Embedded System Project: Pelmanism
CSEE 4840

Can Ilhan (ci2137@columbia.edu)
Chintan Shah (cds2127@columbia.edu)
Sungjun Kim (sk3062@columbia.edu)
Zenan Li (zl2174@columbia.edu)

1. Introduction

The goal of our project is to develop an interactive, picture-based memory game, so called Pelmanism. The basic idea of the Pelmanism is that the player should match two same cards on the screen, recalling what he or she has seen before. In the beginning of the game, all of the cards are shown of the back-side on the screen, hiding the front image; then, the player starts to match the same two cards by turning over two cards in every turn. Provided the player matches the same pictured cards, since there are always two same front-imaged cards, these cards are deleted on the screen; otherwise, the cards are turned over again. The game will be over when all the cards are matched, showing how long or how many clicks it have taken. Figure 1.1 illustrates the Pelmanism.

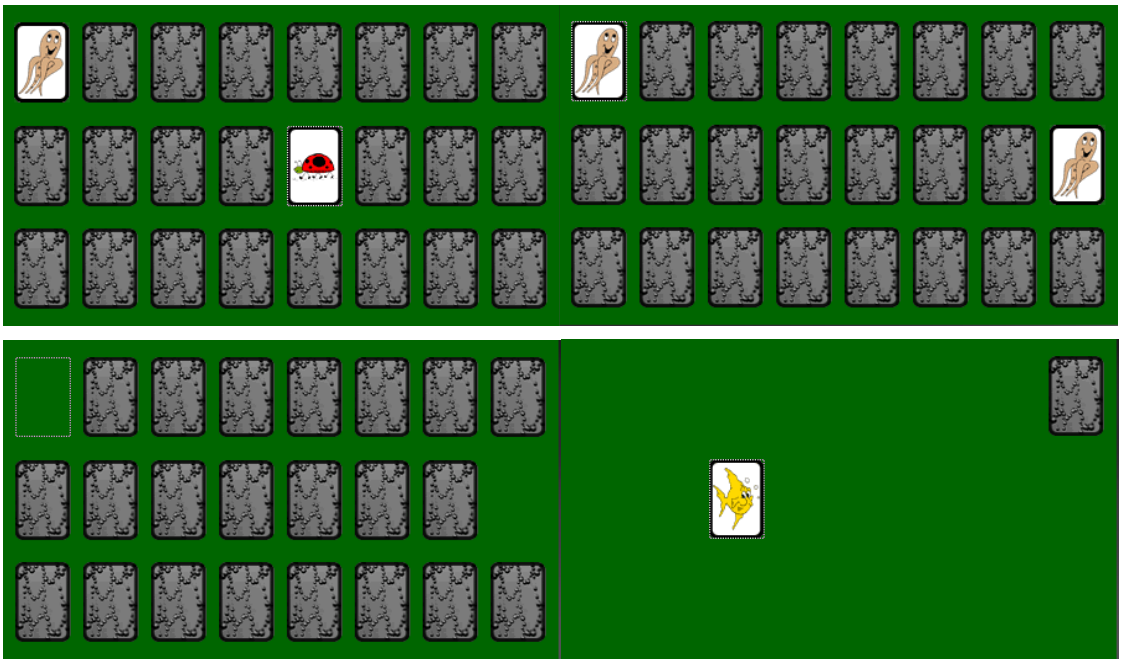


Figure 1.1: An illustration of the Pelmanism

To implement the Pelmanism, we will activate the main processor, the VGA, the PS2, the SD card interfaces of the ALTERA DE2 board, the SRAM and the SDRAM. Finally, for user-interface, PS2 mouse and LCD monitor will be used.

In general, the design will be divided into hardware design and software design. The main structure of our hardware design is shown in Figure 1.2 and the levels of our software design are shown in Figure 1.3.

