E4840 Embedded Systems
Project Design
March 22, 2011

Dawei Liu
Xiaoliang (Lee) Zhu

Funny Soundboard

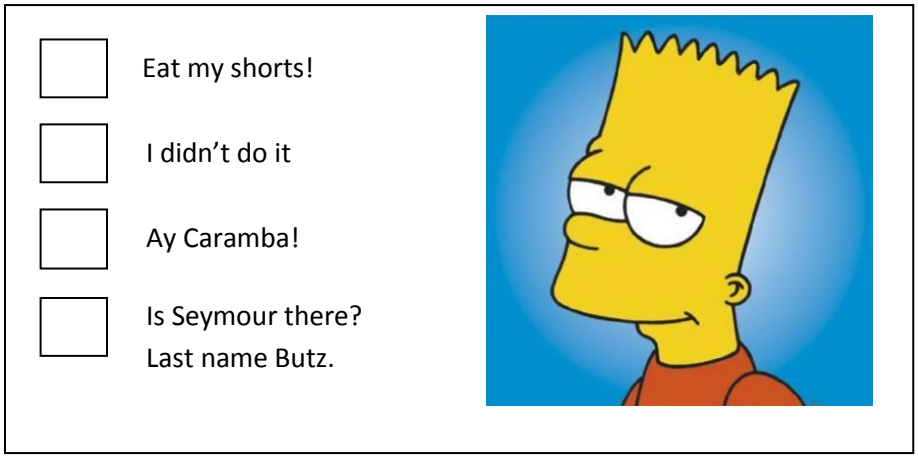
Goal:

We plan to create an embedded system that plays back various sound clips when the user presses different buttons. The purpose is to generate laughs, and the potential application is for the system to be placed in theme parks, restaurants, or anyplace that people wait in line.

Description:

The system will consist mainly of an Altera DE2 board. It will have a VGA monitor output, on which a grid of buttons are displayed, along with text and animation representing the theme of the soundboard. The animation will look like a GIF image and will be restricted to an area about half the screen wide and half the screen high. This means we are not doing sprite based animation.

A PS2 mouse is connected to the system and a cursor will be displayed on screen. When the user clicks on a button using the cursor, the associated sound clip is output through the Analog sound output of the DE2 board. The soundboard will have multiple themes (sets of picture/sound files) by reading from the SD card slot of the DE2 board. An example screen shot is below[1]:



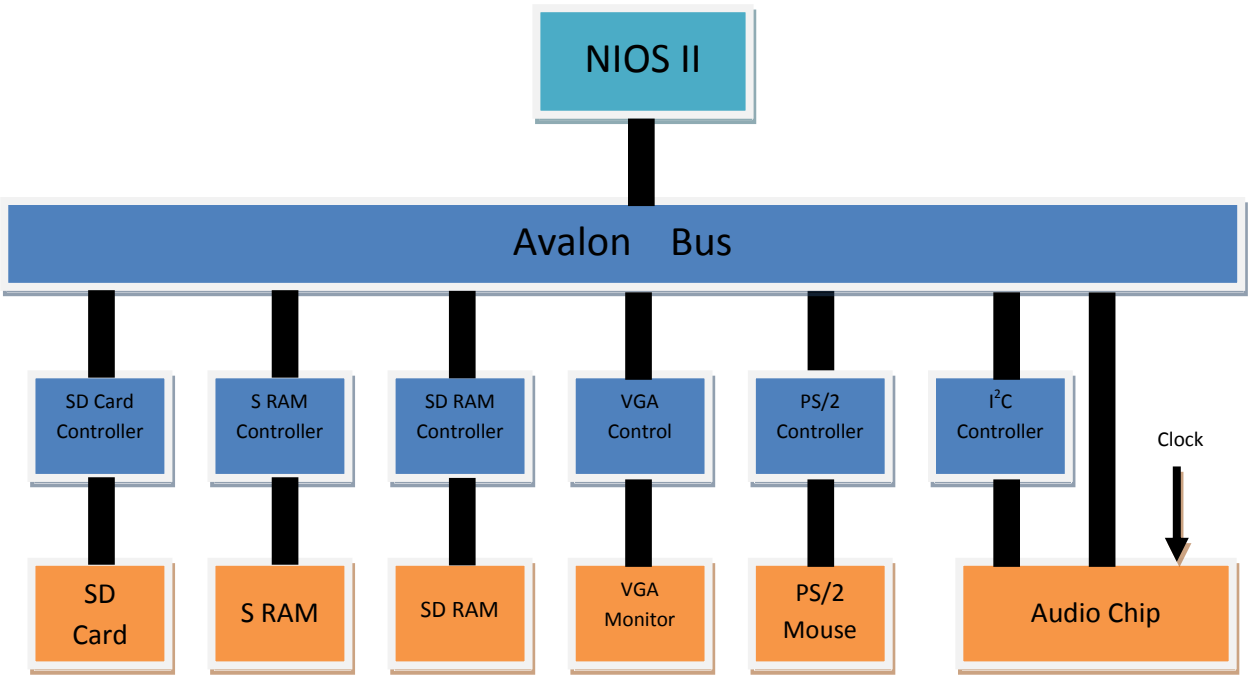
Implementation:

We will use an NIOS II microprocessor implementation on the Cyclone II FPGA as the CPU. A program written in C will be responsible for processing mouse input, generating screen background, process cursor movement, generating animation, and output sound.

