M.A.S.H. Design Document

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

April 3, 2004

1 Introduction

For our Embedded Systems Final Design Project, we are implementing a basic karaoke machine. Audio files in the form of WAVs will be played from the CompactFlash while simultaneously displaying the lyrics of the song on the screen.

2 Design Approach

Our initial step is to first implement the CompactFlash which will store the audio and lyrics. Once we can read specific blocks off the CompactFlash, we need to be able to understand the FAT16 file system to read the audio and lyrics files properly. Next, we intend to implement the SDRAM memory system to provide a way to buffer data from the CF to the SDRAM. Timing will be especially critical. With these components in place, we will then interface with the audio codec. We now need to properly share the extracted audio data between the CompactFlash, the memory buffer, and ultimately the audio codec. Once the basic parts work, we can implement the actual synchronized audio and display of lyrics.

3 CompactFlash Interface

3.1 Introduction

The CompactFlash technology is an open standard. It provides a high capacity data storage and I/O functionality. Since our goal is to build a karaoke machine, we will need some space to store all the songs and the text for song lyrics. The CompactFlash standard provides us a relatively inexpensive and simple way to accomplish our goal.

3.2 Access Modes

The CompactFlash standard provides three access modes: Memory Mode, I/O Mode, and True IDE Mode. The True IDE mode works exactly like the standard IDE interface. I/O mode is often used for non-storage type devices such as GPS and digital cameras. The Memory Mode method treats the CompactFlash card as a single memory buffer. Since our karaoke machine will necessitate the implementation of many individual components, it will also depend heavily on RAM. The Memory Mode of the CompactFlash works similar to RAM access. Thus, we plan to use the Memory Mode in order to shorten the development time.

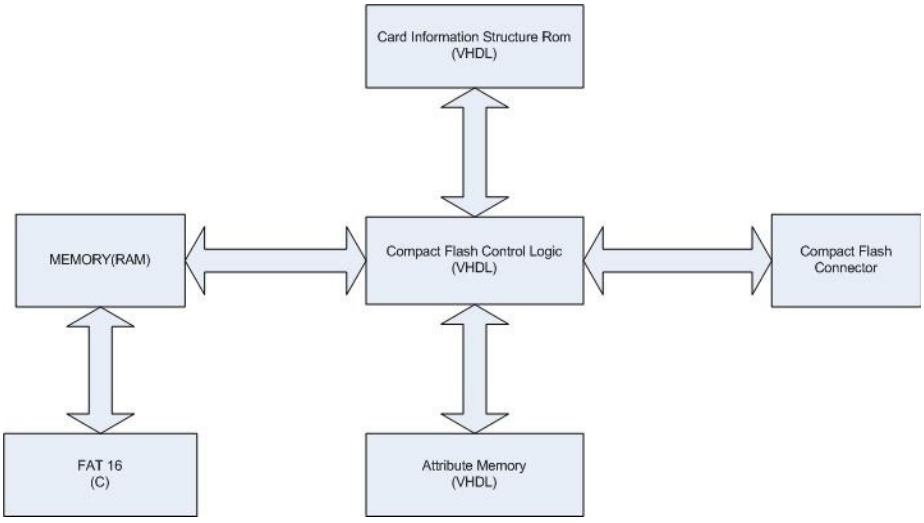


Figure 1: Overall Implementation of CompactFlash

3.3 Design of CompactFlash

The main part of the CompactFlash is the Control Logic. We plan on using VHDL. First, we have to understand the timing diagram in the following section and follow the diagram to control the CompactFlash card. Also, we should build some auxiliary components. One of them, the Card Information Structure ROM, will provide the CompactFlash Control Logic with information about the CompactFlash card. For example, the CompactFlash Control Logic may require data from the CompactFlash card; it will use the Card Information Structure ROM to retrieve the real meaning of the data. We will also need an Attribute Memory which will store information about the CompactFlash card configuration.

In order to control the CompactFlash Control Logic, we should monitor the CompactFlash Control Logic using some memory buffer where the FAT16 file system will reside and communicate with the CompactFlash Control Logic. In summary, the CompactFlash Control Logic will deal with all the operations related to reading, controlling, and storing data in the memory which will be shared by other parts of our design.

3.4 CompactFlash Pins in Memory Mode

The following two tables detail the pins of the CompactFlash interface.

