle(28)	dummy_in(0)	t0_read_from_fifo_sel_to_right(0)	osmiddle_to_bmright(28)	bm_to_osright(28)	t0_read_from_fifo_sel_from_mid(0)
t0_dfifo2_we_to_mid	osleft_to_bm(27)	bmleft_to_osmiddle(27)	t0_dfifo2_we_from_left	t0_dfifo2_we_to_right	osmiddle_to_bmright(27)	bm_to_osright(27)	t0_dfifo2_we_from_mid
t0_dfifo2_data_to_mid(7)	osleft_to_bm(26)	bmleft_to_osmiddle(26)	t0_dfifo2_data_from_left(7)	t0_dfifo2_data_to_right(7)	osmiddle_to_bmright(26)	bm_to_osright(26)	t0_dfifo2_data_from_mid(7)
t0_dfifo2_data_to_mid(6)	osleft_to_bm(25)	bmleft_to_osmiddle(25)	t0_dfifo2_data_from_left(6)	t0_dfifo2_data_to_right(6)	osmiddle_to_bmright(25)	bm_to_osright(25)	t0_dfifo2_data_from_mid(6)
t0_dfifo2_data_to_mid(5)	osleft_to_bm(24)	bmleft_to_osmiddle(24)	t0_dfifo2_data_from_left(5)	t0_dfifo2_data_to_right(5)	osmiddle_to_bmright(24)	bm_to_osright(24)	t0_dfifo2_data_from_mid(5)
t0_dfifo2_data_to_mid(4)	osleft_to_bm(23)	bmleft_to_osmiddle(23)	t0_dfifo2_data_from_left(4)	t0_dfifo2_data_to_right(4)	osmiddle_to_bmright(23)	bm_to_osright(23)	t0_dfifo2_data_from_mid(4)
t0_dfifo2_data_to_mid(3)	osleft_to_bm(22)	bmleft_to_osmiddle(22)	t0_dfifo2_data_from_left(3)	t0_dfifo2_data_to_right(3)	osmiddle_to_bmright(22)	bm_to_osright(22)	t0_dfifo2_data_from_mid(3)
t0_dfifo2_data_to_mid(2)	osleft_to_bm(21)	bmleft_to_osmiddle(21)	t0_dfifo2_data_from_left(2)	t0_dfifo2_data_to_right(2)	osmiddle_to_bmright(21)	bm_to_osright(21)	t0_dfifo2_data_from_mid(2)
t0_dfifo2_data_to_mid(1)	osleft_to_bm(20)	bmleft_to_osmiddle(20)	t0_dfifo2_data_from_left(1)	t0_dfifo2_data_to_right(1)	osmiddle_to_bmright(20)	bm_to_osright(20)	t0_dfifo2_data_from_mid(1)
t0_dfifo2_data_to_mid(0)	osleft_to_bm(19)	bmleft_to_osmiddle(19)	t0_dfifo2_data_from_left(0)	t0_dfifo2_data_to_right(0)	osmiddle_to_bmright(19)	bm_to_osright(19)	t0_dfifo2_data_from_mid(0)
t0_dfifo2_sel_to_mid(2)	osleft_to_bm(18)	bmleft_to_osmiddle(18)	t0_dfifo2_sel_from_left(2)	t0_dfifo2_sel_to_right(2)	osmiddle_to_bmright(18)	bm_to_osright(18)	t0_dfifo2_sel_from_mid(2)
t0_dfifo2_sel_to_mid(1)	osleft_to_bm(17)	bmleft_to_osmiddle(17)	t0_dfifo2_sel_from_left(1)	t0_dfifo2_sel_to_right(1)	osmiddle_to_bmright(17)	bm_to_osright(17)	t0_dfifo2_sel_from_mid(1)
t0_dfifo2_sel_to_mid(0)	osleft_to_bm(16)	bmleft_to_osmiddle(16)	t0_dfifo2_sel_from_left(0)	t0_dfifo2_sel_to_right(0)	osmiddle_to_bmright(16)	bm_to_osright(16)	t0_dfifo2_sel_from_mid(0)
t0_dfifo1_re_to_mid	osleft_to_bm(15)	bmleft_to_osmiddle(15)	t0_dfifo1_re_from_left	t0_dfifo1_re_to_right	osmiddle_to_bmright(15)	bm_to_osright(15)	t0_dfifo1_re_from_mid
t1_ipfifo1_empty_to_mid	osleft_to_bm(14)	bmleft_to_osmiddle(14)	t1_ipfifo1_empty_from_left	t1_dfifo2_we_to_right	osmiddle_to_bmright(14)	bm_to_osright(14)	t1_dfifo2_we_from_mid
t1_ipfifo1_data_to_mid(7)	osleft_to_bm(13)	bmleft_to_osmiddle(13)	t1_ipfifo1_data_from_left(7)	t1_dfifo2_data_to_right(7)	osmiddle_to_bmright(13)	bm_to_osright(13)	t1_dfifo2_data_from_mid(7)
t1_ipfifo1_data_to_mid(6)	osleft_to_bm(12)	bmleft_to_osmiddle(12)	t1_ipfifo1_data_from_left(6)	t1_dfifo2_data_to_right(6)	osmiddle_to_bmright(12)	bm_to_osright(12)	t1_dfifo2_data_from_mid(6)
t1_ipfifo1_data_to_mid(5)	osleft_to_bm(11)	bmleft_to_osmiddle(11)	t1_ipfifo1_data_from_left(5)	t1_dfifo2_data_to_right(5)	osmiddle_to_bmright(11)	bm_to_osright(11)	t1_dfifo2_data_from_mid(5)
t1_ipfifo1_data_to_mid(4)	osleft_to_bm(10)	bmleft_to_osmiddle(10)	t1_ipfifo1_data_from_left(4)	t1_dfifo2_data_to_right(4)	osmiddle_to_bmright(10)	bm_to_osright(10)	t1_dfifo2_data_from_mid(4)
t1_ipfifo1_data_to_mid(3)	osleft_to_bm(9)	bmleft_to_osmiddle(9)	t1_ipfifo1_data_from_left(3)	t1_dfifo2_data_to_right(3)	osmiddle_to_bmright(9)	bm_to_osright(9)	t1_dfifo2_data_from_mid(3)
t1_ipfifo1_data_to_mid(2)	osleft_to_bm(8)	bmleft_to_osmiddle(8)	t1_ipfifo1_data_from_left(2)	t1_dfifo2_data_to_right(2)	osmiddle_to_bmright(8)	bm_to_osright(8)	t1_dfifo2_data_from_mid(2)
t1_ipfifo1_data_to_mid(1)	osleft_to_bm(7)	bmleft_to_osmiddle(7)	t1_ipfifo1_data_from_left(1)	t1_dfifo2_data_to_right(1)	osmiddle_to_bmright(7)	bm_to_osright(7)	t1_dfifo2_data_from_mid(1)
t1_ipfifo1_data_to_mid(0)	osleft_to_bm(6)	bmleft_to_osmiddle(6)	t1_ipfifo1_data_from_left(0)	t1_dfifo2_data_to_right(0)	osmiddle_to_bmright(6)	bm_to_osright(6)	t1_dfifo2_data_from_mid(0)
arpfifo_empty_to_mid	osleft_to_bm(5)	bmleft_to_osmiddle(5)	arpfifo_empty_from_left	t1_dfifo1_re_to_right	osmiddle_to_bmright(5)	bm_to_osright(5)	t1_dfifo1_re_from_mid
sendfifo_empty_to_mid	osleft_to_bm(4)	bmleft_to_osmiddle(4)	sendfifo_empty_from_left	t1_dfifo2_sel_to_right(2)	osmiddle_to_bmright(4)	bm_to_osright(4)	t1_dfifo2_sel_from_mid(2)
t0_finished_to_mid	osleft_to_bm(3)	bmleft_to_osmiddle(3)	t0_finished_from_mid	t1_dfifo2_sel_to_right(1)	osmiddle_to_bmright(3)	bm_to_osright(3)	t1_dfifo2_sel_from_mid(1)
audiofifo_empty_to_mid	osleft_to_bm(2)	bmleft_to_osmiddle(2)	audiofifo_empty_from_left	t1_dfifo2_sel_to_right(0)	osmiddle_to_bmright(2)	bm_to_osright(2)	t1_dfifo2_sel_from_mid(0)
t1_audiofifo2_full_to_mid	osleft_to_bm(1)	bmleft_to_osmiddle(1)	t1_audiofifo2_full_from_left	os_right_reset	osmiddle_to_bmright(1)	bm_to_osright(1)	rst
rs232_tx_to_mid	osleft_to_bm(0)	bmleft_to_osmiddle(0)	rs232_tx_from_left	rs232_tx_to_right	osmiddle_to_bmright(0)	bm_to_osright(0)	rs232_tx_from_mid