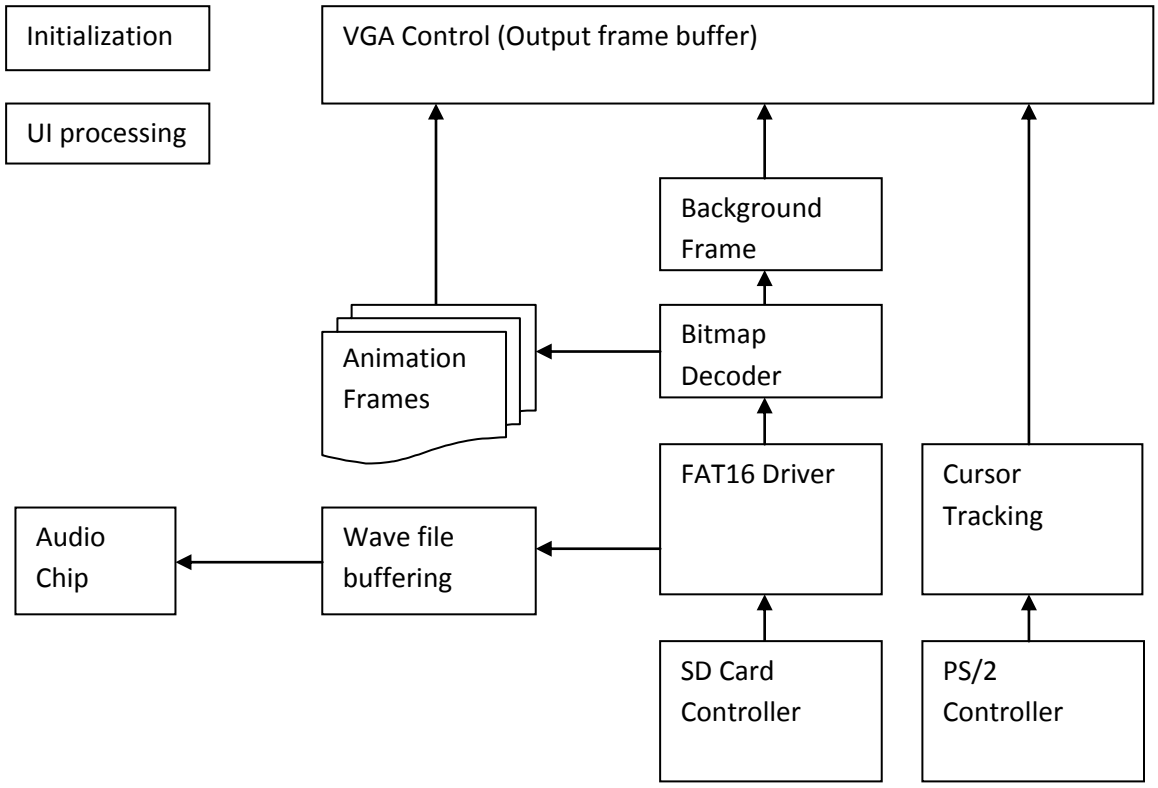
Hardware used will be: SD card slot, PS2 interface, VGA interface, and Analog audio out
Bitmap and Wave file formats will be used to save image and sound files.

[1] Copyright 20th Century Fox, all rights reserved.

Block Diagram:



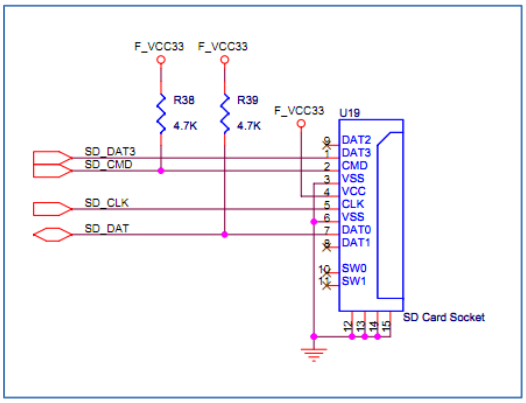
Software Architecture:



HARDWARE

SD Card reader

We choose to use SD card because there is an SD socket is built into the DE2 board. The SD card socket is connected physically as:



With the following pin-out definitions:

Pin	Name	Function (SPI Mode)
1	DAT3/CS	Chip Select/Slave Select (SS)
2	CMD/DI	Master Out Slave In (MOSI)
3	VSS1	Ground
4	VDD	Supply Voltage
5	CLK	Clock (SCK)
6	VSS2	Ground
7	DAT0/DO	Master In Slave Out (MISO)
8	DAT1/IRQ	Unused or IRQ
9	DAT2/NC	Unused

The single data connection of DAT0 pin means that the SD card is set up to be accessed in SPI mode from the FPGA, and data will be read serially one bit at a time. We will set up the SD Card Socket as an Avalon peripheral, and then modify the factory shipped SD Card driver to ensure performance.

The interface to the SD card is simple. Commands are sent from the host through the CMD line, and the card responds with a response frame also on the CMD line, followed by data token and data bytes on DAT0 when applicable. Each command has an expected response type and error conditions. Please see references for details on SD card specification.

During initial boot up, the SD card defaults to SD mode. To switch to SPI mode, we send a GO_IDLE_STATE command to the card while holding the DAT3 pin low. During this process the clock must be below 400 KHz, and we must wait at least 74 clock cycles before accessing the card. During normal operation, a read block command will be issued from the host, and the argument is the starting LBA address. The host will then have to pull the card's status to see if card is ready to transfer data, very much like the process of accessing a hard drive.

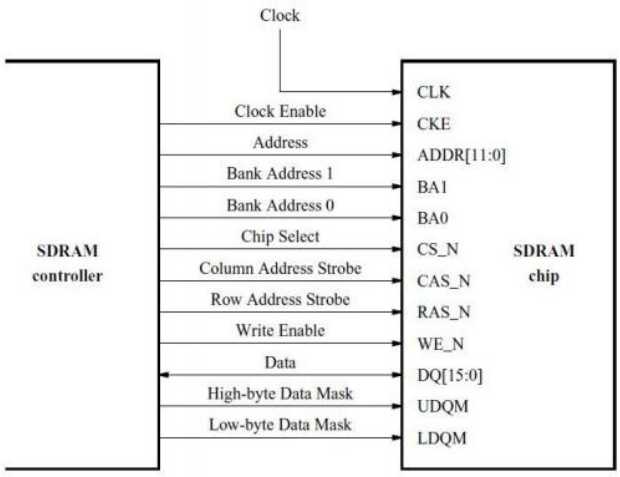
The maximum data clock for a standard SD card in SPI mode is 25 MHz, and this is the upper speed limit to access data. The SD card clock will be generated in software to easily accommodate the slow clock requirements during initialization.

SRAM

The 512K SRAM will be used for program, audio, and video data buffering. It will be added as an Avalon peripheral and initialized in the same way as in Lab3. The existing code and implementation should work. Please see software section for detailed allocations.

SDRAM

The 8MB SDRAM will be used for bit-map processing and as video memory. It will also be added as an Avalon peripheral. We will supply the controller VHDL description to satisfy the read and write timing requirements of the chip. The connection diagram is copied below. We believe the SOPC builder will handle most of the timing as part of the PC100 standard, so implementing this component should not be too difficult.



PS2 Interface

The PS2 interface will be used to connect a mouse. Its implementation is very similar to that of a keyboard used in Lab2. Instead of key codes sent to the host, the mouse will send three bytes to represent the movement and the buttons clicked. The typical bytes received from the mouse will be:

MSB LSB

Y Overflow	X Overflow	Y Sign	X Sign	1	Middle	Right	Left
					Buttons		
X movement							
Y movement							

There will be a software function to initialize the mouse. Then we will wait for the mouse input in the UI loop to update the cursor location and process button presses.

VGA Controller

For our project we will modify the VGA controller supplied in Lab3. We will initialize it and build its clock the same way as Lab3. We will have a "cursor" object built into the controller (think of it as a hardcoded single sprite), and it will be addressed in the same way as the bouncing ball in lab3.

The address processing will be modified so we can give it 2 additional addresses. One for the static background image, which is at the resolution 640 by 480, and another for the data in the animated part of the screen. The controller will do a raster scan for the designated background part of the screen, and when it hits the animation frame (say at the middle right half of the screen), it will switch over to the other address and start reading data off of there.

We can also save memory by having the static section black and white while the animation section in color. See software section for more info and discussion about bit depth.

Audio output

The I2C interface is used to initialize and send data to the audio decoding chip. The DE2 board shipped with an I2C interface that works to communicate to the audio chip. We will build on this component.

To output audio, we set the sampling rate on the WM8731 Audio CODEC to the appropriate value for our input wave file (i.e. 44.1 kHz), and then output the wave data directly to the WM8731 Audio CODEC using the I2C interface. The chip will perform the conversion to analog and output it accordingly. We will also provide a separate clock for the audio codec, as specified in the manual.

