EE 552 Final Report
Porta-AMP Project

December 8, 1999

Kevin Mlazgar
Edmund Fung
John Koob
Dan Leder

Declaration of Original Content

"The project and the contents of this report are entirely the original work of the authors except as follows:"

sdram_inferface.vhd and associated files are borrowed from Kevin’s M. Cmpe 498 project.

MP3 decoder circuitry is borrowed from Kevin and Dan’s EE 582 project. [4]

_____

_____

_____

_____

Table of Contents

Abstract

Our project entails the design and implementation of an FPGA-based MP3-Audio Player that can have individual MP3-encoded songs downloaded on to ram via a parallel port. Our Porta-AMP™ (Portable **A**dvanced **M**P3 **P**layer) player design would have the ability to play MP3 files from memory. At present all of the primary components are functionally tested. The final integration of our project is not complete due to an inoperative MP3 DSP.

Achievements

Porta-AMP Module

This module integrates the MP3 streaming interface, parallel port interface, and the DRAM interface. At present a song can be downloaded into 8 MB of EDO DRAM and then automatically streamed from DRAM to the MP3 interface that converts the parallel stream to a serial stream that is subsequently sent to the MAS3507D MP3 DSP. The serial data-stream data is valid on the falling edge of a clock (<~1MHz) that the MP3 interface internally generates. The MP3 DSP is supposed to generate I^2S stereo data that is sent to a CS4334 DAC that should then generate a stereo analog signal. At present all signals and data up to the MP3 DSP are as expected, however the MP3 DSP is failing to generate the 16 bit serial data samples that in conjunction with the Left/Right clock will be used by the DAC to generate stereo audio.

Keypad Interface

The keypad interface is complete. It successfully simulates and is implemented without any known problems in hardware. The keypad Interface is able to detect which key is being pressed according to the number of the row and drive line. The external asynchronous key press signal will first go through the sampling register which samples at a period of 1 ms. This should be long enough to wait out any key bounces and synchronize the asynchronous input. Thus sampling will always occur every 1 ms and the key identified if one is pressed. If no key press is detected for over one sampling period it can be determined that no key is pressed or that a key that was once pressed has now been released. Please refer to Appendix C for a description of the test cases and the simulations.

Parallel Port Interface

The PPI module can successfully download MP3 songs to memory using the Enhanced Parallel Port (EPP) mode. Various methods were attempted to obtain downloads with varying degrees of reliability. This included implementing mode negotiation and writing a device driver in C. As well, severe ringing on the parallel port signals had to be avoided.

Attempts to download under Windows using the default printer driver were not successful. This was the case even with mode negotiation implemented according to [1]. This reference did not provide complete information on the mode negotiation protocol. After some investigation, the only document known to provide this information was the standard titled *1284-1994 IEEE Standard Signaling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers*. This document could only be purchased from the IEEE at a cost too high for the budget of this project.

Therefore, a device driver was written in C to download data to the PC. Since a PC running DOS could be set to run in EPP mode by the device driver, mode negotiation became unnecessary. Simply writing each byte of a file to the EPP data register would cause the appropriate handshaking signals to be generated by the parallel port controller.

Another problem successfully solved was that of excessive ringing was observed on the handshaking and data signals originating from the PC. This resulted in data transmissions that would not complete reliably. The hypothesis is that ringing caused the finite state machine to advance unpredictably through some states. This was avoided at first by using counters to introduce delays to avoid the ringing. As well, all handshaking input signals were synchronized to the global clock in registers to reduce the chance of signal transitions violating setup and hold times. The ultimate solution was to implement sampling at the clock frequency of control lines. Due to the fact that the clock had a period of ~40ns, and the parallel port interface operates at least an order of magnitude slower, the control lines were sampled and then once a transition detected, a delay was used to allow the ringing to subside.

In this process, a number of handshaking signals were omitted to reduce the complexity of the state machine. Since the other interfaces operated at speeds higher than the bit rate of the parallel data, there was no concern regarding data overruns. Please refer to Appendix C-PP1 for a description of the test cases and the simulations.

Memory Management Interface

This module (mem_mgmt_cont.vhd) handles the interaction between the main module of the PORTA-AMP and the DRAM Interface (dram_int.vhd). It implements memory framing and internally keeps track of all DRAM addressing. All the calling module can do is write or read a single song sequentially to or from memory. All reads by the calling module will be ignored unless a song has previously been written. At present complete DRAM refresh (CBR) circuitry with a period of ~10us is implemented and 512KB of error-free DRAM is interfaced via an 8MB DRAM SIMM. Only 512KB of DRAM is available due to bad address lines. The data throughput (ignoring delays due to refreshes) to DRAM is: READ – 1.8 MB/s, WRITE - 2.12 MB/s. The throughput can be increased threefold if the aggressive delays that are inserted in the DRAM Interface module are removed.

The DRAM interface interfaces a Micron 8MB EDO SIMM that has 16 DRAM devices mounted on it. The DRAM devices are already in groups of four as defined by RAS0#, RAS1#, RAS2#, RAS3#. To make a 16-bit datapath the RAS signals are used as bank selects and all the CASN lines are physically connected together. There are ten address lines (A0-A9) which are used in conjunction with the four RAS# (Row Address Strobe) and a common CAS# (Column Address Strobe) to time multiplex the 20 address lines (in addition to the RAS# bank selects) required to address 8MB of ram with a 16 bit word size. Only normal accesses to memory are be utilized, no EDO or FPM. The RAS# and CAS# and WE# signals are asynchronous edge triggered and not tolerant of glitches. The only effective way found to remove glitches was to have the asynchronously read outputs generated in processes that are sensitive only to the rising edge of the clock, effectively registering them. Once in the process a decision to output a signal is made based on the value of the state when the rising edge occurs. This has the effect of delaying the asynchronous outputs by one clock period. It was feared that hold time violations would occur by checking a state that could be about to change after a rising edge. However, during timing simulations this never occurred and the actual implementation encountered no known problems in this regard.

Framing is implemented in the Memory Management Interface as a means of increasing the registered performance of the module. This is due to the 22 bits of internal addressing required to address 8MB of DRAM with a word size of 16 bits. With framing, the addressing is divided into a 12 bit frame address which is not incremented until the associated 10 bit offset address has wrapped. This means two medium-sized ripple-carry adders can be used instead one large 22 bit ripple-carry adder. Although it is possible to get a 22 bit ripple-carry adder to operate in the EPF10K20RC240-4 FPGA at over 30 MHz, once any logic is attached to the decision to increment a counter based on the adder the speed drops drastically to <25 MHz.

During implementation only the first 2^7 addressing of words were found to be error free. The trouble was isolated to address lines A8 and A9. Due to the fact that there are only ten time-multiplexed address lines to the DRAM of which the upper two are bad, it was determined that absolute address lines 8, 9, 18, and

19 were effectively useless. A simple way of circumventing this problem without attempting pin reassignments on the FPGA or rewiring the complete circuit was to utilize the split nature of the addressing in the Memory Management Interface to permanently set address lines 8, 9, 18, and 19 (which are fed to the DRAM interface) to ground.

Two hardware-based test bench modules were created to exercise the actual DRAM SIMM. One (dramtest_simm.vhd) interfaces directly with the DRAM Interface module (dram_int.vhd) and the other (dramtest_memcont.vhd) interfaces with the Memory Management Interface (mem_mgmt_cont.vhd.)

The SIMM test bench keeps an internal counter that it uses as both the address to write to and the test vector to write. Once the data is written the module will reset the count to zero then proceed to read from the memory locations, once again using the count as the address. The count is then also used to verify the read data with the data that was previously written to that location. The count is used to limit the amount of memory tested. If desired, the length of the count can be modified if the width of the signal is changed and then the module recompiled. The count must be padded with zeros (or ones) to 16 bits to generate the test vector that will be written to the address location. The count is padded with zeros to 22 bits to generate the address to write and read from. An initial loop writes all the vectors to their respective addresses. A second loop then reads from those addresses and verifies that the result is that expected. If an error is detected the address location is displayed on the 7 segment display. Sixteen bit values are displayed on the two 7 segment displays by displaying first the two high order HEX digits then the two low order HEX digits. When the high order HEX digits are being displayed, the decimal points will be turned on. At the end of the memory test the total count of errors is displayed on the 7-segment display (it will blink at twice the rate of that when the bad addresses are being displayed.)

The Memory Management Interface test bench also keeps an internal counter as well as an error counter. However, since the Memory Management Interface abstracts all addressing from the modules that use it, the count is only used as the test vector as well as to verify that validity of the read data.

These test benches were invaluable when testing the DRAM and identifying the address lines A8 and A9 as being the root cause of data errors. Please refer to Appendix D for the DRAM test bench code and simulations.

CD-ROM Interface

The basic architecture of the CD-ROM Interface has been designed and implemented in VHDL. The simulations show that the CDI is able to receive data from the CD-ROM modeled after the handshaking signals defined by the ATAPI specification. Accomplishments include the fact that the CDROM drive is able to enter the hardware reset mode and power-on mode. However, it will not transfer data out from the device's buffer. Due to time constraints and the fact that the project board was frequently used to test other modules, there was insufficient time to test the CD-ROM interface in hardware.

LCD Interface

The LCD module has been simulated and tested using a physical keypad and LCD component. Various command displays can be shown on the LCD in response to key presses on the keypad. Both simulation and hardware testing function as specified. One notable achievement was that the number of logic cells required by the LCD module was reduced from 577 cells to 221 cells, which is a 62% reduction in module size. In addition, the logic cell count is independent of the number of screens. This was accomplished by using ROM lookup tables that would carry the LCD character data, user-defined command set and display addresses. The MIF file defined in the EAB significantly decreased the amount of code in the LCD controller program because logic blocks are no longer needed to store the LCD displays. Please see Appendix C for simulations.

MP3 Interface

The MP3 interface has been implemented and simulated. The simulations work without known problems. The MP3 decoder chip and wiring has been tested using a module used in [4]. The VHDL module implementation has been verified to work in hardware.

One main problem is that a <~1MHz clock needs to be created from a 25MHz clock that would still allow the rest of the circuit to run at the full 25MHz clock. That is, a 1MHz clock is needed with rising and falling edges of equal distance. It was also necessary not to adversely impact the rest of the circuit and create a reliable and fast clock divider. In addition, a clock divider that wasn't limited to powers of two was desirable.