Signal Name	Dir	Pin	Description
A10 - A0	I	8, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20	These address lines along with the -REG signal are used to select the following: The I/O port address registers within the CompactFlash Storage Card or CF+ Card, the memory mapped port address registers within the CompactFlash Storage Card or CF+ Card, a byte in the card's information structure and its configuration control and status registers.
BVD1	I/O	46	This signal is asserted high, as BVD1 is not supported.
BVD2	I/O	45	This signal is asserted high, as BVD2 is not supported.
-CD1, -CD2	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
-CE1, -CE2	I	7, 32	These input signals are used both to select the card and to indicate to the card whether a byte or a word operation is being performed. -CE2 always accesses the odd byte of the word.-CE1 accesses the even byte or the Odd byte of the word depending on A0 and -CE2. A multiplexing scheme based on A0, -CE1, -CE2 allows 8 bit hosts to access all data on D0-D7
-CSEL	I	39	This signal is not used for this mode, but should be connected by the host to PC Card A25 or grounded by the host.
D15 - D00	I/O	31, 30, 29, 28, 27, 49, 48, 47, 6, 5, 4, 3, 2, 23, 22, 21	These lines carry the Data, Commands and Status information between the host and the controller. D00 is the LSB of the Even Byte of the Word. D08 is the LSB of the Odd Byte of the Word.
GND	–	1, 50	Ground.
-OE	I	9	This is an Output Enable strobe generated by the host interface. It is used to read data from the CompactFlash Storage Card or CF+ Card in Memory Mode and to read the CIS and configuration registers

Signal Name	Dir	Pin	Description
READY	O	37	This signal is set high when the CompactFlash Storage Card or CF+ Card is ready to accept a new data transfer operation and is held low when the card is busy. At power up and at Reset, the READY signal is held low (busy) until the Compact Flash Storage Card or CF+ Card has completed its power up or reset function. No access of any type should be made to the CompactFlash Storage Card or CF+ Card during this time. Note, however, that when a card is powered up and used with +RESET continuously disconnected or asserted, the reset function of this pin is disabled and consequently the continuous assertion of +RESET will not cause the READY signal to remain continuously in the busy state
-REG	I	44	This signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. High for Common Memory, Low for Attribute Memory.
RESET	I	41	When the pin is high, this signal Resets the Compact Flash Storage Card or CF+ Card. The CompactFlash Storage Card or CF+ Card is Reset only at power up if this pin is left high or open from power-up. The CompactFlash Storage Card or CF+ Card is also Reset when the Soft Reset bit in the Card Configuration Option Register is set.
VCC	–	13, 38	+5 V, +3.3 V power.
-VS1, -VS2	O	33, 40	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.
-WAIT	O	42	The -WAIT signal is driven low by the CompactFlash Storage Card or CF+ Card to signal the host to delay completion of a memory or I/O cycle that is in progress.
-WE	I	36	This is a signal driven by the host and used for strobing memory write data to the registers of the CompactFlash Storage Card or CF+ Card when the card is configured in the memory interface mode. It is also used for writing the configuration registers.
WP	O	24	The CompactFlash Storage Card or CF+ Card does not have a write protect switch. This signal is held low after the completion of the reset initialization sequence.

3.5 Timing Diagrams and Signal Interface

The CompactFlash is very complex. We emulate the following timing diagrams. Attribute Memory access time is defined as 300ns. Detailed timing specifications are shown in following tables.

Speed Version			300 ns	
Item	Symbol	IEEE Symbol	Min ns.	Max ns.
Read Cycle Time	tc(R)	tAVAV	300	
Address Access Time	ta(A)	tAVQV		300
Card Enable Access Time	ta(CE)	tELQV		300
Output Enable Access Time	ta(OE)	tGLQV		150
Output Disable Time from CE	tdis(CE)	tEHQZ		100
Output Disable Time from OE	tdis(OE)	tGHQZ		100
Address Setup Time	tsu (A)	tAVGL	30	
Output Enable Time from CE	ten(CE)	tELQNZ	5	
Output Enable Time from OE	ten(OE)	tGLQNZ	5	
Data Valid from Address Change	tv(A)	tAXQX	0	

Note: All times are in nanoseconds. Dout signifies data provided by the CompactFlash Storage Card or CF+ Card to the system. The -CE signal or both the -OE signal and the -WE signal must be de-asserted between consecutive cycle operations.