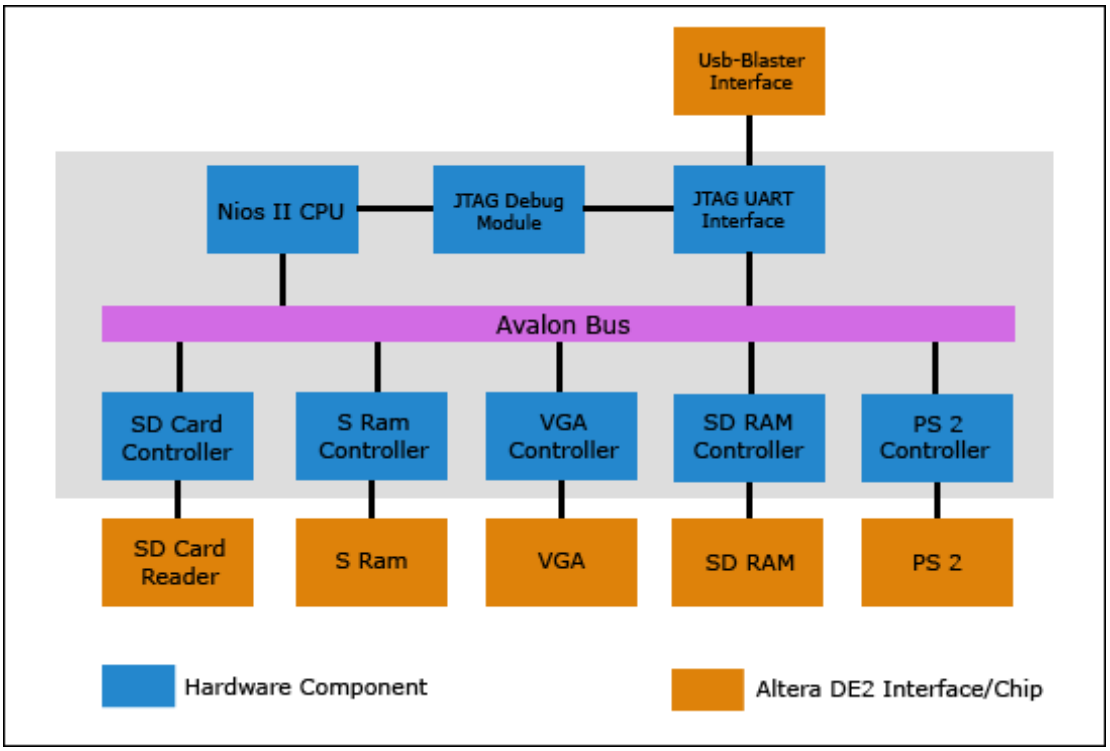


Figure 1.2: Overall structure of our hardware design

Figure 1.3: Overall architecture of our software design

2. Hardware Design

2.1 SD Card

We would be reading the JPEG images stored on the SD card for this project. It will be assumed that the images stored on the card are of the resolution 136 x102 and would not need to be scaled. The SD card will be searched for the JPEG images and if there are more than 8, others will be ignored.

The SD (Secure Digital) Card supports three protocols which can be divided into two classes, the proprietary SD Mode and the open SPI mode. The DE2 board supports the SPI mode and we will be using this mode. It is to be noted that the SPI communications mode supports only a subset of the full SD communications protocol. However, these functionalities are enough to achieve our purpose. A diagram of the SD card is shown below.

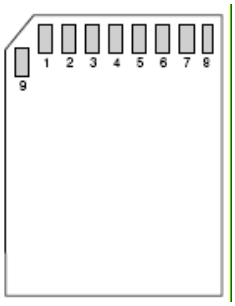


Figure 2.1: SD Card Diagram

The table below lists the pin assignments for the SD card and their functions in the SPI mode.

Table 2.1: SD card pin assignments in the SPI mode

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

In SPI mode, four signals (clock, data in, data out and chip select) are used for the interface. The clock is used to drive data out on the data out pin and receive data on the data in pin. The host drives commands and data to the SD card over the SD card's data in pin. The host receives response and data from the card on its data out pin. The chip select signal is used to enable the card during data and command transfer. The connection of the FPGA with the SD Card slot is as shown in the diagram below.