The solution to the problem was to create a clock using the carry-save counter developed in LAB 3 of EE 552. This carry-save counter would ensure that if the divisor needed to be increased, there would be little detrimental effects to the clock. For example, if we wanted to decrease the clock to 500kHz there would there would detrimental effects to the clock if we were to use a ripple carry counter. The solution to creating the fully function clock was to have the divisor cut in half such that for half of the cycle the new clock would be '1' and for the second half the value would be '0'. Please see Appendix C for test cases and simulations

Implementation and Design

Summary of Operation

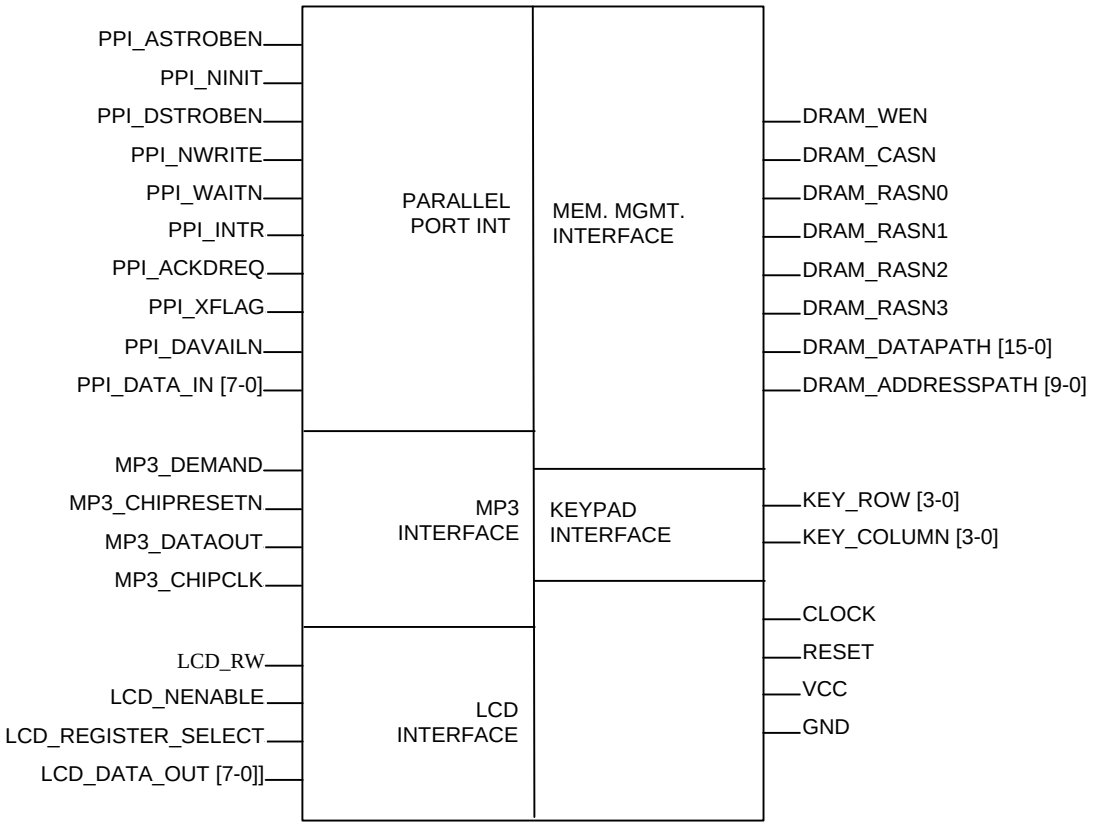
The Porta-AMP player should allow the user to download, stream and play MP3 files. If these files reside on a PC, the player downloads them through the parallel port and stores them in DRAM. If the song files were stored on a CD-ROM, the CD-ROM interface could be used to import the song files from disk into memory. Another option is to stream the song data directly from the parallel port or CD-ROM to the MP3 decoder. A song in memory could be played or deleted using the user interface consisting of a keypad and LCD. This interface is also used to select a song and to view song names. Essentially, operation is similar to commercial MP3 players and CD player with the enhancement of streaming. Please refer to Appendix F for more details about the operation of Porta-AMP.

Data Sheet

See Appendix A for the system-level Porta-AMP product datasheet. This datasheet discusses only external connections, maximum ratings and electrical characteristics of the Porta-AMP prototype as if it were a commercial product.

The following pages serve as an FPGA-level datasheet that describes internal FPGA signals. Please refer to available documentation for the EPF10K20RC240-4 FPGA for the electrical specifications and timing parameters of this device.

Figure 1 - FPGA Pinout Diagram



The pins in Figure 1 represent external pins from the Altera EPF10K20RC240-4 FPGA. The pins on the left hand side are connected to the FLEX_EXPAN_A header. The pins on the right hand side except for the keypad pins are connected to the FLEX_EXPAN_C header. The clock and reset signals do not appear on either header, while reset and ground appear on both headers.

Table 1 lists all external connections to the FPGA pins and their descriptions. Pin numbers not shown have no external connection. Note that the pin numbers shown do not represent the hole numbers on the headers on the UP1 board. Also, not that signal vectors start pin numbering from the least significant bit. It is important that a common ground be made between the UP1 board and the prototype board. Signal names also can be found in the top-level architectural file called porta_amp.vhd, which can be found in Appendix H.

Pin Label	Pin Direction	Pin Number	Pin Description
Key_row	in (3 downto 0)	74, 76, 79, 81	input from keypad – row vector (keypadsize = 4)
Key_column	out (3 downto 0)	75, 78, 80, 82	output to keypad - column vector for keypad
lcd_data_out	out (7 downto 0)	86, 87, 88, 94, 95, 97, 98, 99	Output to lcd – datapath (lcd_data_out_width = 8)
lcd_register_select	out std_logic	67	Select whether accessing command or data register
lcd_rw	out std_logic	83	Read or write to LCD
lcd_nenable	out std_logic	84	Select device
ppi_data_in	in (7 downto 0)	45, 48, 50, 53, 55, 61, 63, 65	Input from parallel port
ppi_nastrobe	in std_logic	46,	Mode negotiation signal
ppi_ninit	in std_logic	49	Initialization signal
ppi_ndstrobe	in std_logic	51	Sent from PC, latch data that came in
ppi_nwrite	in std_logic	54	Data coming in
ppi_nwait	out std_logic	56	Sent to PC to tell it wait or that ready to latch
ppi_intr	out std_logic	62	Mode negotiation signal
ppi_ackdreq	out std_logic	64	Mode negotiation signal – negotiation complete
ppi_xflag	out std_logic	66	Mode negotiation signal - confirmation of mode
pi_davailn	out std_logic	68	Mode negotiation signal
dram_wen	out std_logic	231	Write enable active low
dram_cas	out std_logic	230	Column Address Strobe – last part of address latched on falling edge
dram_ras	out (3 downto 0)	226, 227, 228, 229	Row Address Strobe – first part of address latched on falling edge
dram_datapath	inout (15 downto 0)	191, 192, 193, 194, 195, 196, 198, 199, 200, 201, 202, 203, 204, 206, 207, 208	16-bit bi-directional datapath
dram_addresspath	out (9 downto 0)	214, 215, 217, 218, 219, 220, 221, 222, 223, 225	10-bit address path
mp3_done	in std_logic	70	input from MP3 chip - mp3_decoder requires data (1)
mp3_chipclk	out std_logic	73	output to MP3 chip - new clock ~1MHz
mp3_chipresetn	out std_logic	71	output to MP3 chip - reset the MP3 decoder chip
mp3_dataout	out std_logic	72	output to MP3 chip - output a single bit which is valid on falling edge of mp3_chipclk
Vcc	N/A	N/A	Supply voltage
Gnd	N/A	N/A	Ground
clock	in std_logic	91	Global system clock
reset	in std_logic	28	Global system reset

Table 1 – FPGA Pin Descriptions

Design Hierarchy

As is shown in Figure 2, the major components of the system architecture have been designed, implemented, and simulated. Cells with dashed lines represent modules not yet compiled. Cells with solid lines are the modules that have compiled without errors but have not been simulated. Shaded cells represent modules that compile and simulate with expected behavior.

The Porta-AMP Module currently compiles and simulates without errors. This module integrates the MP3 streaming interface, parallel port interface, and the DRAM interface and is functioning in hardware. At present, a song can be downloaded into DRAM and then automatically streamed from DRAM to the MP3 interface. Refer to the Testing and Simulation section for more details on currently implemented features.

The Parallel Port Interface has been completely designed, implemented and simulated. A simplified version of the PPI module was synthesized and tested with the I/O connected to a PC parallel port, which used a C program as a device driver to transmit data to the Porta-AMP using EPP. Various versions of the PPI module have been successfully tested in hardware. Refer to Appendix C for more information on simulation and testing.

The LCD Interface is functional; the design compiles, simulates without errors, and works with an actual LCD. Recent work was successful in reducing the size of the synthesized logic. See the following section on the logic cells.

The MP3 Interface and Keypad Interface have been completely designed and simulated. The MP3 interface successfully converts the parallel stream to a serial stream that is subsequently sent to the MP3 DSP. This behavior has been verified on an oscilloscope, in spite of the inoperative MP3 DSP. The Keypad Interface has been functionally implemented and is known to work.

Details of the Porta-AMP architecture are presented in the Detailed Design Hierarchy below. As is shown, a central Master Control Module should coordinate six interface modules. Note that abbreviations of internal signals are included while external signals are excluded. For more information, the architecture of the originally proposed design is outlined in the Internal Interface Diagram in Appendix G.

For a complete overview of the architecture of our system with both external and internal signals with respect to the FPGA, please see the previous section and porta_amp.vhd in Appendix H. This VHDL file is the top-level architecture file that structurally connects most of the modules in the Design Hierarchy diagram. If one looks at its entity, all the external I/O pins to the FPGA are defined.

