COMS 4840 - Project Report
Mostly Asians Singing Horribly (M.A.S.H.)

William Dang, Chia-Hung Lee, Stephen Lee, Oleg Mironov, and Zijian "Alfredo" Zhou

(wd2113,cl2208,sl2285,om2003,zz2107)@columbia.edu

May 11, 2004

1 Proposal

We planned on implementing a basic karaoke machine. While we would like to stream video with music playing and the lyrics on screen, we initially implemented the audio synchronized with on-screen lyrics. Our implementation will be broken into the following modules.

- **Sound**

  In order to complete the audio processing unit, we need to interface with the AK4565 20bit stereo codec. We need to create a codec driver in vhdl to interface with various components in the design. The codec driver module will need to handle important signals such as 20bit parallel I/O signals and controls signals that needs to synchronize with the processing of data flow. Data will be accessed and processed from the onboard SDRAM where a circular buffer will be implemented so that the oldest data in memory can be continuous overwritten with new data.

- **Data Storage**

  We are going to use Compact Flash to store and retrieve our data. Compact Flash use 16-bit for data, 12-bits for address and an simple 8-bits interface for controlling purpose. The XSB board provides three different kinds of interface to access CF card. We will use memory mode and write C program to control it.

  We would like to use the compact flash interface to store all the multimedia. The file system on the flash cards is FAT16. It is composed of an MBR which describes the partitioning scheme. The begining of a partition has a root directory which allows via pointers to locate any subdirectory or file. its uncertain whether there would be a significant performance penalty for dynamically selecting tracks and this would have to be worked out during the project.

- **Video Display**

  Full motion video would probably too ambitious, but as a secondary goal, we'd like to use the compact flash to store a bunch of images to be used in a slide show to accompany the karaoke. However, this will probably also prove difficult to implement, so we will leave these as secondary goals if we are successful implementing basic audio playback with synchronized displayed lyrics.

- **Lyrics/Audio Synchronization**

  We plan on examining existing karaoke video disc specifications and other A/V-text synchronization standards, and implement something close to what the standard karaoke machines use.

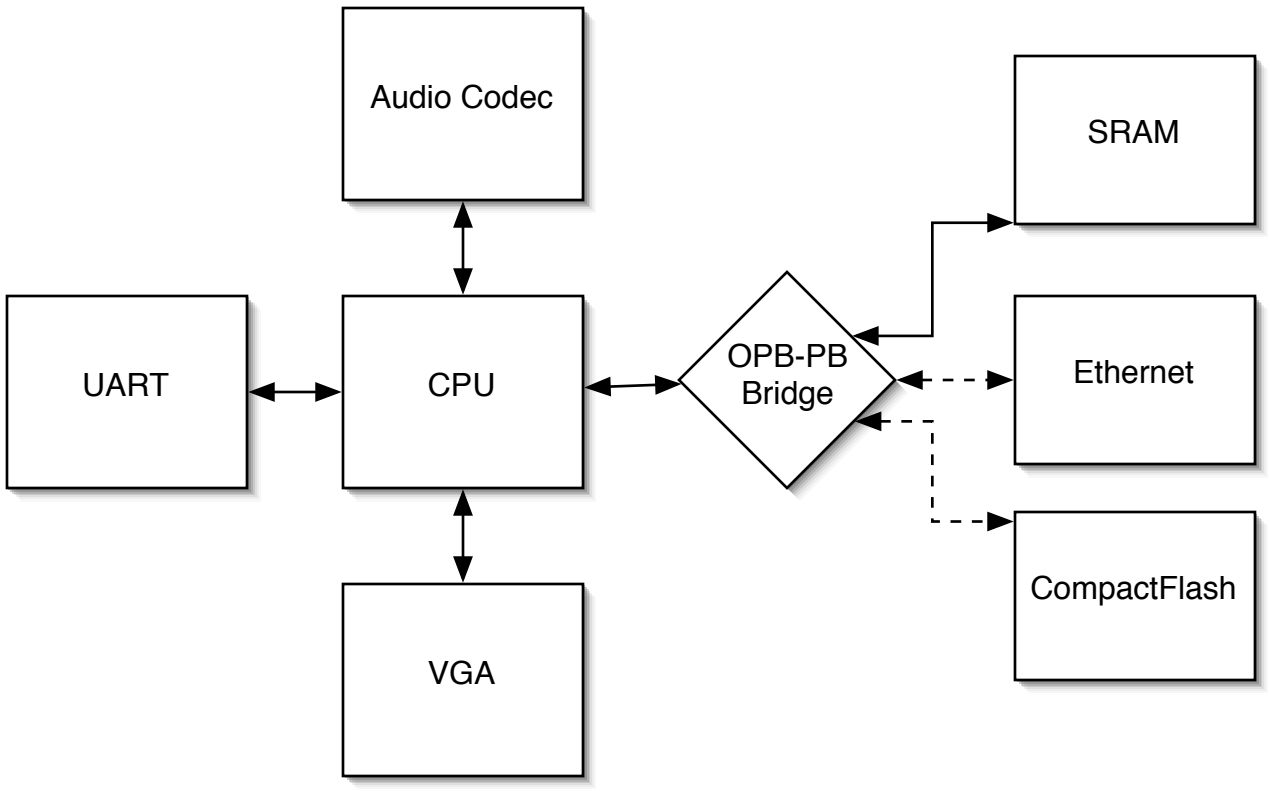


Figure 1: Overall Implementation of Karaoke Machine

2 Design

Figure 1 shows the overall components we attempted to implement. The dotted lines between the OPB-PB Bridge and the CompactFlash and Ethernet devices represent unsuccessful attempts to use these devices. Our original intention was to use the CompactFlash to store both the lyrics and audio for the songs played by our karaoke machine.

The biggest lesson learned from our project was the immense difficulty in designing, implementing, and debugging embedded systems. Designing with such a board becomes a delicate balance of the various constraints placed upon the various components required for the final solution. Specifically, since the XESS board places many of the interesting peripherals on the same bus (PB) without any additional controllers for devices like IDE and CompactFlash, which have vastly different timing diagrams when compared to other PB devices like SRAM or SDRAM, the majority of our problems arrose from building an appropriate state machine to arbitrate between software requests to and from the SRAM and CompactFlash. To further complicate matters, the audio codec also requires the PB bus to set the input control register which allow microphone input and audio output.

After implementing a simple audio player which utilizes the UART to read the entire audio file, the SRAM to store the audio file (limited to about 300KB), and the code to play the audio, we concluded that we should remove as many dependencies on the SRAM as possible. Specifically, since we planned on only displaying a couple lines of text on the screen, we could use a BRAM, which could store the equivalent of 4KB, instead of the SRAM for use in the VGA Character Display. Four kilobytes should give us about six lines of text, which is more than sufficient for our purposes.

If the CompactFlash, as shown in Figure 2, could be utilized, it would be fast enough to stream audio without the need for an SRAM buffer. Furthermore, removing the SRAM from the project would make formulating a state machine for the OPB-PB Bridge much simpler. Unfortunately, removing the SRAM completely would limit the amount of software that could be written. Although a file system like FAT16 is simple enough on its own, it still occupies enough memory that would make SRAM removal not feasible.