Tabelle B.1: IFCCs Richtung rechts

OS Left			OS Middle	OS Middle			OS Right
	bm_to_osleft(31)	osmiddle_to_bmleft(31)		dummy_in(7)	bmright_to_osmiddle(31)	osright_to_bm(31)	
tasktoaudio_sel_from_mid	bm_to_osleft(30)	osmiddle_to_bmleft(30)	tasktoaudio_sel_to_left	dummy_in(6)	bmright_to_osmiddle(30)	osright_to_bm(30)	
t1_audiofifo2_we_from_mid	bm_to_osleft(29)	osmiddle_to_bmleft(29)	t1_audiofifo2_we_to_left	dummy_in(5)	bmright_to_osmiddle(29)	osright_to_bm(29)	
rs232_rx_from_mid	bm_to_osleft(28)	osmiddle_to_bmleft(28)	rs232_rx_to_left	dummy_in(4)	bmright_to_osmiddle(28)	osright_to_bm(28)	
t0_enable_from_mid	bm_to_osleft(27)	osmiddle_to_bmleft(27)	t0_enable_to_mid		bmright_to_osmiddle(27)	osright_to_bm(27)	
t0_reset_from_mid	bm_to_osleft(26)	osmiddle_to_bmleft(26)	t0_reset_to_mid		bmright_to_osmiddle(26)	osright_to_bm(26)	
SW1_from_mid	bm_to_osleft(25)	osmiddle_to_bmleft(25)	SW1_to_left	SW1_from_right	bmright_to_osmiddle(25)	osright_to_bm(25)	SW1_to_mid
SW3_from_mid	bm_to_osleft(24)	osmiddle_to_bmleft(24)	SW3_to_left	SW3_from_right	bmright_to_osmiddle(24)	osright_to_bm(24)	SW3_to_mid
iptotask_switch_from_mid(1)	bm_to_osleft(23)	osmiddle_to_bmleft(23)	iptotask_switch_to_left(1)	fifo6002_empty_from_right	bmright_to_osmiddle(23)	osright_to_bm(23)	fifo6002_empty_to_mid
iptotask_switch_from_mid(0)	bm_to_osleft(22)	osmiddle_to_bmleft(22)	iptotask_switch_to_left(0)	fifo6001_empty_from_right	bmright_to_osmiddle(22)	osright_to_bm(22)	fifo6001_empty_to_mid
t0_ipordata_fifo_switch_from_mid(1)	bm_to_osleft(21)	osmiddle_to_bmleft(21)	t0_ipordata_fifo_switch_to_left(1)	fifo6000_empty_from_right	bmright_to_osmiddle(21)	osright_to_bm(21)	fifo6000_empty_to_mid
t0_ipordata_fifo_switch_from_mid(0)	bm_to_osleft(20)	osmiddle_to_bmleft(20)	t0_ipordata_fifo_switch_to_left(0)	rs232_rx_from_right	bmright_to_osmiddle(20)	osright_to_bm(20)	rs232_rx_to_mid
t0_dfifo2_full_from_mid	bm_to_osleft(19)	osmiddle_to_bmleft(19)	t0_dfifo2_full_to_left	t0_dfifo2_full_from_right	bmright_to_osmiddle(19)	osright_to_bm(19)	t0_dfifo2_full_to_mid
t0_dfifo1_empty_from_mid	bm_to_osleft(18)	osmiddle_to_bmleft(18)	t0_dfifo1_empty_to_left	t0_dfifo1_empty_from_right	bmright_to_osmiddle(18)	osright_to_bm(18)	t0_dfifo1_empty_to_mid
t0_dfifo1_data_from_mid(7)	bm_to_osleft(17)	osmiddle_to_bmleft(17)	t0_dfifo1_data_to_left(7)	t0_dfifo1_data_from_right(7)	bmright_to_osmiddle(17)	osright_to_bm(17)	t0_dfifo1_data_to_mid(7)
t0_dfifo1_data_from_mid(6)	bm_to_osleft(16)	osmiddle_to_bmleft(16)	t0_dfifo1_data_to_left(6)	t0_dfifo1_data_from_right(6)	bmright_to_osmiddle(16)	osright_to_bm(16)	t0_dfifo1_data_to_mid(6)
t0_dfifo1_data_from_mid(5)	bm_to_osleft(15)	osmiddle_to_bmleft(15)	t0_dfifo1_data_to_left(5)	t0_dfifo1_data_from_right(5)	bmright_to_osmiddle(15)	osright_to_bm(15)	t0_dfifo1_data_to_mid(5)
t0_dfifo1_data_from_mid(4)	bm_to_osleft(14)	osmiddle_to_bmleft(14)	t0_dfifo1_data_to_left(4)	t0_dfifo1_data_from_right(4)	bmright_to_osmiddle(14)	osright_to_bm(14)	t0_dfifo1_data_to_mid(4)
t0_dfifo1_data_from_mid(3)	bm_to_osleft(13)	osmiddle_to_bmleft(13)	t0_dfifo1_data_to_left(3)	t0_dfifo1_data_from_right(3)	bmright_to_osmiddle(13)	osright_to_bm(13)	t0_dfifo1_data_to_mid(3)
t0_dfifo1_data_from_mid(2)	bm_to_osleft(12)	osmiddle_to_bmleft(12)	t0_dfifo1_data_to_left(2)	t0_dfifo1_data_from_right(2)	bmright_to_osmiddle(12)	osright_to_bm(12)	t0_dfifo1_data_to_mid(2)
t0_dfifo1_data_from_mid(1)	bm_to_osleft(11)	osmiddle_to_bmleft(11)	t0_dfifo1_data_to_left(1)	t0_dfifo1_data_from_right(1)	bmright_to_osmiddle(11)	osright_to_bm(11)	t0_dfifo1_data_to_mid(1)
t0_dfifo1_data_from_mid(0)	bm_to_osleft(10)	osmiddle_to_bmleft(10)	t0_dfifo1_data_to_left(0)	t0_dfifo1_data_from_right(0)	bmright_to_osmiddle(10)	osright_to_bm(10)	t0_dfifo1_data_to_mid(0)
t1_ipfifo1_re_from_mid	bm_to_osleft(9)	osmiddle_to_bmleft(9)	t1_ipfifo1_re_to_left	t1_dfifo2_full_from_right	bmright_to_osmiddle(9)	osright_to_bm(9)	t1_dfifo2_full_to_mid
rst	bm_to_osleft(8)	osmiddle_to_bmleft(8)	os_left_reset	t1_dfifo1_empty_from_right	bmright_to_osmiddle(8)	osright_to_bm(8)	t1_dfifo1_empty_to_mid
t1_audiofifo2_data_from_mid(7)	bm_to_osleft(7)	osmiddle_to_bmleft(7)	t1_audiofifo2_data_to_left(7)	t1_dfifo1_data_from_right(7)	bmright_to_osmiddle(7)	osright_to_bm(7)	t1_dfifo1_data_to_mid(7)
t1_audiofifo2_data_from_mid(6)	bm_to_osleft(6)	osmiddle_to_bmleft(6)	t1_audiofifo2_data_to_left(6)	t1_dfifo1_data_from_right(6)	bmright_to_osmiddle(6)	osright_to_bm(6)	t1_dfifo1_data_to_mid(6)
t1_audiofifo2_data_from_mid(5)	bm_to_osleft(5)	osmiddle_to_bmleft(5)	t1_audiofifo2_data_to_left(5)	t1_dfifo1_data_from_right(5)	bmright_to_osmiddle(5)	osright_to_bm(5)	t1_dfifo1_data_to_mid(5)
t1_audiofifo2_data_from_mid(4)	bm_to_osleft(4)	osmiddle_to_bmleft(4)	t1_audiofifo2_data_to_left(4)	t1_dfifo1_data_from_right(4)	bmright_to_osmiddle(4)	osright_to_bm(4)	t1_dfifo1_data_to_mid(4)
t1_audiofifo2_data_from_mid(3)	bm_to_osleft(3)	osmiddle_to_bmleft(3)	t1_audiofifo2_data_to_left(3)	t1_dfifo1_data_from_right(3)	bmright_to_osmiddle(3)	osright_to_bm(3)	t1_dfifo1_data_to_mid(3)
t1_audiofifo2_data_from_mid(2)	bm_to_osleft(2)	osmiddle_to_bmleft(2)	t1_audiofifo2_data_to_left(2)	t1_dfifo1_data_from_right(2)	bmright_to_osmiddle(2)	osright_to_bm(2)	t1_dfifo1_data_to_mid(2)
t1_audiofifo2_data_from_mid(1)	bm_to_osleft(1)	osmiddle_to_bmleft(1)	t1_audiofifo2_data_to_left(1)	t1_dfifo1_data_from_right(1)	bmright_to_osmiddle(1)	osright_to_bm(1)	t1_dfifo1_data_to_mid(1)
t1_audiofifo2_data_from_mid(0)	bm_to_osleft(0)	osmiddle_to_bmleft(0)	t1_audiofifo2_data_to_left(0)	t1_dfifo1_data_from_right(0)	bmright_to_osmiddle(0)	osright_to_bm(0)	t1_dfifo1_data_to_mid(0)