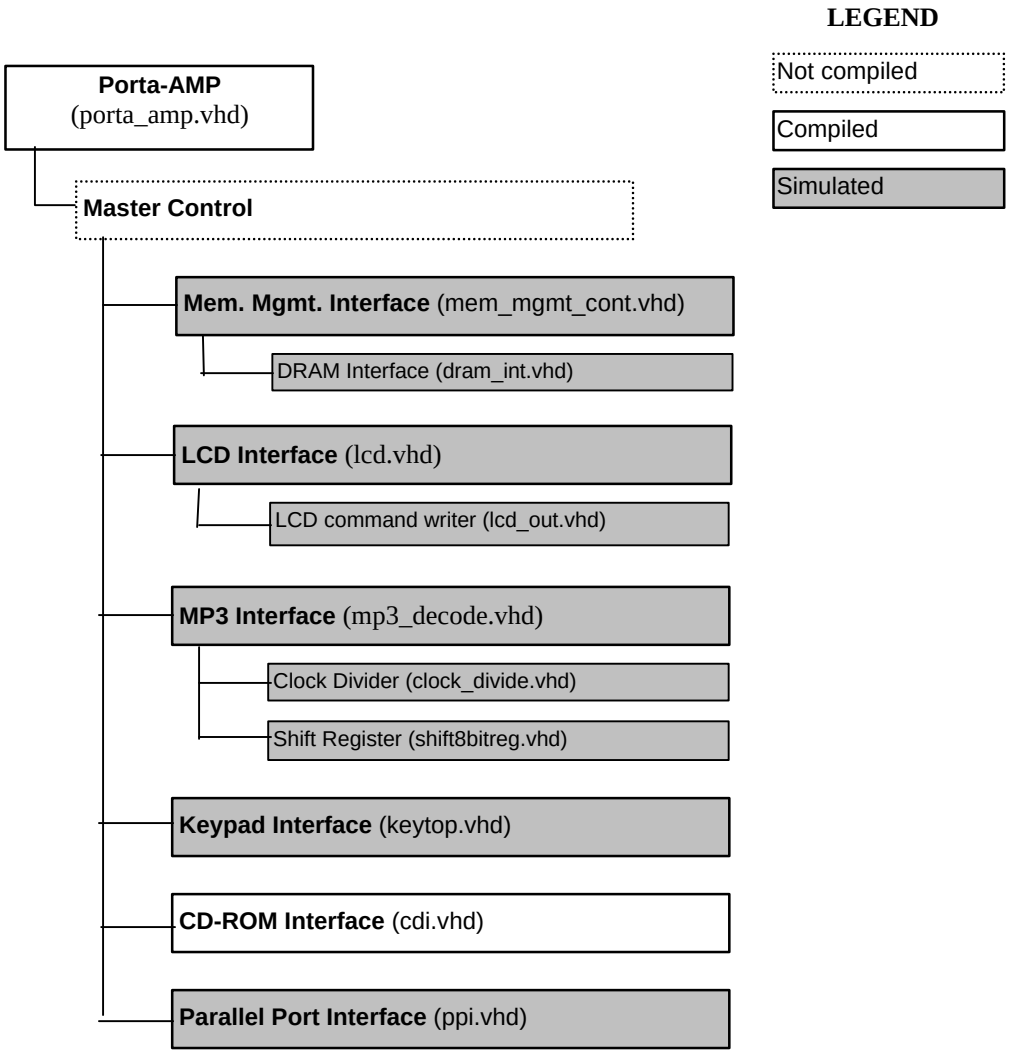


Figure 2 – Design Hierarchy

Logic Cells

Table 2 indicates the number of logic cells found in the fitting report for each module. Also included is an estimate of the numbers of cells a complete module would require. The percentage of the total available logic cells each module would require is also listed. As shown by the totals, the design will fit on one FLEX10K20 FPGA according to the total sizing requirements of the simulated modules. The estimated size requirements shows that the design would still fit in the FPGA, but only 9% of the total number of logic cells would be unused.

Module	Simulated	
	No. of Cells	**Percentage**
Keypad Interface	74	5 %
Parallel Port Interface	111	9 %
Mem. Mgmt. Interface	258	22 %
LCD Interface	221	19 %
MP3 Interface	86	7 %
TOTAL	**627**	**51 %**
PPI, MP3, Mem. Mgmt	544	47 %
LCD and Keypad	~390	34 %

Table 2 – Simulated and estimated logic cell count

Earlier in the project, the number of logic cells taken up by the LCD component was 577 out of 1152 on the FPGA, which amounts to 50% of the total capacity. The reason for this incredibly huge size for a seemingly simple implementation is the size of the arrays used to hold LCD character data and instructions. An array of 32x8 characters was used to hold the display data at every state of the LCD system. Because the LCD is a 16x2 screen, 16 x 2 = 32 characters are required, and each character requires 8-bit encoding in the LCD. Every character is defined as a constant at the start of the code. It is assumed that every bit of the character data is fed into one large multiplexer that has too many configurations to base its decisions on. Thus, the amount of logic cells that is required for the algorithm grows. The solution to this was to use ROM lookup tables that would carry the LCD character data. The MIF file defined in the EAB significantly decreased the amount of code in the LCD controller program because logic blocks are no longer needed to store the LCD displays. The size of the LCD code was reduced to 221 from 577, which is approximately 19% capacity of the FPGA.

The logic cell count other modules did not increase significantly from the Simulation Documentation. The memory management interface increased from 178 to 258 cells during the effort to get DRAM functional. Similarly, the parallel port interface increased from 68 to 111 cells since counters needed to be added to introduce the delays that avoid ringing. The MP3 Interface, LCD Interface and Keypad Interface did not have any increase in logic cells during the last phase of the project since these modules were completed earlier.

With PPI, MP3, and DRAM modules integrated, the logic cell count is 544 or 47 %.

Problems, Experiments and Solutions

Few experiments such as numerical simulations or using synthesis speed-versus-area trade-offs were necessary for this project.

One notable experiment done was changing the optimization of speed vs. area for the dram_ppi_test.vhd module. Optimizing for speed decreased the number of logic cells by 12 (from 452 to 440) and increased the speed from 4.6MHz (from 30.7MHz to 35.3MHz). This is unexpected since logic cell count should not normally decrease when speed increases.

For the same module, another interesting modification had a ripple-carry counter was increased from 5 bits to 10 bits. This increased the number of logic cells by 5, as expected. Inexplicably, the speed increased to 37.3 MHz.

However, a number of problems had to be solved when the simulated VHDL did not work in hardware. This section will outline each problem encountered and present the solutions devised.

Problem: Initial LCD module was too large

The original problem with the LCD was that it was necessary to create multiple screens that would be static such that they would always display the same message. There were multiple (20+) message screens that had to be implemented. However, when we created the LCD module it wouldn't compile in a reasonable amount of time. In order to get the original LCD module to compile, we needed to scale back the number of screens to 3 and it required about 800 logic cells. The original design had all the character constants and command constants defined. Building a string of constants and writing them to the screen created a static message.

Solution: Use the embedded array blocks

We solved the problem of the number of logic cells being used by implementing ROM using the EABs on the FLEX10K board and by creating a MIF file. This significantly reduced the compile time and the size of the module. The number of logic blocks was reduced to about 200 and could support all screens.

Problem: Displaying a Screen on the LCD using ROM

This problem involved creating a screen such that the module could write the desired sequence to the LCD without having to compare every character each time. This comparison needed to be done to determine if it is a character to be displayed or a command to alter the LCD's display characteristics.

Solution: Label the LCD data

It was decided that to alleviate the need for checking every piece of data, an extra bit was needed to label the type of data. This bit was used to indicate whether the data in the ROM address location was a command or a character that needed to be sent to the LCD. Since the LCD already had an interface line that would indicate where the data was to be sent, we used this as the 9th bit of the data in ROM. When the 9th bit was a '1' then the data was a character command, and when the bit was a '0' then the data was a command to change the LCD (e.g. clear screen). However to make it easier to code the data, we extended the number of bits used by ROM to 12bits. This would allow us to use a hexadecimal radix for the data, and would alleviate the necessity to evaluate a binary number to determine the ASCII data value.

Problem: Telling the LCD Module when Screen was Complete

The next problem of knowing when a screen we had reached the end of the screen that was being displayed had to be addressed, since we would next be trying to implement multiple screens in the same ROM. We would need some way to indicate that we had finished a screen so that we would not continue writing out another screen to the LCD.

Solution: Create an end-of-display character

A constant that indicated the end of a display was created. It was decided to use "0x100" since this corresponds to the null character "0x00" with the necessary 9th bit to tell the LCD module that the data is a character. This null character is not unlike a null character in a C string.

Problem: Multiple Screens and Not Altering LCD Module

The next problem was how to separate all the screens and quickly reference them. This needed to be done without having to create a LCD module that would have to be hard coded with memory addresses that might change if the user decided to alter the screens. Once the general design of the LCD module was complete, it was not wise to make extensive changes to the MIF file to add an extra screen. As well, the LCD module should not have to be changed if an extra screen was added.

Solution: Use a double addressing system

A double addressing system was developed. That is, we made the first part of ROM the screen display number whose data contained the address to the screen that we wanted to display. For example, if we wanted to display screen 13, then the LCD module would go to memory address 13 where it would find the address to screen 13. After reading the address of screen 13, it would go to this address and begin writing the contents of the address to the LCD and then would increment to the next address, and continue.

Also, there had to be an initialization routine added to setup the LCD to display correctly. It was decided that this would be mode zero and that the user couldn't access this display mode other than by resetting the LCD module. This was done to prevent the chance of the user accidentally re-initializing the display and lose what was being displayed. In addition, the LCD module was designed so that it would wait for an initialization time before it would let the user's system proceed to communicate with the LCD. This was to ensure that the LCD was indeed ready to accept commands and not still initializing itself. It was also necessary to create minimum delays between writes to the LCD since the LCD had its own timing requirements.

In developing this LCD module, a universal module was created such that anyone could take this module and merely update the LCD.MIF file to their specifications.

Problem: Creating the 1MHz MP3 clock using clock division

The main problem is that a true 1MHz clock needs to be created from a 25MHz clock that would still allow the rest of the circuit to run at the full 25MHz clock. That is, a 1MHz clock is needed with rising and falling edges of equal distance. For the decoder, we needed to have data valid on the falling edge of a clock and we needed to have the data there for a long enough time that it would be stable. The rising edge of the clock must be used to trigger a shift register and the falling edge must indicate valid data. It was also necessary not to adversely impact the rest of the circuit and create a reliable and fast clock divider. In addition, a clock divider that wasn't limited to powers of two was desirable.

Solution: Create a clock using a carry-save counter

The solution to the problem was to create a clock using the carry-save counter developed in LAB 3 of EE 552. This carry-save counter would ensure that if the divisor needed to be increased, there would be little detrimental effects to the clock. For example, if we wanted to decrease the clock to 500kHz there would there would detrimental effects to the clock if we were to use a ripple carry counter. The solution to creating the fully function clock was to have the divisor cut in half such that for half of the cycle the new clock would be '1' and for the second half the value would be '0'.

Problem: Obtaining CD-ROM Interface information

When researching on the technical specification of a CDROM drive or on how to interface a CDROM device, it can be difficult to obtain detailed information. Once this information is found, there are nearly four hundred pages of technical information and it can be difficult to sort out the relevant information.

Solution: Find the ATAPI specification from the T13 Technical Committee

In particular, one should avoid focusing directly on the term "CDROM Interface", especially when doing web searches, as it is almost impossible to find useful information about CDROM Interface that way. Instead, a research on ATAPI, IDE or SCSI controller has proved to be the most effective approach for gathering design specification. In the Porta-AMP project, the draft document **d1321r1c** from Technical Committee T13 is used as a reference for the CDROM Interface design. The document contains 380 pages of information and specifications of ATAPI between a host system and storage device. To save time, the designer suggests one to begin reading at Chapter 5 (Internal Signal Assignments and Descriptions) to understand the function of each signal on the 40-pin connector of the CDROM drive. Next, one can study Chapter 9 (Protocols) to learn how a CDROM drive or hard drive process through different states in order to perform the correct operation of the device. Finally, Chapter 7 (Internal Register Definitions and Descriptions) and Chapter 8 (Command Descriptions) should be carefully studied to determine the correct sequence of read/write commands to be employed by the interface design.