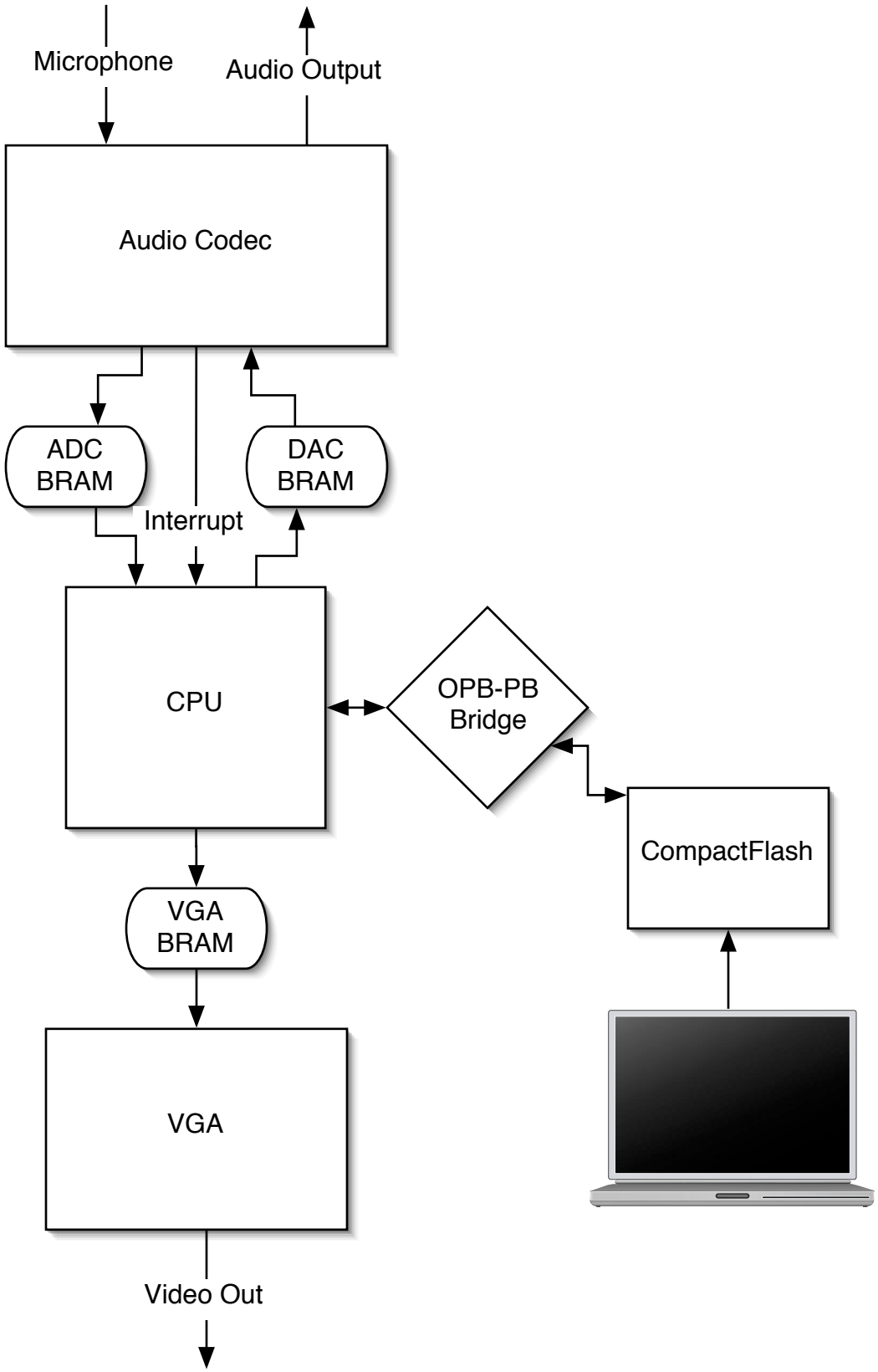


Figure 2: Implementation of Karaoke Machine Using CompactFlash

This was the biggest balancing issue we discovered in this project.

In order to partially circumvent this cyclic dependency between the FAT16 requiring the SRAM and implementing the CompactFlash without the SRAM, we considered not using FAT16 at all. We wanted to write raw blocks sequentially to the CompactFlash card and wastefully use an entire block as a file length prefix to the sequence of blocks that comprise the entire file.

Our final and working implementation, as shown in Figure 3 utlizes the UART to provide the data that would have come from either the CompactFlash or the Ethernet. The UART, even when its baudrate has been upped to 115200, is still only fast to stream relatively low quality audio (8b/mono/11kHz), the same quality that our simple, first-pass audio player could handle. Another good sideeffect of using either the CompactFlash or the Ethernet would have been the ability to stream much higher quality audio.

2.1 Hardware Implementation

2.1.1 Various CompactFlash Implementation Attempts

As part of our original design, we had hoped to use a flash card to stream audio directly. The flash card was described as having an access time of about 300 ns (15 cycles), which was fast enough for our purposes. In our original plan we were going to use the card in memory card mode (one of the three modes specified in the flash documentation). We expected it to behave to a great degree like a more complicated memory chip. The following code represents our first attempt at combining the SRAM and CompactFlash controller.

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cfctrl is
Port ( rst : in std_logic;
           clk : in std_logic;
           cf_cs : in std_logic;       -- pushes state machine out of idle state
   cf_rdy : in std_logic;      -- compactflash ready signal

   --cf_go_addr : out std_logic;
   cf_busy : out std_logic;    -- not finished full read sequence
   cf_ce0 : out std_logic;      -- cf chip enable 0
   cf_ce1 : out std_logic;      -- cf chip enable 1
   cf_oe : out std_logic;      -- cf output enable
   cf_read : out std_logic;    -- data available as of this cycle
   cf_bram_addr : out std_logic_vector( 7 downto 0 );
);
end cfctrl;

architecture Behavioral of cfctrl is

signal state : std_logic_vector( 2 downto 0 );
--signal go_addr : std_logic;
signal busy : std_logic;
signal ce : std_logic;
signal oe : std_logic;
signal read : std_logic;
```

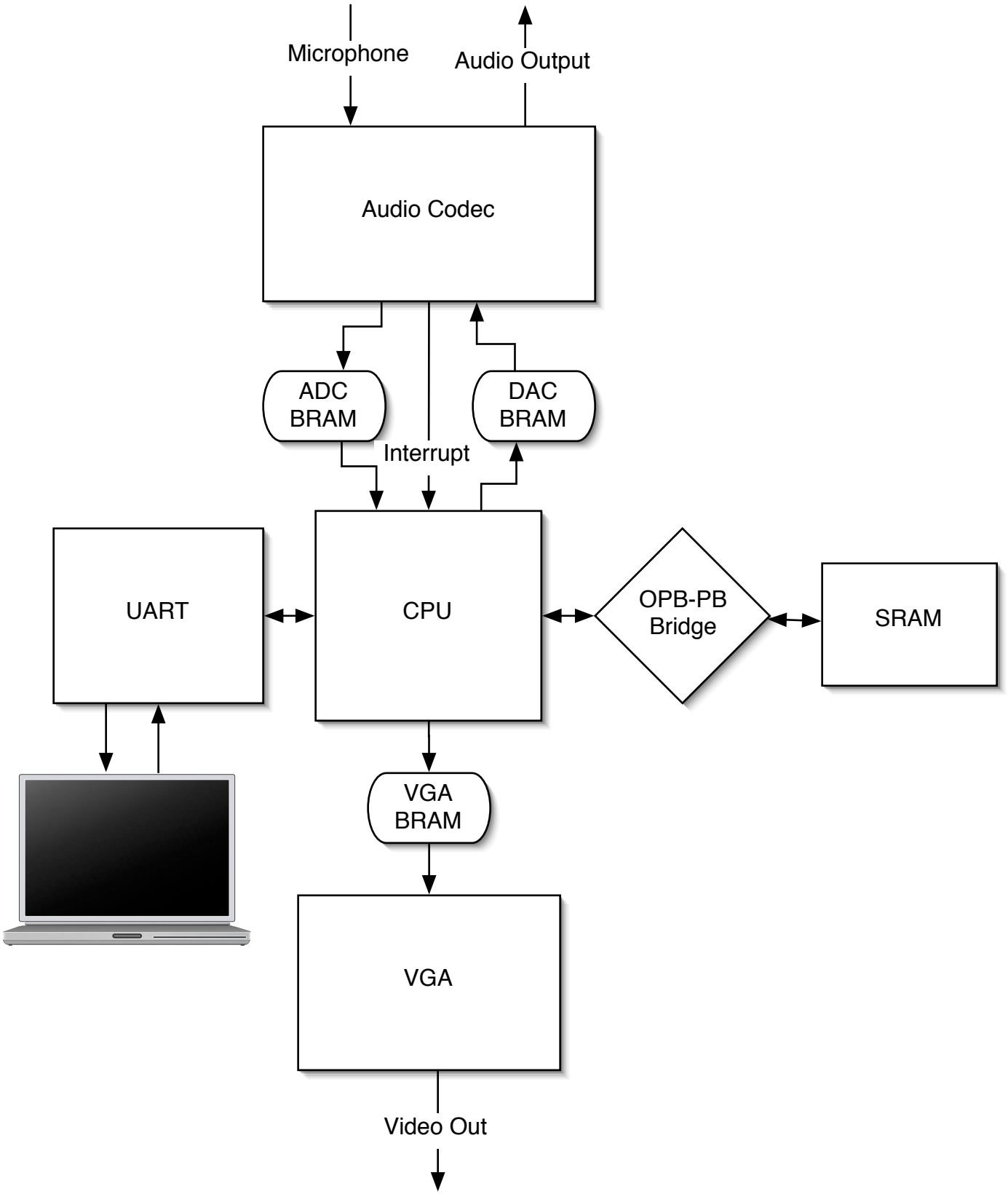


Figure 3: Implementation of Karaoke Machine Using UART