Figure 2: Various Actual Timings of Attribute Memory Reads

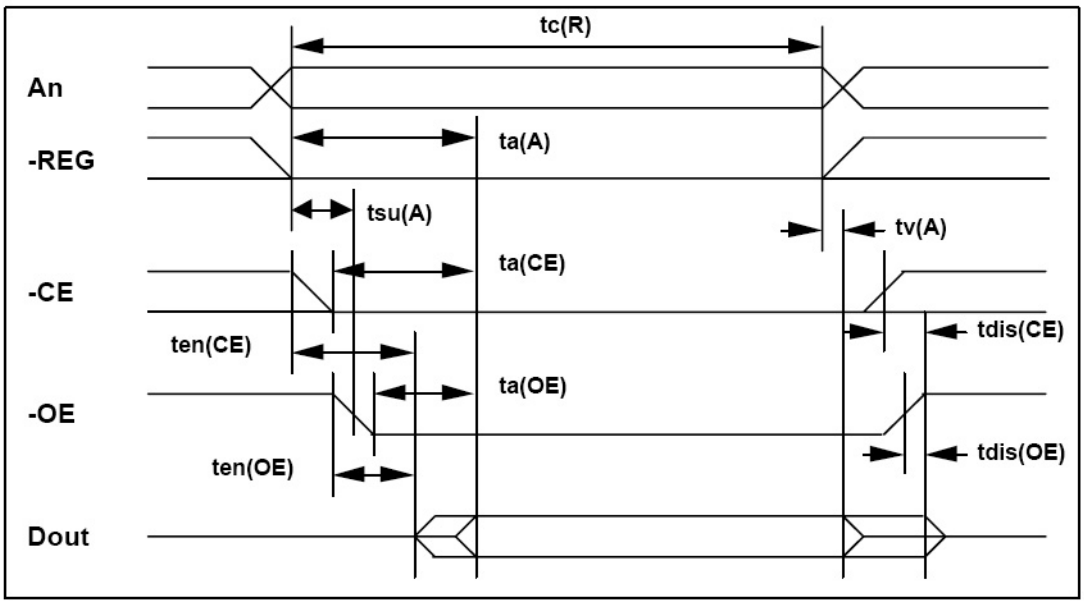


Figure 3: Timing Diagram of Attribute Memory Read

Item	Symbol	IEEE Symbol	Min ns.	Max ns.
Output Enable Access Time	ta(OE)	tGLQV		125
Output Disable Time from OE	tdis(OE)	tGHQZ		100
Address Setup Time	tsu(A)	tAVGL	30	
Address Hold Time	th(A)	tGHAX	20	
CE Setup before OE	tsu(CE)	tELGL	0	
CE Hold following OE	th(CE)	tGHEH	20	
Wait Delay Falling from OE	tv(WT-OE)	tGLWTV		35
Data Setup for Wait Release	tv(WT)	tQVWTH		0
Wait Width Time	tw(WT)	tWTLWTH		350 (3000 for *CF+*)

Note: The maximum load on -WAIT is 1 LSTTL with 50 pF total load. All times are in nanoseconds. Dout signifies data provided by the CompactFlash Storage Card or CF+ Card to the system. The -WAIT signal may be ignored if the -OE cycle to cycle time is greater than the Wait Width time. The Max Wait Width time can be determined from the Card Information Structure. The Wait Width time meets the PCMCIA specification of 12µs but is intentionally less in this specification.