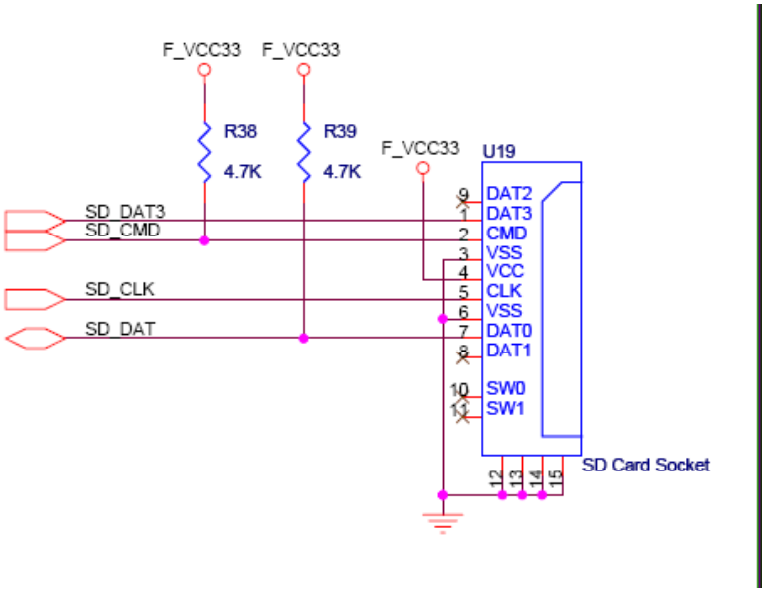


Figure 2.2: Connection with the FPGA

2.1.1 Protocol

The SD protocol is a simple command-response protocol. All commands are initiated by the master. The SD card responds to the command with a response frame and then, depending on the command, may be followed by a data token indicating the beginning of a bulk data transfer or an error condition. SD commands are issued to the card in a packed command frame, a 6-byte structure sent over the SPI port. The command frame always starts with 01 followed by the 6-bit command number. Next the 4-byte argument is sent, MSB first. The 7-bit CRC with a final stop bit '1' is sent last. All bytes of the command frame are sent over the MOSI pin MSB first. The figure shows the command frame format. The CRC is optional in SPI mode and by default CRC checking is disabled and we would not be enabling it.

Table 2.2: SD Command Format

First Byte			Bytes 2-5	Last Byte	
0	1	Command	Argument (MSB First)	CRC	1

The SD card responds to each command frame with a response. Every command has an expected response type. The type of response used for a particular command depends only on the command number, not on the content of the frame. Three response types are defined for SPI mode: R1, R2, and R3. For example, the R1 response is always 1 byte long and its MSB is always 0. The other bits in the response indicate error conditions.

Table 2.3: R1 Response Format

Bit	7	6	5	4	3	2	1	0
Field	0	Parameter Error	Address Error	Erase Seq Error	Com CRC Error	Illegal Command	Erase Reset	In Idle State

2.1.2 Initializing the SD Card in SPI Mode

At power-up, the SD card defaults to the proprietary SD bus protocol. To switch the card to SPI mode, the command 0 (GO_IDLE_STATE) is issued. The SD card detects SPI mode selection by observing that the card select (CS) pin is held low during the GO_IDLE_STATE command. The card responds with response format R1 (as shown above). The idle state bit is set high to signify that the card has entered idle state. The SPI clock rate must not exceed 400 KHz at this stage. Next, at least 74 clocks must be issued by the master before any attempt is made to communicate with the card. This allows the card to initialize any internal state registers before card initialization proceeds. Next, the card is reset by issuing the command CMD0 while holding the CS pin low. This both resets the card and instructs it to enter SPI mode. While the CRC, in general, is ignored in SPI mode, the very first command must be followed by a valid CRC, since the card is not yet in SPI mode. The CRC byte for a CMD0 command with a zero argument is a constant 0x95. For simplicity, this CRC byte is always sent with every command. Next, the card is continuously polled with the commands CMD55 and ACMD41 until the idle bit becomes clear, indicating that the card is fully initialized and ready to respond to general commands.

2.1.3 Reading the Images

The SPI mode supports single block read operations only, which will be used to read the images. The READ_SINGLE_BLOCK command with a starting byte address as the argument is to be used. This address must be aligned with the beginning of a block on the media in the valid address range of the card. The SD card then evaluates this byte address and responds back with an R1 command reply. If the read is completed from the SD media without error, a start data token is sent followed by a fixed number of data bytes. The start data token is not sent if the SD card encounters a hardware failure or media read error. Rather, an error token is sent and the data transfer is aborted.

2.2 VGA Controller

One of the critical hardware components that we are going to implement will be the VGA controller that will be talking to VGA DAC present on DE2 board. The skeleton of this controller will be the VHDL file that we had used in lab3 which has a rectangular sprite. Since porch values, sync declarations and other important values for 640x480 resolution are already defined, we can continue adding processes that help us to achieve our expectations for that controller.