```
signal word_counter : std_logic_vector( 7 downto 0 );

begin

process( rst, clk, cf_cs, cf_rdy )
begin
if rst = '1' then
state <= "000";
--go_addr <= '0';
ce <= '0';
oe <= '0';
read <= '0';
busy <= '0';
word_counter <= X"00";
elsif clk'event and clk = '1' then -- positive clk edge
if state = "000" and cf_cs = '1' then
-- set address
state <= "001";
--go_addr <= '1';
busy <= '1';
elsif state = "001" then
-- chip enable and output enable
state <= "010";
--go_addr <= '1';
ce <= '1';
oe <= '1';
elsif state = "010" and cf_rdy = '1' then
-- done waiting, ready to receive data
state <= "011";
read <= '1';
word_counter <= X"00";
elsif state = "011" then
if word_counter = X"FF" then
-- done retrieving sector
word_counter <= X"00";
state <= "100";
ce <= '0';
oe <= '0';
else
word_counter <= word_counter + 1;
end if;
elsif state = "100" then
if word_counter = X"04" then
state <= "000";
--go_addr <= '0';
ce <= '0';
oe <= '0';
read <= '0';
busy <= '0';
word_counter <= X"00";
else
word_counter <= word_counter + 1;
end if;
end if;
```

```
end if;

--cf_go_addr <= go_addr;
cf_busy <= busy;
cf_ce0 <= ce;
cf_ce1 <= ce;
cf_oe <= oe;
cf_read <= read;
cf_bram_addr <= word_counter;
end process;

end Behavioral;

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity cf is
    Port (
OPB_Clk : in std_logic;
OPB_Rst : in std_logic;
opb_addr : in std_logic_vector( 31 downto 0 );
cf_data : in std_logic_vector( 15 downto 0 );     -- data from pad_io (cf)
cf_cs : in std_logic;
cf_rdy : in std_logic;                            -- iordy signal from pad_io (cf)
opb_data : inout std_logic_vector( 31 downto 0 );
cf_addr : out std_logic_vector( 19 downto 0 );
cf_ce0 : out std_logic;
cf_ce1 : out std_logic;
cf_oe : out std_logic;
cf_busy : out std_logic;
cf_xfer : out std_logic
);
end cf;

architecture Behavioral of cf is

component cfctrl
Port ( rst : in std_logic;
   clk : in std_logic;
   cf_cs : in std_logic;       -- pushes state machine out of idle state
   cf_rdy : in std_logic;      -- compactflash ready signal

   --cf_go_addr : out std_logic;
   cf_busy : out std_logic;    -- finished full read sequence
   cf_ce : out std_logic;      -- cf chip enable
   cf_oe : out std_logic;      -- cf output enable
   cf_read : out std_logic;    -- data available as of this cycle
   cf_bram_addr : out std_logic_vector( 7 downto 0 );
```

```
);
end component;

signal cf_rdata : std_logic_vector (15 downto 0);
signal cf_rdata0 : std_logic_vector (15 downto 0);
signal cf_rdata1 : std_logic_vector (15 downto 0);

signal cf_addr_stat : std_logic := '0';
signal cf_bram_counter : std_logic_vector( 7 downto 0 );
signal addr_mux : std_logic_vector(19 downto 0);
signal cs : std_logic;
signal addr : std_logic_vector (23 downto 0);
signal be : std_logic_vector (3 downto 0);
signal pb_bytesel : std_logic_vector (1 downto 0);

signal cf_addr_reg : std_logic_vector( 16 downto 0 );   -- block address to read off cf
signal xfer : std_logic := '0';                         -- let opb bus about xfer
--signal go_addr : std_logic := '0';                         -- let opb bus about xfer
signal busy : std_logic;
signal ce0 : std_logic;
signal ce1 : std_logic;
signal oe : std_logic;
signal write_enable : std_logic;
signal bram_select : std_logic := '0';
signal cf_read : std_logic;

begin

cf_controller : cfctrl port map (
rst => OPB_Rst,
clk => OPB_Clk,
cf_cs => cs,
cf_rdy => cf_rdy,
--cf_go_addr => go_addr,
cf_busy => busy,
cf_ce0 => ce0,
cf_ce1 => ce1,
cf_oe => oe,
cf_read => cf_read,
cf_bram_addr => cf_bram_counter
);

cf_read0 : RAMB4_S16_S16 port map (
-- write
CLKA => OPB_Clk,
RSTA => OPB_Rst,
DIA => cf_data,
ENA => '1',
WEA => not bram_select and cf_read,
ADDRA => cf_bram_counter,
DOA => open,

-- read
CLKB => OPB_Clk,
```

```
RSTB => OPB_Rst,
DIB => X"0000",
ENB => '1',
WEB => '0',
ADDRB => opb_addr,
DOB => cf_rdata0
);

cf_read1 : RAMB4_S16_S16 port map (
-- write
CLKA => OPB_Clk,
RSTA => OPB_Rst,
DIA => cf_data,
ENA => '1',
WEA => bram_select and cf_read,
ADDRA => cf_bram_counter,
DOA => open,

-- read
CLKB => OPB_Clk,
RSTB => OPB_Rst,
DIB => X"0000",
ENB => '1',
WEB => '0',
ADDRB => opb_addr,
DOB => cf_rdata1
);

process( OPB_Clk, OPB_Rst )
begin
if OPB_Rst = '1' then
bram_select <= '0';
elsif OPB_Clk'event and OPB_Clk = '1' then
-- write to cf block addr register
if opb_addr( 11 downto 0 ) = X"1FF" then
if busy = '0' then
cf_addr_reg <= opb_data( 16 downto 0 );
xfer <= '1';
cs <= '1';
bram_select <= not bram_select;
else
xfer <= '0';
end if;
elsif opb_addr( 11 downto 8 ) = X"0" then
cf_rdata_mux <= bram_select;

if bram_select = '1' then
cf_rdata <= cf_rdata0;
else
cf_rdata <= cf_rdata1;
end if

xfer <= '1';
else
```

```
xfer <= '0';
end if;
end if;
end process;

process( OPB_Clk, OPB_Rst )
begin
if OPB_Rst = '1' then
elsif OPB_Clk'event and OPB_Clk = '1' then
opb_data <= X"0000" & cf_rdata;
end if;
end process;

cf_addr( 19 downto 17 ) <= "101";
cf_addr( 16 downto 0 )  <= cf_addr_reg;
cf_ce0 <= ce0;
cf_ce1 <= ce1;
cf_xfer <= xfer;
--cf_go_addr <= go_addr;
cf_busy <= busy;
cf_oe <= oe;


end process;