Problem: Parallel Port IEEE-1284 Mode Negotiation

Part of the IEEE-1284 standard states that the PC and peripheral should negotiation the mode before data downloads can begin. It was decided to implement the mode negotiation phase for this project so that the default Windows printer driver could be used to download data to the Porta-AMP. In this way, no device driver would need to be created for the project.

However, all attempts at working with the mode negotiation used by Windows were futile. This was caused by a lack of understanding of the complete IEEE-1284 standard, since available documentation only summarized the information. After some investigation, the only document known to provide this information was the standard titled *1284-1994 IEEE Standard Signaling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers*. This document could only be purchased from the IEEE at a cost too high for the budget of this project.

Solution: Create custom device driver

The solution to this problem was to control both ends of the handshaking process. In other words, a device driver was created in C to be run under DOS. This device driver configured the parallel port to operate in EPP mode. The inportb() system call were used to read particular status register bits so that the current parallel port status could be determined. The outportb() system call was used to set bits in the control register write data to the EPP data register. This device driver ensured that the PC used the EPP mode so mode negotiation became unnecessary.

Problem: Unpredictable behavior of Parallel Port Interface

The state machine used in the parallel port interface did not reliable handle the download of data. It was initially suspected and eventually verified utilizing an oscilloscope that these errors are caused by full-swing ringing on the line. This ringing lasted between 100 and 300ns. Since this ringing occurs on all the parallel port data and asynchronous control lines, unpredictable behavior of the state machine was the result.

Solution 1: Insert damping resistors in series with all lines

To reduce the ringing, 50-100 ohm resistors could be inserted in series with the lines in question. This solution was discovered after the following solution had solved the problem.

Solution 2: Synchronize and sample input signals

The ultimate solution was to implement sampling at the clock frequency of control lines. Due to the fact that the clock had a period of ~40ns, and the parallel port interface operates at least an order of magnitude slower, the control lines were sampled and then once a transition detected, a delay was used to allow the ringing to subside. Care was taken to ensure that this delay was significantly smaller than the minimum specified period of the control line. After this delay, the relevant data could be latched safely from the data lines, since their ringing would also have subsided.

This method was used for both high-to-low and low-to-high transitions. Reliable data transmission was visually verified in conjunction with the DRAM test bench (dramtest_ppi.vhd).

Problem: SDRAM Voltage Issues

SDRAM devices including those on the PC100 Micron DIMM chosen for this project utilize only LVTTL I/O. In other words, only 3.3 V interfacing is allowed. The EPF10K20RC240-4 FPGA from Altera is 5V with only 5V I/O. The Flex 10K family supports multi-volt I/O, which would allow for interfacing to SDRAM. Unfortunately, the 240-pin PQFP package on the UP1 development board was the only device in the Flex 10K family, which does not support multi-volt I/O in order to reduce the pin count. This information was found from the information in the FAQ at Altera's website.

As a result, the interface to SDRAM became significantly more problematic.

Attempted Solution 1: Utilize Max 7000S on UP1 board which has multi-volt I/O

One of the possible solutions to the SDRAM voltage issue was to utilize a FPGA, which provided multi-volt I/O. One such device is the EPM7128SLC84-7 Max7000S device located on the UP1 development board. Unfortunately, this device only has 128 logic cells and the present implementation of memory management controller requires 258 logic cells. This makes the EPM7128SLC84-7 device useless for this purpose.

In addition, the "designers" at Altera chose to hardwire the VCCIO pins on the EPM7128SLC84-7 device to 5 V. This is unfortunate since the VCCIO pins are the input pins to the second rail required for multi-volt I/O and should be connected to 3.3 V if one wishes to interface to a 3.3 V device. An 84-pin PLCC socket was purchased for the sole purpose of wiring up the EPM7128SLC84-7 device external to the UP1 board. Due to the complexity and frailty of the wiring required, this was eventually discarded as a solution.

Attempted Solution 2: Utilize 3.3V tolerant buffers and transceivers

The EPM7128SLC84-7 could readily be implemented as a dual-rail 5V to 3.3V buffer. In addition, the Toshiba TC74LCX245 3.3V, 5V-tolerant I/O transceiver could be used in conjunction with the FPGA to bi-directionally buffer the data lines while the FPGA buffers the control and address lines. Unfortunately, with 23 control and address lines as well as 16 data lines, this results in a complex wiring network which would be difficult to troubleshoot and susceptible to noise.

Attempted Solution 3: Utilize a larger FPGA with multi-volt I/O

Instead of using the EPM7128SLC84-7, an FPGA such as the multi-volt tolerant EPM7256ATC100-7 could be used to implement the SDRAM interface external to the EPF10K20RC240-4 on the UP1 development board. Unfortunately, the EPM7256ATC100-7 is a TQFP 100-pin surface mount device with 0.5 mm pin spacing. In other words, a custom PCB design would be required. Due to time constraints this solution along with solution 1 were not considered to be feasible.

Ultimate Solution: Utilize legacy 5V EDO DRAM

To solve the above problem and obtain adequate memory storage for our project, it was decided to forgo the SDRAM interface. Instead, it was decided to design an interface for a legacy 8 MB EDO SIMM from Micron, which operates at 5V TTL levels. In this way, all address and control lines could be directly connected to EPF10K20RC240-4 FPGA.

Problem: DRAM data errors

Upon initial implementation of the DRAM interface, only the first 2^7 words of data could be written and subsequently read without errors. It was suspected that ringing on the line caused the errors. Data errors were detected by downloading data from the parallel port and viewing the data on 7-segment displays once it was in DRAM. Once the errors were detected, two separate hardware-based test benches were created to write and read data from DRAM and indicate any bad address that they came across along with the final count of words in error. The first test bench tests the DRAM interface module to the SIMM directly (dramtest_simm.vhd). The second tests the memory management interface which itself instantiates the DRAM interface module (dram_test_memcont.vhd)

Solution 1: Insert damping resistors in series with all lines in the DRAM interface

When utilizing the oscilloscope to view the control lines, a considerable amount of ringing after signal transitions was observed. Upon closer examination it was seen that the ringing was barely below the threshold voltage for an active low signal. To reduce the ringing 50-100 ohm resistors could be inserted in series with the lines in question. However, when it was considered that the oscilloscope probe itself introduced capacitance on the measured line that could make the ringing appear worse that it really was, this was considered a last resort and was not attempted.

Solution 2: Eliminate 5-inch 40-pin ribbon cable between DRAM and FPGA

The 40-pin ribbon cable between the DRAM and the FPGA was removed. Instead, the DRAM daughter card was plugged directly into the UP1 header. This resulted in much more reliable data transmission above 2^7 words of addressing, but did not remove all errors.

Solution 3: Utilize different I/O pins on the UP1 board

This solution was not attempted, but would be considered if time was not a factor.

Solution 4: Modify DRAM interface code to insert delays

After the transition of data lines and address lines, delays were inserted before any control lines (such as RAS#, CAS#, WE#) were asserted. It was hoped that this would reduce the data error rate, which it did to a certain extent when this solution was implemented.

Solution 5: Isolate bad address lines

Utilizing the hardware-based test bench programs, it was determined that all errors occurred when address lines A8 and A9 were in use. To circumvent this, the memory management interface which implements framing was modified so that those address lines were permanently driven low. Due to the time-multiplexing nature of the addressing to the SIMM this resulted in the net loss of 4 address lines, leaving a total 512 Kbytes of addressable and reliable DRAM. Note that the initial size of DRAM SIMM was 8 MB.

Problem: Questionable MP3 DSP chips and Fried DACs on the side

On more than one occasion, it was found that the CS4334 DACs wired on the board ceased to operate. Applying a voltage double its rated maximum damaged the first DAC. This is due to the ATX power supply having numerous power rails, ranging from –12V to 12V. Unfortunately, a sleep-deprived and slightly color-blind student used the 12V rail as a 5V rail. This proved to be too much for the DAC to handle. The next DAC used in the Porta-AMP appeared to have been damaged from earlier uses, so this one was discarded as well. A third DAC has been obtained and is currently wired on the board.

The MP3 decoder chip appears to be malfunctioning when supplied with data. This conclusion was reached because it is not possible to obtain an I^2S data stream from the DSP. A possible reason for this may be that when the DAC was damaged there may have been enough back voltage to cause damage to the DSP chip. Also, when we replaced the DSP that appeared to malfunction it turned out that the other DSP was also probably damaged. It was not certain as to whether the second DSP was operative, and switching to this DSP confirmed our suspicions that it was defective.

Solution: Unknown

These unfortunate incidents occurred within the last several days of the project. No time was available to develop workarounds. Very sad indeed.

References

1. Axelson, Jan, *Parallel Port Complete*, Lakeview Research, Madison, WI, 1997, pp. 113, 200-284
2. Diamond RIO: n. pag. On-line. Internet. 23 Sept. 1999 Available: http://www.diamondmm.com/rio/
3. eSafe – EE552 Project: n. pag. On-line. Internet. Available: http://www.ee.ualberta.ca/~elliott/ee552/projects/98f/esafe/
4. Leder, Mlazgar, and Carstensen: MP3 player – EE582 Project: Available: Dr. N. Durdle (CEB238)
5. Fraunhofer Gesellschaft (1999): n. pag. On-line. Internet. 10 Apr. 1999.Available: http://www.iis.fhg.de/amm/techinf/layer3/layer3faq/index.html
6. Interfacing the PC (1999): n. pag. On-line. Internet. 12 Oct 1999.Available: http://www.geocities.com/SiliconValley/Bay/8302/parallel.htm
7. MAS3507D Preliminary Data Sheet: n. pag. On-line. Internet. 11 Apr. 1999 Available: http://www.micronas.com/pdf/mas3507d.pdf
8. MP3-2000 (1999): n. pag. On-line. Internet. 10 Apr. 1999. Available: http://www.mp3-2000.com/tutorials/whatismp3.shtml
9. MPEG (Moving Picture Experts Group): n. pag. On-line. Internet. 23 Sept. 1999 Available: http://www.cselt.stet.it/mpeg/.
10. Whitney Master's Thesis by Björn Wesén (1999): n. pag. On-line. Internet. 10 Apr. 1999 Available: http://www.sparta.lu.se/~bjorn/whitney/whitney.zip
11. WINAMP: n. pag. On-line. Internet. 23 Sept. 1999 Available: http://www.winamp.com.
12. AppleCyber: ACPlay: n. pag. On-line. Internet 11 Jan. 1999 Available: http://gussie.alaska.net/acplay/index.spml Note: link is dead
13. Welcome to t13.org; n. pag. On-line. Internet Available: http://www.t13.org
14. Zm06.p65 and 64MSDRAM.p65 On-line. Available: http://www.micron.com
15. "How to Write to an LCD With Minimal Pain and Anguish". n. pag. On-line. Fall 1997 Student App Notes. Available: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/97f/lcd/lcd.html
16. "LCD Driver Application Note". n. pag. On-line Fall 1998 Student App Notes. Available: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/98f/LCD_driver/
17. "Using an LCD in Hardware Projects". n. pag. On-line Winter 1998 Student App Notes. Available: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/98w/Liquid_Crystal_Display/

Appendix A – Porta-AMP Product Datasheet

Appendix B – Datasheets of Components

Appendix C – Test Cases and Verification

Keypad Interface

The functionality of the keypad interface needed to be tested in two stages. The first is to see if the decoder will correctly detect which key is pressed. Secondly, the de-bouncer needs to be tested to see if it can provide a slower sampling period to the key decoder.