Tabelle B.2: IFCCs Richtung links

B.2.2 Standard-Task-Interfaces

OS Left			Task Left
	IF_osleft_to_bm(15)	IF_bmleft_to_taskleft(15)	
	IF_osleft_to_bm(14)	IF_bmleft_to_taskleft(14)	
	IF_osleft_to_bm(13)	IF_bmleft_to_taskleft(13)	
t0_fifo1_din(7)	IF_osleft_to_bm(12)	IF_bmleft_to_taskleft(12)	t_fifo1_din(7)
t0_fifo1_din(6)	IF_osleft_to_bm(11)	IF_bmleft_to_taskleft(11)	t_fifo1_din(6)
t0_fifo1_din(5)	IF_osleft_to_bm(10)	IF_bmleft_to_taskleft(10)	t_fifo1_din(5)
t0_fifo1_din(4)	IF_osleft_to_bm(9)	IF_bmleft_to_taskleft(9)	t_fifo1_din(4)
t0_fifo1_din(3)	IF_osleft_to_bm(8)	IF_bmleft_to_taskleft(8)	t_fifo1_din(3)
t0_fifo1_din(2)	IF_osleft_to_bm(7)	IF_bmleft_to_taskleft(7)	t_fifo1_din(2)
t0_fifo1_din(1)	IF_osleft_to_bm(6)	IF_bmleft_to_taskleft(6)	t_fifo1_din(1)
t0_fifo1_din(0)	IF_osleft_to_bm(5)	IF_bmleft_to_taskleft(5)	t_fifo1_din(0)
t0_fifo1_empty	IF_osleft_to_bm(4)	IF_bmleft_to_taskleft(4)	t_fifo1_empty
t0_rs232_rx	IF_osleft_to_bm(3)	IF_bmleft_to_taskleft(3)	t_rs232_rx
t0_fifo2_full	IF_osleft_to_bm(2)	IF_bmleft_to_taskleft(2)	t_fifo2_full
t0_enable	IF_osleft_to_bm(1)	IF_bmleft_to_taskleft(1)	t_enable
t0_reset	IF_osleft_to_bm(0)	IF_bmleft_to_taskleft(0)	t_rst

Tabelle B.3: STI-Verbindungen Slot 0, Richtung rechts

OS Middle			Task Right
	IF_osmiddle_to_bmright(15)	IF_bmleft_to_taskright(15)	
	IF_osmiddle_to_bmright(14)	IF_bmleft_to_taskright(14)	
	IF_osmiddle_to_bmright(13)	IF_bmleft_to_taskright(13)	
t1_fifo1_din(7)	IF_osmiddle_to_bmright(12)	IF_bmleft_to_taskright(12)	t_fifo1_din(7)
t1_fifo1_din(6)	IF_osmiddle_to_bmright(11)	IF_bmleft_to_taskright(11)	t_fifo1_din(6)
t1_fifo1_din(5)	IF_osmiddle_to_bmright(10)	IF_bmleft_to_taskright(10)	t_fifo1_din(5)
t1_fifo1_din(4)	IF_osmiddle_to_bmright(9)	IF_bmleft_to_taskright(9)	t_fifo1_din(4)
t1_fifo1_din(3)	IF_osmiddle_to_bmright(8)	IF_bmleft_to_taskright(8)	t_fifo1_din(3)
t1_fifo1_din(2)	IF_osmiddle_to_bmright(7)	IF_bmleft_to_taskright(7)	t_fifo1_din(2)
t1_fifo1_din(1)	IF_osmiddle_to_bmright(6)	IF_bmleft_to_taskright(6)	t_fifo1_din(1)
t1_fifo1_din(0)	IF_osmiddle_to_bmright(5)	IF_bmleft_to_taskright(5)	t_fifo1_din(0)
t1_fifo1_empty	IF_osmiddle_to_bmright(4)	IF_bmleft_to_taskright(4)	t_fifo1_empty
t1_rs232_rx	IF_osmiddle_to_bmright(3)	IF_bmleft_to_taskright(3)	t_rs232_rx
t1_fifo2_full	IF_osmiddle_to_bmright(2)	IF_bmleft_to_taskright(2)	t_fifo2_full
t1_enable	IF_osmiddle_to_bmright(1)	IF_bmleft_to_taskright(1)	t_enable
t1_reset	IF_osmiddle_to_bmright(0)	IF_bmleft_to_taskright(0)	t_rst