end Behavioral;
```

To do this, we would have to modify the opb_xsb300 module leaving only the SRAM and the flash. Since we did not appreciate the full difficulty of adding the flash functionality even in its most basic form, we tried to do it all at once. The code was bloated and it quickly became too complicated to follow. Besides, as Cristian later explained, the flash is fundamentally incompatible with the SRAM due to its being an unclocked, level sensitive device. The SRAM on the other hand is an edge sensitive peripheral.

At this point we realized that the only way the flash could be used, would be as the only peripheral on the peripheral bus. Our next attempt was still trying to use the flash in memory card mode. According to the CompactFlash documentation, we needed to provide a sequence of events: assert an address, turn on the enables, and wait for a ready signal; at that point, the data would show up on the data bus. Output-Enable would act like a clock causing the next word to be read from a sector on every falling edge. However, we could not get the card to respond intelligibly. The following VHDL comprises our next attempt.

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity opb_xsb300_cf is
  generic (
    C_OPB_AWIDTH    : integer := 32;
    C_OPB_DWIDTH    : integer := 32;
    C_BASEADDR      : std_logic_vector := X"2000_0000";
    C_HIGHADDR      : std_logic_vector := X"2000_00FF"
```

```
    );

    Port (  OPB_Clk : in std_logic;
OPB_Rst : in std_logic;
OPB_ABus : in std_logic_vector (31 downto 0);
OPB_BE : in std_logic_vector (3 downto 0);
OPB_DBus : in std_logic_vector (31 downto 0);
OPB_RNW : in std_logic;
OPB_select : in std_logic;
OPB_seqAddr : in std_logic;
UIO_DBus : out std_logic_vector (31 downto 0);
UIO_errAck : out std_logic;
UIO_retry : out std_logic;
UIO_toutSup : out std_logic;
UIO_xferAck : out std_logic;
PB_A : out std_logic_vector (19 downto 0);
PB_WE_N : out std_logic;
PB_OE_N : out std_logic;
PB_D : inout std_logic_vector (15 downto 0);
               CS0_N : out std_logic;
               CS1_N : out std_logic;
               IOCS16_N : in std_logic;
               IORDY : in std_logic;
               INTRQ : in std_logic
     );
end opb_xsb300_cf;

architecture Behavioral of opb_xsb300_cf is

component pad_io
    Port ( clk : in std_logic;
           rst : in std_logic;
           PB_A : out std_logic_vector(19 downto 0);
           PB_WE_N : out std_logic;
           PB_OE_N : out std_logic;
           PB_D : inout std_logic_vector(15 downto 0);
           CS0_N : out std_logic;
           CS1_N : out std_logic;
           IOCS16_N : in std_logic;
           IORDY : in std_logic;
           INTRQ : in std_logic;
           pb_addr : in std_logic_vector(19 downto 0);
           pb_wr : in std_logic;
           pb_rd : in std_logic;
           pb_dread : out std_logic_vector(15 downto 0);
           pb_dwrite : in std_logic_vector(15 downto 0);
           cf_cs_pad0 : in std_logic;
           cf_cs_pad1 : in std_logic;
           cf_rdy : out std_logic;
   cf_cs16 : out std_logic;
   cf_intr : out std_logic
);
end component;
```

```
signal cs : std_logic;

signal addr: std_logic_vector( 19 downto 0 );

signal xfer : std_logic;

signal oe_timer :std_logic_vector(8 downto 0) := "000000000";

signal state :std_logic;




signal cf_rdy : std_logic;
signal cf_addr : std_logic_vector( 19 downto 0 );
signal cf_data : std_logic_vector( 15 downto 0 );
signal cf_ce0 : std_logic :='1';
signal cf_ce1 : std_logic :='1';
signal cf_oe : std_logic :='0';
signal cf_intr : std_logic;
signal cf_cs16 : std_logic;


begin

pad_io1 : pad_io port map (
clk => OPB_Clk,
rst => OPB_Rst,
PB_A => PB_A,
PB_WE_N => PB_WE_N,
PB_OE_N => PB_OE_N,
PB_D => PB_D,
CS0_N => CS0_N,
CS1_N => CS1_N,
IOCS16_N => IOCS16_N,
IORDY => IORDY,
INTRQ => INTRQ,
pb_addr => cf_addr,
pb_rd => cf_oe,
pb_wr => '0',
pb_dread => cf_data,
pb_dwrite => "0000000000000000",
cf_cs_pad0 => cf_ce0,
cf_cs_pad1 => cf_ce1,
cf_rdy => cf_rdy,
cf_cs16 => cf_cs16,
cf_intr => cf_intr
);




process( OPB_Clk, OPB_Rst )
```

```
begin
if OPB_Rst = '1' then
cs <= '0';
elsif OPB_Clk'event and OPB_Clk = '1' then
if OPB_ABus(31 downto 20) = X"008" then
cs <= OPB_select;
        else
            cs <= '0';
end if;
end if;
end process;


process(OPB_Clk)
begin
if OPB_Rst = '1' then
oe_timer <= "000000000";
cf_ce0<='0';
cf_ce1<='0';
cf_oe<= '0';
elsif OPB_Clk'event and OPB_Clk = '1' then
if oe_timer="100000000" then
cf_oe <= not cf_oe;
--elsif oe_timer = "000000000" then
--cf_oe <= not cf_oe;
--elsif oe_timer = "000000011" then
--cf_ce0 <= not cf_ce0;
--cf_ce1 <= not cf_ce1;
--elsif oe_timer="011111101" then
--cf_ce0 <= not cf_ce0;
--cf_ce1 <= not cf_ce1;

end if;
oe_timer <= oe_timer +1;
end if;
end process;

process( OPB_Clk, OPB_Rst )
begin
if OPB_Rst = '1' then
state <= '0';
elsif OPB_Clk'event and OPB_Clk = '1' then
  state <= (not state) and cs;
  if state='1' then

UIO_DBus <= cf_data & cf_data;
         xfer <= '1';
          else
            UIO_DBus <= X"00000000";
            xfer <='0';
          end if;
end if;
cf_addr( 19 downto 17 ) <= "101";
cf_addr( 10 downto 0 )  <= " 0110100100";
```

```
cf_addr(16 downto 11) <= "000000";

-- tie unused to ground
UIO_errAck <= '0';
UIO_retry <= '0';
UIO_toutSup <= '0';
UIO_xferAck <= xfer;

end process;
end Behavioral;
```

We found the documentation on the official CompactFlash "Consortium" website to be vague. While it gives a general description of how the flash is organized, however, it does not describe the exact specifics of operations.

Our final attempt is shown in the piece of code below. The ata_cntl module was posted on the XESS website. We started working on it fairly late, since we had hoped to use the flash in memory mode which we hoped that it would be fairly simple. The following VHDL represents our attempt at hard coding a simple read/write operation using the controller provided by XESS. We should have received a pulsing interrupt signal, but the oscilloscope never registered any such response.

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity opb_xsb300_cf is
  generic (
    C_OPB_AWIDTH    : integer := 32;
    C_OPB_DWIDTH    : integer := 32;
    C_BASEADDR      : std_logic_vector := X"2000_0000";
    C_HIGHADDR      : std_logic_vector := X"2000_00FF"
    );

    Port (  OPB_Clk : in std_logic;
                OPB_Rst : in std_logic;
                OPB_ABus : in std_logic_vector (31 downto 0);
                OPB_BE : in std_logic_vector (3 downto 0);
                OPB_DBus : in std_logic_vector (31 downto 0);
                OPB_RNW : in std_logic;
                OPB_select : in std_logic;
                OPB_seqAddr : in std_logic;
                UIO_DBus : out std_logic_vector (31 downto 0);
                UIO_errAck : out std_logic;
                UIO_retry : out std_logic;
                UIO_toutSup : out std_logic;
                UIO_xferAck : out std_logic;

                PB_A : out std_logic_vector (19 downto 0);
                PB_WE_N : out std_logic;
                PB_OE_N : out std_logic;
```

```
                PB_D : inout std_logic_vector (15 downto 0);
                CS0_N : out std_logic;
                CS1_N : out std_logic;
                DMACK_N : out std_logic;
                --IOCS16_N : in std_logic;-- get rid of this
                --IORDY : in std_logic; -- get rid of this
                INTRQ : in std_logic;
                FPGA_INIT_N : out std_logic

   );
end opb_xsb300_cf;

architecture Behavioral of opb_xsb300_cf is

component ataCntl
generic(
FREQ: natural := 50_000 -- operating frequency in KHz
);
port(
-- host side
clk: in std_logic; -- master clock
rst: in std_logic; -- reset
rd: in std_logic; -- initiate read operation
wr: in std_logic; -- initiate write operation
abort: in std_logic; -- aborts read/write sector operation
head: in std_logic_vector(3 downto 0); -- disk head for data access
cylinder: in std_logic_vector(15 downto 0); -- cylinder for data access
sector: in std_logic_vector(7 downto 0); -- sector for data access
hDIn: in std_logic_vector(15 downto 0); -- data from host to disk
hDOut: out std_logic_vector(15 downto 0); -- data from disk to host
done: out std_logic; -- read or write operation is done
status: out std_logic_vector(3 downto 0); -- diagnostic status