Due to the limitation of the MaxplusII simulator, (i.e. its inability to realize the actually behavior of the electrical circuit of the keypad), it is very difficult to test the design using a simulator. Therefore, the test case of the keypad design was only setup to test the two areas mentioned above.

In the test, the programmer wants to ensure the keypad decoder is able to detect key "4" being pressed with a sampling period of 500ns from the de-bouncer. According to the simulation waveform diagram, the de-bouncer is able to correctly feed the keypad decoder the key press input after 25 counts. With a clock period of 20ns, the de-bouncer released signals to the decoder after approximately 500ns and the decoder correctly identify the key press as "4". See Appendix F for the simulation diagrams.

Test Cases

1. The test case only simulates the cases for drive-1 state
2. When state = drive 1 => key_column = 0111
3. Key input is sampled after 25 counts plus delay (wait for state change)
4. Noise before to the input is less than a half clock cycle thus will not affect the decoding process
5. Key press "8" is detected, key_data output to the master control and the machine goes to the foundkey state until a reset or key release.
6. Next key input "12"is detected
7. Next key input "4"is detected
8. Next key input "0"is detected
9. The signal data_valid is output to the master control whenever a key press and sampled and detected

Parallel Port Interface

The Parallel Port Interface receives 8-bit asynchronous input data from the parallel port of a PC using the Enhanced Parallel Port protocol. Handshaking signals are used between the PC and the PPI module.

During normal operation, the Master Control module asserts the ppi_download signal to indicate that it is ready to receive a word of data. The PPI module responds by bringing the ppi_ready signal low. At the appropriate time during the handshaking sequence, the first byte on the data inputs is loaded synchronously into an 8-bit output register using a register enable. When the second byte is received, it is loaded into a second 8-bit output register. Once both bytes have been latched, the ppi_ready signal is asserted, which indicates to the Master Control module that a word is available.

The reader should note that the final parallel port module used in the prototype utilizes a custom EPP device driver. This made mode negotiation unnecessary, as the PC was forced to use EPP mode, and the PPI module was specifically designed to use EPP mode. In addition, excessive ringing was observed on the handshaking and data signals originating from the PC. This resulted in data transmissions that would not complete reliably. The hypothesis is that ringing caused the finite state machine to advance unpredictably through some states. This was avoided by using counters to introduce delays to avoid the ringing. As well, all handshaking input signals were synchronized to the global clock in registers to reduce the chance of signal transitions violating setup and hold times. In this process, a number of handshaking signals were omitted to reduce the complexity of the state machine. This included sending the ppi_waitn acknowledgement signal to the PC. Since the other interfaces operated at speeds higher than the bit rate of the parallel data, there was no concern regarding data overruns. The test cases and simulations below are still valid, as various signals are simply omitted in the final implementation.

Test Cases

The PPI module was implemented in one VHDL file (see Appendix E) and simulated with MaxPlusII. Refer to Appendix F for annotated waveform diagrams for more detailed information. The first simulation demonstrates the operation of mode negotiation.

The annotations refer to the following test cases:

1. Test if ppi_xflag, ppi_intr, ppi_davailn, and ppi_ackdreq are set to 1, 0, 1, and 1 respectively when ppi_dstrobe is low and ppi_astrobe is high.
2. Test if ppi_xflag goes low when the extensibility byte is not X"40". This occurs after the ppi_dstroben and ppi_writen signals go high.
3. Test if the time between ppi_ackdreq going low and ppi_intr going high meets the minimum delay of 500ns. (This was accomplished using a delay counter).
4. Test if the negotiation mode can proceed to next extensibility byte if the first byte is invalid.
5. Test if ppi_xflag remains high when the extensibility byte is X"40".
6. Test if normal EPP mode is entered after negotiation complete.

Four simulation diagrams of the EPP handshaking cycle itself follow the mode negotiation simulation. A summary of test cases used in the simulation of the operation of the EPP mode is presented below. More test cases are shown directly on the waveform diagrams.

- Ensure that ppi_ready is low while the pair of bytes are being read
- Timing constraints should be met. For example, the ppi_waitn is short at the current clock period of 66 ns. One version of the EPP protocol requires that the time between the rising edge of the ppi_dstroben signal and the falling edge of ppi_waitn should not be more than 125ns. In the simulations, this is at most 84ns, which is the clock period (66.7ns) plus some propagation delay.
- Data is latched on the rising edge of the clock after the ppi_dstroben signal goes high.

- When a pair of bytes of data are downloaded, ppi_ready should be low. The first byte is latched and appears in the output word, but since ppi_ready is low, this output is still invalid. When the second byte of data is latched, ppi_ready should go high.
- The ppi_writen and ppi_dstroben signals are handshaking inputs from the parallel port. In this simulation, these signals were manually specified to simulate normal EPP protocol behavior.
- Likewise the ppi_reset and ppi_download signals are inputs from the Master Control module that are manually specified in this simulation.
- The outputs ppi_ready and ppi_waitn function as expected, while the remaining outputs are used to handle timeouts and have yet to be implemented

Design Verification

The design of the mode negotiation and handshaking protocol is verified by the above test cases. The first extensibility byte is not X"40" so the PC is not requesting EPP mode. The appropriate behavior of the ppi_xflag signal is observed at this time. The next extensibility byte is X"40", which indicates that the PC is prepared to transmit data in EPP mode. Since this is what is required, the mode negotiation completes with ppi_intr going high. The EPP handshaking cycle begins in the s_init state so that the first byte can be downloaded.

The registered performance of the PPI module is 105.26 MHz. The critical path is latching the bytes for output to the Master Control module. At this speed, there is no need to make it operate faster.

Memory Management Interface

Annotations:

NOTE: ALL data from external sources is inserted manually

1. verify that cannot read when have not yet written to memory
2. verify that mc_ram_ready goes low upon master issuing a read/write
3. verify that wait for DRAM client interface to be ready before
   issuing a read/write
4. verify that on a write to DRAM client interface:
   dpath_r_w = '0'
   dpath_csn = '0'
   dpath_primary_datapath is being driven with the write data
5. verify that will wait till write is over
6. verify that address incremented for subsequent writes/reads
7. verify address starts at zero for first read
8. verify only drive read data to master when it is valid:
   during wait_for_end_of_master_read since mc_ram_ready signal
   returns to high in that state after a read...
9. verify address returns to zero on a read when reach end of song...
10. increment_write_offset_address
11. increment_write_frame_address
12. increment_read_offset_address
13. increment_read_frame_address

LCD Interface

TEST CASES for LCD

- The LCD interface initializes before it indicates that it has completed.
- Correct initialization command sequence is written out on lcd_data_out.
- The LCD interface reacts correctly in response to the lcd_mode_chg line being activated while it is in the wait_for_mode State.
- Is the correct address in the MIF actually latched as the new address for ROM. (i.e. Are we accessing the correct command sequence.)
- The LCD interface responds correctly when the null termination character of a command sequence is reached. That is does it stop incrementing the address and return to the wait_for_mode State.
- The LCD interface asserts the lcd_done output when it has completed outputting the command sequence stored in ROM.
- The LCD interface asserts the lcd_register_select output when it is about to output a character that is to be written to the LCD.

DESIGN VERIFICATION for LCD (TESTING)

For testing purposes the delays (count limits) were decreased to allow for the simulation to be simulated under reasonable conditions for MAXPLUS simulations. The actual values for the count would require 40us, 2ms, and 15ms timers which take extremely long amounts of time to simulate.

In the following simulations, it is being shown that the LCD interface is indeed satisfying all of the test cases that were presented for this interface. The simulation shows that for individual stimuli that were introduced, the correct operation was demonstrated. However due to the real time nature of the delays, the delays were shortened to allow for relatively simple simulation. For the simulations, the rising edges trigger all latching and state change operations. It can also be seen that the LCD interface is progressing through all states of operation and implementing the corresponding output for that particular state.

The maximum speed of operation for this module was determined using the Timing Analyzer, which gave a result of 45.3ns or 22.07MHz. The critical paths of this module are in the clock_counter that determines the amount of time we need to wait for the LCD to initialize itself before writing any commands to it. If need be this initialization wait could be removed assume that we are going to turn the LCD on before we power on the rest of the system.

The following correspond to annotations of the LCD simulations that follow:

For LCD Figure 1: page 1

1. The initialization_wait state
2. This rising edge triggers the state change to the initialization state where the rest of the LCD initialization starts to take place.
3. The get_mode_address state where since we are in the initialization still, this will latch the address for mode 0 (the initialization sequence for the LCD).

4. The wait_mode_address state where we wait for the two addresses (the mode_address, which is where to find the mode we want, and the address that is going to hold this mode_address) are equal. This ensures that we are looking at the right mode before we go looking for the command sequence.
5. The load_address state where we load the address of the first command in the mode's command sequence.
6. The wait_address state where we ensure that we are at the correct address before we proceed to write commands to the LCD.
7. Here we can see that the address has been correctly latched and now we can proceed.
8. Here is where we move to the next state, where we can get the first character from the ROM.
9. This state (display_char) is used to check the character to see if it is the command sequence terminator. It also tells the module underneath (lcd_out.vhd) that the character is valid and to latch it. Here is also where we wait for lcd_out to indicate that it has finished sending the character to the LCD.
10. This state (inc_address) is used merely to increment the ROM address so that we can begin reading the next character from memory.
11. This delay from the time that display_char state was entered into and the time to the output of the command to the LCD is attributed to the lcd_out module, which operates slightly different. It waits for the display_char to indicate that valid data is available. At this point it latches the data into an intermediate register in case the original signal is lost. It then moves through its own state machine processing.
12. In the mean time while we are waiting for the lcd_out to complete we pre-fetch the command from the ROM. This allows for quicker display and a more stable execution.
13. Here we see that the lcd_nenable output is cleared (this is because the LCD requires a falling edge to latch data). At this point the data on lcd_data_out is valid and from this point we begin our counter to wait the required execution time of the command (as described in the LCD data sheets). Also here it is possible to see that the data being output is identical to the data in the MIF file for the first initialization command.

For LCD Figure 1: page 2

14. Here we see the same thing as in 13 except the command has changed. It corresponds to the next command in the MIF file.
15. Same as 14
16. Same as 15
17. Since we have completed writing the initialization sequence to the LCD, we can now assert the lcd_done output, which corresponds to this state.
18. Here we see that we have intercepted the command sequence terminator. The null character (0x100), so now we are done this is why 17 occurred.
19. Here we have returned to where we wait for the next lcd_mode_chg to come along, which will indicate a change in the LCD needs to be made.

