

# **VLSI Design**

## **of a simple MP3 player using VHDL**

**Prof. Dr.-Ing. Ulf Schlichtmann**

**Oct. 2008**

**Institute for Electronic Design Automation  
Department of Electrical Engineering  
Technische Universitaet Muenchen  
Univ.-Prof. Dr.-Ing. Ulf Schlichtmann  
Arcisstr. 21, 80333 Munich  
Phone: +49 89 289 23666  
Email: eda@ei.tum.de**

Copyright © by Institute for Electronic Design Automation, Technische Universitaet Muenchen, 2008.  
No part of this publication may be reproduced or distributed in any form without the prior written permission  
of the Institute for Electronic Design Automation, Technische Universitaet Muenchen.

---

*2005-current* Bing Li

*2008-current* QingQing Chen

---

## Contents

<b>0. README</b>	<b>5</b>
<b>1. Introduction</b>	<b>7</b>
1.1 VLSI Design Flow	7
1.2 VHDL	8
<b>2. Lab Overview</b>	<b>9</b>
2.1 System Structure	9
2.2 Hardware	10
<b>3. Module Specifications</b>	<b>12</b>
3.1 KBC Specification	12
3.2 FIO Specification	12
3.3 LCDC Specification	14
3.4 Decoder Specification	17
<b>4. VHDL Design Guidelines</b>	<b>21</b>
<b>5. Displaying File Names</b>	<b>24</b>
5.1 KBC Interface	24
5.2 Reading File Information	24
5.3 Arbiter&Multiplexer	26
5.4 Receiving File Information	29
5.5 Building the List Function	31
<b>6. Making a Simple MP3 Player</b>	<b>34</b>
6.1 Playing the Music	34
6.2 Customizing the MP3 Player	35
<b>7. Implementing the MP3 System</b>	<b>36</b>
7.1 Simulating and Synthesizing the PLCN Module	36
7.2 Implementing the Complete MP3 System	37
7.3 Testing the MP3 Player	38

## Figures

<b>1 VLSI design flow</b>	<b>7</b>
<b>2 MP3 system structure</b>	<b>9</b>
<b>3 The Virtex-II Pro development board</b>	<b>10</b>
<b>4 The structure of the MP3 decoder</b>	<b>18</b>
<b>5 Key code comparator</b>	<b>24</b>
<b>6 Block diagram of a finite-state machine</b>	<b>25</b>

7	State diagram representation.....	25
8	Displaying file information state machine.....	26
9	The diagram of the arbiter and multiplexer .....	27
10	Arbiter structure.....	28
11	File name displaying.....	29
12	Top structure of the list function .....	33
13	Top structure of the PLCN module .....	34
14	Project implementation structure .....	36

## Tables

1	Abstraction levels and domains .....	8
2	Interface signals of the KBC module .....	12
3	Predefined scan codes.....	12
4	Scan codes of the keypad .....	13
5	Interface signals of the FIO module .....	13
6	The commands of the FIO module .....	14
7	File information format of the FAT16 file system .....	14
8	Interface signals of the LCDC module.....	15
9	The commands of the LCDC module.....	15
10	CCMD definition.....	16
11	ASCII table.....	18
12	Interface signals of the decoder module.....	19
13	Clock specifications .....	38

## 0 README

### 1. Login/logout the linux system

In this lab the work environment is linux system. To login the system, input the username and password in the login screen. The username and password are assigned by the tutor after the lab introduction. To logout the system, click the first icon on the left side of the toolbar and select *Log Out*, then select *End Current Session*. **Note:** Please do not shutdown or reboot the computer anytime, because other users may also work on the computer through remote login.

### 2. Documents

All documents can be found online at: <http://eda.ei.tum.de/lehrangebot/L-VLSI/index.html#doc>. Additionally an FAQ can be found at: <http://eda.ei.tum.de/lehrangebot/L-VLSI/index.html#faq>.

In Section 1 of the lab manual a short introduction about digital design flow and VHDL is given. In Section 2 the overview about this lab is presented. The specifications of the given modules are listed in Section 3. In Section 4 some VHDL guidelines can be found, which are useful when describing RTL circuits using VHDL. The design tasks of this lab are listed in Section 5 and 6. Finally the instructions for simulating the *playcontrol* module and implementing the MP3 design are given in section 7.

### 3. Project

The MP3 project is in the *\$HOME/prj* directory. The work directory for the playcontrol module is *\$HOME/prj/playcontrol*. The entity for the playcontrol module is provided in *playcontrol.vhd* and a simple testbench for the playcontrol module is also given with the name *playcontrol\_tb.vhd*. The top level design of the MP3 system and the download command are in the *\$HOME/prj/top\_system* directory. To submit the lab results, all the finished VHDL code should be saved in the *\$HOME/prj/playcontrol* directory, which will be checked just before the final exam. The implemented MP3 project including all intermediate files should be kept for demonstration.

### 4. Design tools and hardware

The VHDL simulator used in this lab is ModelSim; The FPGA implementation tool is Xilinx ISE. To run ModelSim and ISE, input *vsim&* and *ise&* in the terminal window respectively (& makes the program run at the background so that the current terminal window can still accept new commands). To download the finished design, change to *\$HOME/prj/top\_system* directory and issue the *MP3download* command. The Xilinx VirtexII Pro development boards in the lab room 2961 can be used to test the FPGA design. **Note:** Please do not take off the compact flash card from the slot anytime. This may damage the board or the card slot.

### 5. Recommended work schedule

week 1	introduction
week 2 ~ week 4	exercise 1 ~ exercise 6
week 5	test the list function in hardware
week 6 ~ week 7	exercise 7 ~ exercise 8
week 8	test the play function
week 9 ~ week 10	exercise 9
week 11 ~ week 12	test the complete system

### 6. Final exam

The final exam is an oral test and lasts for 20 Minutes. During the exam, the result of the MP3 design should firstly be demonstrated. Thereafter the requested state machines should be explained. Finally some questions about the state machines and VHDL will be asked.

The exact date of the exam will be announced during the semester per email and on the web page of this lab. Both the exam and design results contribute to the final grade and the participation in the final exam is mandatory for receiving course credits.

### 7. Basic linux commands

All the design tools should be invoked using terminal commands. To open a terminal window, right click with the mouse on the desktop, select *Konsole*. To run a linux command input the command in the

terminal window followed with the [Enter] key. The useful linux commands in this lab are listed below:

```
cd dir      change the current directory to dir. $HOME (equivalent to ~) is the default
              directory (the rsexx directory).
ls dir      list files in the dir directory. If dir is omitted, the current directory is listed.
cp src dst  copy the file src to the file dst.
mv src dst  rename/move the file src to the file dst.
rm file     delete file.
mkdir dir   create a new sub-directory in the current directory with the name dir.
```

## 8. OpenAFS usage

The *\$HOME* directory *rsexx* can be accessed outside of the institute. In the following are the instructions to install and configure the OpenAFS client.

### ■ Linux System

From <http://www.openafs.org/release/index.html> the rpm packages can be found for Fedora and RHEL. The *openafs-version.rpm* and *openafs-client-version.rpm* should be installed. Then the kernel module package *kmode-openafs-version.rpm* corresponding to the system kernel version should be installed. The configuration files are located in */usr/vice/etc*.

For configuration, input the following information to corresponding files.

SuidCells.local:

```
regent.e-technik.tu-muenchen.de
```

CellServDB.local:

```
129.187.230.2 #refi.regent.e-technik.tu-muenchen.de
129.187.230.9 #refile.regent.e-technik.tu-muenchen.de
```

CellAlias:

```
regent.e-technik.tu-muenchen.de regent
```

By issuing “klog rsexx -cell regent” the afs token can be fetched from the server. Thereafter, *\$HOME* can be accessed through */afs/regent/home/rsexx/rsexx* (*rsexx* is the login name).

For debian/ubuntu systems, the package “openafs-client” in the repository should be installed. To compile and install the kernel module, please refer */usr/doc/openafs-client/README.modules*.

### ■ Windows System

Download and install the OpenAFS client from <http://www.openafs.org/windows.html> Double-click the *AFS Client Configuration* icon in the system setting; The AFS Client Configuration utility opens, displaying the *General* tab; In the *Cell Name* box, enter the name of the AFS cell “regent.e-technik.tu-muenchen.de”; Select the *AFS Cells* tab; Select *Add* button. In the *AFS Cell* entry, input “regent.e-technik.tu-muenchen.de”; Select the *Add* button in this window; In the *Server Name* entry, input “refi.regent.e-technik.tu-muenchen.de”; Click the *OK button*; Click the *Add* button, input “refile.regent.e-technik.tu-muenchen.de”; Then click the *OK* button;

To access *\$HOME* directory, run the openafs-client program. In the *tokens* window input you login name and password to get a token from the server. In the *Drive Letters* window add your directory as “*/afs/regent/home/rsexx/rsexx*”. You can access your *\$HOME* directory through the selected drive letter.

## 1 Introduction

Traditionally integrated circuits have been developed using schematics. With shrinking silicon structures and higher integration densities, automated tools for Electronic Design Automation (EDA) have been developed. Growing complexities of very large scale integrated circuits (VLSI) and time-to-market pressures resulted in research effort for computer aided design, which led to new, more efficient design methods. In this section the flow of the current VLSI design will be sketched and the history and features of VHDL (VHSIC<sup>†</sup> Hardware Description Language), the hardware design language used in this lab, will be briefly introduced.

### 1.1 VLSI Design Flow

Developing technical products typically consists of a sequence of construction (synthesis) and validation (analysis) steps. This also applies to the design of VLSI circuits. Figure 1 shows a typical VLSI design flow starting at **system level** with a specification of the system behavior and the interfaces to its operating environment. Through several abstraction levels, this initial specification is gradually refined until a detailed description (mask data) necessary for fabrication of the integrated circuit is obtained.

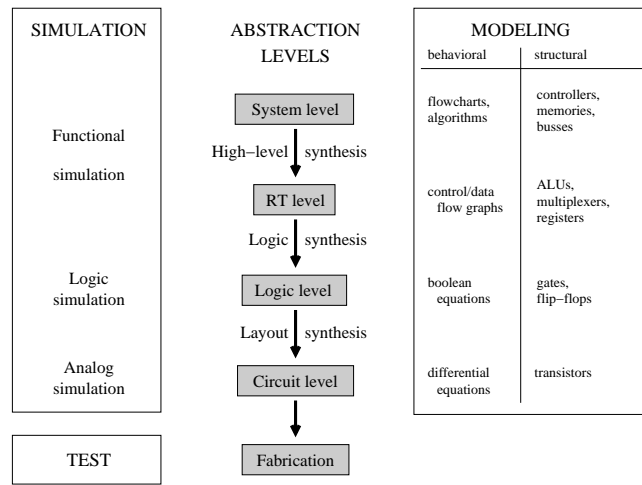


Figure 1: VLSI design flow

**High level synthesis** transforms the system level description to a **register transfer level (RTL)** description. Typically a controller part for providing the correct sequence of actions, and a dataflow part for executing arithmetic operations are generated. On system and on RTL level the function of the circuit can be simulated, but there is not enough information yet for doing a detailed timing simulation.

Through **logic synthesis** and logic optimization, an RTL description is refined to **logic level**. Boolean equations or gates and flip-flops provide a detailed logic description enabling also timing simulation. In the last step of logic synthesis, the circuit description is mapped to a technology specific library of circuit components (technology mapping). On logic level it is very important to validate the design by **logic simulation** of function and timing.