-- disk side
dior_n: out std_logic; -- disk register read-enable
diow_n: out std_logic; -- disk register write-enable
cs0_n: out std_logic; -- disk command block register select
cs1_n: out std_logic; -- disk control block register select
da: out std_logic_vector(2 downto 0); -- register address
ddIn: in std_logic_vector(15 downto 0); -- data from disk
ddOut: out std_logic_vector(15 downto 0); -- data to disk
ddOutEnbl: out std_logic; -- enable data outputs to disk
intrq: in std_logic; -- interrupt from disk
dmack_n: out std_logic -- DMA acknowledge
);
end component;

signal cf_rd : std_logic;
signal cf_wr : std_logic;
signal cs, rnw, q0, q1, wq0 : std_logic;

signal addr : std_logic_vector(7 downto 0);
signal wdata : std_logic_vector(15 downto 0);
```

```
signal head : std_logic_vector( 3 downto 0 );
signal cylinder: std_logic_vector(15 downto 0);
signal sector: std_logic_vector(7 downto 0);

signal pb_data_tri_n, pb_data_tri : std_logic;
signal pb_data_in : std_logic_vector( 15 downto 0 );
signal pb_data_out : std_logic_vector( 15 downto 0 );
signal diskdata : std_logic_vector( 15 downto 0 );

signal done : std_logic;
signal status : std_logic_vector( 3 downto 0 );
signal state : std_logic;
signal xfer : std_logic;

signal pb_a_i : std_logic_vector(19 downto 0);
signal pb_we_n_i, pb_oe_n_i : std_logic;
signal intrq_i : std_logic;

signal cs0_n_i, cs1_n_i, dmack_n_i : std_logic;
signal start : std_logic;

signal startup :std_logic_vector(2 downto 0) := "000";
signal abort : std_logic;

signal rst : std_logic;


begin

cf : ataCntl port map(
-- host side
clk => OPB_Clk,
rst => rst,
rd => '0', --cf_rd,
wr => '1', --cf_wr,
abort => '0',
head => head,
cylinder => cylinder,
sector => sector,
hDIn => X"5657", --OPB_DBus( 15 downto 0 ),
hDOut => diskdata,
done => done,
status => status,

-- disk side
dior_n => pb_a_i(18),
diow_n => pb_a_i(19),
cs0_n => cs0_n_i,
cs1_n => cs1_n_i,
da => pb_a_i( 2 downto 0 ),
ddIn => pb_data_in,
ddOut => pb_data_out,
ddOutEnbl => pb_data_tri_n,
intrq => intrq_i,
```

```
dmack_n => dmack_n_i
);



FPGA_INIT_N <= '1';

UIO_errAck <= '0';
UIO_retry <= '0';
UIO_toutSup <= '0';

process(OPB_Clk, OPB_Rst)
begin
if OPB_Rst = '1' then
startup <= "000";
rst <= '1';
elsif OPB_Clk'event and OPB_Clk ='1' then
if startup <= "111" then
rst <= OPB_Rst;
else
rst <='1';
startup <= startup+1;
end if;
end if;
end process;

process(OPB_Clk)
begin
  if OPB_Clk'event and OPB_Clk='1' then
    addr <= OPB_ABus(7 downto 0);
    rnw <= OPB_RNW;
    wdata <= OPB_DBus(31 downto 16);
  end if;
end process;


cs <= OPB_select when OPB_ABus(31 downto 20) = X"00A" else '0';

-- state 00 is idle
-- state 01 is accepted transaction
-- state 10 is xfer

process(OPB_Rst, OPB_Clk)
begin
  if OPB_Rst = '1' then
    q0 <= '0';
    q1 <= '0';
  elsif(OPB_Clk'event and OPB_Clk = '1') then
    q0 <= cs and not q1 and not q0;
    q1 <= not q1 and q0;
  end if;
end process;

UIO_xferAck <= q1;
```

```
UIO_DBus <= X"00000000";


head <= X"0";
cylinder <= "0000000000000011";
sector <= X"05";

--state 0 is idle
--state 1 is writing

process(OPB_Rst, OPB_Clk)
begin
  if OPB_Rst = '1' then
    wq0 <= '0';
  elsif OPB_Clk'event and OPB_Clk='1' then
    wq0 <= (q0 and not rnw) or not done;
  end if;
end process;

cf_wr <= wq0;
cf_rd <= '0';

-- I/O
--pb_a_i ( 19 downto 18) <= "10";
pb_a_i ( 17 downto 3 ) <= "00000000000" & done & cf_wr & status(0) & wq0;
pb_oe_n_i <= '0';
pb_we_n_i <= '1';

abuf : for i in 0 to 19 generate
  abuf : OBUF_F_24 port map (
    O => PB_A(i),
    I => pb_a_i(i));
end generate;

webuf :  OBUF_F_24 port map (
    O => PB_WE_N,
    I => pb_we_n_i);

oebuf :  OBUF_F_24 port map (
    O => PB_OE_N,
    I => pb_oe_n_i);

cs0buf :  OBUF_F_24 port map (
    O => CS0_N,
    I => cs0_n_i);
cs1buf :  OBUF_F_24 port map (
    O => CS1_N,
    I => cs1_n_i);
mackbuf :  OBUF_F_24 port map (
    O => DMACK_N,
    I => dmack_n_i);

intrqbuf : IBUF port map (
  I => INTRQ,
```

```
  O => intrq_i);

pb_data_tri <= not pb_data_tri_n;
dbuf : for i in 0 to 15 generate
  dbuf : IOBUF_F_24 port map (
    IO => PB_D(i),
    O => pb_data_in(i),
    I => pb_data_out(i),
    T => pb_data_tri
  );
end generate;


end Behavioral;
```

In the design document accompanying the code the author said that the user could hard-code the address values and assert a write or read signal and the state machine should have performed the requested operations.

However, we found the code to be no better than our previous attempts. After writing an OPB interface code for the module and adding proper buffering for the PB signals. We tried to hard code address values for accessing the data, but we could not get an active response from the card. Although the card did occasionally respond with a high interrupt signal, (a prerequisite of functionality that was the response saying the card had performed the operation) however the XESS state machine did not respond to it. In fact, we could not get any varying signals out of the card when we tested the flash card accesses with an oscilloscope.

An additional point is that the CompactFlash card works on all platforms we have tried. While it is possible that the card is not compatible with our board, or it may speak a different protocol.

Our next alternative, as shown in 4, was to stream the audio over the Ethernet. We had given up on the CompactFlash rather late and thus, starting the Ethernet from scratch would have required a Herculaen effort. Luckily, the JayCam group graciously provided their work on the Ethernet coupled with the SRAM. Unfortunately, their work only enabled sending data from the XESS board to the PC connected via Ethernet. We need bidirectional Ethernet with a heavier emphasis on the ability to receive data from the PC on to the XESS board.

2.1.2 VGA Character Display: Replacing the SRAM with a BRAM

With the exception of some minor changes to the original Lab 5 memory controller and pad_io components, the following comprises the changes we made to the Character Display to remove dependence on the SRAM.

```
----------------------------------------------------------------------------
--
-- VGA video generator
--
-- Uses the vga_timing module to generate hsync etc.
-- Massages the RAM address and requests cycles from the memory controller
-- to generate video using one byte per pixel
--
-- Cristian Soviani, Dennis Lim, and Stephen A. Edwards
--
----------------------------------------------------------------------------

