Campus Fighter

CSEE 4840 Embedded System Design

Haosen Wang, hw2363

Lei Wang, lw2464

Pan Deng, pd2389

Hongtao Li, hl2660

Pengyi Zhang, pnz2102

March 2011

Project Introduction

In this project we aim to implement a fighting video game similar to the early version of Street Fighter which was first released in 1987 by Capcom. While the game series is a big success even in today, the version we try to implement is quite simple due to resource and time limitations.
Our final goal is to make a game played by two people. The control interface is PS2 keyboard and the players must control their game roles and fight against each other.

Two key factors of fighting games are the high quality of dynamic display of game roles with different actions and the reliable interface of control which can handle fast and multiple inputs. Thus in our implementation, making a good display of roles and actions is the most important work. Exploiting the most of PS2 keyboard's performance is also a challenge. Apart from those, our implementation also includes a simple audio generator for some sound effects in the game.

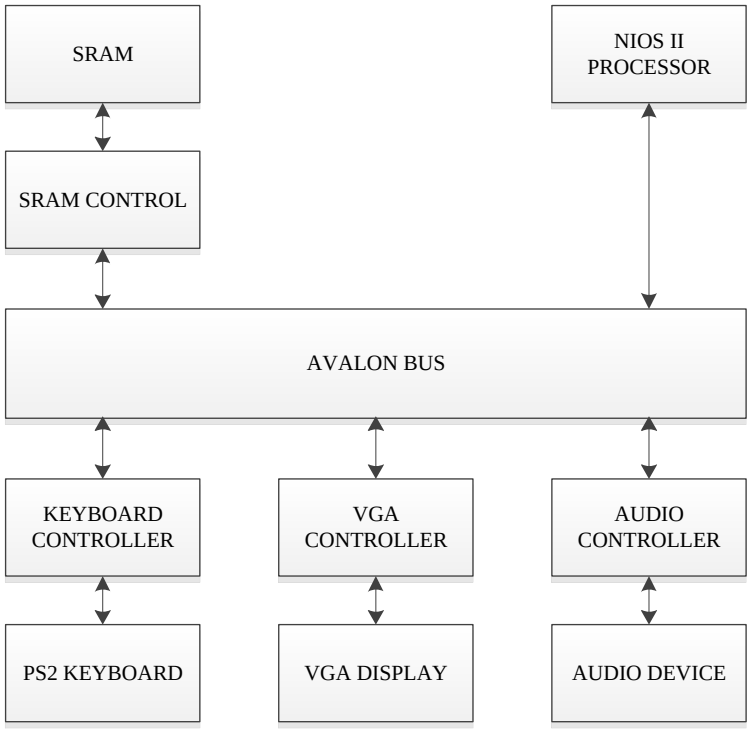
Updated Milestone

Milestone1:
Finish the sub-images for game roles
Finish the display of back ground
Display basic roles on the screen
Finish the basic sprite structure

Milestone2:
Finish the basic modification of PS2 keyboard driver
Game role can perform simple movement on the screen
Finish basic display elements and menus of the game

Milestone3:
Finish the attack judgment and movement interaction of two roles
Finish the hit point and score record
Can generate simple sound effect

Implementation detail: Hardware Structure



The hardware structure is shown above. As can be seen,we need to build SRAM, NIOS processor, VGA controller, audio controller and keyboard module all connected to the avalon bus, which makes the communication between CPU, peripheral devices, such as keyboard and LCD displayer, and Avalon bus come true. This structure is similar to what we have done in lab2 and lab3.

The keyboard module, VGA controller and the Audio controller are interfaces aiming at communicating through the Avalon bus. Here, since the difference between controlling fighting actions and typing word, VHDL codes for the keyboard are to be improved by ourselves and are not the same as that used in Lab2. The VGA controller and Audio controller use VHDL codes are improved based on those provided in Lab3. These three parts will be bind to the Avalon bus with SOPC builder in the Quartus.