**Layout synthesis** calculates the positions of the circuit elements on the chip and generates geometrical layout data at **circuit level**. Here the elements are assigned places on the chip by placement and the interconnect between these elements is realized by routing wires between them.

After logic synthesis the circuit elements and their physical implementation are known. Layout synthesis adds information about the wiring, which is very important for exact timing analysis as wire delay dominates component delay in advanced process technologies. Finally, the implementation of the circuit on transistor level is given and a precise **analog simulation** is possible.

When the function and the timing of the placed and routed circuit have been validated and the constraints are met, the design is handed over to a semiconductor foundry for **fabrication**.

<sup>†</sup> VHSIC = Very High Speed Integrated Circuit

## 1.2 VHDL

In recent years, VHDL has emerged as one of the most important electronic design languages. VHDL was originally proposed by the U.S. Department of Defense (DoD) in order to provide a means for documentation and simulation of military electronic components. In 1987, it was adopted by IEEE as a standard (IEEE-1076-1987).

By utilizing VHDL as a specification language it is possible to start simulation of complex systems before their implementations are fully specified. Furthermore, VHDL facilitates the top-down design process where a higher level specification is developed, debugged and finally used to validate the correctness of the next lower level implementation. Through supporting mixing different abstract levels during simulation, VHDL makes it possible to adopt also the bottom-up or mixed design styles.

VHDL is derived from the Ada programming language. Consequently, it provides the typical means of software languages for describing functionality and control constructs. Additionally, it incorporates the necessary constructs for modeling hardware components, e.g. methods for describing concurrency and timing.

In a typical VLSI design environment, VHDL can be used in three distinct fashions employing various levels of abstraction and modeling styles:

- High-level specification
- Logic/standard component level design
- Standard component model library support

During the high-level specification phase, VHDL is used as an architectural tool to aid analysis and evaluation of design alternatives. At this level, virtually all language features of VHDL are used, especially abstraction mechanisms (e.g. user defined data types). This stage of the design is often performed without knowledge or concern on specific implementation details.

At the logic/standard component level, the actual implementation of the design is determined. The design is implemented as a structural composition of predefined (or previously developed) standard components. In this design stage, typically only a subset of VHDL which is synthesizable by commercial logic synthesis tools is used. When a synthesis tool meets a high level modeling description written in VHDL, e.g. the *AFTER* statement, it may simply ignore this description. As a result, the mismatch between the high level simulation and the behavior of the synthesized circuit may probably happen. Therefore when describing circuits using VHDL on the RTL level, many of the advanced features of VHDL should be carefully avoided.

Standard component libraries are precompiled and provided to the design team. These libraries are usually targeted to a specific semiconductor technology. and are usually highly parameterizable so that can be reused in different circuit designs.

In Table 1 a more detailed overview of the application of VHDL on the different levels in the VLSI design flow with respect to the individual description domains is given.

Level	Behavior	Structure	Data	Timing
System level	Algorithms	Processes	Abstract data types	Causality
Register-transfer level (RTL)	Dataflow, Finite state machines	Register, ALU	Bitvectors	Clock cycles
Logic level	Boolean equations	Gates, Flipflops	Bits	Delays

Table 1: Abstraction levels and domains

VHDL is already supported by all major EDA tool vendors and accepted as an industry standard. Using VHDL as a standard circuit description language, design tools from different vendors can be easily integrated into one environment and it is also easier to migrate to new technologies and different foundries. Furthermore VHDL provides both an executable specification with well-defined simulation semantics and a human readable documentation of a design. Finally, with its flexibility to parameterize design models (macros) with generic variables, modification of existing designs and reuse of previously developed components in VHDL format are easy.



## 2 Lab Overview

In this section the content and task structure of this lab will be introduced. The main content of this lab is to design a simple MP3 player. The first part of the tasks is to design the playcontrol (PLCN) sub module for the MP3 player using VHDL. Then the top level FPGA (Field Programmable Gate Array) project will be created to integrated the PLCN with other given modules to form a functioning system. The complete FPGA design process, including timing constraints, pin assignment, synthesis, placement&routing, bitstream generation will be fulfilled. Finally, the MP3 design will be downloaded to hardware for testing.

### 2.1 System Structure

Figure 2 shows the structure of the MP3 player. The modules inside the dash box are implemented in the FPGA chip, and the modules outside are implemented by the other ASIC chips on the testing board.

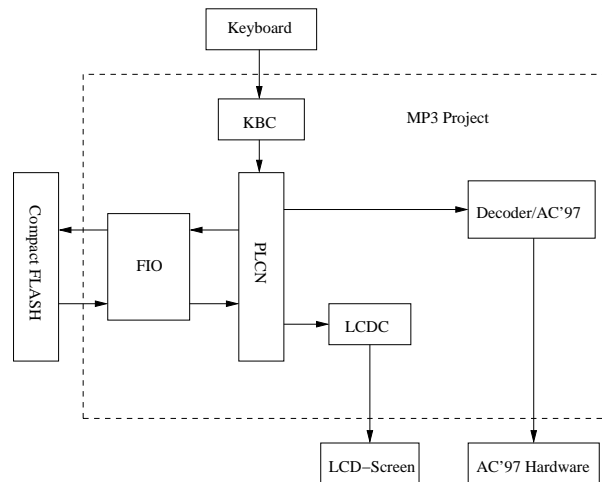


Figure 2: MP3 system structure

The functions of the modules inside the FPGA are listed below:

- FIO Module  
This module reads raw data saved in the Compact Flash (CF) card, parses the FAT16 file system and returns file data and file information, e.g. file name, file size etc..
- KBC module  
This module monitors the keypad connected to the FPGA chip through a PS/2 interface and sends corresponding key scan codes to the PLCN module whenever a key is pressed.
- Decoder&AC'97 Module  
This module receives and decodes MP3 data from the PLCN module. The decoded samples are saved in an internal buffer. These samples are fetched and played by the AC'97 hardware automatically. The AC'97 hardware can be directly controlled by sending hardware control commands. The status of the input MP3 data buffer (DBUF), the sample buffer (SBUF) and the decoder can be monitored through the interfacing signals.
- LCDC module  
This module controls the 16×2 character LCD. Besides displaying characters on fixed positions, this module provides flexible scroll functions.
- PLCN module  
This module integrates all the modules above together to form a functioning MP3 player. The main task of this lab is to design and test this module. The modules above (except PLCN) have been implemented and provided as netlist in the MP3 project. The functions of the given modules and their interfaces to the PLCN module are completely described in Section 3, which should be read thoroughly and carefully before starting the lab.

The lab is partitioned into the following steps:

1. Creating the PLCN FPGA project

This project is used to simulate and synthesize the sub module PLCN. Because this module will be integrated to the top level design, IO pads should not be implemented for interface signals during synthesis. Instructions for this step can be found in Section 7.

2. Displaying file information

The file information, including file name, access time, file size etc., is read from the FIO module by PLCN. Then the file name is written to the LCDC module, which displays the corresponding characters on the LCD screen. The key scan codes from the KBC module control the switching of the files to be listed. After finishing this step, the MP3 system can be tested by creating the *top\_system* project and implementing the design (step 5). After downloading the design into FPGA, the names of the files saved on the CF card can be browsed.

3. Playing the music

The PLCN module reads MP3 data periodically from the FIO module. These data are fed to the MP3 Decoder module, and the decoded PCM samples are written to SBUF for playing. The Start and Stop of the play process are controlled by pressing the keys on the keypad.

4. More control functions

Additional functions are implemented, which include Mute, Pause, Change volume etc..

5. Creating the top level design

The top project which integrates the sub modules is created in this step. Unlike the PLCN FPGA project, the top design should run through the complete FPGA design flow and generate the downloadable bitstream for testing. Instructions for creating the *top\_system* project can be found in Section 7.

6. System Test

In this step, the FPGA bitstream is downloaded into the FPGA chip to check if the system works. If there are problems, the VHDL code of the PLCN module and the project settings should be checked.

## 2.2 Hardware

In this lab the Xilinx Virtex-II Pro development board (Figure 3) is used for hardware testing. The FPGA on the hardware board is Virtex II-Pro XC2VP30-7FF896C. Two PowerPC CPU cores are integrated inside this FPGA, one of which runs the MP3 decoder migrated from MAD library.

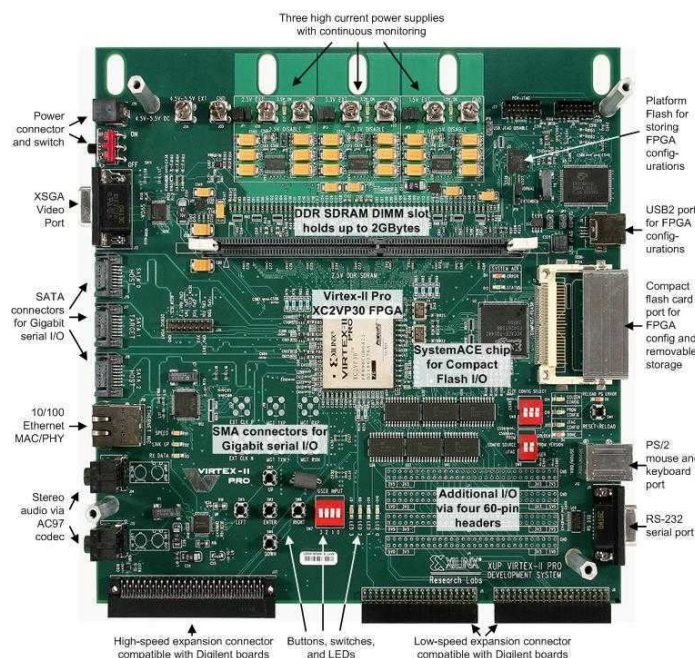


Figure 3: The Virtex-II Pro development board

In details, the hardware test environment contains:

- Xilinx Virtex-II Pro FPGA XC2VP30-7FF896C with 2448kbit Block RAM (BRAM), 30816 logic cells and two PowerPC 405 processors
- 256MB DDR266 SDRAM
- AC'97 audio chip (LM4550 from National Semiconductor)
- Xilinx System ACE configuration chip with 512MB Compact Flash memory card
- PS/2 controller with keypad
- A 16×2 character LCD
- Xilinx USB download cable

### 3 Module Specifications

In this section the interfaces of the given modules, including PS/2 keyboard controller (KBC), File I/O module (FIO), LCD controller (LCDC) and decoder (DEC) will be described. The PLCN module uses the functions of these given modules to form the MP3 system. Depending on the specific design, not all the signals in the following specifications are needed to implement the PLCN module in this lab. **Note: Each output signal in the view of the *playcontrol* module should have a driver in the *playcontrol* module. Otherwise, errors may occur during generating the bitstream for the *top\_system* module.**

#### 3.1 KBC Specification

The keyboard controller initializes and monitors the keys from the keyboard in the MP3 system. When a key is pressed its scan code is fetched and saved in an FIFO whose output is connected to the PLCN module. The interface signals of the KBC to the PLCN module are listed in Table 2.

Signal Name	Width	Direction (KBC view)
RD	1	input
RD_ACK	1	output
DATA	8	output
EMPTY	1	output

Table 2: Interface signals of the KBC module

- RD

When the RD signal is '1' at a rising clock edge, a scan code is read from the FIFO.

