UNIVERSITY OF LEEDS

ELEC5620M
Mini Project

DE1-SoC Pong

Alexander Bolton - 200938078
Sam Wilcock - 201285260
John Jakobsen -

May 2019

The University of Leeds
School of Electronic and Electrical Engineering

Contents

1 Introduction

This report will discuss the group project for the Embedded Microprocessor System Design module. The groups members were Alexander Bolton, Sam Wilcock, and John Jakobsen. The projects aim was to create a game of Pong on the DE1-SoC's microprocessor unit (MPU) which utilised the LT24 LCD Screen, a VGA screen, PS2 keyboard controls, button controls, and have audio output.

This report will be broken down into sections with section 1 being the introduction. Section 2 will discuss the display and graphics side of the project including the VGA driver which controls the monitor, the display driver which controls both LCD and VGA screens with a frame buffer, sprites and text which will go into depth of how the sprites are created, finally game engine graphics which will go into how the game engine uses the sprites including destroying, creating, and moving the sprites.

Section 3 will discuss...

Section 4 will discuss...

Section 5 will be the conclusion which will summarise the report and discuss if we have met the aims of the project. It will discuss what could be improved upon and changed. All code will be placed in the end of the report in the appendices.

2 Display and Graphics

2.1 VGA Driver

This subsection discusses the VGA driver and how it was implemented in the project. The VGA video out supports 640x480 however in this project is set to the default value of 320x240 pixels. The image displays from the VGA controller which is addressed from a pixel buffer. Each pixel value is write addressable using equation 1. An example of the pixel at 0,1 is shown in equation 2. The default base address for the pixel buffer is 0xC8000000 as stated in the manual. [1]

$$VGA_{baseaddress} + (pixelX_{coordinates}\, pixelY_{coordinates}\, 0_2) \tag{1}$$

$$C8000000_{16} + (00000001\,000000000\,0)_2 = C8000400_{16} \tag{2}$$

The pixels are layed out with the y coordinate starting from the top to bottom of the screen. The x coordinate is from right to left of the screen as shown in figure 1.

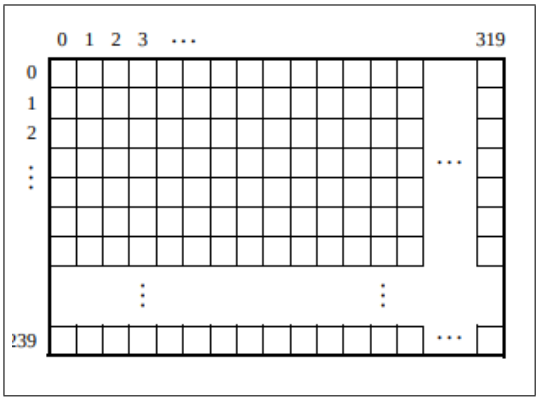

Figure 1: Pixel layout for pixel buffer of VGA controller [1]

Each pixel once addressed can be set to a value of colour with by setting bits for red, green, and blue. Each colour is allocated 5 bits which indicate the strength of colour for the pixel as shown in figure 2.

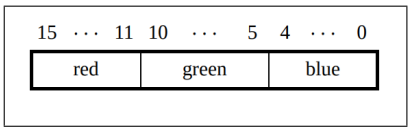

Figure 2: Pixel Colour Layout [1]

This code extracted from the project shows how a pixel is set in C.

```
void VGA_drawPixel(int x, int y, short colour){
  //Call address of pixel
    //Address base + [Pixel Y][Pixel X]0
  volatile short *vga_addr=(volatile short*)(0xC8000000 + (y<<10) + (x<<1));
  *vga_addr=colour; //Set pixel to colour
}
```

Figure 3: Code used to set pixel to a colour.

2.2 Display Driver

This subsection discusses the display driver which allows pixels to be set to a frame buffer and refreshed on command. The frame buffer is made up of 2 types of arrays which were a front frame buffer (what's currently on screen) and a rear frame buffer (what wants to be put onto the screen). The advantage of this is that it allows pixels to be checked and allows refresh on command.