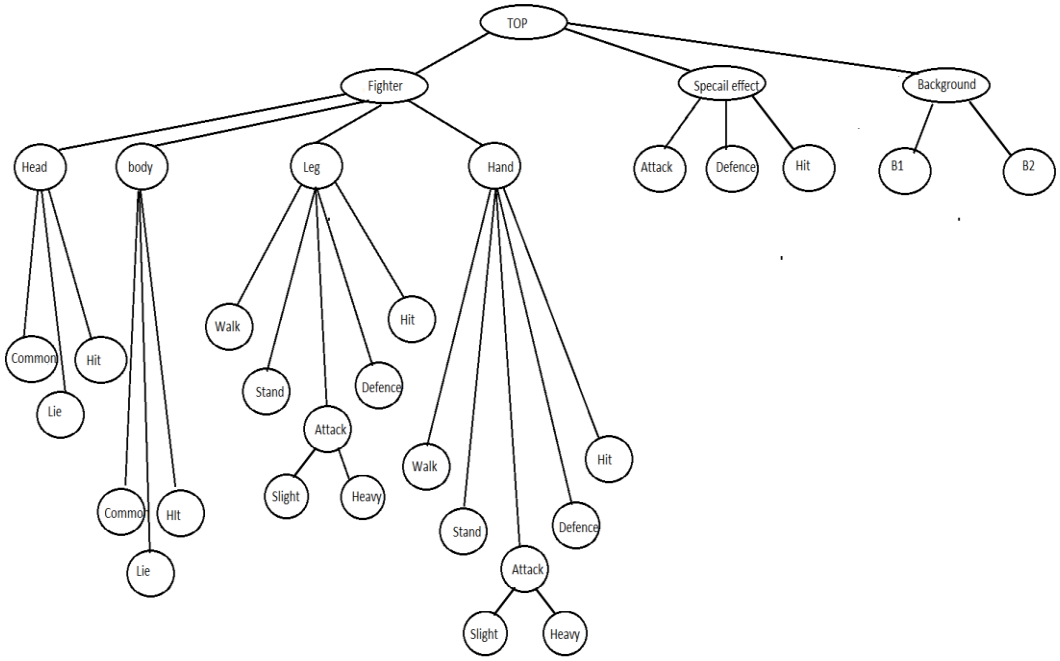
In the VGA block, sub-images of different parts of the character of two fighters will be stored in flash memory and can be displayed on the screen. Those parts include arms, legs, body, head and some background images and their relative position. To display the correct images on the screen with all the position and movement information, we implement the VGA controller based on the sprite graphics. This will be described in detail below.

Additionally, there is also proper simple music accompany with the fighting, and certain corresponding sound needed for actions like the kick and jump. Since the sound effects in this game is not too complex, we just implement this part based on lab3.

Implementation detail: Graphic Display

The video display is complex because in the fighting game. In spite of just dealing with plain movements of pre-drawn parts in other simple video games, there are lots of the different actions for fighters which are more like animations. We intend to use 15 sprites to implement the video display. Even for a signal fighter, different actions will lead to different locations for his or her hands, head, body, foots and so on. Also, the image loaded into every sprite needs to be refreshed very quickly to achieve smooth movement display.

Because the memory of Cyclone II FPGA board is big enough and processes to calculate the location when people are moving (including rotation calculation) is really resource-consuming, we use the method that store the fixed figures for every possible movement for roles in the flash memory. In that way, different matrixes will be used in the project to store different information. And for different information, the sizes of the different matrixes are different. A component tree will be used to describe the different figures which represent different actions.



As the figure above shows that, in order to save memory and improve the speed, we choose divided a fighter into 4 parts: head, arm, body and leg. And in different situation, they will perform different actions as the design which has been shown in the figure above. And in order to improve the game quality, we add some special effects for the game as the figure, when the fighter attack, defend and be hit, there will be different effects. For the back ground part, we give the player two choices to choose the background they wish to continue the game.

To realize the action display in an animation way, we plan to prepare 3~4 images for a certain action and display them by refreshing the sprite frames frequently. This is not shown in the above graph. Because the total screen we use to play the game is 680 * 480 pixels, so that we design the size of the combination of one fighter will nearly reach to 150*150 (full movement range, not necessarily

all solid images). For the details, we would like to choose 5 kinds of the matrixes to store the sub-image information. The matrix sizes are (20* 60), (20*80),(40*80), (40*40), (20*20).

Eight kinds of color will be used in the game. And we can implement it by 3 bit vector. In our project, we use the different values to represent the different colors. The table follows will show the details:

Color	Value
Red	000
Yellow	001
Blue	010
Green	011
White	100
Black	101
Orange	110
Brown	

Instead of drawing the pixel matrices by ourselves, **we choose to shoot certain photographs of real actors and use Photoshop to process the pictures** so that they can be used in the game display. We do so because we need to collect multiple images for different actions and it requires really good drawing skills to make it all by hand. Using photos is more efficient and can offer better quality.
On the other hand, since our image matrices are relatively larger, we choose to use Matlab to read in a source image and translated the pixels into VHDL codes. Again, this offers higher efficiency and better quality. But the work is never straightforward. A careful match between color space and pixel coordinate range is needed. This is our first work in the following week.

Implementation detail: PS2 Keyboard

For a fighting game, the appropriate functionality of the keyboard is of great significance. While the basic PS2 driver in lab2 can still be utilized here, software treatment of keyboard inputs need more attentions. The high requirement of keyboard in a fighting game can be described as the following three points.

First, compared to ordinary character inputs, the system may face far most fast input speed in a fighting game. Players in game may need to push a series of combo keys in very short time to get in advance against his/her opponent. Since the basic key set of fighting game control is no more than 10 keys, it is more likely than ordinary typing to push key in very high speed. If the software driver misses some of the fast key pushes or makes mistake decode, there will be unexpected role actions which can greatly decrease the joy of the game.

Second, since this is a 2 players game using the same keyboard, it can be roughly regarded as a double in key input speed. Thus if not handled carefully, a similar degrade of game performance as above will happen.