- RD\_ACK

In response to a valid RD signal (RD is '1' at a rising clock edge), the RD\_ACK signal switches to '1' after a definite number of clock periods. If the RD signal is '1' but the FIFO is empty, the KBC module returns no valid RD\_ACK.

- DATA

When RD\_ACK is '1', the data on the DATA bus is a valid scan code.

- EMPTY

The EMPTY signal shows the status of the key FIFO. When at least a scan code exists in the FIFO, EMPTY is '0', otherwise '1'.

Some scan codes listed in Table 3 are predefined as control keys of the MP3 player. Other useful scan codes can be found in Table 4, which can be used to implement new functions of the MP3 player.

Key	Scan Code	Key Name	Description
'8'	0x75	LISTPREV	list the previous file name.
'2'	0x72	LISTNEXT	list the next file name.
'Esc'	0x76	PLAY	start the music.
'Ctrl'	0x14	STOP	stop the music.
'Alt'	0x11	PAUSE	pause the music.
'←'	0x66	MUTE	mute but the play is still continuing.
'+'	0x79	INCVOL	increase the volume.
'-'	0x7B	DECVOL	decrease the volume.

Table 3: Predefined scan codes

**Be noted that this PS/2 controller is quite simple and does not support pressing the keypad without releasing keys for long time.**

#### 3.2 FIO Specification

The FIO module parses the FAT16 file system on the Compact Flash card and returns file information and MP3 data. The interface signals of the FIO module to the PLCN module are listed in Table 5.

Key	Scan Code	Key	Scan Code	Key	Scan Code	Key	Scan Code
'Esc'	0x76	'Ctrl'	0x14	'Alt'	0x11	'←'	0x66
'Num'	0x77	'÷'	0x4A	'×	0x7C	'—'	0x7B
'7'	0x6C	'8'	0x75	'9'	0x7D	'+'	0x79
'4'	0x6B	'5'	0x73	'6'	0x74		
'1'	0x69	'2'	0x72	'3'	0x7A	'Enter'	0x5A
'0'	0x70			'.'	0x71		

Table 4: Scan codes of the keypad

Signal Name	Width	Direction (FIO view)
BUSIV	1	in
CTRL	1	in
BUSI	8	in
BUSY	1	out
BUSO	32	out
BUSOV	1	out

Table 5: Interface signals of the FIO module

- BUSIV

BUSIV is the signal to define if the data on BUSI is valid. Only when BUSIV is '1' can the command/parameter of the BUSI signal be processed by the FIO module.

- CTRL

CTRL is the control signal of the data bus BUSI. When CTRL='1', the data on BUSI is a command. When CTRL='0', the data on BUSI is the parameter for the READ/FFSEEK/BFSEEK command.

- BUSI

BUSI is the input data bus to the FIO module. When the BUSIV signal is '1', the data on the BUSI signal is a valid command/parameter if CTRL is '1'/'0'. Six commands are defined for the FIO module, as listed in Table 6.

When CTRL='0', BUSI holds the data size parameter to FIO module. To specify the read data size (for READ command), the value 0-255 of the parameter designates the data size 1-256 DWORDs (Double Words, 32 bits) respectively. To specify the parameter for the FFSEEK/BFSEEK command, the value 0-255 designates the size 1-256 kDWORDs ( $2^{10}$  DWORDs). Before the READ/FFSEEK/BFSEEK command is sent, the parameter must be properly set.

- BUSY

BUSY shows the status of the FIO module. Only when BUSY is '0' can a command/parameter be sent to the FIO module.

- BUSOV

BUSOV is the valid signal for the output signal BUSO. Only when BUSOV='1' are the data of BUSO valid. Note that BUSOV may be continuous '1' when more than 1 DWORD data are returned, but the FIO module does not guarantee that the valid BUSOV (BUSOV='1') is always continuous when returning data.

- BUSO

BUSO returns the requested file information or file data. The FIO module returns 8 DWORDs of file information in response to the FILENEXT/FILEPREV command. The 32 bits file information is listed in Table 7. The file information DWORDs with lower index are returned before the DWORDs with higher index. The file information bytes with lower index are returned in the lower bytes of a DWORD. The returned MP3 data in response to the READ command are sequentially output from the BUSO bus. If the requested data goes beyond the end of the file, which means the file end is reached during a read command, the FIO module returns the valid data which are already read and packs the further data with unpredictable content so that the number of the returned DWORDs is always the same as requested. The file data DWORDs with lower index are returned before the DWORDs with higher index. The file data bytes with lower index are returned in the lower bytes of a DWORD.

Command Value	Command Name	Description
0x00	FILENEXT	Get next file information.
0x01	FILEPREV	Get previous file information.
0x02	READ	Read specified number of data DWORDs. Before this command is sent, PLCN must send the requested data size (max. 256 DWORDs).
0x03	OPEN	Close the previously opened file and open the file which is currently listed (on the LCD). Before the data of a file is read, the open command should be sent by PLCN.
0x04	FFSEEK	Forward move the read address of the current opened file. The parameter to the FIO module specifies the data size of the forward moving in range of 1 to 256 kDWORDs ( $2^{10}$ DWORDs). If the new read address goes beyond the end of the file, the current read address is set to the file end.
0x05	BFSEEK	Backward move the read address of the current opened file. The parameter to the FIO module specifies the data size of the backward moving in range of 1 to 256 kDWORDs ( $2^{10}$ DWORDs). If the new read address is less than the beginning of the current opened file, the current read address is set to the file beginning.
others		reserved, do not sent at any time.

Table 6: The commands of the FIO module

Whether the output data are file information or MP3 data should be decided by the PLCN itself. The output data should be counted so that the end of the transfer can be identified.

Bytes	Description
10:0	MS-DOS (8+3) filename, padded with spaces.
11	File attribute descriptor byte: Bit 7, 6: unused bits; Bit 5: archive bit: '1' if the listed file / directory can be archived; Bit 4: read-only bit: '1' if the listed file / directory is write protected; Bit 3: system bit: '1' if the listed FAT entry is a system file; Bit 2: hidden bit: '1' if the listed FAT entry is hidden; Bit 1: directory bit: '1' if the listed FAT entry is a directory; Bit 0: volume bit: '1' if the listed FAT entry is a volume label.
12	Reserved for Windows NT, meaningless here.
13	Millisecond portion of the create time.
15:14	Hour, minute and second portion of the create time.
17:16	Date portion of the create time.
19:18	Last access date.
21:20	Extended attribute, set to 0 for FAT16.
23:22	Hour, minute and second portion of the file modification time.
25:24	Date portion of the file modification time.
27:26	Cluster number.
31:28	File size in bytes.

Table 7: File information format of the FAT16 file system

### 3.3 LCDC Specification

The LCDC module accepts character codes and displays them on a 2 line×16 character LCD. This module provides character displaying and additional flexible screen scrolling functions. The LCDC module is

Signal Name	Width	Direction (LCDC view)
CMD	2	input
BUSY	1	output
CCRM_WDATA	36	input
CCRM_ADDR	5	input
CCRM_WR	1	input
CHRM_WDATA	8	input
CHRM_ADDR	8	input
CHRM_WR	1	input

Table 8: Interface signals of the LCDC module

Command Value	Command Name	Description
00	No command	The LCDC module does nothing when receiving this value. Anytime when no command is sent, CMD should be set to 00.
01	CLEAR	This command clears the display on the LCD screen. This command sets all the data in the inside character memory (CHRM) to 0x20, which is the ASCII code of the blank character.
10	REFRESH	This command refreshes LCD screen with the data from the inside memories. The control and character data are saved in the Character Command Memory (CCRM) and Character Memory (CHRM). Anytime when some data in these memories are changed, the REFRESH command should be applied to update the displaying on the LCD screen.
11	Reserved	Reserved by the LCDC module. Users should not send this command in any case.

Table 9: The commands of the LCDC module

connected to the PLCN module through the signals listed in Table 8.

- CMD

This signal is used by PLCN to send commands to LCDC. Commands corresponding to each value of CMD are listed in Table 9.

- BUSY

When this bit is logic '1', the LCDC module is busy and can not accept any command more. PLCN can send commands to LCDC only when this bit is '0'.

- CCRM\_WDATA

Character Command Memory (CCRM) write data. The 36 bits are stored in a 36bits×32 on-chip memory addressed by the CCRM\_ADDR signal.

- CCRM\_ADDR

CCRM address. Users can write to CCRM addressed by this signal.

- CCRM\_WR

Write control signal for CCRM. Users write data to CCRM by setting this signal to '1' at a rising clock edge.

- CHRM\_WDATA

Character Memory (CHRM) write data. This memory is an 8bits×256 memory, which is used to save character codes. Each character code is 8 bit wide in ASCII format.

- CHRM\_ADDR

CHRM address. Users can write to CHRM addressed by this signal.

- CHRM\_WR

Write control signal for CHRM. Users write data by setting this signal to '1' at a rising clock edge.

Two dual port RAMs, CHRM (8bits×256) and CCRM (36bits×32) are used to control the displaying on

LCD screen. CHRM is used to store character codes; CCRM is used to store character commands (CCMD). Dual port means that each of these memories has two read/write ports which can be accessed at the same time. Both CCRM and CHRM are synchronous RAM, which can accept pipelined read/write operations.

One port of each memory (port A of the dual port RAM) is connected to PLCN. Users can write data in both memories to change the displaying on the LCD screen. Users write character codes, which are to be displayed on LCD, into the CHRM; The CCMDs (display command) are written into the CCRM. The port B of both memories are used by LCDC internally.

Inside LCDC a piece of circuit computes the addresses for all characters corresponding to the LCD screen positions from the content (CCMDs, 36 bits) of CCRM. Then the character codes are read from CHRM and displayed on the LCD screen. This computing and displaying process is triggered when the command REFRESH appears on the CMD bus. The format of the CCMD, which consists of V, CTR, CPOS, CLTH, CBAD, CSAD and WLTH, is listed in Table 10.

Name	Bits	Definition
V	35	Valid bit. If V='1', the circuit inside the LCDC executes the command defined in the other sections of this entry when a REFRESH command is received. If V='0', this memory entry is ignored.
34:32		reserved.
CTR	31	Refresh direction. If CTR='0', each time the character index on the LCD screen is subtracted by 1 and the corresponding position on the LCD screen is updated. If CTR='1', the character index is added by 1 during refreshing.
CPOS	30:26	The start character position on the LCD screen. The LCDC circuit displays the character codes read from CHRM from the CPOS, in the direction defined by CTR.
CLTH	25:21	Character length. CLTH defines the number of the characters (CLTH+1) to be refreshed on the LCD screen.
CBAD	20:13	The start address of the character storage section. When WLTH+1 characters are read from CHRM memory, the LCDC wraps back to the address defined by CBAD to read the following character codes from CHRM.
CSAD	12:5	The address which the LCDC starts to read character codes from CHRM.
WLTH	4:0	Wrap length. The LCDC circuit wraps to CBAD when WLTH+1 characters have been read from CHRM memory.