We are planning to define a cursor sprite (the way we defined the ball for lab3) in this controller. Therefore PS-2 controller will be talking to this component as well as with our software via Avalon bus with a predefined address reserved for cursor drawing process.

We are also planning to develop our animation/visual effect as a process in our VGA controller that will be triggered after the user has clicked on any image. During the animation, instead of showing a sprite for a clicked image, a predefined animation will take place before revealing the whole image. We have some different ideas about how to implement an animation/visual effect.

One idea is showing some pixels of the image at a time (i.e only the diagonal and some other parallel lines to that diagonal) and after waiting some refreshes, revealing some more and more) to give impression of revealing slowly (fading in). Other idea is turning the sprite until it rotates 90 degrees and start revealing the image from 91st degree to 180th degree rotation (totally revealed). After that time, VGA will be constantly showing that image until it is folded off or deleted. We will define some signals for that so that our software can take control of those operations. We are going to develop VGA controller without any animation component in the beginning and add one or a few animations on top of that later (check milestones for detail).

Most importantly, VGA component will draw a fix sprite for each folded-off card (or a default image that we are going to supply in our SD card). Whenever user has clicked an image, VGA controller will get the appropriate image from the SRAM (where those images are previously saved randomly) and draw continuously until the card(s) is/ are removed or folded-off again (when user can not match two cards). For simplicity, we will put the images on SD card with previously defined/calculated resolution. We calculated the height and width of the images so that 512K SRAM can handle. We know from lab3 that VGA DAC needs 10 bit signal for each of the R/G/B components. However we think 10 bit scope for one color component is too high for our simple card game. Also 512K is very limited since we are converting JPEG to bitmap and then saving it into SRAM. That's why we are planning to reserve at most 2 bytes per pixel to save some space (otherwise we have to use 4 bytes per pixel [10x3 bits]). That's why we are going to define each color with 5 bit resolution. The first 5 bits of each color component will be padded zero and from 5th bit to 9th bit (little-endian representation) will be our actual color representation. This will be more than enough for our project since 32x32x32 color tones can be represented. We are planning to use most of the SRAM (i.e. 85%, not all of them since we may use some remaining part in the future) for saving 16 pictures in it. The detailed calculation for each picture's resolution is as follows (assuming 4:3 ratio pictures):
4k*3k*16*2= 1024*512*0.85 ➔ k=34 ➔ So each picture will be resized to 136x102 resolution before saving into our SD card. Also this resolution will not use the screen fully therefore we decided to add some spacing between pictures. After the calculations of spacing, the final GUI will look like:

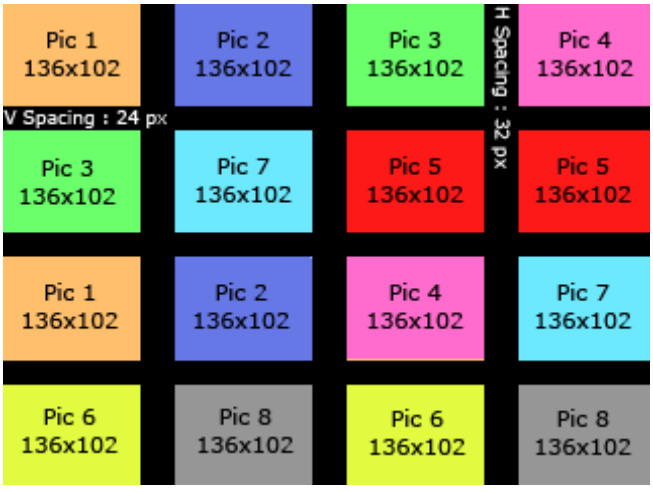


Figure 2.3: An illustration of GUI

Address definitions of our VGA controller will probably look like:

Address Value	Processes
0x00	Cursor
0x01	SRAM address offset to read (high)
0x02	SRAM address offset to read (low)
0x03	Data read from SRAM
0x04	Animation - Visual Effect

Figure 2.4: Address definitions of the VGA controller

Therefore our address signal will be at least 3 bit and we may use a few more addresses if we can define more than one animation so that we can call randomly from our software using their specific address values.