Tabelle B.4: STI-Verbindungen Slot 1, Richtung rechts

OS Left			Task Left
	IF_bm_to_osleft(27)	IF_taskleft_to_bmleft(27)	
	IF_bm_to_osleft(26)	IF_taskleft_to_bmleft(26)	
	IF_bm_to_osleft(25)	IF_taskleft_to_bmleft(25)	
t0_fifo2_sel(2)	IF_bm_to_osleft(24)	IF_taskleft_to_bmleft(24)	t_fifo2_sel(2)
t0_fifo2_sel(1)	IF_bm_to_osleft(23)	IF_taskleft_to_bmleft(23)	t_fifo2_sel(1)
t0_fifo2_sel(0)	IF_bm_to_osleft(22)	IF_taskleft_to_bmleft(22)	t_fifo2_sel(0)
t0_fifo2_dout(7)	IF_bm_to_osleft(21)	IF_taskleft_to_bmleft(21)	t_fifo2_dout(7)
t0_fifo2_dout(6)	IF_bm_to_osleft(20)	IF_taskleft_to_bmleft(20)	t_fifo2_dout(6)
t0_fifo2_dout(5)	IF_bm_to_osleft(19)	IF_taskleft_to_bmleft(19)	t_fifo2_dout(5)
t0_fifo2_dout(4)	IF_bm_to_osleft(18)	IF_taskleft_to_bmleft(18)	t_fifo2_dout(4)
t0_fifo2_dout(3)	IF_bm_to_osleft(17)	IF_taskleft_to_bmleft(17)	t_fifo2_dout(3)
t0_fifo2_dout(2)	IF_bm_to_osleft(16)	IF_taskleft_to_bmleft(16)	t_fifo2_dout(2)
t0_fifo2_dout(1)	IF_bm_to_osleft(15)	IF_taskleft_to_bmleft(15)	t_fifo2_dout(1)
t0_fifo2_dout(0)	IF_bm_to_osleft(14)	IF_taskleft_to_bmleft(14)	t_fifo2_dout(0)
t0_fifo2_we	IF_bm_to_osleft(13)	IF_taskleft_to_bmleft(13)	t_fifo2_we
t0_fifo1_re	IF_bm_to_osleft(12)	IF_taskleft_to_bmleft(12)	t_fifo1_re
t0_rs232_tx	IF_bm_to_osleft(11)	IF_taskleft_to_bmleft(11)	t_rs232_tx
t0_ledbar(9)	IF_bm_to_osleft(10)	IF_taskleft_to_bmleft(10)	t_ledbar(9)
t0_ledbar(8)	IF_bm_to_osleft(9)	IF_taskleft_to_bmleft(9)	t_ledbar(8)
t0_ledbar(7)	IF_bm_to_osleft(8)	IF_taskleft_to_bmleft(8)	t_ledbar(7)
t0_ledbar(6)	IF_bm_to_osleft(7)	IF_taskleft_to_bmleft(7)	t_ledbar(6)
t0_ledbar(5)	IF_bm_to_osleft(6)	IF_taskleft_to_bmleft(6)	t_ledbar(5)
t0_ledbar(4)	IF_bm_to_osleft(5)	IF_taskleft_to_bmleft(5)	t_ledbar(4)
t0_ledbar(3)	IF_bm_to_osleft(4)	IF_taskleft_to_bmleft(4)	t_ledbar(3)
t0_ledbar(2)	IF_bm_to_osleft(3)	IF_taskleft_to_bmleft(3)	t_ledbar(2)
t0_ledbar(1)	IF_bm_to_osleft(2)	IF_taskleft_to_bmleft(2)	t_ledbar(1)
t0_ledbar(0)	IF_bm_to_osleft(1)	IF_taskleft_to_bmleft(1)	t_ledbar(0)
t0_finished	IF_bm_to_osleft(0)	IF_taskleft_to_bmleft(0)	t_finished

Tabelle B.5: STI-Verbindungen Slot 0, Richtung links

OS Middle			Task Right
	IF_bmright_to_osmiddle(27)	IF_taskright_to_bmleft(27)	
	IF_bmright_to_osmiddle(26)	IF_taskright_to_bmleft(26)	
	IF_bmright_to_osmiddle(25)	IF_taskright_to_bmleft(25)	
t1_fifo2_sel(2)	IF_bmright_to_osmiddle(24)	IF_taskright_to_bmleft(24)	t_fifo2_sel(2)
t1_fifo2_sel(1)	IF_bmright_to_osmiddle(23)	IF_taskright_to_bmleft(23)	t_fifo2_sel(1)
t1_fifo2_sel(0)	IF_bmright_to_osmiddle(22)	IF_taskright_to_bmleft(22)	t_fifo2_sel(0)
t1_fifo2_dout(7)	IF_bmright_to_osmiddle(21)	IF_taskright_to_bmleft(21)	t_fifo2_dout(7)
t1_fifo2_dout(6)	IF_bmright_to_osmiddle(20)	IF_taskright_to_bmleft(20)	t_fifo2_dout(6)
t1_fifo2_dout(5)	IF_bmright_to_osmiddle(19)	IF_taskright_to_bmleft(19)	t_fifo2_dout(5)
t1_fifo2_dout(4)	IF_bmright_to_osmiddle(18)	IF_taskright_to_bmleft(18)	t_fifo2_dout(4)
t1_fifo2_dout(3)	IF_bmright_to_osmiddle(17)	IF_taskright_to_bmleft(17)	t_fifo2_dout(3)
t1_fifo2_dout(2)	IF_bmright_to_osmiddle(16)	IF_taskright_to_bmleft(16)	t_fifo2_dout(2)
t1_fifo2_dout(1)	IF_bmright_to_osmiddle(15)	IF_taskright_to_bmleft(15)	t_fifo2_dout(1)
t1_fifo2_dout(0)	IF_bmright_to_osmiddle(14)	IF_taskright_to_bmleft(14)	t_fifo2_dout(0)
t1_fifo2_we	IF_bmright_to_osmiddle(13)	IF_taskright_to_bmleft(13)	t_fifo2_we
t1_fifo1_re	IF_bmright_to_osmiddle(12)	IF_taskright_to_bmleft(12)	t_fifo1_re
t1_rs232_tx	IF_bmright_to_osmiddle(11)	IF_taskright_to_bmleft(11)	t_rs232_tx
t1_ledbar(9)	IF_bmright_to_osmiddle(10)	IF_taskright_to_bmleft(10)	t_ledbar(9)
t1_ledbar(8)	IF_bmright_to_osmiddle(9)	IF_taskright_to_bmleft(9)	t_ledbar(8)
t1_ledbar(7)	IF_bmright_to_osmiddle(8)	IF_taskright_to_bmleft(8)	t_ledbar(7)
t1_ledbar(6)	IF_bmright_to_osmiddle(7)	IF_taskright_to_bmleft(7)	t_ledbar(6)
t1_ledbar(5)	IF_bmright_to_osmiddle(6)	IF_taskright_to_bmleft(6)	t_ledbar(5)
t1_ledbar(4)	IF_bmright_to_osmiddle(5)	IF_taskright_to_bmleft(5)	t_ledbar(4)
t1_ledbar(3)	IF_bmright_to_osmiddle(4)	IF_taskright_to_bmleft(4)	t_ledbar(3)
t1_ledbar(2)	IF_bmright_to_osmiddle(3)	IF_taskright_to_bmleft(3)	t_ledbar(2)
t1_ledbar(1)	IF_bmright_to_osmiddle(2)	IF_taskright_to_bmleft(2)	t_ledbar(1)
t1_ledbar(0)	IF_bmright_to_osmiddle(1)	IF_taskright_to_bmleft(1)	t_ledbar(0)
t1_finished	IF_bmright_to_osmiddle(0)	IF_taskright_to_bmleft(0)	t_finished

Tabelle B.6: STI-Verbindungen Slot 1, Richtung links

Anhang C

Modular Design How-To

In diesem How-To sollen in kompakter Form die notwendigen Schritte des Modular Design Flows für partielle Rekonfiguration gemäss Xilinx Application Note 290 [24] wiedergegeben werden. Zu Beginn sollte die in Abbildung C.1 dargestellte Verzeichnisstruktur erzeugt werden.