Figure 4: Various Actual Timings of Common Memory Reads

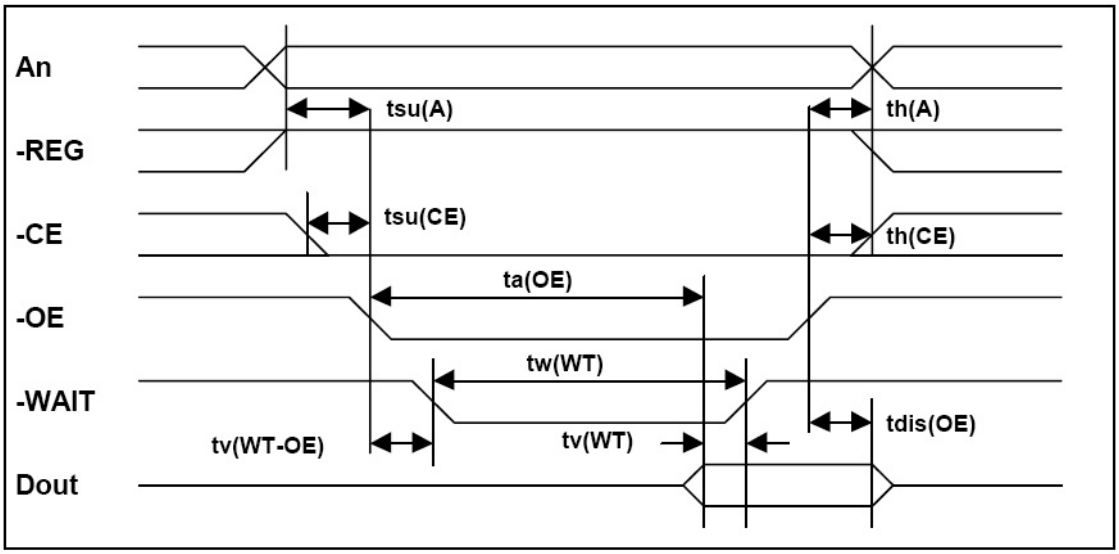


Figure 5: Timing Diagram of Common Memory Read

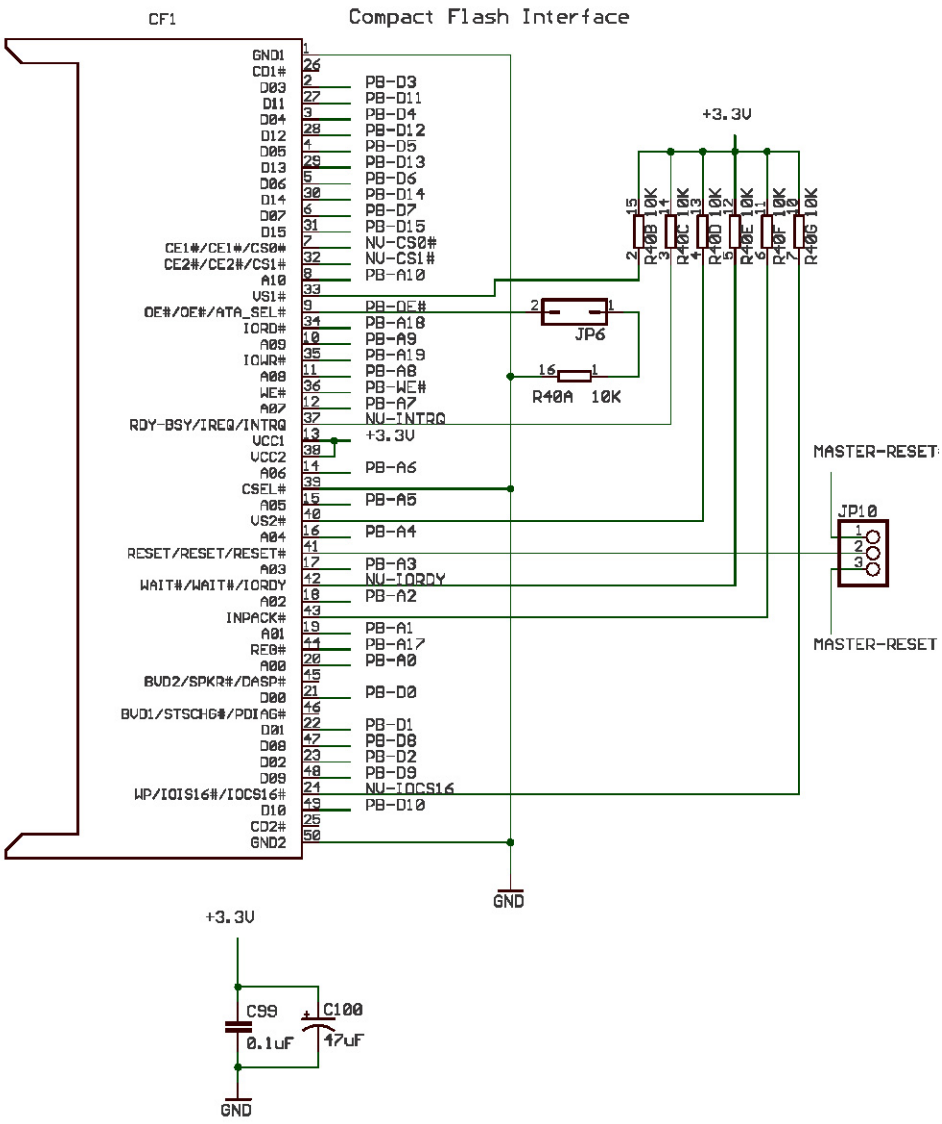


Figure 6: Mapping of the FPGA Pins to the CompactFlash Connector

4 FAT16

4.1 Introduction

The FAT16 file system is a simple file system used by CompactFlash cards to store files. Before we can actually start using a file system, we need to understand how the hard drive organizes partitions that divide up the various file systems that might be present on a CompactFlash card. Considering the scope of M.A.S.H., we intend to assume that the first partition on the CompactFlash will be a FAT16 one.