SOFTWARE

FAT16 Driver

The FAT16 file system is standard on SD cards 4 GB in size or less. FAT16 is also less complicated than FAT32 file systems. Since the files required in our system will be substantially less than the 4 GB storage limit, we will work with FAT16 and implement a software file system parser.

Having file system support means we can work with directories and distinguish different file types from the SD Card. This capability is necessary and eliminates the need to generate custom container formats for sound and bitmap files stored on the SD card.

The FAT16 file system breaks the volume into clusters of 32 or 64 KB (2 or 4 GB cards), and within each cluster the data is organized in 512 byte sectors. There is a Master boot record at the very beginning of the volume, giving partition information about the volume. Following that is the File Allocation Table (FAT), containing which clusters in the volume is used, and how each cluster links to other clusters. Following the FAT table is the Root Directory Entry Table, here is the listing of all the files and directories in the drive, their attributes, and the start cluster location for the file.

In our system we will build on an academic release of a FAT16 file parser. Included are functions to parse the master boot record, traverse the Root Directory Entry Table, get file attributes, and reading data from the clusters.

FAT16 Drive Layout

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)

Bitmap Decoder

This component is used to extract the input bitmap files into various raw formats we need to use for video output. The bitmap images themselves are already very "raw", consisting of a header section followed by data section. Each pixel is represented by 3 bytes of Red, Green, and Blue, and then an "Alpha" overlay field. We won't care about most of the header fields, and our component can just read the "data offset" field in the header then jump directly to the data section. One thing to note is that the pixels start from lower left, so it is in reverse raster order.

We need to have two outputs from the bitmap decoder, one is the background frame of buttons and text, and the other is the series of frames for video. All picture data will be read from SD card and processing done on the fly with minimal buffering. This is because the bitmap images tend to be very big and we won't waste memory to load-then-process . We will store the background picture in SRAM, and store all of the video frames in SDRAM.

Cursor Tracking

The cursor is generated based on the mouse position. The UI processing loop will keep track of cursor position, updating, keeping it in bounds, and pass it along to the VGA controller. When a button is pressed the location information will be checked against the list of button locations specified in a configuration file (or hard coded).

Initialization, memory, and video limitations

When system starts we plan to process all the bitmap files. Store the background bitmap in raster order inside SRAM, and then store each frame of the animation in SDRAM.

We anticipate there should be enough room in the 512KB SRAM to store the background image, as long as we use a bit depth less than 24 bits per pixel.
For the animation we are restricted by the 8MB SDRAM size. A quick calculation shows that:

Image size:	640/2 * 480/2	(quarter screen size animation)
Bits/ pixel:	24	
Frame rate:	x 15	
Product:	3.5 MB	

So we can play 2 seconds of video at full color depth and frame rate. A lot of tinkering with this will be required as actual system performance is measured, and as we make trade-offs on animation size and length.

Audio Buffering

Since the video and background is read during initialization, we can do real time transfer of audio data when the user presses a button. This saves us buffering of audio data in memory and allows for arbitrary size audio to be played.

The factory shipped demo employed no buffering of data between the SD card and the audio chip. This could work, but we will test to see if we need to employ buffering in block (512 byte) sizes, in SRAM.

SD card layout

Possible directory structure is:

Root

- Configfile.txt (list how many wave files there are, where the buttons are on the background, where animation is and how many frame there are)
- Background.bmp (background image, with buttons drawn)
- Wavefile1
- Wavefile2
- …
- *Animation directory*
  - Frame 1.bmp
  - Frame2.bmp
  - …

Milestones

March 29:

Get to the point where we can list the directory structure of a SD card on the connected VGA screen.

- SD card reader interface and driver working.
- FAT16 driver working
- VGA frame buffer working
- SRAM working

April 12:
Display bitmap image on VGA screen, show cursor on screen.
- PS2 mouse driver working
- BMP decoding working
- SDRAM working
- VGA controller interface updated

April 28:
Animation and sound output working.
- VGA controller optimized for animation
- Animation size and style optimized
- Sound output working
- Cursor & UI integration working

References:

Previous project with overview of SD card:
http://www.cs.columbia.edu/~sedwards/classes/2008/4840/designs/Pelmanism.pdf

SD Card specification:
http://www.sdcard.org/developers/tech/sdcard/pls/simplified_specs/Part_1_Physical_Layer_Simplified_Specification_Ver3.01_Final_100518.pdf

FAT16 information:
http://home.teleport.com/~brainy/fat16.htm

FAT 16 file parser:
http://wolverine.caltech.edu/eecs52/projects/188mp3/188mp3.htm

PS2 Mouse Lecture
http://www.cs.columbia.edu/~sedwards/classes/2011/4840/ps2-keyboard.pdf