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

library UNISIM;
```

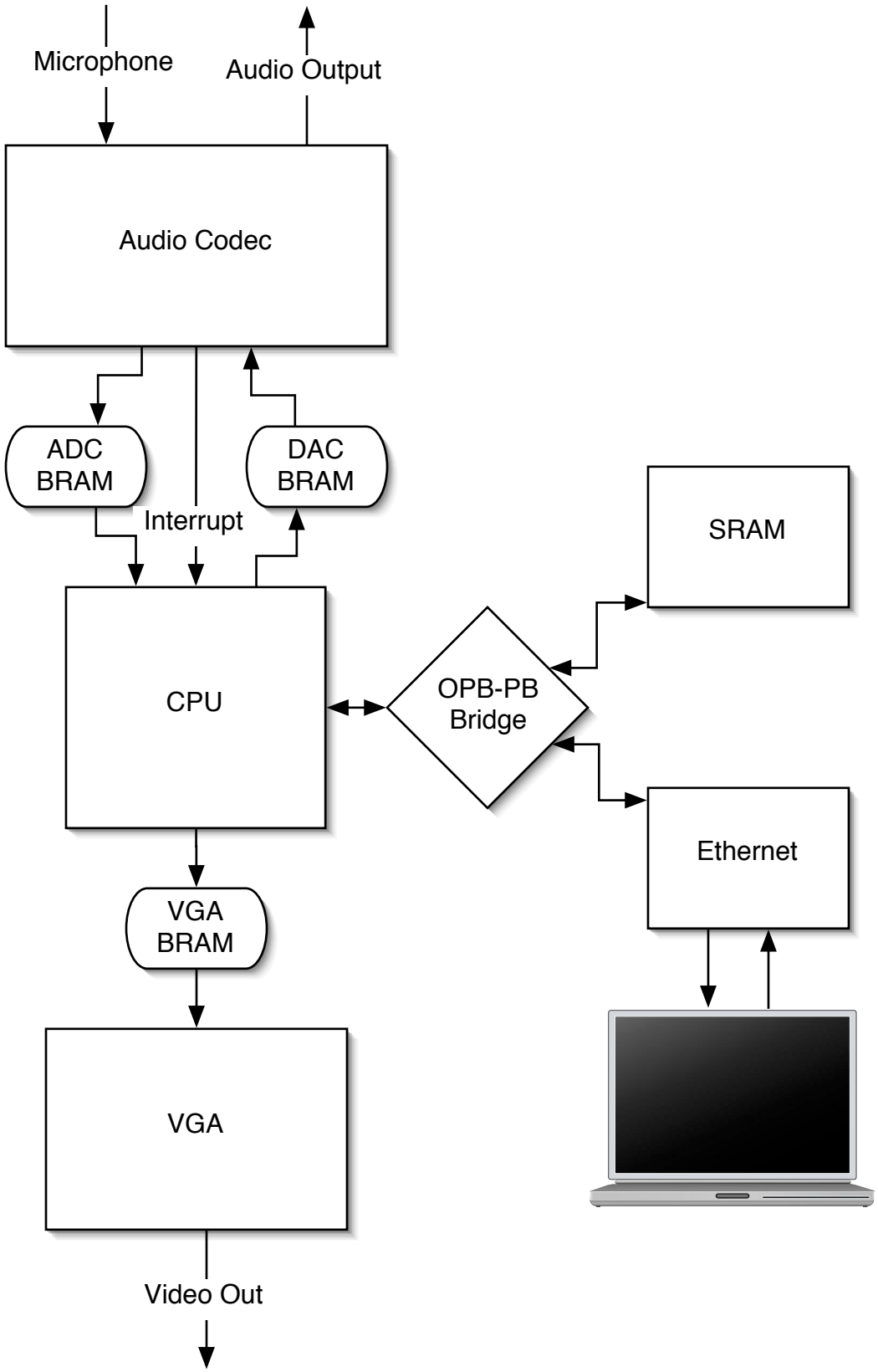


Figure 4: Implementation of Karaoke Machine Using Ethernet

```
use UNISIM.VComponents.all;

entity opb_xsb300_vga is
generic (
C_OPB_AWIDTH : integer := 32;
C_OPB_DWIDTH : integer := 32;
C_BASEADDR : std_logic_vector := X"FEFF_0200";
C_HIGHADDR : std_logic_vector := X"FEFF_03FF");

port (
OPB_Clk : in std_logic;
OPB_Rst : in std_logic;
OPB_ABus : in std_logic_vector (31 downto 0);
OPB_BE : in std_logic_vector (3 downto 0);
OPB_DBus : in std_logic_vector (31 downto 0);
OPB_RNW : in std_logic;
OPB_select : in std_logic;
OPB_seqAddr : in std_logic;
UIO_retry : out std_logic := '0';
UIO_DBus : out std_logic_vector (31 downto 0) := "00000000000000000000000000000000";
UIO_errAck : out std_logic;
UIO_toutSup : out std_logic;
UIO_xferAck : out std_logic;
pixel_clock : in std_logic;
VIDOUT_CLK : out std_logic;
VIDOUT_RCR : out std_logic_vector(9 downto 0);
VIDOUT_GY : out std_logic_vector(9 downto 0);
VIDOUT_BCB : out std_logic_vector(9 downto 0);
VIDOUT_BLANK_N : out std_logic;
VIDOUT_HSYNC_N : out std_logic;
VIDOUT_VSYNC_N : out std_logic);
end opb_xsb300_vga;

architecture Behavioral of opb_xsb300_vga is
-- Fast low-voltage TTL-level I/O pad with 12 mA drive

  component OBUF_F_12
port (
O : out STD_ULOGIC;
I : in STD_ULOGIC);
end component;

-- Basic edge-sensitive flip-flop

component FD
port (
C : in std_logic;
D : in std_logic;
Q : out std_logic);
end component;

-- Force instances of FD into pads for speed

attribute iob : string;
```

```
attribute iob of FD : component is "true";

component vga_timing
port (
h_sync_delay  : out std_logic;
v_sync_delay  : out std_logic;
blank : out std_logic;
vga_ram_read_addresser : out std_logic_vector (7 downto 0);
pixel_clock : in std_logic;
reset : in std_logic;
font_line : out std_logic_vector(2 downto 0);
                        line_loc : out std_logic_vector(9 downto 0);
need_reload : out std_logic
);

end component;


component fontrom128_64
port (
Clk : in std_logic;
en : in std_logic;
addr : in std_logic_vector( 15 downto 0 );
hline : in std_logic_vector( 2 downto 0 );
data : out std_logic_vector( 15 downto 0 )
);
end component;

signal r : std_logic_vector (9 downto 0);
signal g : std_logic_vector (9 downto 0);
signal b : std_logic_vector (9 downto 0);
signal blank : std_logic;
signal hsync : std_logic;
signal vsync : std_logic;
signal vga_ram_read_address : std_logic_vector(7 downto 0);
signal video_data : std_logic_vector(15 downto 0);
signal vreq : std_logic;
signal vreq_1 : std_logic;
signal load_video_word : std_logic;
signal vga_shreg : std_logic_vector(15 downto 0);
signal cur_line : std_logic_vector(2 downto 0);
signal reload : std_logic;
signal font_map : std_logic_vector(15 downto 0);
signal cur_pix : std_logic_vector(2 downto 0);
signal counter : std_logic_vector(3 downto 0) := "0000";
signal cs : std_logic;
signal q0 : std_logic;
signal d0 : std_logic;
signal xfer : std_logic;
signal wdata : std_logic_vector( 7 downto 0 );
signal addr : std_logic_vector( 8 downto 0 );
        signal yloc:std_logic_vector (9 downto 0);
        signal mask : std_logic_vector(2 downto 0);
        signal mask2 : std_logic_vector(2 downto 0);
```

```
--  signal beg_time_reg : std_logic_vector( 8 downto 0 );
--  signal end_time_reg : std_logic_vector( 8 downto 0 );
  signal write_enable : std_logic;
--  signal color_mask : std_logic_vector( 2 downto 0 ) := "111";
begin
st : vga_timing port map (
pixel_clock => pixel_clock,
reset => OPB_Rst,
h_sync_delay => hsync,
v_sync_delay => vsync,
blank => blank,
vga_ram_read_addresser => vga_ram_read_address,
font_line => cur_line,
need_reload => reload,
                line_loc => yloc
);

buff : RAMB4_S8_S16 port map (
-- write
CLKA => OPB_Clk,
RSTA => OPB_Rst,
DIA => wdata,
ENA => '1',
WEA => write_enable,
ADDRA => addr,
DOA => open,

-- read
CLKB => pixel_clock,
RSTB => OPB_Rst,
DIB => X"0000",
ENB => '1',
WEB => '0',
ADDRB => vga_ram_read_address( 7 downto 0 ),
DOB => video_data
);

ft : fontrom128_64 port map (
Clk=>pixel_clock,
en=>'1',
addr=>video_data,
hline=>cur_line,
data=>font_map
);

UIO_DBus <= X"0000_0000";
cs <= OPB_select when ( OPB_ABus(31 downto 8) = X"FEFF02" or OPB_ABus( 31 downto 8 ) = X"FEFF03" ) else

process( OPB_Clk, OPB_Rst )
begin
if OPB_Rst = '1' then
d0 <= '0';
elsif OPB_Clk'event and OPB_Clk = '1' then
d0 <= ( not q0 ) and cs;
```