Since the CompactFlash will be read using Memory Mode, which has also been called Common Mode, the FAT16 file system layer will access the CompactFlash through simple I/O memory reads. In other words, the FAT16 layer will write a specific memory location to reference a specific block on the CompactFlash. The data from the CompactFlash will be written to a specific memory location accesible to the FAT16 layer.

4.2 Master Boot Record

Located at Cylinder 0, Head 0, Sector 1, in the first 512 bytes, the Master Boot Record keeps track of the various partitions that exist on the hard drive. The first partition entry, whose details are encapsulated in 16 bytes, is located at offset 0x1BE.

4.3 Partition Entry

The block of 16 bytes is organized according to the following table. Since we are using a 64MB CompactFlash card, the "Type of Partition" should contain a value of 0x06, which indiciates that the partition of a 16-bit FAT (Partition Larger than 32MB).

Offset	**Description**	**Size (bytes)**
00h	Current State of Partition (00h=Inactive, 80h=Active)	1B
01h	Beginning of Partition - Head	1B
02h	Beginning of Partition - Cylinder/Sector	2B
04h	Type of Partition	1B
05h	End of Partition - Head	1B
06h	End of Partition - Cylinder/Sector	2B
08h	Number of Sectors Between the MBR and the First Sector in the Partition	4B
0Ch	Number of Sectors in the Partition	4B

4.4 FAT16 Drive Layout

The following table describes how each partition is laid out. Since we will assume that the files will be located directly in the root file directory, we will not have to traverse further than the root directory entry, which should point directly the files to be used.

Offset	**Description**
Start of Partition	Boot Sector
Start + # of Reserved Sectors	FAT Tables
Start + # of Reserved + (# of Sectors Per FAT * 2)	Root Directory Entry
Start + # of Reserved + (# of Sectors Per FAT * 2) + ((Maximum Root Directory Entries * 32) / Bytes per Sector)	Data Area (Starts with Cluster #2)

4.5 FAT16 Boot Record

At the beginning of a FAT16 partition, within the first sector, the boot record and its various details reside. The following table breaks down the information located in the boot record.

Offset	Description	Size (bytes)
000h	Jump Code + NOP	3B
003h	OEM Name	8B
00Bh	Bytes Per Sector	2B
00Dh	Sectors Per Cluster	1B
00Eh	Reserved Sectors	2B
010h	Number of Copies of FAT	1B
011h	Maximum Root Directory Entries	2B
013h	Number of Sectors in Partition Smaller than 32MB	2B
015h	Media Descriptor (F8h for Hard Disks)	1B
016h	Sectors Per FAT	2B
018h	Sectors Per Track	2B
01Ah	Number of Heads	2B
01Ch	Number of Hidden Sectors in Partition	4B
020h	Number of Sectors in Partition	4B
024h	Logical Drive Number of Partition	2B
026h	Extended Signature (29h)	1B
027h	Serial Number of Partition	4B
02Bh	Volume Name of Partition	11B
036h	FAT Name (FAT16)	8B
03Eh	Executable Code	448B
1FEh	Executable Marker (55h AAh)	2B

4.6 Clusters

Though we do not have to look beyond the root directory entry, we will still need to refer to the FAT Tables in order to traverse the clusters that comprise the entire file. The FAT16 drive actually maintains multiple copies of the FAT table, one for updating and one for backup. Each row contains a value and associated meaning according to the following table.

FAT Code Range	Meaning
0000h	Available Cluster
0002h-FFEFh	Used, Next Cluster in File
FFF0h-FFF6h	Reserved Cluster
FFF7h	BAD Cluster
FFF8h-FFFF	Used, Last Cluster in File

4.7 Directory Table

Each entry in a directory table consists of 32 bytes. Assuming the directory entry is formated using the traditional DOS 8.3, the 32 bytes will consist of the following information (contained in yet another table).

Offset	Length	Value
0	8B	File Name
8	3B	Extension
11	1B	Attribute (00ARSHDV)
22	2B	Time
24	2B	Date
26	2B	Cluster
28	4B	File Size

The Attribute byte has the following breakdown:

- 0: unused bit
- A: archive bit,
- R: read-only bit
- S: system bit
- D: directory bit
- V: volume bit

4.8 Summary

Now that the FAT16 File System Format has been thoroughly broken into many tables, its use in the M.A.S.H. karaoke system can be described.