We will be using NIOS2's internal 50 MHz clock. However, we also know that VGA's pixel refresh rate should be calculated by 25 MHz for 640x480 resolution with given porch, sync values. We are going to divide the main clock in this peripheral and make sure that all processes that are responsible for pixel generation will be using 25 MHz down sampled clock.

2.3 SRAM Controller

Even tough SRAM Controller is trivial and already defined in SOPC builder before; we would like to mention how it is going to store our images. After we fetch the 8 images from SD-Card, our JPEG decoder will convert it into bitmap and the result will be stored into our SRAM with a random memory offset order (so that the first sprite will not be necessarily covering the first image, etc). Upon user's click, software will send the address of the clicked image and VGA controller will display it after showing a short animation. We are thinking to use 85% of the SRAM (443904bits). Therefore the registers will be holding:

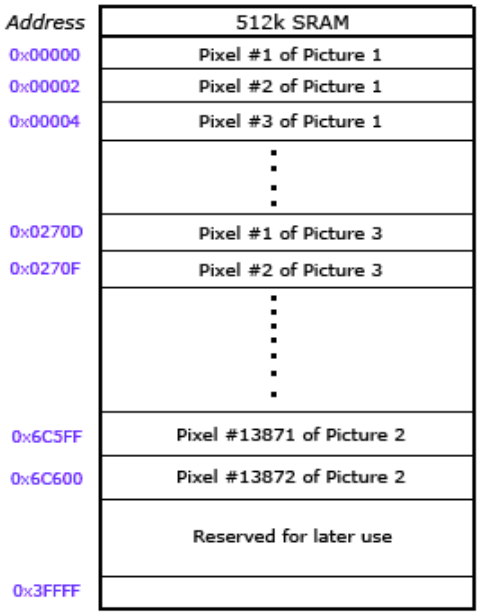


Figure 2.5: Register Values of our 512k SRAM

2.4 SDRAM Controller

Since we will be using most of the SRAM for saving decoded Jpegs (bitmaps), we are not going to save our C++ code in the same memory which we think it would not fit after defining Jpeg decoder, file system, drivers and API's in software. That's why we are planning to interface off-chip 8Mb SDRAM via Avalon bus. The controller definition is trivial since it is already defined in SOPC builder and we don't need to modify it. It is actually using PC100 standard to talk to SDRAM. All of the signals except the Clock signal are already defined in SOPC controller and we will connect our global clock directly to it to make it work.

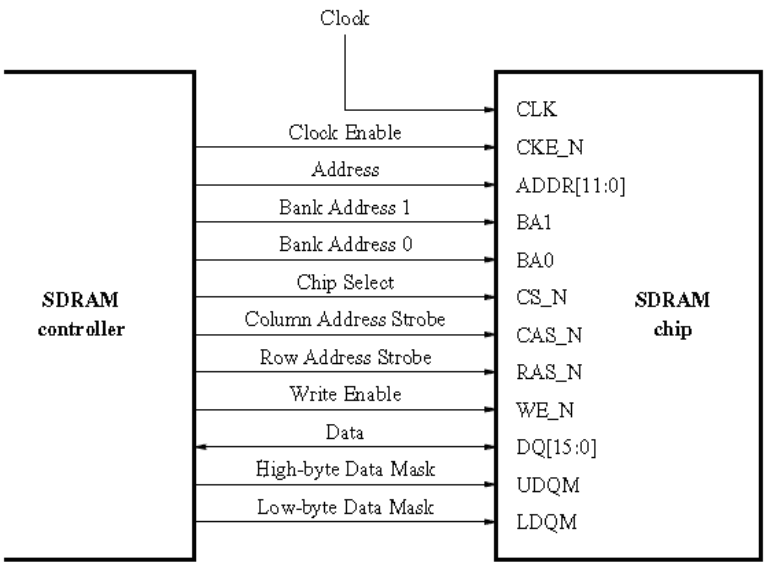


Figure 2.6: The SDRAM signals

2.5 PS2 controller

In this project, since PS2 mouse is used, PS2 controller will be implemented. Data communication is done according to the clock as follows:

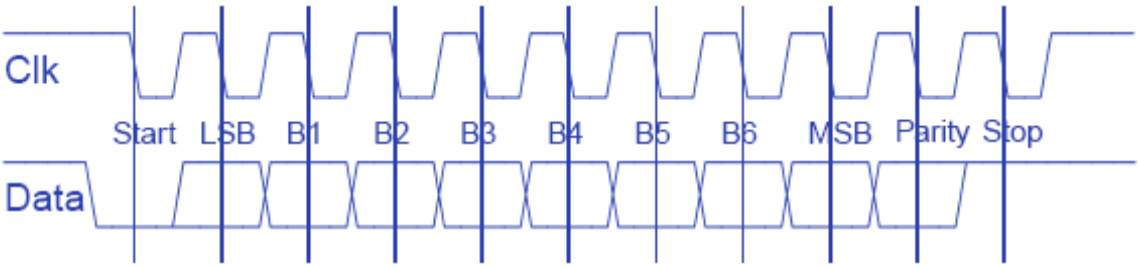


Figure 2.7: The clocks of data communication

Also, in the PS2 mouse control, the transferring data is as follows:

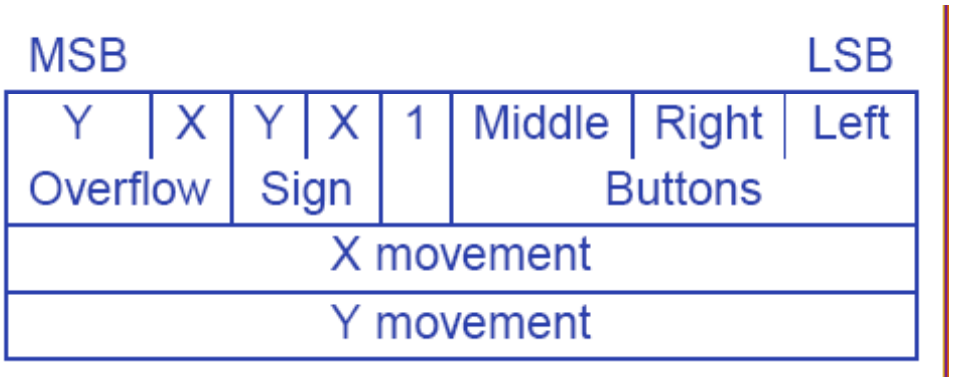
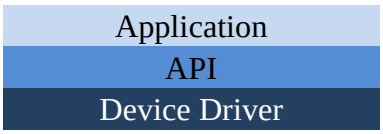


Figure 2.8: The transferring data of PS2 mouse control

Therefore, in the hardware part, PS2 controller reads these data and sends them to the software side. Software mouse driver will process the data: Overflow, Sign, Buttons, X movement and Y movement.

3. Software Architecture

Our basic software design is 3-layered architecture: application, API, device driver. In the device driver layer, code will be written as microcode with C language; in the API and application, it will be coded as object-oriented programming with C++ language.



3.1 Device driver layer

Device driver layer abstracts hardware devices such as VGA, SD card and PS2 mouse. In this layer, all the microcode of our game software will be isolated. Basic role of this layer is providing abstracted functionality to the API layer.

3.1.1 VGA driver

VGA driver provides drawing pixels, giving visual effects and cleaning LCD screen or pixel. This driver is interfaced with the hardware VGA interface. Since most of functionalities will be implemented in the hardware side, this driver simply interfaces the hardware

3.1.2 SD card driver

SD card driver communicates with SD card interface in the hardware layer. This driver reads data from SD card or writes data to the card. Additionally, it checks whether SD card is inserted.

3.1.3 Mouse driver

Mouse driver receives data from user's mouse control. This driver's output will be movement direction, movement speed and information whether a mouse is connected to the PS2 port.

3.2 API layer

API layer gives useful toolsets or specific functionalities to the application layer. Through the API, the application layer can deals with file data, receives mouse status, decodes JPEG image and write text image on the LCD screen.

3.2.1 File

File API has file system, open, close, read and write functions. In our project, this API's main role is image file transferring because the game image size is so big that this data will be stored in the SD card.

3.2.2 I/O

In our I/O API, input will be mouse data processing; output will be printf function through JTAG. The printf function will be wrapped in this API because it will be only activated in the debug mode. Also, since mouse is the only input device of our game project, input API will just process data from mouse driver. In the input API, function of getting position of the mouse cursor will be implemented; cursor image is also stored here.