For LCD Figure 2: page 1

20. We wait in this state to ensure that the LCD has been initialized such that if we start sending commands to it, it will respond correctly. This is mainly only needed if the MAX7000 or some other non-volatile FPGA were to be used. However we chose to keep this anyway (the delay is only 15ms).
21. At this point the clock_count which is what is being used to determine the delay has reached the required count value (this was reduced for simulation).

22. This can be seen in LCD Figure 1 (blown up)
23. Here is where we receive the first request to change what is being displayed on the LCD.
24. In response to the request the lcd_done output is cleared since we are no longer done and we will need to inform the controller later when we are done.
25. Since the mode is 0, this is the mode that will be latched.
26. We used a 0 indexed mode but for the MIF we added one to the index, this is to ensure that we have a mode for the initialization of the LCD. (initialization is commands that setup the LCD for writing and reading)
27. Here we have retrieved the address to the first command in mode 0.
28. This command is held longer since the execution time required for this command is ~2ms as defined on the LCD data sheets.
29. The lcd_register_select output is asserted here because the next command to be output to the LCD is going to a character and we need to change to the data register and not the command register.
30. Next request for change of the LCD display. A new LCD mode as well

MP3 Interface

TEST CASES for MP3 Decoder Interface

- The MP3 interface when reset will activate the MP3 chip's reset which is active low.
- The MP3 interface will wait for the mp3_enable input to be asserted and the mp3_done input to be asserted before proceeding to the next state of operation.
- When the MP3 interface receives the asserted mp3_enable, the mp3_ready output will be cleared on the next rising edge of the clock (the next state).
- When the MP3 interface receives the asserted mp3_enable, the data on the mp3_datain input will be latched in. This is to ensure that the correct data is latched and that we don't lose this data if the data bus supplying this interface is changed.
- After the data has been latched the data must be loaded into a shift register so that the data can be streamed out to the MP3 chip. Therefore make sure that the shift register is loaded with the correct byte of the two-byte input.
- Is a clock of 1MHz frequency generated and on when there is valid data to be shifted out of the shift register, this is the mp3_chipclk. Must be a fully functional clock with rising and falling edges at half period intervals.
- Is the shift register shifting the correct bit out on the rising edge of the mp3_chipclk, and is this bit still valid on the falling edge of the mp3_chipclk.
- Does the MP3 interface completely shift out a byte to the MP3 chip and does it correspond to the byte that was input into the shift register.
- Does the MP3 interface correctly load the second byte of the two-byte input data into the shift register such that we complete the 16-bit streaming operation?
- Does we return to a state where we are waiting for more data, and do we assert the mp3_ready line at this point.
- Does the mp3_chipclk turn off and on at the correct times. This clock should only be active during the state where we are supposed to be shifting the data out to the MP3 chip.
- Does it react to two totally different mp3_datain words, such that the output data will change to mimic what the input would be if you output bit-wise from MSB to LSB.

DESIGN VERIFICATION for MP3 Decoder Interface (TESTING)

For testing purposes the clock into the MP3 decoder interface was set to 10MHz, to allow for a more uniform division using the carry save counter. Using 5 as the divider gives a fully functioning 1MHz clock that can be redirected to the MP3 chip.

In the following simulations, it is being shown that the MP3 decoder interface is indeed satisfying all of the test cases that were presented for this interface. The simulation shows that for individual stimuli that were introduced, the correct operation was demonstrated. For the simulations, the rising edges trigger all latching and state change operations. It can also be seen that the MP3 decoder interface is progressing through all states of operation and implementing the corresponding output for that particular state.

The maximum speed of operation for this module was determined using the Timing Analyzer, which gave a result of 31.7ns or 31.54MHz. The critical paths of this module are in the clock dividers' counters that determine the amount to divide the clock by. The main guts behind this module is a 7

State Finite-State-Machine, that shifts 2 bytes into a shift register and shifts out the MSB from the shift register. At this time we were using the carry-save adder/counter to divide the clock (faster than a ripple carry adder), so I don't have a suggestion of any change that could be used to make this faster.

The following correspond to annotations of the MP3 decoder simulations that follow:

For MP3 Figure 1: page 1

1. The mp3_reset being asserted forces the interface to be in the startstate State, were we ensure that the MP3 chip is being reset (the mp3_chipresetn output is low – is active low reset).
2. Here we have finally received the mp3_enable signal we were waiting for before we start streaming and retrieving data. This state is the load_data State where we latch the data in that is on the mp3_datain bus.
3. This clearing of the mp3_ready output is essentially an acknowledgement signal that is being sent to the controller to tell it that we have latched the data that was on the bus and it can continue to do what it needs to do. This is also the shift1 State where we load the shift register with the first byte of the input data.
4. Here we have selected the first byte and know we load the shift register with this value.
5. This rising edge of the mp3_chipclk is used to control the shift register. The rising edge tells the shift register to output the MSB on to the mp3_dataout output. At this point we are also shifting a zero into the shift register to push the data out. This is essentially implementing a shift left register. It would also be possible to shift in a one but the bit to shift in is arbitrary. (0x55 → '01010101': shift in a 0 → '10101010' → 0xAA)
6. Here is where the bit shifted out is deemed to be valid to the MP3 decoder chip. The MP3 decoder chip requires that data be valid on the falling edge of the clock that is being input to it.
7. Here we shift out the next bit of the byte. We shift the data left to ensure that the next bit is in the MSB location of the shift register.
8. Here we have shifted out the last bit of the first byte of the data.

For MP3 Figure 1: page 2

9. Count has reached 8, which indicates that 8 bits have been shifted out and therefore we are done shifting the byte.
10. This is actually where the mp3_chipclk should be zero while we load the second byte into the shift register. After the byte is loaded we need to restart the clock to again begin shifting out the byte to the MP3 chip.
11. This shows that the mp3_enable line can remain up as long as we need, however we want it to be off before the mp3 requests more data.
12. The shift2 State where we load the second byte into the shift register.
13. It should be noted that the shift register counter is triggered by the falling edge of the clock. This is because the valid data is valid on the falling edge so the count should correspond to this implementation. This ensures that the clock shuts off after the eighth bit has been validated.

For MP3 Figure 1: page 3

14. We have completed the transfer of the 16-bit data to the MP3 chip and now we are waiting for the next mp3_enable to be asserted to start loading and streaming the next data word.
15. The mp3_datain must be available before the mp3_enable input is asserted.

For MP3 Figure 1: page 4

16. Here we see that the MP3 decoder has finished again and is ready for more data.
17. It is possible to notice that when you look at mp3_chipresetn it is not cleared at any point, since this would cause a reset in the MP3 decoder chip.

Appendix D – Test Benches

Appendix E – Schematics

Appendix F – Proposed Design

Master Control Module

This module is the main control module of the project, in that all the other modules in the project interact with it. The behavior of this module is as follows. When the system is first powered on, the main controller will wait for the DRAM module to indicate that it is ready to proceed, however while it is waiting it will display a Startup Banner.

After the DRAM module has indicated that it is ready and if there are any songs in memory, then the name of the first song will be indicated along with a stop icon. There are multiple next steps that can occur from this point, for example, start playing the displayed song, change displayed song to next song in memory or previous song in memory or enter into command mode. When you choose to enter play mode, the display will be refreshed to make sure that song name is correct and a new icon indicating that you are in play mode will be displayed. Also when in play mode the MP3 decode module will request data from this module and this request will be forwarded to the DRAM module, who when the data is valid will drive the data onto the bus. This will continue until the end of the song is reached at which point in time DRAM will move to the next song in memory and will trigger a signal, which indicates this switch in song. At this point this module will stop play and update the display with the new song name, and then resume by playing the new song. From the play mode operation there are also a number of possible inputs that will result in new operation. If pause is selected the display will be updated with a new icon that indicates operation paused. Also if next or previous song, are selected the display will be updated to represent the corresponding change by displaying the next or previous song respectively, after which it will return to the play mode to resume operation. Alternatively if stop is selected the display will change to represent this new state, while here the address will need to be reset to ensure that if play is chosen that we start playing song from the beginning. Another possibility is to select command mode, which will stop the playing of the song and reset the song's addresses (stopping the song is necessary to avoid conflicts that may arise such as deleting the playing song.) If the user selects pause, the operation of playing the song is suspended until play/pause is pressed to return to the play mode, while here the only possible options available are play, pause, and command mode. Command mode is a valid input in any mode of operation, since it has a certain priority over memory.

If however the DRAM module indicates that it is ready but that there are no songs in memory, then the system will continue to display the startup banner until command mode is selected. Once command mode has been selected, a message with a number of choices will be displayed. The keypad is used to enter the choices, if at anytime during command mode 'ESC' is selected, the system will cancel the current operation and step back one step at a time. The first choice to be made in command mode is whether to delete a song, to download a song, or to stream a song.

If delete a song is chosen, then the display would be updated with the new menu. This new menu would display a song name and the choice to delete, from this point the next and previous options are also available to move through available songs to delete. Once the song that is to be deleted is found, the delete option is chosen and the DRAM module is told to delete the selected track. When the deleting of the song is complete the DRAM module will indicate that it is ready to continue, at this point the Deleted message is displayed and we wait for the user to respond OK. When an OK is received we will return to the delete menu that displays the first song with the choice to delete, this allows us to delete songs continually.

If download were chosen, then the display would be updated with a new menu that would display choices of where to download data from. The two current choices are between the parallel port and a CD-ROM, each requiring different operation to be performed. For the CD-ROM, the user must cycle through the

CD-ROM files to determine the file that they want to download. For the parallel port the download would go into a waiting for transmission to start, and once the transmission starts the display will indicate that downloading. The CD-ROM will also indicate downloading once the track is selected. Once downloading is complete from either the CD-ROM or parallel port, either will set a line that indicates that it is done, after which the display will indicate download complete and will wait for an OK press. After a download has completed the display will return to the initial command mode display of the choices.

If streaming were chosen, then the display would be updated with a new menu that would display choices of where to stream data from. The two current choices are between the parallel port and a CD-ROM, each requiring different operation to be performed. For the CD-ROM, the user must cycle through the CD-ROM files to determine the file that they want to stream. For the parallel port the streaming would go into a waiting for transmission to start, and once the transmission starts the display will indicate that streaming is occurring, with a name of parallel port. The CD-ROM will also indicate streaming is occurring with the name of the track once the track is selected. Once downloading is complete from either the CD-ROM or parallel port, either will set a line that indicates that it is done, after which the display will indicate streaming complete and will wait for an OK press. After a streaming has completed the display will return to the initial command mode.