1. Read the 16 bytes that consists of the First Partition Entry of Master Boot Record.
2. Check to ensure that the partition is "Active". If Active, record the type, location, and dimensions of the drive.
3. Read the first partition's Boot Record.
4. Record the partition's meta information and pay particular attention to the "Sectors per Cluster" and "Sectors per FAT" which will help locate the start of the FAT Tables.
5. Load the first directory entry from the "Root Directory Entry".
6. Check to make sure the first entry is a file with a file name that ends in a supported audio format (ie. WAV format) or some text file that maintains lyrics or other necessary data.
7. Starting at the Cluster referenced by the directory entry, collect each cluster included by the file until a FAT Table lookup returns 0xFFFF.

5 Memory Controller

The memory controller acts like a traffic light at an intersection of a busy highway and a farm road. The Audio DAC is a car running on the highway, since its memory access must have a higher priority than the CPU, otherwise the audio playback will skip.

The audio player software is in charge of putting audio data into a circular buffer. The audio player will not signal the Audio DAC to start playback until the circular buffer wraps around and reaches the beginning of the buffer. When this happens, the audio player signals the Audio DAC to commence. When the memory controller sees the Audio DAC requesting access to the circular buffer, the memory controller will prevent the audio player from writing to the buffer and force it to wait until the DAC is finished.

The implementation of the memory controller will take the form of the included finite state machine. When the audio player needs to access memory, the access will granted provided that there is no other processes of higher priority such as the Audio DAC reading or writing to memory. When the DAC reads from memory, accesses from the audio player are restricted.