Table 10: CCMD definition

Each entry defines a command to refresh a region on the LCD screen using the character codes saved in CHRM. The maximal number of CCMDs saved in CCRM is 32, because the LCD can display only 32 characters. Each time when a REFRESH command is detected on the CMD bus, the LCDC reads all the entries of the CCRM one by one. The corresponding displaying command from each entry is executed, if the V bit is '1' in this entry. If V='0', the entry is ignored by LCDC. The algorithm to display characters on the LCD screen is listed in the following code.

```

for each entry i in the CCRM {
  if V='1' {
    addr:=CSAD
    lcdpos:=CPOS
    for j in 0 to CLTH {
      display CHRM(addr) on LCD position lcdpos
      if CTR='0' {
        if lcdpos = 0 {
          lcdpos:=31
        }
      }
      else {

```



```

        lcdpos:= lcdpos-1
    }
}
else {
    if lcdpos = 31 {
        lcdpos:=0
    }
    else {
        lcdpos:= lcdpos+1
    }
}

if addr = CSAD+WLTH {
    addr:=CBAD
}
else {
    addr:=addr+1
}
}
}
}
}

```

The default value of the 36 bits of the first entry in the CCRM is 0x“883E0001F”. The V bits of all the other entries are set to ‘0’ by default.

When the command on the CMD bus is CLEAR, all the character codes in the CHRM are filled with 0x20, which is the ASCII code for blank character. Be noted that this command does not refresh the LCD screen. To clear the LCD screen an additional REFRESH command should be issued.

During executing any command the BUSY signal is set to ‘1’ by the LCDC to signify that the current state of LCDC is busy and no more command should be sent.

In short, users can manage the content in CCRM and CHRM to form different displaying solutions (for example scrolling). The LCD screen is updated using the instructions in the CCRM and the character codes in the CHRM by issuing the REFRESH command.

The character codes in the CHRM are in ASCII encoding format. For example, to display the character ‘a’, 0x“61” should be written to the CHRM. The ASCII table is listed in Table 11.

### 3.4 Decoder Specification

Figure 4 shows the internal structure of the decoder. The MP3 decoder reads MP3 data from its input buffer (DBUF) and decodes them to PCM samples, which are written to an output sample buffer (SBUF). The AC’97 chip is configured and controlled through the hardware configuration instruction buffer (IBUF). During play process, the AC’97 chip reads the PCM samples from SBUF automatically and plays them using the configured sample rate and sample format by the decoder. When DBUF is empty, the decoder has no MP3 data to decode and can not write valid samples to SBUF. In this case, the decoder keeps waiting for the next valid DWORD in the DBUF unless the DEC\_RST signal is valid. When SBUF is empty, the AC’97 chip plays all ‘0’s as the PCM sample. After power on the decoder starts a decoding process automatically. The interface signals of the decoder module to the PLCN module are listed in Table 12.

#### ■ DBUF\_ALMOST\_FULL

When DBUF\_ALMOST\_FULL is ‘0’, the DBUF can accept at least 256 DWORDs. This signal is used to trigger the monitoring process, which fills the DBUF with the MP3 data regularly. When the DBUF\_ALMOST\_FULL is ‘0’ during the play process, a 256 DWORD data block (or less) should be requested to fill DBUF, so that the fluent playing process can be maintained. After power on, the DBUF is empty and DBUF\_ALMOST\_FULL is ‘0’.

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	,	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table 11: ASCII table

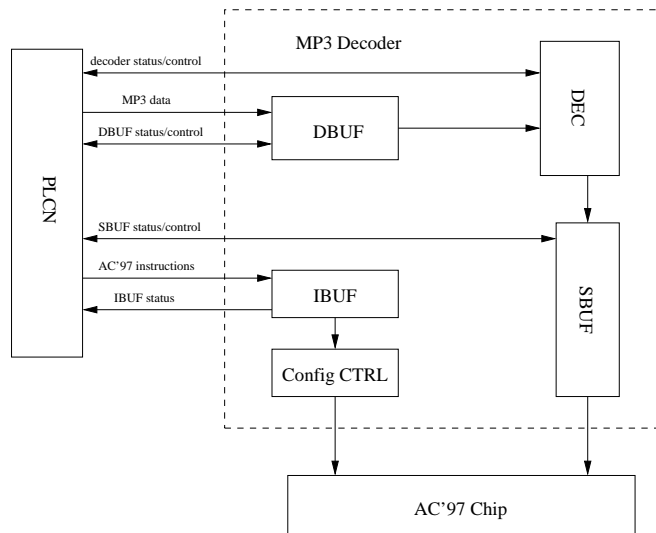


Figure 4: The structure of the MP3 decoder

- **DBUF\_WR**  
Write enable of the DBUF. When this signal is '1' at a rising clock edge, the data on the DBUF\_DIN is written into DBUF at the same rising clock edge.
- **DBUF\_DIN**  
Input data bus of the DBUF.
- **DBUF\_RST**  
This signal clears the data in the DBUF. If this signal is '1' at a rising clock edge, the DBUF will be cleared to empty and the DBUF\_ALMOST\_FULL will be '0' at the next rising clock edge. DBUF\_RST should be only one clock period '1' to clear the DBUF.
- **SBUF\_FULL**  
The full status of the SBUF. When SBUF\_FULL='1', the SBUF is full.
- **SBUF\_EMPTY**  
SBUF\_EMPTY='1' means there is no data in the SBUF; otherwise there is at least one PCM sample in SBUF. When SBUF is empty, the AC'97 chip plays all '0's as the PCM sample automatically.

Signal Name	Width	Direction (decoder view)
DBUF_ALMOST_FULL	1	out
DBUF_WR	1	in
DBUF_DIN	32	in
DBUF_RST	1	in
SBUF_FULL	1	out
SBUF_EMPTY	1	out
SBUF_RST	1	in
HW_FULL	1	out
HW_WR	1	in
HW_DIN	32	in
DEC_RST	1	in
DEC_STATUS	1	out

Table 12: Interface signals of the decoder module

■ SBUF\_RST

This signal clears the decoded samples in the SBUF. If this signal is ‘1’ at a rising clock edge, the SBUF will be cleared to empty. The SBUF\_FULL will be ‘0’ and SBUF\_EMPTY will be ‘1’ at the next rising clock edge. SBUF\_RST should be only one clock period ‘1’ to clear the SBUF.

■ HW\_FULL

The full status of the decoder instruction buffer (IBUF). When HW\_FULL=‘1’, the buffer is full and cannot accept more instructions. Otherwise, IBUF can accept at least one instruction.

■ HW\_WR

Write enable of the IBUF. When this signal is ‘1’ at a rising clock edge, the instruction on the HW\_DIN at the same rising clock edge is written into the IBUF.

■ HW\_DIN

Instruction data bus of the IBUF. The bit 31 of this vector is the command valid bit. If this bit is ‘1’, this command is a valid command. Otherwise, this command is simply ignored by the decoder. The 28 to 30 bits of HW\_DIN contain the command of the instruction. Currently only two commands are supported: “000” is the *change volume* command; “001” is the *pause* command; all the other values are ignored by the decoder.

When the command is “000”, the value of 0 to 4 bits of the HW\_DIN is the volume of the right channel of the player; the value of the 8 to 12 bits is the volume of left channel of the player. The maximal volume is “00000” and the minimal volume is “11111”. The default volume of the player is “00000”. The 15th bit of HW\_DIN is the mute control bit. When the bit 15 is ‘1’, both the left and right channels are muted so that no sound can be heard but the decoding and play process still proceed. After power on, the default mute status is unmute.

When the command is “001”, the player toggles its status between pause and playing. During pause state, the decoding and play process is stopped and the AC‘97 chip plays all ‘0’s so that no sound can be heard. The default pause status of the player is playing.

■ DEC\_RST

If this signal is ‘1’ at a rising clock edge, the decoder is informed to stop its current decoding loop and initialize for the decoding of the next MP3 file. DEC\_RST should be only one clock period ‘1’ to reset the decoder.

■ DEC\_STATUS

This signal reflects the internal reset status of the decoder. When the decoder detects the DEC\_RST signal is ‘1’ at a rising clock edge, it starts to reinitialize itself and sets the DEC\_STATUS to ‘1’ instantly. After the reinitialization the DEC\_STATUS signal is switched back to ‘0’, which means a new play process is started and the decoder starts to read the MP3 data from the DBUF.

During the reinitialization of the decoder, it firstly writes the internal cached decoded samples to SBUF, then clears its own internal buffers and finally starts reading DBUF again. In short, the decoder may write some data to SBUF when DEC\_STATUS=‘1’.

Any time when the DEC\_STATUS is '1', MP3 data should not be written into the DBUF and hardware configuration instructions should not be written into the IBUF. The default value of this signal is '0'. Be noted that the DBUF and SBUF are not cleared during the reinitialization of the decoder.

## 4 VHDL Design Guidelines

In this section some VHDL design guidelines are listed. Abiding by these rules can benefit the simulation and the design consistence. Be noted that for ASIC and FPGA designs, some design rules may differ. But most of the rules are common to both design types. The rules listed below are specified for FPGA design. *reset.state* and *clk\_polarity* in the code examples are constants and defined in *\$HOME/prj/comp\_def/system\_constants.vhd*.

### 1. Use the AFTER statement carefully

AFTER statement might cause mismatch between simulation and the synthesized netlist, if the time parameter is larger than one clock period. To write RTL level code, do not use the AFTER statement.

### 2. Avoid gated clocks unless absolutely necessary

The clock and asynchronous reset/set signals are routed by dedicated global wires. The number of the global wires is very limited. If some logic gates appear in the path of the global wires, a new global wire may be needed to route the gate output forwardly. In this case the placement&routing tool must spend more time to route all these global routings, and possibly there are no enough resources to fulfill the complete routing task.

### 3. Avoid latches unless absolutely necessary

Latches cause potential problems for static timing analysis and testability.

### 4. Complete reset is mandatory for sequential processes

No logic gates should appear on the reset path. The incomplete reset below generates an inverter at the asynchronous reset path, which should be avoided.

```
reg: process (clk, res)
begin -- reg
    if res=reset_state then --<- reset branch
        outbit1 <= '0';      --<- reset branch
        -- outbit2 <= '0'; --<- if this assignment is commented an inverter
                           -- is generated.
    elsif clk'event and clk=clk_polarity then
        outbit1 <= ctl;
        outbit2 <= ctl;
    end if;
end process reg;
```

### 5. Avoid internal three-state buses unless absolutely necessary

Internal three-state signals are difficult to handle at fabrication test and during constraining for logic synthesis. An alternative is to use a multiplexer solution. But this may result in high wire areas and problems during routing, depending on the number of channels. Use internal three-state buses only if such routability problems are obvious. Generate three-state buffers using the following concurrent statement:

```
bus <= ram_dat when ena = enable_state else (others => 'Z');
```

Not all FPGA structures contain internal tri-state buffers so that the usage of tri-state buses should be avoid in FPGA designs. For the chip level bidirectional pins, which are used to communicate with other external chips, for example the data bus of a RAM chip, the tri-state buffers should be directly instantiated/described. The sub module which interfaces tri-state buffers should use three signals (input, output and enable) and connect them to the respective ports of the tri-state buffer instances.

### 6. Refer to the same clock edge

The mixture of rising edge and falling edge triggered flip-flops may cause timing analysis problems. Special consideration should be given when mixed clock edges/signals are used.

### 7. Do not use combinational feedback loops

Combinational feedback loops are often the source of problems in tools relying on static timing information.

**8. Balance clock to delta accuracy**

Gated clocks or clock assignments imply delta cycles in RTL models. This corrupts the memory function or serially connected register processes (shift-registers). As an example the synthesis result of the following VHDL description does not match simulation.

```
r1: process (mclk)
begin
    if mclk'event and mclk=clk_polarity then
        dat <= d_i;
    end if;
end process r1;
gclk <= mclk; --<- this infers one delta!
r2: process (gclk)
begin
    if gclk'event and gclk=clk_polarity then
        d_o <= dat;
    end if;
end process r2;
```

Generally all registers should be directly clocked by the same signal in the respective clock domain. Do not assign the original clock signal to another signal.