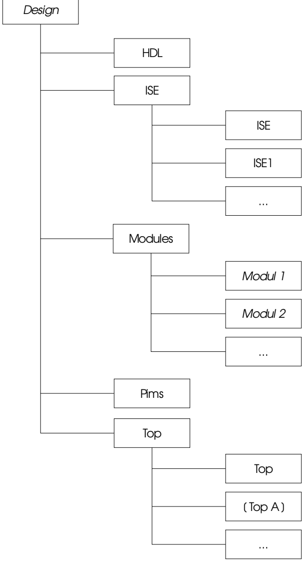


Abbildung C.1: Verzeichnisstruktur

C.1 Design Entry / Synthesis Process

1. VHDL-Code in HDL-Verzeichnis ablegen

2. *Top*-Synthese

   (a) Neues Projekt im Verzeichnis ISE/ISE erstellen

   (b) *Add Source* nur für Top.vhd ausführen

   (c) Im Menu *Edit→Preferences...* bei *Processes* den *Property Display Level* auf *Advanced* setzen

   (d) In den Synthesize-Properties unter Synthesis Options den *Bus Delimiter* auf () einstellen und unter Xilinx Specific Options den Punkt *Add I/O Buffers* aktivieren

   (e) Synthesize ausführen

3. Modul-Synthese

   - Erste Möglichkeit :
     (a) Für jedes Modul n ein neues Projekt im Verzeichnis ISE/ISEn erstellen
     (b) In jedem Projekt n *Add Source* für Modul$n$.vhd ausführen
     (c) In den Synthesize-Properties unter Xilinx Specific Options den Punkt *Add I/O Buffers* deaktivieren
     (d) Synthesize ausführen
   - Zweite Möglichkeit :
     (a) Ursprüngliches Projekt im Verzeichnis ISE/ISE öffnen.
     (b) In den Synthesize-Properties unter Xilinx Specific Options den Punkt *Add I/O Buffers* deaktivieren.
     (c) Das gewünschte Modul hinzufügen : *Add Source* Modul$n$.vhd
     (d) Synthesize ausführen.
     (e) Das Modul wieder entfernen : Im Fenster *Sources in Project* das entsprechende Modul mit der anklicken. Delete-Taste drücken.
     (f) Die weiteren Module durch wiederholen dieser Schritte synthetisieren. Bei erneuter *Top*-Synthese : Alle Module entfernen und *Add I/O Buffers* wieder aktivieren.

Als Ergebnis erhält man die Netzliste Top.ngc, sowie für jedes Modul Modul$n$.ngc im ISE-Verzeichnis bzw. im entsprechenden ISEn-Verzeichnis.

C.2 Initial Budgeting Phase

1. Top.ngc und Bus-Makro bm_4b.nmc ins Verzeichnis Top/Initial kopieren

2. `ngdbuild -p xcv800-hq240-5 -modular initial Top.ngc` ausführen.

3. Timing-Constraints definieren mit : `constraints_editor Top.ngd`. Die Constraints in Top.ucf speichern.

4. Flächen-Constraints für die einzelnen Komponenten und die Pinbelegung[1] ebenfalls in Top.ucf speichern mit `floorplanner Top.ngd`

5. Top.ucf mit Editor öffnen

   - Überprüfen, ob die Flächen-Constraints für die Komponenten jeweils ein Vielfaches von vier Slices breit sind und die Grenzen auf einer Spalte liegen, die eine ungerade CLB-Spaltennummer trägt.
   - Für jede Komponente die zusätzliche Beschränkung AREA_GROUP "component$n$"ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE; einfügen.
   - Bus-Makros einfügen, z.B. : INST "bm"LOC = "TBUF_R28C41.0";. Dabei wird die Spaltenposition durch Spaltennummer der Modulgrenze - 4 berechnet (Bei diesem Beispiel wäre die Modulgrenze bei C45).

C.3 Active Module Phase

1. Top.ucf aus Top/Initial-Verzeichnis in jedes Modulverzeichnis /Modules/Modul$n$ kopieren.

2. Die Modulnetzlisten Modul$n$.ngc in die entsprechenden Verzeichnisse kopieren.

3. Nun folgende Befehle ausführen :

   (a) `ngdbuild -p xcv800-hq240-5 -modular module -active Moduln ../../Top/Initial/Top.ngc`

   (b) `map -pr b Top.ngd -o Top_map.ncd Top.pcf`

[1] Dies kann auch schon im letzten Schritt, im Constraints-Editor erfolgen.

(c) `par -w -ol 5 -n 1 -s 1 Top_map.ncd mppr.dir Top.pcf`

(d) `copy mppr.dir\5_5_1.ncd Top.ncd`

(e) `copy ..\..\bitgen_jtag.ut .`

(f) `bitgen -d -f bitgen_jtag.ut -g ActiveReconfig:yes Top.ncd`

(g) `trce Top.ncd Top.pcf`

(h) Wenn das Modul in der folgenden Final Assembly Phase in den Bitstream der Startkonfiguration eingebunden werden soll :
`pimcreate -ncd Top.ncd -ngm Top_map.ngm ../../Pims`
Wird nur der partielle Bitstream des Moduls benötigt, kann dieser Befehl weggelassen werden.

In jedem Modulverzeichnis befindet sich nun ein partieller Bitstream mit dem Namen top.bit. Zur Vermeidung von Missverständnissen sollte er in Modul$n$.bit geändert werden.

C.4 Final Assembly

1. Top.ucf, Top.ngc (Startkonfiguration) und bm_4b.nmc nach Top\Assemble kopieren.

2. Folgende Befehle ausführen :

   (a) `ngdbuild -p xcv800-hq240-5 -modular assemble -pimpath ../../Pims Top.ngc`

   (b) `map -pr b Top.ngd -o Top_map.ncd Top.pcf`

   (c) `par -w Top_map.ncd Top.ncd Top.pcf`

   (d) `copy ..\..\bitgen_par.ut .`

   (e) `bitgen -f bitgen_par.ut Top.ncd`

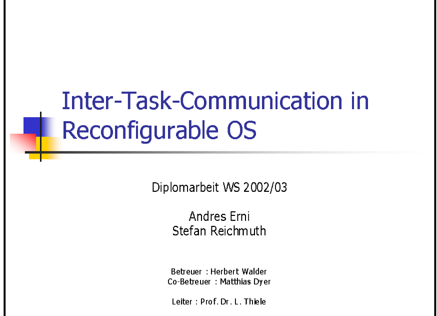
   (f) `trce Top.ncd Top.pcf`

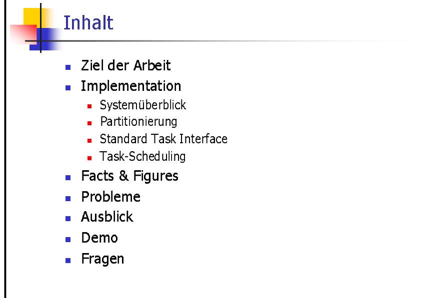
Nach diesem Schritt erhält man einen kompletten Bitstream für die Startkonfiguaration.

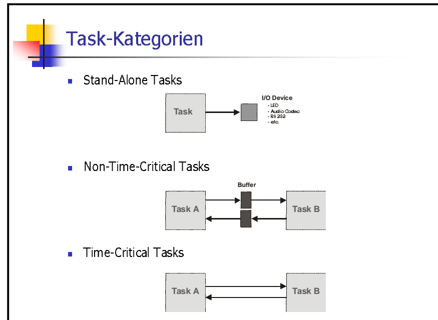
[1] Mit der Option `n` wird die Anzahl an Iterationen festgelegt. Um eine möglichst kurze Ausführungszeit zu erreichen, wird hier der Wert 1 verwendet. Bei kritischen Designs sollte jedoch ein grösserer Wert benutzt werden.

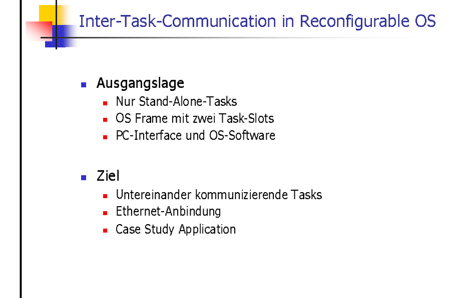
Anhang D

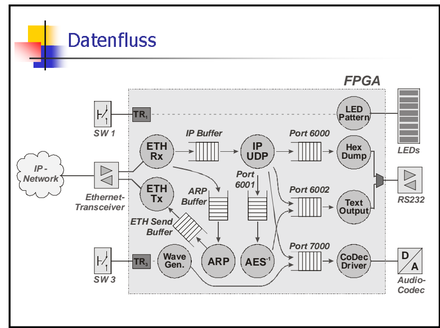
Präsentation

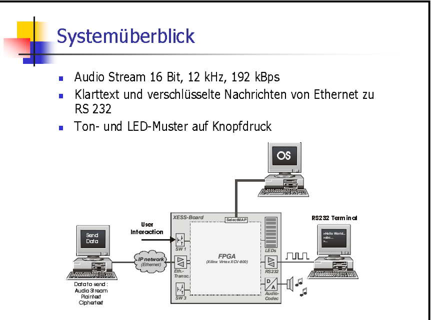

Inter-Task-Communication in Reconfigurable OS
Diplomarbeit WS 2002/03
Andres Erni
Stefan Reichmuth
Betreuer : Herbert Walder
Co-Betreuer : Matthias Dyer
Leiter : Prof. Dr. L. Thiele