```
q0 <= d0;
end if;
end process;

xfer <= q0;
write_enable <= q0;

process( OPB_Clk )
begin
if OPB_Clk'event and OPB_Clk = '1' then
  --wdata <= OPB_DBus( 31 downto 24 );
                 wdata <= OPB_DBus( 7 downto 0 );
end if;
end process;

process( OPB_Clk )
begin
if OPB_Clk'event and OPB_Clk = '1' then
--addr <= OPB_ABus( 9 downto 1 );
               addr <= OPB_ABus( 8 downto 0 );
end if;
end process;

--        process( OPB_Clk )
--        begin
--         if OPB_Clk'event and OPB_Clk = '1' then
--            if vga_ram_read_address( 7 downto 0 ) = X"FE" then
--              beg_time_reg <= video_data( 0 ) & video_data( 15 downto 8 );
--            elsif vga_ram_read_address( 7 downto 0 ) = X"FF" then
--              end_time_reg <= video_data( 0 ) & video_data( 15 downto 8 );
--            end if;
--          end if;
--        end process;

-- Video request is true when the RAM address is even

-- FIXME: This should be disabled during blanking to reduce memory traffic

--vreq <= reload ;--not vga_ram_read_address(0);

--vreq <= not vga_ram_read_address(0);
-- Generate load_video_word by delaying vreq two cycles

--process (pixel_clock)
--begin
--if pixel_clock'event and pixel_clock='1' then
-- if counter="1111" then
-- load_video_word <= '1';

                          --       counter <= "0000";
--else
-- load_video_word <='0';
-- counter <= counter+1;
--end if;
```

```
--end if;
--end process;

-- Generate video_req (to the RAM controller) by delaying vreq by
-- a cycle synchronized with the pixel clock

--process (clk)
--begin
--if clk'event and clk='1' then
--video_req <= pixel_clock ;--and vreq;
--end if;
--end process;

-- The video address is the upper 19 bits from the VGA timing generator
-- because we are using two pixels per word and the RAM address counts words

--video_addr <= vga_ram_read_address(19 downto 0);

-- The video shift register: either load it from RAM or shift it up a byte

process (pixel_clock)
begin
if pixel_clock'event and pixel_clock='1' then
if reload = '1' then
--if reload = '1' then
vga_shreg <= font_map(7 downto 0) & font_map(15 downto 8); --video_data;
-- Shift the low byte of read video data into the high byte
else
                          vga_shreg <= vga_shreg(14 downto 0) & "0";
end if;
                        if yloc>255 and yloc<304 then
                          mask <= "111";
                        else
                          mask <= "000";
                        end if;
                        if yloc>255 and yloc <264 then
                          mask2 <="000";
                        else
                          mask2 <= "111";
                        end if;
end if;
end process;

--        process( pixel_clock )
--        begin
--          if pixel_clock'event and pixel_clock = '1' then
--            if vga_ram_read_address( 8 downto 0 ) = beg_time_reg then
--              color_mask <= "000";
--            elsif vga_ram_read_address( 8 downto 0 ) = end_time_reg then
--              color_mask <= "111";
--            end if;
--          end if;
--        end process;
```

```
-- Copy the upper byte of the video word to the color signals
-- Note that we use three bits for red and green and two for blue.

r(9 downto 7) <= vga_shreg(15)&vga_shreg(15)&vga_shreg(15) and mask; -- and color_mask;
r(6 downto 0) <= "0000000";
g(9 downto 7) <= (vga_shreg(15)&vga_shreg(15)&vga_shreg(15) and mask) and mask2;
g(6 downto 0) <= "0000000";
b(9 downto 8) <= vga_shreg(15)&vga_shreg(15) and mask(1 downto 0); -- and color_mask(
b(7 downto 0) <= "00000000";

-- Video clock I/O pad to the DAC

vidclk : OBUF_F_12 port map (
O => VIDOUT_clk,
I => pixel_clock);

-- Control signals: hsync, vsync, and blank

hsync_ff : FD port map (
C => pixel_clock,
D => not hsync,
Q => VIDOUT_HSYNC_N );

vsync_ff : FD port map (
C => pixel_clock,
D => not vsync,
Q => VIDOUT_VSYNC_N );

blank_ff : FD port map (
C => pixel_clock,
D => not blank,
Q => VIDOUT_BLANK_N );

-- Three digital color signals

rgb_ff : for i in 0 to 9 generate

r_ff : FD port map (
C => pixel_clock,
D => r(i),
Q => VIDOUT_RCR(i) );

g_ff : FD port map (
C => pixel_clock,
D => g(i),
Q => VIDOUT_GY(i) );

b_ff : FD port map (
C => pixel_clock,
D => b(i),
Q => VIDOUT_BCB(i) );

end generate;
```

```
UIO_errAck <= '0';
UIO_retry <= '0';
UIO_toutSup <= '0';
UIO_xferAck <= xfer;

end Behavioral;
```

2.1.3 Enabling the Interrupt from the Audio Codec

```
BEGIN opb_intc
 PARAMETER INSTANCE = intc
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0xFFFF0000
 PARAMETER C_HIGHADDR = 0xFFFF00FF
 PORT OPB_Clk = sys_clk
 PORT Intr =  ak4565_intr & uart_intr
 PORT Irq =   intr
 BUS_INTERFACE SOPB = myopb_bus
END