**9. The assignment of a signal should appear only in one process**

The motivation for this recommendation is to avoid multi-driver errors. These may easily occur if a signal of resolved type is assigned twice.

```
reg: process (clk, reset)
begin -- reg
    if reset=reset_state then
        output <= '0';
    elsif clk'event and clk=clk_polarity then
        output <= ctl;      --<-----,
    end if;                  \
end process reg;             |
-- ... --<- lots of code here |
output <= ctl;               --<- overlooked, that 'output' has
                             -- been assigned before
```

The result of such description is an unintended multi-driver in the synthesized netlist as well as simulation behavior.

**10. All combinational conditional assignments should be checked for completeness**

Incomplete combinational conditional assignments lack a default statement or an else clause. This may lead to unintended latch inference from non-clocked processes.

```
latch: process (en, pn_gen_ssrg_outbit)
begin
    if en='1' then
        outbit <= pn_gen_ssrg_outbit;
    end if;
end process latch;
```

To avoid this problem, always use an else branch:

```
combo: process (en, pn_gen_ssrg_outbit)
begin
    if en='1' then
```

```

        outbit <= pn_gen_ssrg_outbit;
    else
        outbit <= other_value;
    end if;
end process combo;

```

or a default assignment in front of the conditional structure:

```

combo: process (en, pn_gen_ssrg_outbit)
begin
    outbit <= other_value;
    if en='1' then
        outbit <= pn_gen_ssrg_outbit;
    end if;
end process combo;

```

## 11. Use standard templates for clocked processes

```

async_res: process (clk, rst)
begin
    if rst=reset_state then
        <actions_to_perform_at_async_reset>
    elsif clk'event and clk=clk_polarity then
        <actions_to_perform_at_pos_clock_edge>
    end if;
end process async_res;

```

Clock and asynchronous reset must be in the sensitivity list to avoid that the simulation result mismatches the behavior of the circuit after pre- and post-synthesis. The *reset\_state* is a one-bit constant defined by the system designer.

```

sync_res: process (clk) --<- synchronous reset (rst_sync) should not
-- be in sensitivity list
begin
    if clk'event and clk=clk_polarity then
        if rst_sync='0' then
            <actions_to_perform_at_sync_reset>
        else
            <actions_to_perform_at_pos_clock_edge>
        end if;
    end if;
end process sync_res;

```

## 12. Minimize the number of signals of the sensitivity list

Minimize the number of signals of the sensitivity list so that processes are only activated when a re-evaluation of the outputs is required. For synchronous circuit the clock signal is required in the sensitivity list. If asynchronous reset exists, the reset should also be in the sensitivity list. Except the clock and reset signals, no other signals can present in the sensitivity list of a synchronous circuit.

## 5 Displaying File Names

In this section the displaying file name function of the play control (PLCN) module will be implemented. The PLCN module reads the file information through the FIO module, then displays the file name on the LCD screen through the LCDC module (see Figure 2).

### 5.1 KBC Interface

The KBC module provides an FIFO (First In First Out) output interface to PLCN. When keys on the keypad are pressed, the respective scan codes are saved in this FIFO inside the KBC module. These keys should be read and used to activate the respective function blocks in PLCN.

The KBC FIFO output interface has 4 signals specified in Section 3.1. The *empty* shows the status of this FIFO. When *rd* is '1' at a rising clock edge, a valid read command is sent to KBC. When *rd\_ack* is '1' at a rising clock edge, a valid key scan code is on the 8bit *data* bus. When the FIFO is empty (*empty*='1'), no valid *rd\_ack* will be returned even though *rd* is valid.

In the MP3 system, the key scan codes are read instantly from the FIFO, once it is not empty. The *empty* signal is directly connected to the *rd* across an inverter. The key scan codes from the KBC module are used to activate different function modules. 0x72 and 0x75 are the scan codes of keys '2' (also '↓') and '8' (also '↑'), which are used to list the next file name and the previous file name. When a '↓'/'↑' key is detected, the corresponding signal *listnext*/*listprev* is generated to trigger the list block.

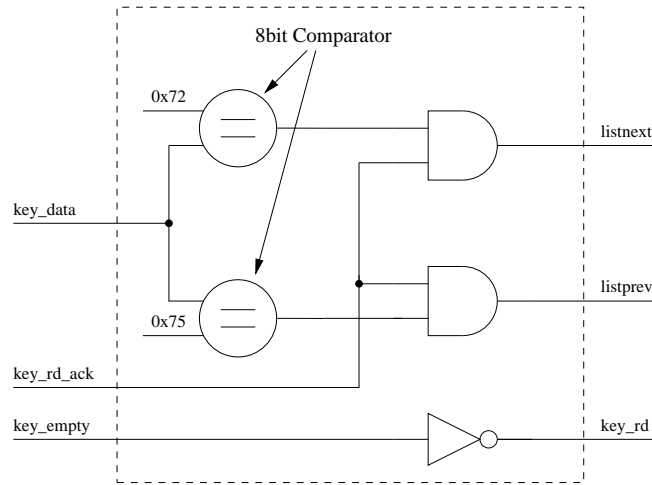


Figure 5: Key code comparator

#### Exercise 1:

Please describe the circuit in Figure 5 as a VHDL entity. The entity definition and input/output signals are listed in Code 1.

### 5.2 Reading File Information

File information of the FAT16 file system is listed in Table 7. The file information can be fetched from the FIO module by issuing the FILENEXT/FILEPREV commands through the BUSI/BUSIV/CTRL signals of the FIO module.

A state machine is used to control the signals to the FIO module. Figure 6 shows the basic structure of a state machine, which consists of three components: the state register, the next state logic and the output logic. State diagram is normally used to represent the transitions and the output functions of a state machine. Each state in the state machine is represented by a node and each transition by an arc. Figure 7 shows an example of such a state diagram with *s\_k* as the default state after reset. The arc from node *s\_k* to node *s\_j* with label '1'/'1' (the first '1' is the input condition; the second '1' is the output) specifies that, for the present state *s\_k* and the input '1', the next state is *s\_j* and the output is '1'.



```

library ieee;
use ieee.std_logic_1164.all;

entity commandcomparator is
  port(
    rd      : out std_logic;
    rd_ack  : in  std_logic;
    data    : in  std_logic_vector(7 downto 0);
    empty   : in  std_logic;
    listnext : out std_logic;
    listprev : out std_logic
  );
end commandcomparator;

architecture commandcomparator_arch of commandcomparator is
  --define signals here

begin
  --describe the circuit here

end architecture;

```

Code 1: Key code comparator

A general form of state machine can be described using three VHDL processes, corresponding to the three components in Figure 6. The output process and the state transition process are both combinatorial and generate the output values and the next state using the current states and input values respectively. The sequential process saves the next state to flip-flops so that it can be used as current state at the next clock edge.

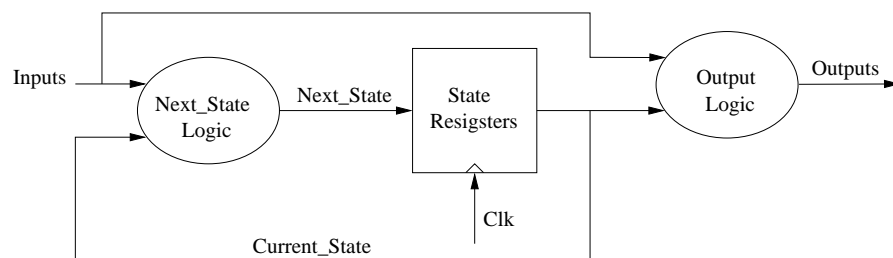


Figure 6: Block diagram of a finite-state machine

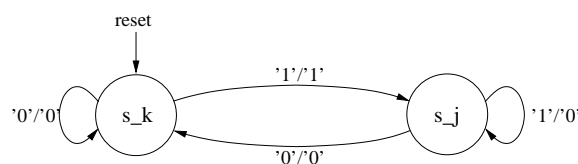


Figure 7: State diagram representation

### Exercise 2:

Please describe the state machine shown in Figure 8 and the circuit for *busi*. The input and output ports of this entity are defined in Code 2. *info\_ready* is defined in Section 5.4. The default state after reset is idle. Only the transition conditions are shown on the transition arcs. Each transition arc is referred with a name, as A\* in Figure 8.

The values of the output signals are explained below:

- **req** is the interface request signal connected to the arbiter described in Section 5.3 to get the access of the FIO module. The interface of the FIO module is shared by several blocks, e.g. the list block and the play block in Section 6.1. When the *req* signal is '1', the list block informs the arbiter that it needs the interface. The arbiter decides the value of the corresponding *gnt* signal according to the values of all the *gnts* and their priorities. If the *gnt* signal of a block is '1', this block is granted to use the FIO interface. *req* is '0' at arc A1 and A6 and '1' at the other arcs.
- **busiv** is the signal to designate that the data/command on the *busi* signal is valid. *busiv* is '1' at arc A4 and '0' at the other arcs.
- **ctrl** designates the meaning of *busi*. *ctrl*='1' means the current value of *busi* is a command. *ctrl*='0' means the current value of *busi* is a parameter. *ctrl* is '1' at arc A4 and don't care at the other arcs.
- **info\_start** informs the other blocks in the PLCN module that the list command is sent to the FIO module. The next data from FIO will be file information. This signal will clear the file information counter in Section 5.4. *info\_start* is '1' at arc A4, and '0' at the other arcs.

**busi** is the data/command to the FIO module. At arc A4 this signal should have value 0x"00"/0x"01" when listing the next/previous file. The *busi* signal is the output of a register instead of the state machine and updated only at arc A2.

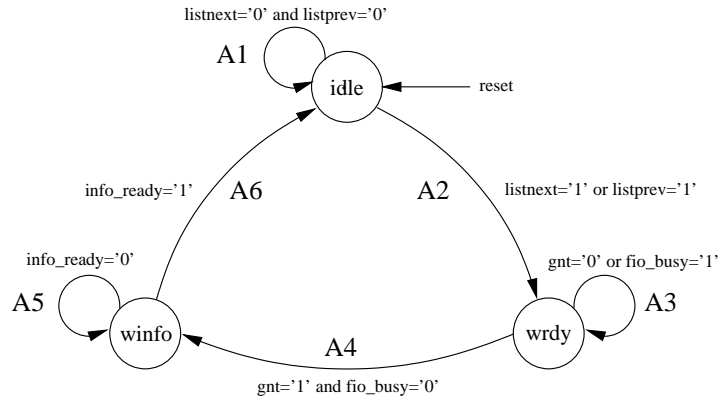


Figure 8: Displaying file information state machine

### 5.3 Arbiter&Multiplexer

The FIO module is used by several blocks in the PLCN module. In order to avoid bus collision, an arbiter and a multiplexer are implemented to coordinate the access of the FIO interface. Figure 9 shows the diagram of the arbiter and the multiplexer. The  $i$ th input signal vector ( $0 \leq i \leq N - 1$ ) is input( $M * (i + 1) - 1$  downto  $M * i$ ), where M and N are generics.