Third, in a fighting game, software has to detect the push of multiple keys simultaneously. This is necessary because fighting game players often tend to keep pushing move arrows or defense keys and push attack keys at the same time. Also, since players need to trigger combo actions, multiple key pushing at the same time is almost inevitable. Thus the software code should make full use of the break mode signal from PS2 keyboard and make a good arrangement of the key push record so that most of the control input can act as the players expected.

To try our best to fulfill the above requirements, first we need to make the decode process of every key input as fast as possible. This can be done by making the code only sensitive to the smallest number of keys which are used in the game. In case a too fast push after one another is missed, a small signal buffer may be used so that every push can be dealt. For the multiple key push judgment, the software need to keep a record list of up to 20 pushed keys and record the time interval of each key push and the break mode bounce up. This structure guarantees that only a fast enough series key push can trigger combo actions.

Implementation detail: Sound effects

For the audio module, it plays the corresponding sound when the fighter conducts attacking and blocking moves. To implement the audio module, the music will be generated through the FM synthesis based on the lab 3. The basic idea for sound note generation is to the modulating frequency ω_C and the modulation depth I of the fundamental sine wave: x(t)=sin(ω_C t+Isin(ω_m t)). A complete cycle of the sine wave with adequate sampling points will be stored in a ROM. This ROM is accessed at different rates, which effectively modulates the frequency of the sine wave. In addition, the period of time each note lasts will determined by a counter that counts the number of clock cycles for the note. Then, the sampled sine waves for different notes will be transmitted sequentially through I2C bus to the WM8731 Audio CODEC on the DE2 board for audio output.

To make a good sound effects, we should record some real-life sound effects and analysis its waveform parameters using CoolEdit. After we get a good data description of the sound we needed, we will store them as ROM and use them to synthesis sounds in the game.

Implementation detail: Software Mechanism

After building up all the hardware controllers and driver parts, we need to make a complete mechanism for the game contents. This part is written in the C programming language and compiled and debugged using the NIOS II Integrated Development Environment.

First is the basic location and movement generation of each display sprite. To do this, we use a very simple animation-like process to call different ROM-stored sub-images of a moving part and display them in a certain coordinate through time sequence. The arrangement of image display related to certain actions are pre-programed in C code and should be executed efficiently.

On the other hand, here are several important points lying in the realization of certain fighting action, dealing with details of how two characters behave under the impact of another and the harmony of different parts of the same character.

Since heavy kick and light kick are two different actions, positions of legs, arms and body are not the same, software needs to give corresponding positions' coordinates for the graphics. Software also needs to judge whether the strike touches the model of the hostile character and how many points reduction of health generated according to the type of the strike. Not only kick or punch by simply pressing one light-kick key or one heavy-punch key, but also more powerful skill can be triggered by certain groups of keys. As long as one character's hit point reaches zero, it will be regarded as the loser.

There are also some complicated situations to deal with in software section. Issues about how to display the overlapping parts when two characters' parts overlap a little, that character will not move forwards any more when it is close enough to another one, that the damage is decreased when defending action is taken, how to process multi-keys contention and to give out corresponding sound when certain events happen are all supposed to be solved in software.

According to the possible problems mentioned above, software section can be realized by the following modules.

Keyboard based Action Module. According different inputs from Keyboard Module, mark different types of attacks and actions(such as defend or jump) with relevant flags in order to make corresponding reduction of hit points.

Model Judging Module. This module aims at judging whether the attack is effective to result in health reduction on the hostile character. And also estimate the distance between two character models to determine if the character can move forwards any more.

Model Judging Based Action Module. If Model Judging Module tells that the attack hits on the target, and at the same time the defense action is effective informed by Keyboard based Action Module, then this action will be judged as an effective defense and give out a certain flag bit in this module. However, when the attack hits on the target without defense, this action will be regarded as a full attack and also generates a sign bit.

Model Display Module. According to flags obtained in the Keyboard based Action Module, this part can read different images from the ROM to realize different actions like move, kick or jump. And control (generate coordinates) the positions of those images which represent different parts of the character. If Model Judging Based Action Module tells that the full attack or the defense achieves, the VGA Module will also be able to generate a graphic showing the full attack or the attack being defended and read relevant images to show the character being attacked or defending.

Audio Module. In this module, background music is given and specific sound effects play according to flags obtained in the Keyboard based Action Module. If Model Judging Base Action Module tells that the full attack or the defense is effective, the Audio Module will also be able to generate a sound showing the attack or defense.

Damage Module. This module will estimate the multitude of the damage due to results from Keyboard based Action Module and Model Judging Based Action Module.

Hit points Calculation Module. Whenever damages are made, subtract the damage from the left hit points according to the results of Damage Module and store hit points for each character.

Score Calculation Module. According to difference of skills resulting in the damage, grade relevant scores and accumulate. Always record the highest final score of the game.

Win or Lose Module. Whenever the hit point of either side decreases to zero, the other side will win this round. The player who wins in more rounds than the other will get the final champion of the game. The total number of rounds can be selected from 3, 5 and 7.