3.2.3 JPEG decoder

JPEG decoder will extract JPEG file and return raw image data. Even though it is more efficient to be implemented in the hardware side, this decoder is placed in the software side due to the decoding algorithm's complexity.

3.2.4 Animation

Since animation effect will implemented by the hardware control logic, this animation API will interface with application layer and device driver layer. Nevertheless, it is necessary because it is 3-layered design; interface object between the lowest layer and the highest layer is necessary.

3.2.5 Text

Text API has text fonts and letter-typing functions. Since the application will use several characters, the text API is needed.

3.3 Application layer

Application layer will be the largest and the most complicated part among 3 layers. The layer will follow MVC (model-view-controller) design because it is an interactive application. Therefore, graphic and control class is designed. Also, model class will be composed of player, card and timer classes.

3.3.1 Graphic

Graphic class is used by the control class and uses model classes: player, card and timer class. When control class changes data of model classes, the graphic class will change the screen. Also, since the graphic class is passive, it will not change the model classes' data.

3.3.2 Control

Control receives user's input, changes model classes' data and sends command to the graphic class. Provided the control should be divided from the main function, it will be implemented as a class; otherwise, it will be included in the main function. Also, for initiating and ending the game, the control has functions such as shuffling cards or starting/stopping timer; it will deal with all the game rules.

3.3.3 Player

Player class has name, number of matched cards and other player-related information. This class is software abstraction of a player in the real card game.

3.3.4 Card

Card class has ID, image, and other card-related information.

3.3.5 Timer

Timer class is used when displaying elapsed time during the game and at the end of the time.

3.3.6 Environment

Environment object has data that will be frequently changed according to the game rule policy or to the software developers' tunings. For example, when the card-turning animation duration should be changed, the developer will find the information here. Therefore, when some application requirement is changed, the developer should look up this object first.

4. Milestones

Until milestone 1, only hardware work will be done; from the milestone 2, however, hardware and software sides will be developed at the same time. To be specific, at first, we finish the basic hardware components; then, we concurrently develop the software part and the difficult part of the hardware module. Basic hardware part development will be done by milestone 1; a little harder hardware part and the basic software design will be implemented by milestone 2. By milestone 3, the hardest hardware part and most of software part will be done. Finally, until the final report day, we will focus on the intensive testing and debugging.

4.1 Milestone 1

VGA interface, PS2 mouse, SRAM, SDRAM will be interfaced; basic nios2 system will be implemented so that software development can start from the milestone 2.

4.2 Milestone 2

In the hardware side, SD card interface will be done; in the software side, on the other hand, skeleton code will be generated.

4.3 Milestone 3

Hardware side will be finished – hardware animation effecter will be implemented. Also, software development will be finished except functions of giving animation effects.

4.4 Final report

Between milestone 3 and final report day, we will interface application with hardware animation effecter. Furthermore, we will focus on validating our hardware and software.

Category	Task	Milestone 1	Milestone 2	Milestone 3	Final Report
Hardware	SD card controller		√		
Hardware	PS2 mouse controller	√			
Hardware	VGA controller (basic operati	√			
Hardware	NIOS2 system build	√			
Hardware	VGA controller (full operation)			√	
Hardware	Integrated testing and validating				√
Software	Skeleton code of design		√		
Software	VGA driver				√
Software	PS2 mouse driver		√		
Software	SD card driver			√	
Software	File API			√	
Software	I/O API		√		
Software	JPEC decoder			√	
Software	Text API		√		
Software	Animation API				√
Software	Graphic (application)				√
Software	Control (application)			√	
Software	Player (application)			√	
Software	Card (application)			√	
Software	Timer (application)		√		
Software	Environment (application)				√
Software	Integrated testing and validating				√
colored box: on-going work					
√: finishing date of the part					

Figure 4.1: Project Schedule

References

[1]Stephen A. Edwards CSEE 4840 Embedded System Design. http://www1.cs.columbia.edu/~sedwards/classes/2008/4840/index.html

[2] Interfacing a MultiMediaCard with ADSP-2126x SHARC Processors (EE-264)

[3] SD Card Association. http://sdcard.org/

[4] Jun Li, Sharp. Interfacing a MultiMediaCard to the LH79520 System-On-Chip

[5] Secure Digital Card Interface for the MSP430, Michigan State University, 2004