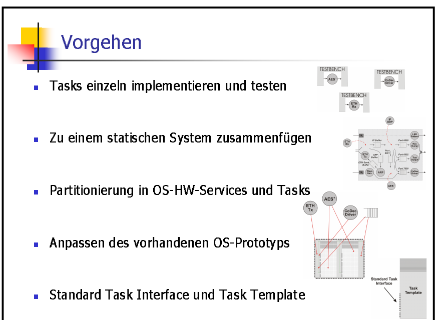

Inhalt
Ziel der Arbeit
Implementation
Systemüberblick
Partitionierung
Standard Task Interface
Task-Scheduling
Facts & Figures
Probleme
Ausblick
Demo
Fragen

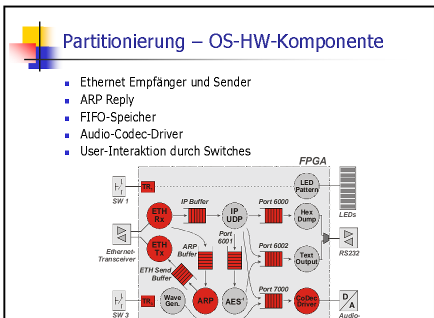

Task-Kategorien
Stand-Alone Tasks
Task
I/O Device
Non-Time-Critical Tasks
Buffer
Task A
Task B
Time-Critical Tasks
Task A
Task B

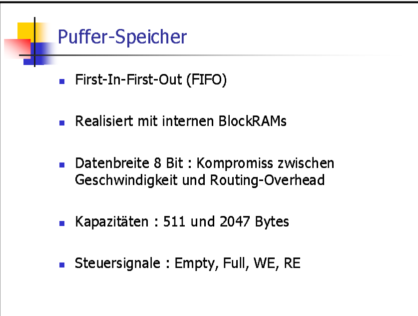

Inter-Task-Communication in Reconfigurable OS
Ausgangslage
Nur Stand-Alone-Tasks
OS Frame mit zwei Task-Slots
PC-Interface und OS-Software
Ziel
Untereinander kommunizierende Tasks
Ethernet-Anbindung
Case Study Application

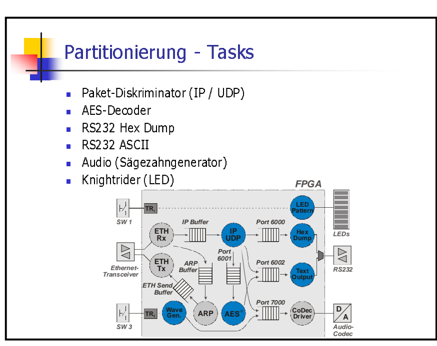

Datenfluss
FPGA
SW 1
LED Pattern
LEDs
IP Buffer
Port 6000
ETH Rx
IP UDP
Hex Dump
IP - Network
Ethernet-Transceiver
Port 6001
ETH Tx
ARP Buffer
Port 6002
Text Output
RS232
ETH Send Buffer
Port 7000
SW 3
Wave Gen.
ARP
AES
CoDec Driver
Audio-Codec

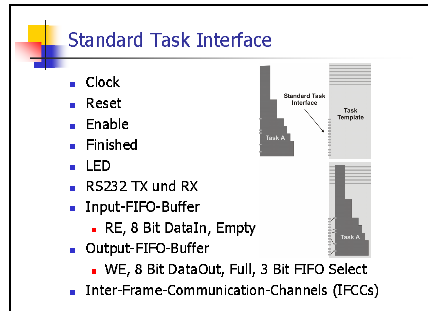

Systemüberblick
Audio Stream 16 Bit, 12 kHz, 192 kBps
Klarttext und verschlüsselte Nachrichten von Ethernet zu RS 232
Ton- und LED-Muster auf Knopfdruck
OS
User Interaction
RS232 Terminal
FPGA

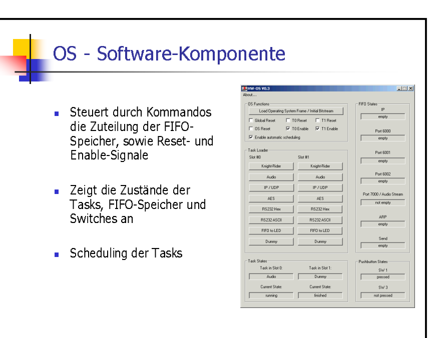

Vorgehen
Tasks einzeln implementieren und testen
Zu einem statischen System zusammenfügen
Partitionierung in OS-HW-Services und Tasks
Anpassen des vorhandenen OS-Prototyps
Standard Task Interface und Task Template

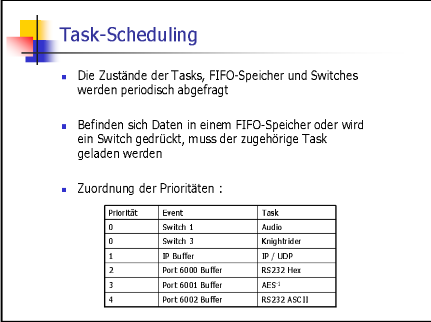

Partitionierung – OS-HW-Komponente
Ethernet Empfänger und Sender
ARP Reply
FIFO-Speicher
Audio-Codec-Driver
User-Interaktion durch Switches
FPGA
LED Pattern
LEDs
IP Buffer
Port 6000
ETH Rx
IP UDP
Hex Dump
Port 6001
Port 6002
ETH Tx
ARP Buffer
Text Output
RS232
Ethernet-Transceiver
ETH Send Buffer
Port 7000
Wave Gen.
ARP
AES
CoDec Driver
Audio-Codec
SW 1
SW 3


Puffer-Speicher
First-In-First-Out (FIFO)
Realisiert mit internen BlockRAMs
Datenbreite 8 Bit : Kompromiss zwischen Geschwindigkeit und Routing-Overhead
Kapazitäten : 511 und 2047 Bytes
Steuersignale : Empty, Full, WE, RE

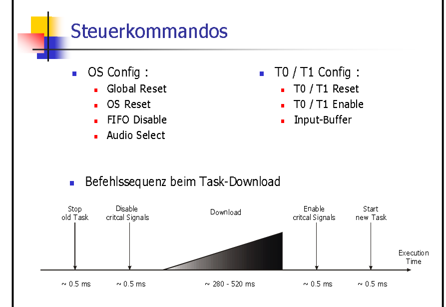

Partitionierung - Tasks
Paket-Diskriminator (IP / UDP)
AES-Decoder
RS232 Hex Dump
RS232 ASCII
Audio (Sägezahngenerator)
Knightrider (LED)
FPGA
LED Pattern
LEDs
IP Buffer
Port 6000
ETH Rx
IP UDP
Hex Dump
Port 6001
Port 6002
ETH Tx
ARP Buffer
Text Output
RS232
Ethernet-Transceiver
ETH Send Buffer
Port 7000
Wave Gen.
ARP
AES
CoDec Driver
Audio-Codec
SW 1
SW 3