BEGIN opb_xsb300_ak4565
 PARAMETER INSTANCE = audio
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFEFE0000
 PARAMETER C_HIGHADDR = 0xFEFEFFFF
 PORT OPB_Clk = sys_clk
 PORT AU_CSN_N =  AU_CSN_N
 PORT AU_BCLK =  AU_BCLK
 PORT AU_MCLK = AU_MCLK
 PORT AU_LRCK = AU_LRCK
 PORT AU_SDTI = AU_SDTI
 PORT AU_SDTO0 = AU_SDTO0
 PORT Interrupt = ak4565_intr
 BUS_INTERFACE SOPB = myopb_bus
END
```

In order to enable the interrupt from the audio codec, we first added a line, ak4565_intr, from the Interrupt line defined by the our AK4564 VHDL component. Then, we concatenated the signal to the interrupt vector used by the interrupt controller.

2.1.4 Microphone Input into the Audio Codec

After we were able to play wav file, the next step of the Karaoke machine is to enable the microphone input.

We are using the AK4565 20 bit codec, that features a four stereo input selector which switches among microphone and line inputs. However the audio codec does not accept input from the microphone by default, therefore we must configure the audio codec through one of its control registers. Figure 5 shows the control register address and content needed in order to command the codec to accept microphone input.

In order to configure the codec we have to abide the following timing rules as shown in figure 7:

However, the designers of the XSB300E made it such that the pins needed for the following signals to be shared with SRAM accesses: CCLK(PB-D0), CDTI(PB-D1), CDTO(PB-D2). This is also demonstrated in the figure 7.

Therefore, in order to arbitrate the use of the PB bus between SRAM accesses and audio codec configurations, we must build a multiplexor inside the OPBbus's Pad_io module.

After many trials we were able to configure the codec to allow the input from the microphone and have its signals amplified. However, we discovered that the configuration process was very capricious, sometimes

■ **Register Definitions**

All registers inhibit writing at PDN pin = "L".
Writing to 08H and 09H is ignored and these addresses respond "0" at reading.
For addresses from 0AH to 1FH, data must not write.

Addr	Register Name	D7	D6	D5	D4	D3	D2	D1	D0
00H	Input Select	0	0	0	0	LINE	EXT	INT1	INT0
R/W		R/W							
Default		0	0	0	0	0	0	0	1

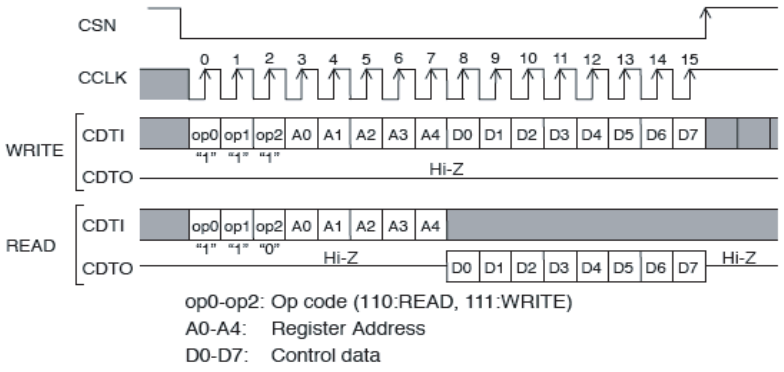
INT0: Select ON/OFF of INTL0 and INTR0 (0: OFF, 1: ON)
INT1: Select ON/OFF of INTL1 and INTR1 (0: OFF, 1: ON)
EXT: Select ON/OFF of EXTL and EXTR (0: OFF, 1: ON)
LINE: Select ON/OFF of LIN and RIN (0:OFF, 1:ON)

When LINE bit is "1", INT0, INT1 and EXT bits are ignored. These inputs are always OFF.
When LINE bit is "1", the gain table of IPGA switches LINE side.

When LINE bit is "0", if INT0, INT1 and EXT bits go to "1" at the same time, the input signals are mixed by 0dB gain.

Figure 5: The address and content of the control register.

CCLK always needs 16 edges of "↑" during CSN = "L". Reading/Writing of the address except 00H – 09H are inhibited. Reading/Writing of the control registers by except op0 = op1 = "1" are invalid.



op0-op2: Op code (110:READ, 111:WRITE)
A0-A4: Register Address
D0-D7: Control data

Figure 16. Control Data Timing

Figure 6: The timing diagram for writing to a control register.

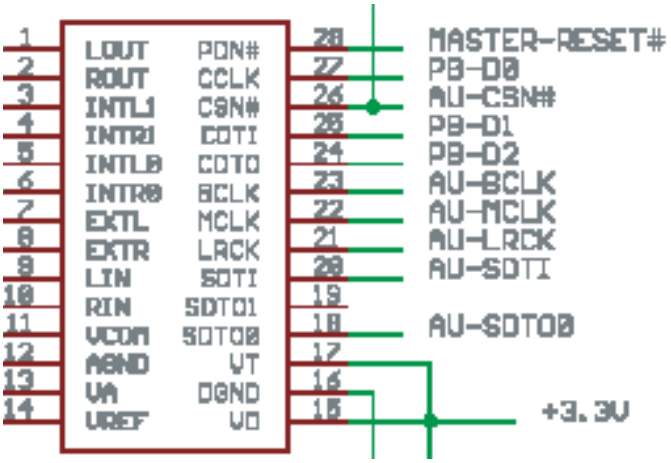


Figure 7: The timing diagram for writing to a control register.

we were able to configure it, sometimes, it just seems it can not be done. After spending countless hours on the microphone, we discovered that there is a glitch in the data line, since we depend on CCLK to generate a steady clock signal to write to the register, any glitch can not be tolerated.

We therefore proceeded to talk to Christian about our issue with configuring the AK4565 codec, it then suggested by him that we use an amplifier instead of using the AK4565 codec for microphone amplification and sound mixing.

Now we have the system set up as follows:

1. The output of the AK4565 Codec is connected to the input of the amplifier.
2. The microphone is connected to the microphone input of the amplifier.
3. The output of the amplifier is connected to two speakers.
4. The amplifier does the mixing of the audio input with its microphone input.

2.2 Software Implementation

2.2.1 Passing Data to the Audio Codec

```
#include "xbasic_types.h"
#include "xio.h"

unsigned long intcounter;
extern volatile unsigned char readybuf[16];

void audio_handler( void *callback ) {
  Xuint32 channel, sample;
  Xuint8 s;
  int i;

  for(i=0;i<64;i++){
    if ( !(i&0x03) ){
      s = readybuf[(i>>2)&0x0FF];
      channel = ( 0x000000FF & s ) << 7;
      sample = ( channel << 16 ) | channel;
      XIo_Out32( 0xFEFE0000 + i*4 , sample );
      continue;
    }
    XIo_Out32( 0xFEFE0000 + i*4, sample );
  }
  intcounter++;
}
```

The audio_handler function read 8b, mono, 11kHz audio samples from a buffer. The audio codec expects 16b, stereo, 44kHz audio samples, so we had to widen each sample and duplicate the widened sample to both the left and right channels. At this point, the audio sample is in a 32b form. Finally, we simply duplicate this 32b samle four times in order to upsample to the full 44kHz that the audio codec expects.

2.2.2 Lyrics

Our on-screen lyrics display is generated by downloading the lyrics from a text file to the SRAM and displaying it one by one with a configured delay. The lyrics are sent through the UART along with the wav file but stored at different locations on the SRAM. Synchronizing the lyrics with the playback of the audio was configured manually by ear. The configured delay for displaying lyrics is based on a counter from the audio handler interrupt that loops consistently to the values specified by the lyric input file. A line of text is first displayed on the screen and then the synchronized version is replayed underneath the original text,

matching the lyrics of the song. The format of our lyric file contains the following: the total time needed to output the line of text, the actual text itself, and the weight values of each letter in the text. A sample of a line of text from the lyric input file is:

```
10                    // Total Display Time for "HELLO WORLD"
HELLO WORLD           // Text to be displayed on screen
1 1 1 1 1 1 2 2 2 2 2 // Time weights per character
...
0                     // Terminates lyrics file
```

Note that the total time, T_{total}, multiplied by a weighted delay value, λ_i, divided by the total weighted delay value, T_w, of all the letters equals the delay for the individual letter, t_i.

$$\begin{aligned} \Lambda &= \sum_{i=0}^{N} \lambda_i \\ t_i &= \frac{\lambda_i}{\Lambda} T_{total} \\ T_{total} &= \sum_{i=0}^{N} t_i \end{aligned}$$

The total time, T_{total}, that the line is present on the screen is equal to the sum of each character's weighted time, t_i, where t_i is determined by weight, λ_i.

As in Lab 5, we have implemented a character ROM that stores all the alphabet letters as 8x8bit characters.

An alternative method of displaying lyrics was to interleave the lyrics text into the audio data to make it more efficient to stream the audio file. A lyrics timing identifier would be embedded in the text and outputted according to the position of the audio data. The main concern was the synchronization of the text with the data, which would be difficult compared to the loop with the counter based on the audio interrupt.

2.2.3 Merging the Lyrics and Audio

To stream the audio and lyrics, we devised the following format for interleaving the audio and text.

Number of Lyrics Blocks (4B)	**Lyric Block (5B)**	...	**.WAV Header**	**Audio Samples**	...

Each lyrics block is broken into the following:

Number of Interrupts (4B)	**Character (1B)**

This format affords us a very fine level of control over when to play audio and how long the cursor should remain on a particular character.

2.2.4 Synchronizing the Lyrics Display and Audio

Synchronizing the display of the lyrics and audio playback depends on the interrupts initiated by the audio codec.

3 Summary

3.1 Who Did What

Will produced sample songs with sychronized lyrics.

Chia-Hung worked with Will on streaming audio and lyrics/display and audio synchronization.

Stephen worked with Oleg on compactflash, basic audio player, and bram character display.

Oleg worked with Stephen on compactflash, basic audio player, and bram character display.

Zijian worked on microphone input and audio mixing.

3.2 What We Learned

We have learned many valuable lessons during these two months. From the beginning, we underestimated the complexity of the project and many of us started working on the complex parts of the project first. After facing many challenges and spending lots of time on these parts, we then started working on simpler things.

This project would have been far more enjoyable if we had fully understood the XESS board's capabilities and especially its limitations. Perhaps, we have been trained to look at a computer's specfications, like a Pentium motherboard, and assume that each listed component actually works flawlessly. With respect to the XESS board, this could not be further from the truth. Considering how much our project depends on the CompactFlash, this lack of foresight made implementing our project idea very difficult.

In addition to these lessons, the most significant fact we learned over the course of implementing our karaoke machine is that **Cristian is a VHDL coding beast**.

3.3 Advice for Next Year's Students

Consequently, it is imperative that next year's students truly understand the implications for using a device like the CompactFlash or Ethernet. The devices might be coupled to the SRAM which would necessitate a rather effort to construct a state machine to arbitrate accesses between these peripherals and across the common bus. Many of the devices are dumped on to the same peripheral bus which makes synchronous access very difficult.