Parallel Port Interface

The parallel port interface uses handshaking to communicate with the PC and other modules. After a mode negotiation phase, the PC will use EPP mode. The host PC brings the ppi_writen signal low and then writes data to the data outputs and asserts ppi_dstroben. This is done only if the ppi_waitn signal is low. The EPP interface brings ppi_waitn high to inform the host that it's ready to latch the data. The host brings ppi_dstroben high and the data is latched. Then, the PPI module asserts ppi_ready to indicate to the master control module that valid data is available and another byte could be fetched. A download is initiated by asserting the ppi_download signal. The handshaking sequence begins again when ppi_waitn is asserted.

Other signals generated by this interface alert the master control interface that the download activity has completed (ppi_dldone) and warn that the expected acknowledgement signal from the PC has not been received within an acceptable timeframe (ppi_timeout).

Data from the fitting report indicates that the Parallel Port Interface requires only **68** logic cells.

```
Total dedicated input pins used:                6/6      (100%)
Total I/O pins used:                           32/183    ( 17%)
Total logic cells used:                        68/1152   (  5%)
Total embedded cells used:                      0/48     (  0%)
Total EABs used:                                0/6      (  0%)
Average fan-in:                              2.77/4      ( 69%)
Total fan-in:                              189/4608      (  4%)
```

Pin No (D-Type 25)	Pin No (Centronics)	SPP Signal	Direction In/Out	Hardware Inverted
1	1	nStrobe	In/Out	Yes
2	2	Data 0	Out	
3	3	Data 1	Out	
4	4	Data 2	Out	
5	5	Data 3	Out	
6	6	Data 4	Out	
7	7	Data 5	Out	
8	8	Data 6	Out	
9	9	Data 7	Out	
10	10	nAck	In	
11	11	Busy	In	Yes
12	12	Paper-out/Paper-End	In	
13	13	Select	In	
14	14	nAuto-Linefeed	In/Out	Yes
15	32	nError/nFault	In	
16	31	nInitialize	In/Out	
17	36	nSelect-Printer/ nSelect-In	In/Out	Yes
18-25	19-30	Ground	GND	

Note 1: See signal descriptions for information on source of these signals

Table – Parallel Port pinout [1]

Memory Management Interface

A client_ram_ready signal is output to the client to indicate whether the Mem. Mgmt. Interface is presently occupied. NO commands should be issued to the Mem. Mgmt. Interface if this line is low. If a read or write had been issued this signal will go low during the access. When it goes high again there is valid data available on the client_primary_datapath_out for a read or if a write was issued it is done. If a delete present track was issued this line will go low until the track has been deleted from the song table. If the track has been incremented or decremented this line will go low until the internal registers that hold the start of the present song, the end of the present song, and the present place in the present song have all been updated with values for the new track.

A client_ram_data_select signal is input from the client to indicate whether raw song data is requested to be streamed for the present track at the present address in that song or whether the song name is requested. This module will respond accordingly.

A client_ram_track_control signal indicates whether the client interface controller's internal track up/down parallel load register should be incremented or decremented. Or whether the present track should be deleted. If the present track is deleted this module will automatically initialize at the first song. This signal will only be listened to if the client_ram_ready signal is high.

A client_ram_no_data signal is output to the client that indicates that the end of the present song has been reached or that there is no song data in memory at all. If the end of the present song has been reached it is expected that the client will use the client_ram_track_control to increment to the start of the next song or decrement to the beginning of the present song.

CD-ROM Interface

Interface Signal Assignments and Descriptions

The physical interface consists of receivers and drivers communicating through a 40-conductor flat ribbon non-shielded cable using an asynchronous interface protocol. Reserved signals shall be left unconnected. This data was obtained by consolidating information available on www.Deja.com.

Description	Source	Pin	Acronym
-Reset	Host	1	cdi_areset
	n/a	2	Ground
-Data bus bit 7	Host/Device	3	cdi_data_in7
-Data bus bit 8	Host/Device	4	cdi_data_in 8
-Data bus bit 6	Host/Device	5	cdi_data_in 6
-Data bus bit 9	Host/Device	6	cdi_data_in 9
-Data bus bit 5	Host/Device	7	cdi_data_in 5
-Data bus bit 10	Host/Device	8	cdi_data_in D10
-Data bus bit 4	Host/Device	9	cdi_data_in 4
-Data bus bit 11	Host/Device	10	cdi_data_in 11
-Data bus bit 3	Host/Device	11	cdi_data_in D3
-Data bus bit 12	Host/Device	12	cdi_data_in 12
-Data bus bit 2	Host/Device	13	cdi_data_in 2
-Data bus bit 13	Host/Device	14	cdi_data_in 13
-Data bus bit 1	Host/Device	15	cdi_data_in 1
-Data bus bit 14	Host/Device	16	cdi_data_in D14
-Data bus bit 0	Host/Device	17	cdi_data_in D0
-Data bus bit 15	Host/Device	18	cdi_data_in 15
-Ground	n/a	19	Ground
?(keypin)	n/a	20	?Reserved
?DMA Request	Device	21	?DMARQ
-Ground	n/a	22	Ground
-I/O Write	Host	23	?DIOW
-Ground	n/a	24	Ground
-I/O Read	Host	25	cdi_read
-Ground	n/a	26	Ground
-I/O Ready	Device	27	cdi_ready
?Cable Select	(note 1)	28	?SPSYNC:CSEL
?DMA Acknowledge	Host	29	?DMACK-
-Ground	n/a	30	Ground
-Interrupt Request	Device	31	cdi_dev_intrq
?16 Bit I/O	Device	32	?IOCS16-
-Device Address Bit 1	Host	33	cdi_dev_addr1
?PASSED DIAGNOSTICS	(note 1)	34	?PDIAG-
-Device Address Bit 0	Host	35	cdi_dev_addrO
-Device Address Bit 2	Host	36	cdi_dev_addr2
-Chip Select 0	Host	37	cdi_chip_sel0-
-Chip Select 1	Host	38	cdi_chip_sel1-
-Device Active	(note 1)	39	cdi_dasp-
-Ground	n/a	40	Ground

Note 1: See signal descriptions for info on source of these signals

LCD Interface

A new state is detected when there is a rising edge on the lcd_mode_chg signal from the Mode Manager. Afterwards, the controller reads in the mode from the lcd_mode vectored signal. The lcd_mode_chg is crucial because without it, the controller would not be able to interpret consecutive occurrences of the same lcd_mode signals. For example, if the mode were at the song refresh state, and then had to refresh the song again right after that, the controller would not notice the new but same mode. However, a rising edge on lcd_mode_chg would let it know that a mode request has been made and thus the lcd_mode would be checked again for a song refresh.

clock - The PortaAMP clock signal, used in this case for the state machine mechanism.

areset - The PortaAMP reset signal, used in this case for setting the state to reset mode.

lcd_data_in - Data from the Mem. Mgmt. Interface, normally characters for the current song title.

lcd_mode_chg - A signal detected on its rising edge which lets the LCD controller know the display must be updated.

lcd_mode - A binary value from the Mode Manager that represents the current mode of the PortaAMP system. 5 bits are used for *lcd_mode* to carry up to 32 different states.

lcd_data_out - Display information sent to the data bus of the LCD.

lcd_register_select - If set to 1, the data register of the LCD is to be used (e.g. for writing characters); if set to 0, the instruction register of the LCD is to be used.

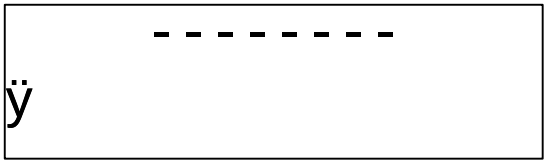
lcd_nenable - A signal detected on its falling edge, which lets the LCD controller, know there is information on the data bus.

The following LCD pins are used with PortaAMP:

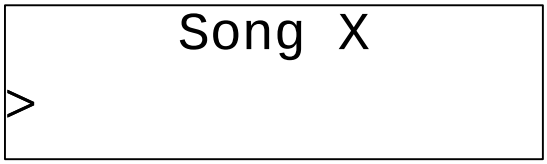
Pin	**Signal**	**Information**
1	V_{ss}	Always grounded
2	V_{dd}	5V power
3	V_o	Contrast; adjustable up to 5V
4	RS	Register Select; 0=data reg., 1=instruction reg.
5	R/W	Read/Write; always set to 0 (write)
6	E	Enable
7-14	DB0-DB7	Data Bus

LCD States (On a 16x2 Display)

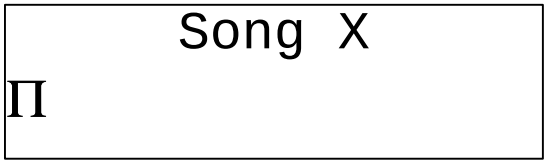
Display	*lcd_mode*	Description
PortaAMP	0000	*power_on* - The reset state.
Command 1=DN 2=DT 3=S	0001	*cmd_default* - Command menu 1 = download songs (*dl_req)*, 2 = delete (*del_what*), 3 = play stream. If deleting songs, then go to state 01011.
From? 1=CD 2=EPP	0010	*source_req* - Download/Stream request (upon selecting Option 1 from *cmd_default*).
Waiting for TX	0011	*source_wait* - Waiting for download/stream source to be ready. If streaming, then go to state 01101.
Downloading	00100	*dling* - Downloading song from source.
Downloading Complete	0101	*completed* - Finished downloading/streaming. Only the second line is updated.
		After *completed*, the system goes back to *cmd_default*.
	0110	*song_default* - Awaiting command from user. Songs ready to be played.



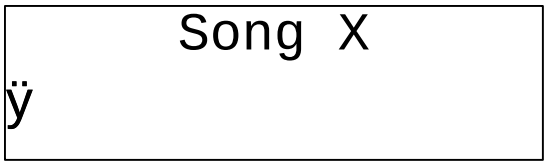
0111 *song_name* - Only the song title on the first line is changed.



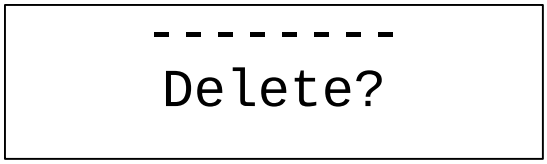
1000 *play* - Current song is being played. Only the second line is updated.



1001 *pause* - Song waits for another *pause* or *play* signal to resume play. Only the second line is updated.



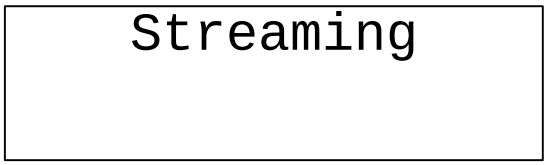
1010 *stop* - Stop playing current song. Only the second line is updated.



1011 *del_what* - Prompt to delete song (upon selecting Option 2 from *cmd_default*). User can use the keypad to scroll through song list.

After *del_what*, the mode changes to *song_name* to display the current song name on the first line.

1100 *del_done* - Deletion of selected song completed. Only the second line is updated. Goes back to *del_what* afterwards.