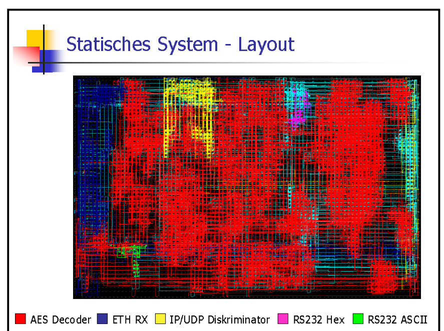

Standard Task Interface
Clock
Reset
Enable
Finished
LED
RS232 TX und RX
Input-FIFO-Buffer
RE, 8 Bit DataIn, Empty
Output-FIFO-Buffer
WE, 8 Bit DataOut, Full, 3 Bit FIFO Select
Inter-Frame-Communication-Channels (IFCCs)
Standard Task Interface
Task Template
Task A

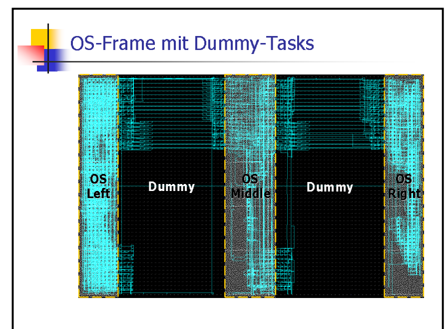

OS - Software-Komponente
Steuert durch Kommandos die Zuteilung der FIFO-Speicher, sowie Reset- und Enable-Signale
Zeigt die Zustände der Tasks, FIFO-Speicher und Switches an
Scheduling der Tasks

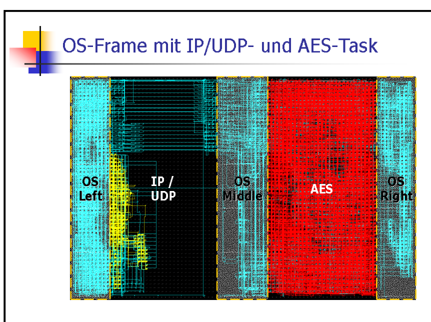


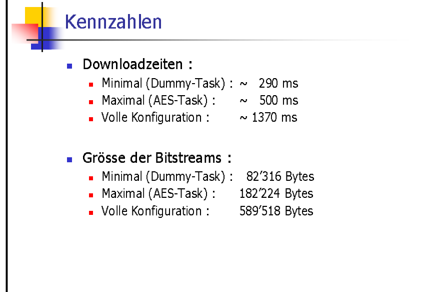
Priorität	Event	Task
0	Switch 1	Audio
0	Switch 3	Knightrider
1	IP Buffer	IP / UDP
2	Port 6000 Buffer	RS232 Hex
3	Port 6001 Buffer	AES^{-1}
4	Port 6002 Buffer	RS232 ASCII





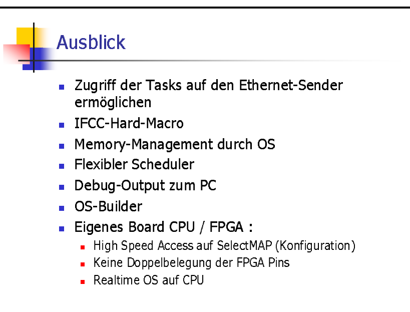


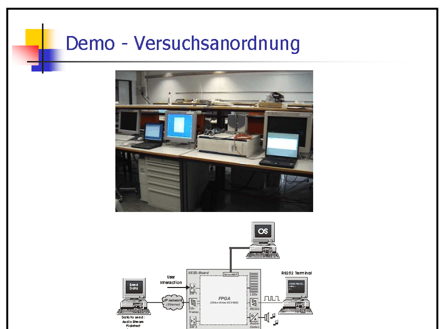


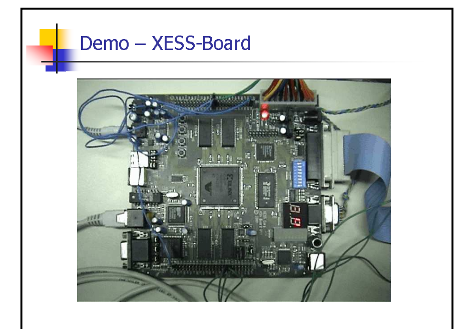


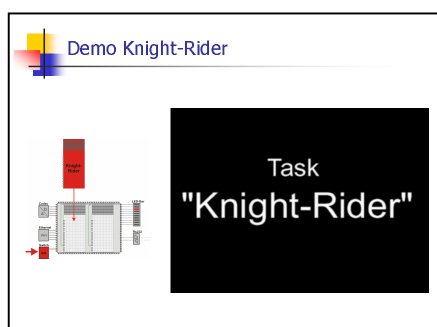
1

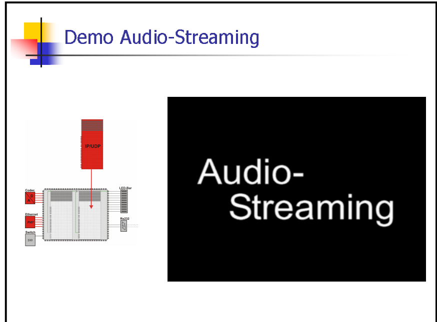
Probleme
Kritische Signale werden während des Downloads unterbrochen
Momentane Lösung : "Freezing"
Besser mit Hard-Macro ?
Einschränkungen durch XESS-Board :
Pins sind z.T schon belegt / überbelegt
Ressourcen die nicht direkt an OS Frame grenzen, können nicht benutzt werden
Windows ist kein Echtzeit-Betriebssystem

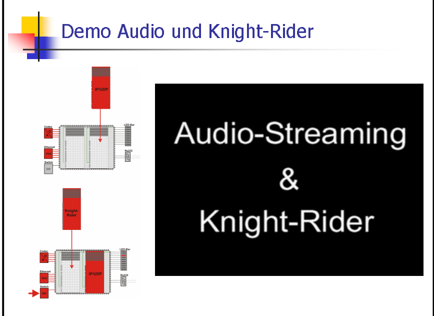

Ausblick
Zugriff der Tasks auf den Ethernet-Sender ermöglichen
IFCC-Hard-Macro
Memory-Management durch OS
Flexibler Scheduler
Debug-Output zum PC
OS-Builder
Eigenes Board CPU / FPGA :
High Speed Access auf SelectMAP (Konfiguration)
Keine Doppelbelegung der FPGA Pins
Realtime OS auf CPU


Demo - Versuchsanordnung
OS


Demo – XESS-Board


Demo Knight-Rider
Task
"Knight-Rider"


Demo Audio-Streaming
Audio-
Streaming


Demo Audio und Knight-Rider
Audio-Streaming
&
Knight-Rider

Anhang E

CD-Inhalt

Die Struktur der wichtigsten Verzeichnisse der CD zu dieser Diplomarbeit zeigt die Abbildung E.1. Zur besseren Übersicht wurde tieferliegende Unterverzeichnisse weggelassen.

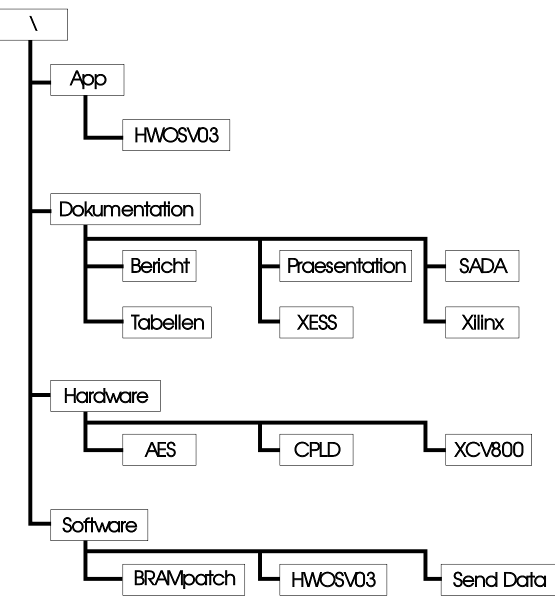


Abbildung E.1: Verzeichnisstruktur der CD

Zur Erläuterung :

- `App` : Hier befinden sich die Dateien, die zu einer Demonstration der entwickelten Applikation nötig sind. Es sind dies einerseits das CPLD-File, eine Audio-Datei (`Test.raw`) und die Applikation zum Versenden von Audio-

oder Text-Daten. Andererseits ein Unterverzeichnis `HWOSV03` mit der Software-Komponente des OS (`PCI7200T1.exe`) und den von ihr benötigten Bitstreams in weiteren Unterverzeichnissen.