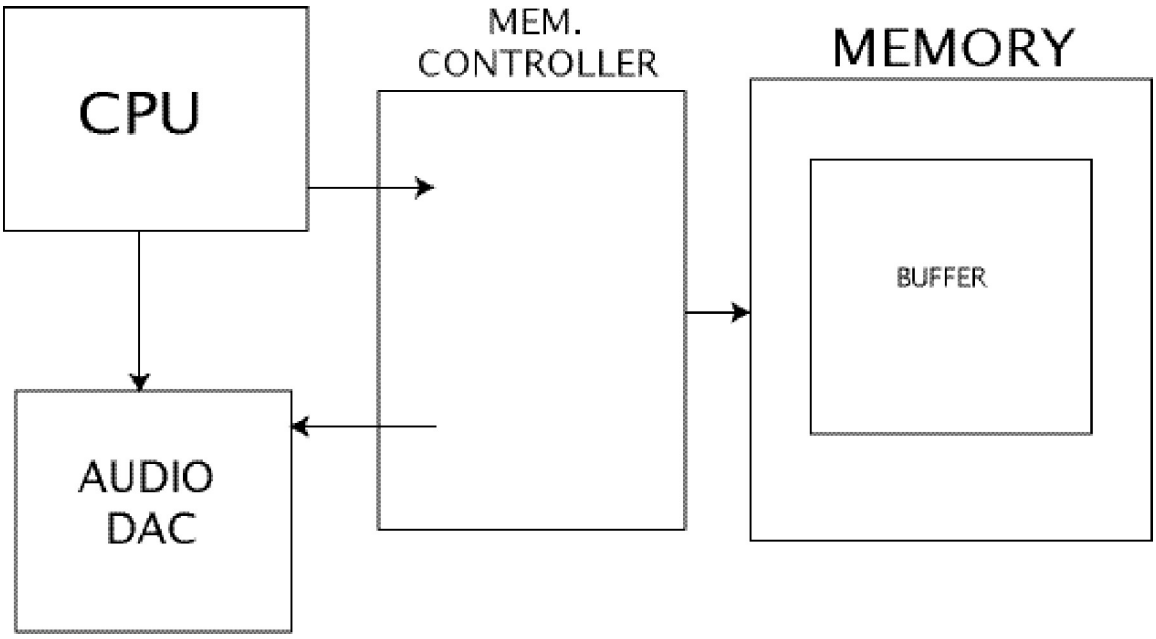


Figure 7: Memory Controller and Neighboring Components

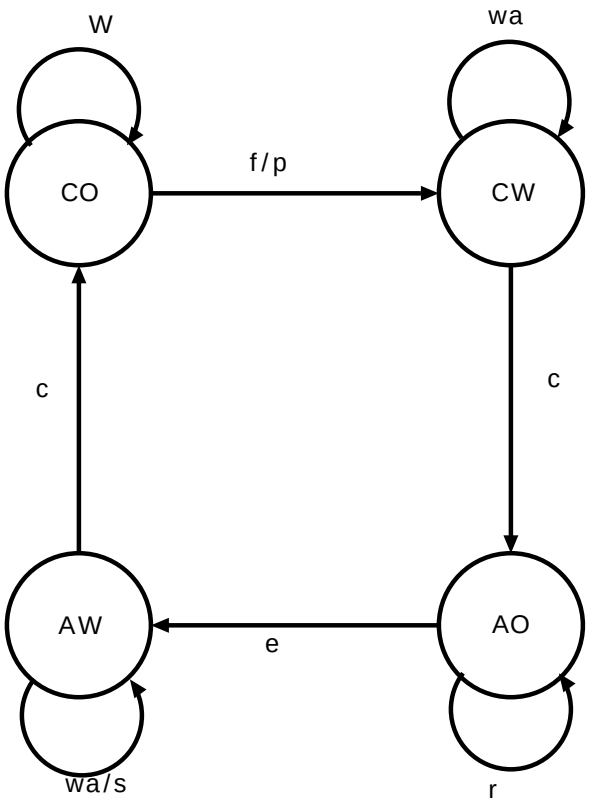
6 Audio Codec

The Audio DAC requires several control signals which control how the audio data is played back. It takes in serial audio streams at a predefined rate. First, our audio player will parse the WAV file header and use the parsed settings to setup the DAC. Next, the audio data will be written to a BRAM buffer by the audio player, a C program, which coordinates the entire audio playback process. The buffer acts as a two port FIFO which will deliver the data to a control module. This control module will send it serially into the DAC. We are not yet sure if we can write to the buffer directly from the flash card or whether we will need an intermediate buffer in the SDRAM. The additional required clocks can be taken from the several availiable system clocks.

The accompanying C program reads in a wave file and will write the data to the buffer and the various contr ol register inside the audio chip, possibly through another buffer.

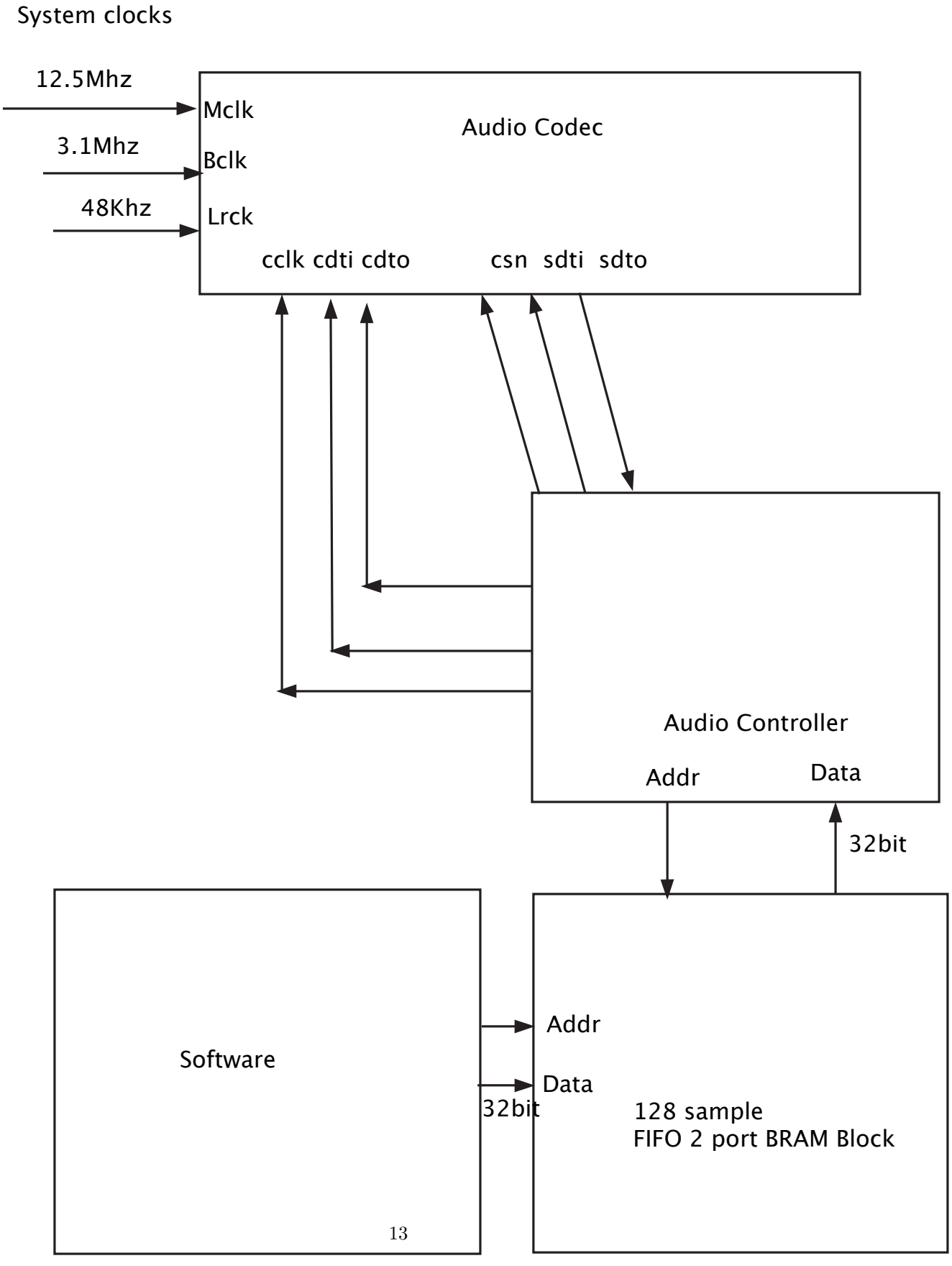
```
#include <stdio.h>

int main() {
  int i=0;
  int channels=0;
  int fmt_length=0;
  int tag=0;
  int sample_rate=0;
  int byte_rate=0;
  int block_size=0;
  int sample_size=0;
  char current4[4]="    ";
  char current2[2]="  ";
  int f_length=0;
  int data_length =0;
```