Each block which uses the FIO interface has a *req* signal. This signal is connected to a bit of the *req* signal of the arbiter. The arbiter decides which block can get the bus access and informs it by the corresponding bit of the *gnt* signal. Only after the block which is currently holding the bus finishes its operation and releases the bus by setting its *req* to '0', can the next block get the bus access, i.e. non-preemptive arbitration. Several blocks can request the bus access at the same time, but they are granted one after another, depending on the priority of the *req* bits they connect. The *req*(0) has the highest priority and the *req*(N-1) has the lowest priority.

Figure 10 shows the diagram of the arbiter. The AND and NOR gates in Figure 10 are used to check if the block which currently holds the interface has released the bus by changing its *req* to '0'. The *gnt* signal is

```

library ieee;
use ieee.std_logic_1164.all;

entity listctrl is
  port(
    clk      : in  std_logic;
    reset    : in  std_logic;
    listnext : in  std_logic;
    listprev : in  std_logic;
    req      : out std_logic;
    gnt      : in  std_logic;
    busi     : out std_logic_vector(7 downto 0);
    busiv    : out std_logic;
    ctrl     : out std_logic;
    busy     : in  std_logic;
    info_start : out std_logic;
    info_ready : in  std_logic
  );
end listctrl;

```

Code 2: Port definition of the state machine of the displaying function

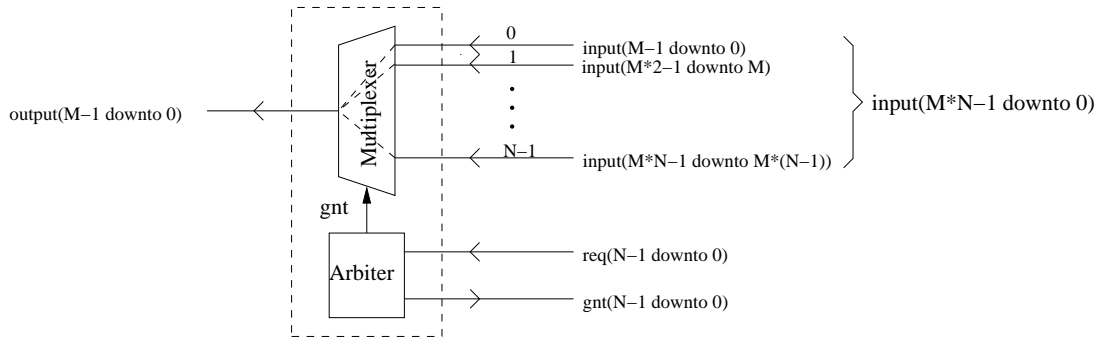


Figure 9: The diagram of the arbiter and multiplexer

used as a mask to *bitwise 'AND'* the *req* signals (see the AND gate in Figure 10). All the bits after the AND gate are *'ORed'* together and the inverted signal is used as the enable of the register (see the NOR gate in Figure 10). If the current granted block releases the bus control, the next grant value from the arbitration logic is assigned to the *gnt* at the output of the register. The arbitration logic decides which logic will get the valid grant signal based on the current values of their *req* signals and the priorities of them. *gnt(i)* can be decided by

```

if i='0' then
  gnt(0) <= req(0);
else
  gnt(i) <= not (req(0) or req(1)...or req(i-1)) and req(i);
endif;

```

Because the arbitration logic is non-preemptive, flip-flops with enable pin are used to store the current *gnt* value. The output of the flip-flops are updated only when the enable signal is valid. Code 3 shows a template to describe such a flip-flop, where *reset\_state* and *clk\_polarity* are constants and defined in *\$HOME/prj/comp\_def/system\_constants.vhd*.

After a block gets the FIO access, its output signals to the FIO module should be selected by the multiplexer to its output, which is connected to the FIO interface directly. The *gnt(N-1 downto 0)* is used

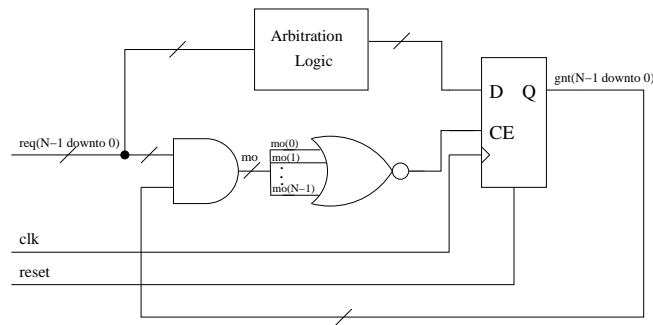


Figure 10: Arbiter structure

```

process(clk, reset)
begin
    if reset=rest_state then
        --insert the reset statements here
    elsif clk'event and clk=clk_polarity then
        if enable=enable_state then --this if has no matching else
            --insert logic before the flip-flop
        end if;
    end if;
end process;

```

Code 3: DFF with enable

as the select signal of the multiplexer.  $output(j)$ , where  $j=0,1,\dots,M-1$ , can be decided by

$$output(j) \leq (gnt(0) \text{ and } input(0*M+j)) \text{ or } (gnt(1) \text{ and } input(1*M+j)) \dots \\ \text{or } (gnt(N-1) \text{ and } input((N-1)*M+j));$$

### Exercise 3:

Please describe the arbiter and multiplexer using generics M and N. The entity is illustrated in Code 4.

```

library ieee;
use ieee.std_logic_1164.all;

entity arbitermultiplexer is
    generic(
        M      : integer;
        N      : integer
    );
    port(
        clk     : in  std_logic;
        reset   : in  std_logic;
        input    : in  std_logic_vector(M*N-1 downto 0);
        output   : out std_logic_vector(M-1 downto 0);
        req     : in  std_logic_vector(N-1 downto 0);
        gnt     : out std_logic_vector(N-1 downto 0)
    );
end arbitermultiplexer;

```

Code 4: Port definition of the arbiter&amp;multiplexer

## 5.4 Receiving File Information

When the FIO module receives the FILENEXT/FILEPREV command, it returns the 32 bytes file information (see Table 7) through the *fio\_buso* and *fio\_busov* signals. When *fio\_busov*='1' at a rising clock edge, the corresponding *fio\_buso* is valid. The 32 bytes file information are sent in the same order as defined in Table 7 by 8 DWORDs. There may be intervals between these returned DWORDs, where the *fio\_buso* is '0' because the FIO block needs some time to load further data from the external SD card.

The first 11 bytes are the file name in the old 8:3 MS-DOS format. The first 8 bytes contain base file name, and the next 3 bytes are the file extension name. There is no "." in the file name, which is added by the OS automatically. The file name should be displayed on the 16×2 LCD. In this section the instructions to display the file name at the first 11 character positions of the LCD will be given.

The LCD controller used in this project provides two RAM interfaces: *CHRM* and *CCRM*. *CHRM* is an 8bits×256 RAM, which can save 256 characters. *CCRM* is a 36bits×32 RAM, which controls which characters from the *CHRM* should be displayed on which positions on the LCD screen. The *lcdc\_cmd* signal controls the refreshment of the LCD. When *lcdc\_cmd*="10", the algorithm described in Section 3.3 is used to update the LCD screen. *lcdc\_cmd* command can be sent only when the *lcdc\_busy* signal is '0'. Figure 11 shows the diagram of displaying file names.

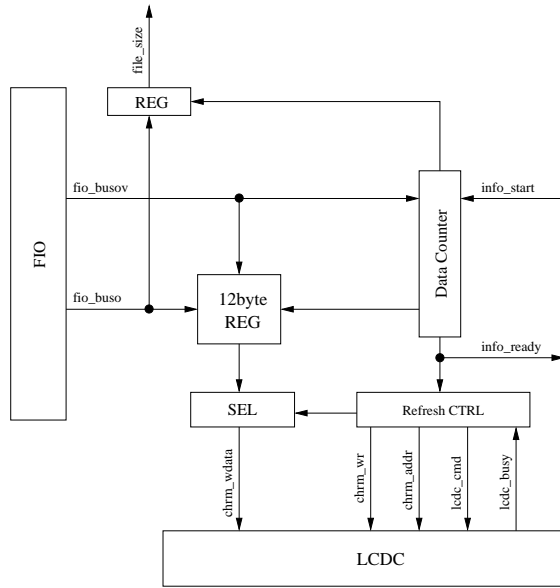


Figure 11: File name displaying

The 4bit Data Counter is used to count the DWORD number of the file information. This counter is reset to "1000" using the global reset signal. When the *info\_start* signal from the state machine in Code 2 changes to '1', which means the coming data from FIO is file information, the counter is cleared to 0. Each time when the *fio\_busov* is '1' at a rising clock edge, the counter increases 1. When the counter reaches 8, it will not increase but keep its old value because all file information has been received. Each time when the *fio\_busov* is '1' and the data counter is less than 3, the output DWORD on the *fio\_buso* bus is registered by flip-flops in the 12-byte register, from which the first 11 bytes are used to save file names. These file name bytes are sent to LCDC for displaying. When the Data Counter is 7 and *fio\_busov* is '1', *fio\_buso* contains the file size, which is registered and will be used in the play function to decide the file end. Code 5 shows an example of the data counter with asynchronous reset.

When the Data Counter changes from 7 to 8, the valid *info\_ready* is generated to inform the state machine in Figure 8 to release the FIO interface. At the same time, *info\_ready* is used to trigger the process to update the LCD screen. Code 6 shows the code to generate *info\_ready*.

After the file information is received, the *Refresh CTRL* block is triggered by the *info\_ready* signal to write the file name into the *CHRM* and send the *REFRESH* command to LCDC. A refresh counter (*refcounter*) inside the *Refresh CTRL* block is used to select the file name bytes sequentially and generate

```

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

...
signal counter : std_logic_vector(3 downto 0);
...

process(clk, reset)
begin
    if reset = reset_state then
        counter <= "1000";           --asynchronous global reset
    elsif clk'event and clk = clk_polarity then
        if info_start = '1' then     --synchronous reset
            counter <= "0000";
        elsif busov = '1' and counter /= "1000" then
            counter <= counter+1;
        end if;
    end if;
end process;

```

Code 5: Counter template

```

library ieee;
use ieee.std_logic_1164.all;

...
signal count3 : std_logic;
...

process(clk, reset)
begin
    if reset = reset_state then
        count3 <= '1';
        info_ready <= '0';
    elsif clk'event and clk = clk_polarity then
        count3 <= counter(3);
        if count3 = '0' and counter(3) = '1' then
            info_ready <= '1';
        else
            info_ready <= '0';
        end if;
    end if;
end process;

```

Code 6: Detect counter switch

the address of CHRM. The file name bytes should be written into the first 11 bytes of the CHRM to display them on the first 11 positions of the LCD. The output of the refresh counter is connected to the four bits (3 downto 0) of *chrm\_addr* and the other bits of the *chrm\_addr* are '0's. The *refcounter* is set to "0000" when the *info\_ready* is valid and increases 1 at each rising clock edge when its value is not "1011", which is the value set by the asynchronous reset. Refer Code 5 when describing the refresh counter. When the value of the refresh counter is less than "1011" the *chrm\_wr* signal is '1' to write the file name bytes to CHRM.