- `Dokumentation` : Hier befinden sich Unterverzeichnisse mit folgendem Inhalt :
  - `Bericht` : Dieser Bericht in PDF-Format, sowie die Latex-Source-Files und Corel-Draw-Bilder.
  - `Praesentation` : Die Schlusspräsentation, inklusive Bildern und Filmen.
  - `SADA` : Zwei Berichte von Diplomarbeiten, die sich ebenfalls mit partieller Rekonfiguration beschäftigt haben ([3] und [8]). Zudem der Bericht zu unserer Semesterarbeit (AES-ASIC-Implementation) [4].
  - `Tabellen` : Die Tabellen aus Anhang B und zu den Steuerkommandos (Abschnitt 3.6) als Excel-Dateien.
  - `XESS` : Die Datenblätter zu den XESS-Board-Ressourcen, sowie das *XSV Board V1.1 Manual* [19].
  - `Xilinx` : Die *Development System Reference Guides* Versionen 4 und 5 (entsprechend der Modular-Design-Version und der Version der Xilinx Tools) und für diese Arbeit wichtige Application Notes ([20], [21], [22], [23], [24] und [25]).
- `Hardware` : Im Verzeichnis `AES` befindet sich der VHDL-Code, für die zwei in unserer Semesterarbeit entwickelten AES-ASICs (`riddler` und `joker`, vgl. [4]). Das Verzeichnis `XCV800` enthält die komplette Verzeichnisstruktur, die wir beim Modular Design Flow benutzt haben. Im Verzeichnis `hdl` befindet sich der VHDL-Sourcecode. Der Code für die Tasks befindet sich dabei im gleichnamigen Unterverzeichnis, mit der Ausnahme des AES^{-1}-Tasks, der sich im `AESDec`-Verzeichnis befindet. Die beiden Task-Templates (`TaskTemplate_left.vhd`, `TaskTemplate_right.vhd`) befinden sich ebenfalls im `Tasks`-Verzeichnis. Die Datei `make.bat` führt den vollständigen Design Flow durch, d.h. vom VHDL-Code bis zu den partiellen Bitstreams und der Startkonfiguration (`osframe.bit`). Die Bitstreams befinden sich am Ende im Verzeichnis `bitstreams`, das File (`osframe.bit`) in `\top\assemble`. Die Ausführungszeit des vollständigen Skripts betrug auf unserem PC (1000 Mhz Pentium III) ca. *drei* Stunden ! Im `CPLD`-Verzeichnis befindet sich schliesslich der Code, für den zur Konfiguration benötigten XC95108-Baustein auf dem XESS-Board.
- `Software` : Hier befinden sich die Verzeichnisse mit den Projekt-Dateien für das HW-OS (`HWOSV03` und das Programm *Send Data*). Zudem ist auch der

Sourcecode zum “patchen” des BlockRAM-Bugs (vgl. Abschnitt 7.1) hier abgelegt (`BRAMpatch`).

Literaturverzeichnis

[1] AKM Semiconductor Inc.: *Datasheet for the AK4520A stereo audio codec.* http://www.xess.com/manuals/AK4520A.pdf

[2] J. Daemen, V. Rijmen: *AES Proposal : Rijndael.* AES Algorithm Submission, 1999

[3] Matthias Dyer, Marco Wirz: *Reconfigurable System on FPGA.* Diplomarbeit ETH Zürich, 2002

[4] Andres Erni, Adrian Lutz, Stefan Reichmuth: *Rijndael Crypto Chip Implementation.* Semesterarbeit ETH Zürich, 2002

[5] Intel Corporation: *LXT970A - Dual-Speed Fast Ethernet Transceiver.* Datasheet, Intel, Januar 15, 2001. http://www.intel.com/design/network/products/LAN/docs/LXT970A_docs.htm

[6] H. Kaeslin, *VLSI I: Lecture notes on Digital VLSI and FPGA Architectures.* Microelectronics Design Center ETH Zürich, 2001

[7] H. Kaeslin, *VLSI II: Lecture notes on Design of Very Large Scale Integration Circuits.* Microelectronics Design Center ETH Zürich, 2001

[8] Michael Lerjen, Christian Zbinden: *Reconfigurable Bluetooth Ethernet Bridge.* Diplomarbeit ETH Zürich, 2002

[9] A. K. Lutz, J. Treichler, F. K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, W. Fichtner: *2 Gbit/s Hardware Realizations of RIJNDAEL and SERPENT: A comparative analysis*, Cryptographic Hardware and Embedded Systems - CHES 2002 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002 Revised Papers, Springer. pp. 144-158.

[10] National Institute of Standards and Technology (NIST): *Federal Information Processing Standards (FIPS) Publication 197: Announcing the Advanced Encryption Standard (AES).* 2001

[11] L. Peterson, B. Davie: *Computer Networks.* San Francisco: Morgan Kaufmann, 2000

[12] J. Postel: *Internet Protocol*, RFC 791, Information Sciences Institute, University of Southern California, Marina del Rey, September 1981.
ftp://ftp.rfc-editor.org/in-notes/rfc791.txt

[13] J. Postel: *Internet Control Message Protocol*, RFC 792, Information Sciences Institute, University of Southern California, Marina del Rey, September 1981.
ftp://ftp.rfc-editor.org/in-notes/rfc792.txt

[14] J. Postel: *User Datagram Protocol*, RFC 768, Information Sciences Institute, University of Southern California, Marina del Rey, August 1980.
ftp://ftp.rfc-editor.org/in-notes/rfc768.txt

[15] David C. Plummer : *An Ethernet Address Resolution Protocol*, RFC 826, MIT, November 1982.
ftp://ftp.rfc-editor.org/in-notes/rfc826.txt

[16] University of Queensland, Brisbane: *VHDL XSV Board Interface Projects.*
http://www.itee.uq.edu.au/ peters/xsvboard/

[17] Michael Ruppen: *Reconfigurable OS Prototype.* Diplomarbeit ETH Zürich, 2003

[18] A. Tanenbaum: *Computernetzwerke.* New Jersey: Prentice-Hall, 1996

[19] X Engineering Software Systems Corporation: *XSV Board V1.1 Manual.*
XESS, Apex NC, 2001

[20] Xilinx Inc.: *Using the Virtex Block SelectRAM+ Features.* Application Note, XAPP130 (v1.4), December 18, 2000.
http://support.xilinx.com/xapp/xapp130.pdf

[21] Xilinx Inc.: *170 MHz FIFOs Using the Virtex Block SelectRAM+ Feature.* Application Note, XAPP131 (v1.4), June 5, 2001.
http://support.xilinx.com/xapp/xapp131.pdf

[22] Xilinx Inc.: *Virtex Configuration and Readback.* Application Note, XAPP138 (v2.7), November 11, 2002.
http://support.xilinx.com/xapp/xapp138.pdf

[23] Xilinx Inc.: *Virtex Series Configuration Architecture User Guide.* Application Note, XAPP151 (v1.5), September 27, 2000.
http://support.xilinx.com/xapp/xapp151.pdf

[24] Xilinx Inc.: *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations.* Application Note, XAPP290 (v1.0), May 17, 2002.
http://support.xilinx.com/xapp/xapp290.pdf

[25] Xilinx Inc.: *Xilinx Alliance 3.1i Modular Design.* Application Note, XAPP404 (v1.2), April 20, 2001.

[26] Xilinx Inc.: *Development System Reference Guide - ISE 4.*
Xilinx, San Jose CA, 2001.

[27] Xilinx Inc.: *Development System Reference Guide - ISE 5.*
Xilinx, San Jose CA, 2002.