States
CO : CPU operating
CW : CPU waiting
AO : Audio operating
AW : Audio waiting

Transitions
f: buffer full
p: Audio playing
e: buffer empty
w: memory writing
r : memory reading
wa: waiting
s: starting timer
c: release control of mem


System clocks
12.5Mhz
Mclk
3.1Mhz
Bclk
48Khz
Lrck
Audio Codec
cclk cdti cdto
csn sdti sdto
Audio Controller
Addr
Data
32bit
Software
Addr
Data
32bit
128 sample
FIFO 2 port BRAM Block

```
FILE *waveFile;
waveFile = fopen("homer_doh.wav", "r");
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
printf("%s\n", current4);
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
f_length=current4[3]*256*256*256+current4[2]*256*256+current4[1]*256+current4[0];
printf("file length-8 = %d\n", f_length);
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
printf("%s\n", current4);
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
printf("%s\n", current4);
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
fmt_length=current4[3]*256*256*256+current4[2]*256*256+current4[1]*256+current4[0];
for(i=0; i<2; i++)
  {
    current2[i]=getc(waveFile);
  }
tag=current2[1]*256+current2[0];
printf("format is %d\n", tag);
for(i=0; i<2; i++)
  {
    current2[i]=getc(waveFile);
  }
channels=current2[1]*256+current2[0];
printf("%d channels\n", channels);
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
sample_rate=current4[3]*256*256*256+current4[2]*256*256+current4[1]*256+current4[0];
printf("sample rate = %d\n", sample_rate);
for(i=0; i<4; i++)
  {
    current4[i]=getc(waveFile);
  }
byte_rate=current4[3]*256*256*256+current4[2]*256*256+current4[1]*256+current4[0];
printf("bytes/second = %d\n", byte_rate);
 for(i=0; i<2; i++)
  {
```

```
      current2[i]=getc(waveFile);
    }
  block_size=current2[1]*256+current2[0];
  printf("bytes/sample %d\n", block_size);
   for(i=0; i<2; i++)
    {
      current2[i]=getc(waveFile);
    }
  sample_size=current2[1]*256+current2[0];
  printf("%d bit samples\n", sample_size);
  for(i=0; i<4; i++)
    {
      current4[i]=getc(waveFile);
    }
  printf("%s\n", current4);
  for(i=0; i<4; i++)
    {
      current4[i]=getc(waveFile);
    }
  data_length=current4[3]*256*256*256+current4[2]*256*256+current4[1]*256+current4[0];
  printf("data_length = %d\n", data_length);
  /*  for(i=0; i<data_length; i++)
    {
      printf("%u ", getc(waveFile));
      if(i%10==0)
printf("\n");
}*/
  printf("\n");
  fclose(waveFile);
  return 0;
}
```