A case statement can be used to select the saved file name bytes to *chrn\_data*. Code 7 shows the case example.

```

case refcounter(3 downto 0) is
  when "0000" =>
    chrn_wdata <= filenamereg(7 downto 0);
  when "0001" =>
    chrn_wdata <= filenamereg(15 downto 8);
  when "0010" =>
    chrn_wdata <= filenamereg(23 downto 16);
  when "0011" =>
    chrn_wdata <= filenamereg(31 downto 24);
  when "0100" =>
    chrn_wdata <= filenamereg(39 downto 32);
  when "0101" =>
    chrn_wdata <= filenamereg(47 downto 40);
  when "0110" =>
    chrn_wdata <= filenamereg(55 downto 48);
  when "0111" =>
    chrn_wdata <= filenamereg(63 downto 56);
  when "1000" =>
    chrn_wdata <= filenamereg(71 downto 64);
  when "1001" =>
    chrn_wdata <= filenamereg(79 downto 72);
  when others =>
    chrn_wdata <= filenamereg(87 downto 80);
end case;

```

Code 7: File name byte selection

After the file name bytes are written to the CHRM, the *REFRESH* command should be sent to LCDC to update the file name on the LCD screen. This command should be sent only when the *lcdc\_busy* signal is '0'. The REFRESH command should last only one clock period. The code example in Code 8 is used to generate the *lcdc\_cmd* signal.

#### Exercise 4:

Please analyze the waveform of the signals *chrn\_ready* and *lcdc\_cmd* in code 8.

#### Exercise 5:

Please write the VHDL code of the complete file information processing block, based on the structure and code described in this section.

## 5.5 Building the List Function

After finishing all the sub blocks of the displaying file name function, all these sub blocks are connected together to form the PLCN module, which can be synthesized and tested in hardware. The instructions for synthesis and implementation of the complete MP3 project can be found in Section 7.

Figure 12 shows the top structure and connections of the sub blocks. The *file\_size* signal is left unconnected in the list function. The top entity of the PLCN module is given in the *MP3\_prj/playcontrol* directory with name *playcontrol.vhd*. If an output signal from the port definition of the PLCN module is not driven in the list function, it should be connected to '0's. All the unused input signals from the port definition of the PLCN should be left unconnected.

When instantiating the arbiter&multiplexer module, the generics M and N are set to 10 and 3. The signals from the state machine are connected to the port input( $M * 3 - 1$  downto  $M * 2$ ) of the multiplexer.

```

library ieee;
use ieee.std_logic_1164.all;

...
signal chrm_ready : std_logic;
...

process(clk, reset)
begin
    if reset = reset_state then
        chrm_ready <= '0';
    elsif clk'event and clk = clk_polarity then
        if lcdc_cmd = "10" then
            chrm_ready <= '0';
        elsif refcounter = "1010" then
            chrm_ready <= '1';
        end if;
    end if;
end process;

process(clk, reset)
begin
    if reset = reset_state then
        lcdc_cmd <= "00";           --no command
    elsif clk'event and clk = clk_polarity then
        if lcdc_cmd = "10" then
            lcdc_cmd <= "00";
        elsif lcdc_busy = '0' and chrm_ready = '1' then
            lcdc_cmd <= "10";       --refresh command
        end if;
    end if;
end process;

```

Code 8: Generate the REFRESH command

The other two ports are reserved for the play function and driven to all '0's. The clock and reset signals are global signals and are defined in the port of the PLCN module.

### Exercise 6:

Please connect all the sub blocks of the list function using VHDL, as shown in Figure 12. The list function can be tested using FPGA by following the instructions in Section 7.



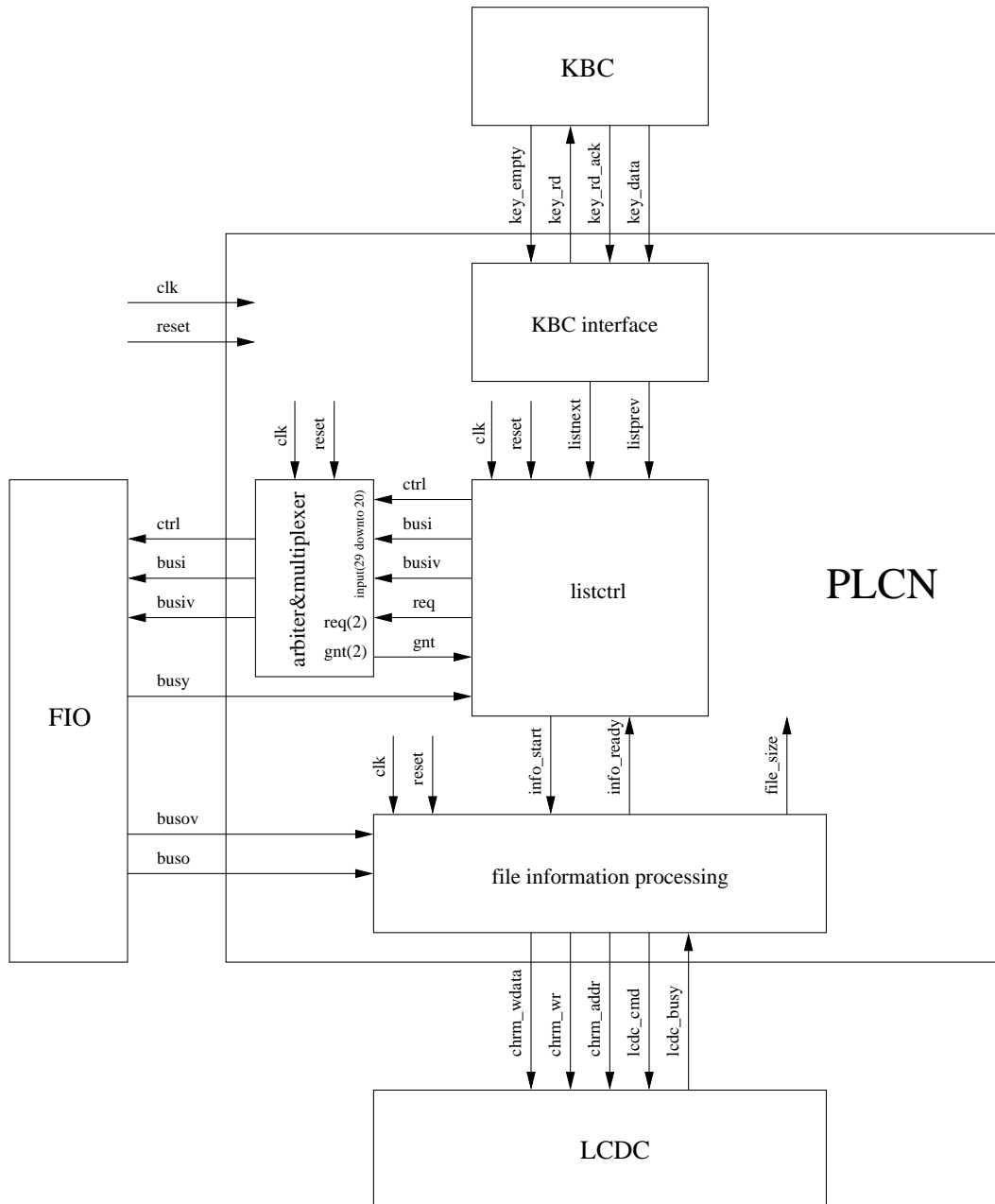


Figure 12: Top structure of the list function

## 6 Making a Simple MP3 Player

In this section, the play function of the MP3 player will be implemented. The file listed on the LCD screen is decoded and the hardware is initialized to start the playing process. The additional functions, e.g. mute, pause etc. should also be implemented.

### 6.1 Playing the Music

Figure 13 shows the structure of the PLCN module including the list and play functions, where not all but a part of illustrative signals are shown. The real signal names can be defined according to Section 3.

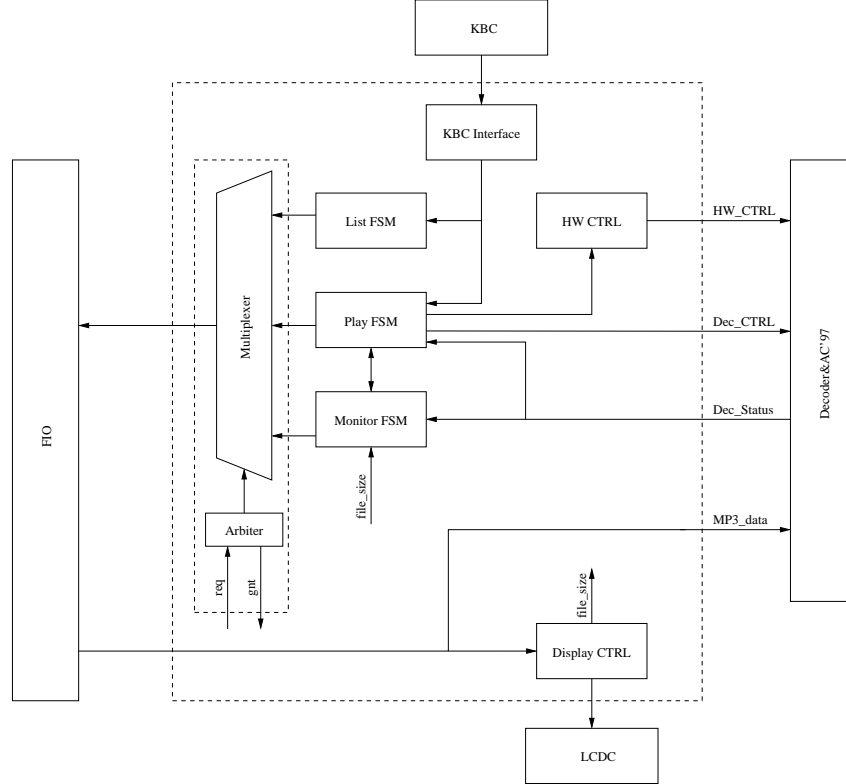


Figure 13: Top structure of the PLCN module

The functions of the blocks are described below:

- **KBC Interface** reads the key scan codes from the KBC FIFO and decodes them to generate control signals, for example the *listnext* and *listprev* signals in the list function. In the play function the scan codes of the keys 'Esc' and 'Ctrl' are decoded to generate signals to trigger the *play start* and *play stop* functions.
- **PLAY FSM** controls the start and end of the playing process. If the 'Esc' key is sent to the PLCN module, this state machine sends the open command to FIO and enables the Monitor state machine (Monitor FSM). During the playing process, the PLAY FSM stops the Monitor FSM when the key 'Ctrl' is pressed, or when the file end is reached.
- **Monitor FSM** periodically checks the status of the MP3 Data buffer (DBUF) via its almost full signal (*dbuf\_almost\_full*). If *dbuf\_almost\_full* signal switches to '0', another read command is sent to the FIO module so that additional MP3 data are requested from the FIO module and fill the MP3 data buffer. Because the file information should not be written into the DBUF, the state of this FSM is also used to control the enable of the writing to the DBUF.

During the playing process, the file end should be always checked before the READ command is sent. Each time when a block data from FIO module is received, the read byte number is added to a read counter, which is initialized with 0. When the read counter is equal to the file size, the file end is

reached. Thereafter the PLAY FSM should be informed to stop the Monitor FSM and return to the idle state. The file size is a part of the file information and transferred from the FIO module to PLCN when listing the file names.

Be noted that the unit of the file size is byte, but the unit to request data from the FIO module is DWORD. If the file size is not at the DWORD boundary, the packed data in the last DWORD of the MP3 data returned from the FIO module can be safely written to DBUF. The decoder works fine when the number of the garbage bytes is no more than 3.

- **HW CTRL** generates the AC'97 control commands when it receives the corresponding key scan codes.