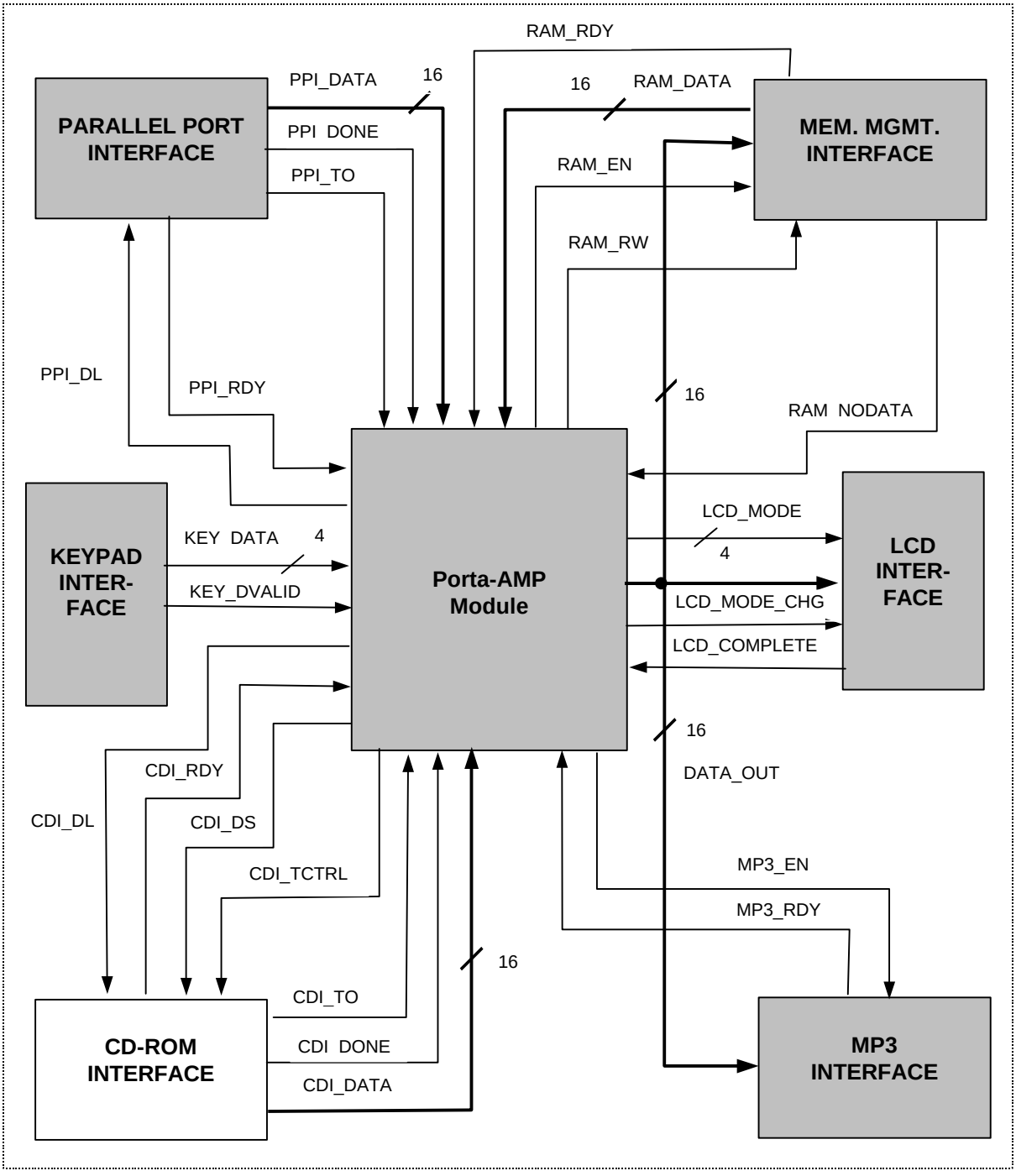


1101 *streaming* - Streaming song from source. Goes to state 01000 to show the *play* icon. Goes to state *completed* when done.

Appendix G – Project Details

System Architecture

The following is the Detailed Design Hierarchy and Internal Interface Diagram. Shaded modules are functional in hardware except that the MP3 Interface does not produce audio due to a defective MP3 decoder chip.



System Features

Optional Features

Optional Feature	Description
CD-ROM Streaming Support	CD-ROM streaming is the ability to play MP3 files directly from CD-ROM Download data into DRAM from the parallel port interface while streaming data from the CD-ROM
Parallel Port Streaming Support	Parallel Port streaming is the ability to play MP3 files directly from the parallel port Download data into DRAM from the CD-ROM while streaming data from the parallel port interface.

Enhancements

Enhancement	Description
IDE CD-ROM Support	ability to play MP3 files directly from CD-ROM ability to download MP3 file into memory from CDROM
Enhanced User Interface	LCD shows time elapsed/remaining in song Create PC Client for downloading and uploading songs
Low Power Mode	System can switch to low-power and retain data in DRAM self-refresh mode
Parallel Port Features	Use the ECP parallel port protocol instead of EPP. ECP is more compatible with Microsoft Windows. uploading files from player to PC act as personal file storage instead of strictly MP3 songs
Memory Management	Reduce memory fragmentation caused by song deletion. This would be implemented using a simple linear paging algorithm.
Battery Operation for Portability	provide the option of using a battery for the system power supply

Components Available or Required

Component	Description	Status	Date Avail.
EPF10K20RC240-4	Altera FLEX10K20 FPGA + UP1	Available	-
ATX PC power supply	+3.3V, +/-5V, +/-12V power supply	Available	-
ECS-300 Oscillator	Dual output CMOS clk oscillator	Available	-
SDRAM socket	168 pin unbuffered DIMM socket	Obsolete	-
MT8LST864A	Micron 64MB 168 pin PC100 SDRAM	Obsolete	-
DRAM socket	72 pin SIMM socket	Available	-
MT16D232M-60 X	Micron 8MB EDO SIMM	Available	-
DSP socket	44 pin PLCC	Available	-
MAS3507D DSP	MP3 DSP decoder chip	Available	-
CS4334	Crystal Cirrus DAC	Available	-
Optrex DMC	16X2 LCD	Available	-
Keypad	4X4 passive keypad	Available	-
CD-ROM	ATAPI interface w/ cable	Obsolete	-
Stereo Amplifier	Amplify line level O/P of DAC	Available	-
Stereo Speakers	Produce near CD-quality sound	Available	-
Parallel Port connector	IEEE-1284B	Available	-
Parallel Port Cable	IEEE-1284	Available	-

Appendix H – Compiled VHDL Code

```
// PPI Device Driver
//
// This driver is used to set PC to use EPP mode
// and download data to the Porta-AMP
//
// It was compiled using Borland C++ 3.0.
//
// Dec 4, 1999
//
#include <conio.h>
#include <stdio.h>
#include <bios.h>
#include <stdlib.h>
#include <ctype.h>
#include <dos.h>
#include <time.h>

// base address of the data register for parallel port on PC (LPT1)
#define base_address 0x378
// address of the status register for parallel port on PC
#define status_address base_address+1
// address of the control register for parallel port on PC
#define control_address base_address+2
// address of the epp data register for parallel port on PC
#define epp_data base_address+4

void EPP_print(char * fname);
void tick_delay(void);

void main(int argc, char **argv)
{
   unsigned short status;
   unsigned short data;
   char filename[256];
   clock_t start, endt;
   int i;

   // check if a file has been specified for transfer
   // if not then display the syntax of the program
   if ( argc < 2 )
   {
      printf("Syntax: lptsend <file>\n");
      exit(0);
   } else {
      strcpy(filename, argv[1]);    // else put the first argument to filenam
   }


   // clear the timeout bit in the status register by writing a '1' to
   // it.  First we want to get current status and maintain it when
   status = inp(status_address);
   status = (status & 0xFE) | 0x01;
   // write to the status register and clear timeout bit
   outportb(status_address, status);
   // set the appropriate lines to values that won't trigger
   // downloading before is ready set the nInit bit high
   outportb(control_address, 0x0C);  // C3-L C2-H C1-H C0-H
   tick_delay();  // add delay to ensure signals have stabilized
   EPP_print(filename); // run the EPP printing procedure

   // clear the nInit line to indicate end of the file
   outportb(control_address, 0x08);         // added for ram test
```

```
   tick_delay();
   // ram
   outportb(control_address, 0x0C);  // ram
}

void tick_delay(void)
{
   clock_t start;

   start = clock();     // get the start clock
   while ( clock()-start < 1 );           // loop for 1 clock_tick 50ms
}

void EPP_print(char * fname)
{
   FILE *fp;
   unsigned short status;
   unsigned short next_byte;
   unsigned short next_byte2;
   clock_t start, endt;
   time_t start_time, end_time;
   unsigned short temp;
   unsigned int delay_count;
   unsigned long max_count;
   unsigned int byte_count;
   unsigned short last_char;

   last_char = 0;
   delay_count = 0;
   max_count = 50000;
   byte_count = 0;

   // ensure that the file that want to transmit is there
   // and is opened before we start sending data
   if((fp = fopen(fname, "rb")) == NULL)
   {
      fprintf(stderr,"Failed to open file: %s\n", fname);
      return;     // return to calling program since couldn't open file
   }
   printf("Sending EPP...\n");

   status = inp(status_address);
   status = (status) | 0x00;
   outportb(status_address,status);

   // print out the initial status of the parallel port
   status = inp(status_address);
   printf("Initial status: %X\n", status);
   start = clock();     // get the initial time (used for calculating speed)
   start_time = time(NULL);   // real time value
   do
   {
         // get next char from file
         next_byte=getc(fp);

         // check if the peripheral is telling us to wait
         while ( (inp(status_address) & 0xD8) != 0xD8)
         {
            delay_count++;          // increment the delay_count
            if (delay_count >= max_count) {     // have reached the maximum delay
               fprintf(stderr,"Max time for transfer exceeded! %X\n", last_char);
               fclose(fp);
               return;
```

```
            }

            // check if we have a timeout while we were waiting
            if ((inp(status_address) & 0x01) == 0x01)  // timeout
            {
               fprintf(stderr,"Timeout occurred in the transfer! %X\n",
last_char);
               fclose(fp);
               return;
            }
         }
         // have finished wait with out timing out so set delay back to 0
         delay_count = 0;

         // send byte to the epp data register.
         // when you write to the epp data register it initiates
         // the handshaking by itself without the knowledge of the
         // programmer
         outportb(epp_data, next_byte);

         // keep track of last byte so that can display it when finished
         // transmitting
         last_char = next_byte;
   }  while( !feof(fp) );

   // display the last byte transmitted sot that can compare
   printf("Last Byte: %X, Next Byte: %X\n", last_char, next_byte);

   // finished transmission so find how long transmission took
   endt = clock();
   end_time = time(NULL);
   printf("EPP TIME: %f - %f \n", endt/CLK_TCK, start/CLK_TCK);
   printf("Elapsed time: %f", difftime(end_time, start_time));
   fclose(fp);
}
```