**Exercise 7:**

Please draw the state machine diagrams of the PLAY FSM and Monitor FSM in the form of Figure 8. The related information can be found in the corresponding specifications.

**Exercise 8:**

Please implement the necessary blocks in Figure 13. These blocks and the blocks in Section 5 should be connected together and tested in hardware. The resulted MP3 player should play MP3 music fluently.

## 6.2 Customizing the MP3 Player

After implementing the simple MP3 player, additional functions can be added to customize the MP3 player individually. The following functions are supported by the the provided modules and the AC'97 hardware.

- Music mute;
- Play pause;
- Forward/Backward
- Change volume;
- Display playing percentage;
- Display playing state;
- Display volume;
- LCD displaying scroll;

**Exercise 9:**

Please implement and test the additional functions above.

## 7 Implementing the MP3 System

In this section the instructions to synthesize the PLCN module and implement the complete MP3 system will be given. Figure 14 shows the implementation structure of the MP3 player. Compared with Figure 2 the FIO and Decoder modules are implemented by the ppc\_core and ppc\_ctrl module together. The sysctrl module implements the clock buffers and reset polarity correction. The test\_modules provides the soft cores for the on-chip logic analyzer.

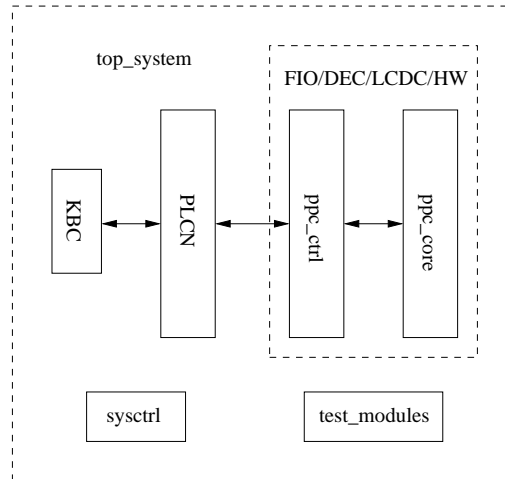


Figure 14: Project implementation structure

### 7.1 Simulating and Synthesizing the PLCN Module

In this section, the instructions to simulate and synthesize the PLCN (*playcontrol*) module will be given.

- Create the *playcontrol* project
  1. Open a terminal, start ISE: *ise&*
  2. Select File → New Project...
  3. Input *playcontrol* as Project Name; *\$HOME/prj/playcontrol* as Project Location; then press *Next*.
  4. Select Family *Virtex2P*; Device *XC2VP30*; Package *FF896*; Speed *-7*; Synthesis Tool *XST (VHDL/Verilog)*; Simulator *ModelSim-SE Mixed*; Preferred Language *VHDL*. Press *Next* three times; Then check your project specification and press *Finish*. The FPGA type is the same as the chip used on the test board.
- Create/Add existing source files to the *playcontrol* project
  1. Click the right mouse button in the *Sources* window. Select *Add Source*. Then select *playcontrol.vhd*. Click *Open* then *OK*. The content of this file can be viewed by double clicking it. The exercises can be directly done in this file, or additional VHDL files can be added in this project for sub-module description, which should be instantiated in *playcontrol.vhd*. Another VHDL source file *\$HOME/prj/comp\_def/system\_constants.vhd* should also be added into the *playcontrol* project. The constants *reset.state* and *clk.polarity* are defined in this file. To view the content of this package file, select the *Libraries* window, then double click the file name under the *work* entry.
  2. To create a new VHDL source file, right click in the *Sources* window, select *New Source*, then input the name of the source file and select the source type as *VHDL Module*. To use the macros from Xilinx, e.g. multiplier and divider, the source type should be *IP (Coregen & Architecture Wizard)*. For detailed instructions to use the macros from Xilinx, please refer the core generator user guide.
- Simulate the *playcontrol* module

In this lab a simple testbench (*playcontrol\_tb.vhd*) is provided to simulate the *playcontrol* module. This testbench tests only the *list* and *play* function. To test the extended functions, more test cases should be added to *playcontrol\_tb.vhd*. The VHDL simulation can be done by directly creating a ModelSim

project. There are tutorials about the usage of ModelSim in *index.html*. Another way to do simulation is to start ModelSim from the ISE project.

1. Add the *playcontrol\_tb.vhd* testbench file to the playcontrol project.
  2. Select *Behavioral Simulation* from the *Sources for:* drop-down list.
  3. Select the testbench in the *Sources* window, then double click the *Simulate Behavioral Model* in the *Processes* window. For further instructions to use ModelSim please refer the tutorials.
- Create the timing constraints
 

Timing constraints should be added just before running synthesis. Timing constraints are used by synthesis tool to judge if a circuit path should be optimized. In digital circuit design, timing constraints mostly stand for clock periods, which are decided by the design specification. The clock frequency for the playcontrol module is 32Mhz, the same frequency as the oscillator Y4 on the VirtexII Pro board used in this lab. To create the timing constraints for the playcontrol module:

    1. Select *Synthesis/Implementation* from the *Sources for:* drop-down list.
    2. Select the *playcontrol* entry in the *Sources* window.
    3. Click the '+' symbol at the left side of the *User Constraints* entry in the *Processes* Window, then double click the *Create Timing Constraints* entry. Then select *Yes* in the appeared window.
    4. In Xilinx Constraints Editor, click the *Global* tab, then double click the *Period* cell in the *clk* entry.
    5. In the clock period editor, specify the clock period as time 31.25 ns, then click *OK* and close the constraints editor.
  - Synthesize the playcontrol module
    1. Select *Synthesis/Implementation* from the *Sources for:* drop-down list.
    2. Select *playcontrol* in the *Sources* window, then right click on the entry *Synthesis - XST* in the *Processes* window, select *Properties*. In the *Synthesis Options* window, select *Xilinx Specific Options* at the left side. Then deselect the *Add I/O Buffers* Option. The reason of doing this is that the playcontrol module is a sub module in the MP3 project and it has no direct external pin on the FPGA. If I/O buffers are generated for a sub module, there will be errors during the implementation of the top level of the design.
    3. Select *playcontrol* in the *Sources* window, then click the '+' symbol at the left side of the entry *Synthesis - XST*, thereafter double click the *View Synthesis Report* entry. This will start the synthesis of the playcontrol module. When the synthesis process is finished, the synthesis report will be displayed in the right side window.
  - Check the synthesis report
    1. No warning like "The following signals are missing in the process sensitivity list".
    2. No warning like "Found latch for signal ...".
    3. Other warning come mostly from circuit optimization and can be safely ignored.

## 7.2 Implementing the Complete MP3 System

In the MP3 system, all the sub modules except PLCN in Figure 14 are given. The MP3 project uses hierarchical design style. Every sub module has its own directory and project. The top\_system design searches and integrates the synthesized sub modules in the specified paths. All the sub modules are connected together in the top\_system.vhd file. **Note: Before running the implementation of the top\_system, the synthesis of the playcontrol module should be rerun, if there is any change in the playcontrol project.**

1. Create a project with the name *top\_system* in the *\$HOME/prj/top-system* directory in the same way as creating the playcontrol project, but do not deselect the *Add I/O Buffers* option in the synthesis properties.
2. Add the *top\_system.vhd* source file to the project.
3. Add  
*playcontrol\_component\_pkg.vhd*, *ps2\_kbc\_component\_pkg.vhd*, *ppc\_core\_component.vhd*,

*ppc\_ctrl\_component\_pkg.vhd*, *sysctrl\_component\_pkg.vhd*, *system\_constants\_pkg.vhd*,  
*test\_modules\_component\_pkg.vhd*

to the *top\_system* project. These files contain the component definitions of the sub modules and the system constants. The directory for these source files is *\$HOME/prj/comp\_def*.

4. Add the macro search paths to the project. Select *Synthesis/Implementation* from the *Sources for:* drop-down list. Select the *top\_system* entry in the *Sources* window. In the *Processes* window, right click on the entry *Translate* under the *Implement Design* entry and select *Properties*. In the *Macro Search Path* cell input *"../playcontrol" -sd "../ppc\_ctrl" -sd "../ppc\_core/implementation" -sd "../ps2\_kbc" -sd "../sysctrl" -sd "../test\_modules"*. These paths are the the locations of the sub modules.
5. Add the *top\_system.ucf* constraints file to the project. The timing constraints and pin locations are actually saved in this constraints file.
6. Create the timing constraints for the following clocks:

Signal Name	Description	Frequency
ext_ace_clk	main system clock	32MHz
ps2_clk	PS/2 keypad interface clock	16.7KHz
bit_clk	AC'97 clock	12.288MHz
ext_cpu_clk	PowerPC clock	100MHz

Table 13: Clock specifications

7. The input/output/inout signals of the *top\_system* design are connected to the other chips on the board through wires. This means the signals of the design should be assigned to corresponding pin locations on the FPGA. The I/O standard and the driving strength should also meet the electrical characteristics of the other chips. Instead of using the graphic interface to fulfill these tasks, a part of the constraints (others are already set in the constraints file) can be added by directly editing the constraints file. To open the constraints file, select *Edit Constraints (Text)* in the under the *User Constraints* entry in the *Processes* window.  
 In *top\_system.ucf*, *LCD\_DB<4>* to *LCD\_DB<7>* and *ext\_ace\_clk* are not assigned to proper locations and standards. These constraints should be added by referring the assignments of the similar signals in the constraints file. The locations for these pins are below:  
*LCD\_DB<4>: P9; LCD\_DB<5>: N2; LCD\_DB<6>: M4; LCD\_DB<7>: R9; ext\_ace\_clk: AH15*  
 Be noted that the clock pin must be assigned to some special pins on the FPGA. Please refer the documents from Xilinx when designing new projects. The definitions and the formats of the different constraints can be found in the Constraints Guide in *index.html*.
8. The synthesis and placement&routing of the MP3 design should be executed to generate the bitstream, which can be downloaded to the FPGA for hardware testing. To run these steps, double click the *Generate Programming File* entry in the *Processes* Window. Thereafter all the implementation steps will be run automatically.

### 7.3 Testing the MP3 Player

1. The following steps should be performed to download the generated bitstream to the FPGA chip for testing. Power on the board (the power switch is beside the SVGA output); run command *./MP3download* in the directory *\$HOME/prj/top\_system*. This script download the bitstream file *top\_system.bit* in the current directory and the PowerPC image for the *ppc\_core* module. After downloading successfully, the MP3 design is ready for testing.
2. During testing the *play* function, if there is no any sound, check if the audio cable is correctly connected and if the loudspeaker of the monitor is muted. If both have no problem, the design of the playcontrol module should be reviewed.
3. The decoded MP3 music files on the Compact Flash card can be found by the link on the lab web page. To play a decoded *cdr* file on PC, using the command: *play filename*.

4. Four LEDs on the PCB board are used to indicate the status of the decoder.  
LED0: dbuf empty; LED1: dbuf almost full; LED2: sbuf empty; LED3: sbuf write
5. If the downloading by issuing the *./MP3download* is unsuccessful, use the following steps to solve the problem:  
Check if the usb cable to the PCB board is correctly connected; issue the command *clean\_cablelock*; if the bitstream still can not be downloaded, turn off the power supply of the board, and turn on it again after some seconds.
6. The IP cores of the on-chip logic analyzer ChipScope is already implemented in *top\_system.vhd*. The ChipScope software can be invoked by command: *start\_analyzer&*. Please refer the ChipScope User Guide for the instructions of using ChipScope.