Once the screen is desired to be refreshed the memory is compared between the front frame buffer and rear frame buffer to check if they are the same. If not the frame buffers are updated by checking each pixel in the frame buffers values. Any differences then update to display and to the front frame buffer. When first creating this frame buffer a problem with articulating occurred which is believed due to a memory overflow. The decision was made to split the frame buffer into 2 halves which reduced the memory used in each array which alleviated the issue.

Checking the pixels in a frame buffer is a very slow process so multiple frame buffers were created to experiment with. This experimentation involved splitting the screen into multiple sections and finding how many sections would be the fastest. This improved performance as multiple frame buffers were checked in memory and making the changes was done by comparing all pixels in a smaller area when there was a change in that area. In this experimentation frame buffers of 1 section to 8 sections were created. It was found that the 8 section frame buffer yielded the faster rendering of the frame buffer to the screens. All frame buffers were kept in the library which can be set. As it was still a slow process for the game a frame skip feature was also added to which skips a number of frames before refreshing. This speeds up the game dramatically. Also added to reduce errors and prevent system crashing was a 'Displays_setWindow' when set this prevents the driver from trying to address pixels outside of the window.

To compare frame buffers quickly the function 'memcmp' was used which set a variable in the function. If a change was found then the frame buffer would update.

```
    for(x = 0; x < PixelWidth; x++){
      short colour = frontFrameBuffer[x][y];
      if(frontFrameBuffer[x][y] != rearFrameBuffer[x][y]){
        short colour = frontFrameBuffer[x][y];
        Displays_drawPixel(x,y,colour);
        rearFrameBuffer[x][y] = colour;
      }
    }

  }
}
//This function fills the displays a set colour.
void Displays_fillColour(short colour){
```

Figure 4: Code used to compare and update the buffers (Quad buffer)

A number of function were created for use by the engine. These are functions such as 'Displays_init' which initialises the displays, 'Displays_mode' to set which buffer to use, 'Displays_setPixel' to set the pixel to a colour in rear buffer, 'Displays_Refresh' to refresh the buffer to hardware, 'Displays_ForceRefresh' to force a display refresh regardless of frame skip count, and finally 'Displays_getPixel' which returns the value of the colour of the pixel requested.

2.3 Sprites and Text

This subsection discusses how sprites are rendered and created in the library and how text is also created. There are 2 main sprites for this game of pong.

The first sprite is the ball. The ball must be initialised into an array before it can be rendered. To initialise the ball Pythagoras theorem was used (as previously completed in the graphics library of a previous assessment in this module). Equation 3 is Pythagoras theorem equation which is used. When the result is equal to or is less than radius squared then the value in the array is set to 1. The code iterates from the centre point for all values of x and y and sets an array. This was created as such to allow the ball to be dynamically set in size. Once rendered the ball can be rendered by providing coordinates of where to render (from centre of ball) and a colour. This will use these values and set the pixels to the frame buffer.

$$Radius^2 = x^2 + y^2 \tag{3}$$

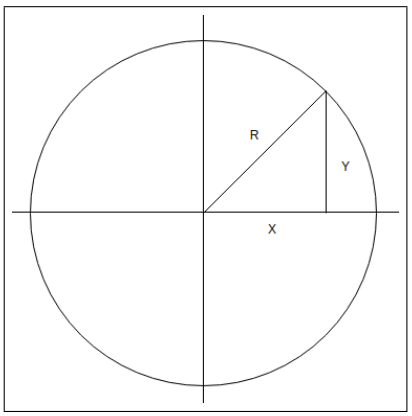

Figure 5: Pythagoras Theorem demonstrated inside circle

The paddle was set by iterating through a square of defined values and setting pixels to the colour desired by iterating through the height and width of the rectangle and setting values to the desired colour.

When generating and rendering text a library of hex values is used. The user inputs the character array of text they wish to use. These hex values are at 90 degree angles so the code first rotates the text 90 degrees to be upright for each letter in the char array into a letter array. A for loop is then used to set the pixels of the letter array to the frame buffer. If small then the is simply reads from the letter array and sets the appropriate pixel. If large 1 pixel in the letter array sets 4 pixels in the frame buffer. This allows large bold text.

2.4 Game Engine Graphics

This section explains how the game engine uses sprites to make a usable resource in the final game engine. In the game engine each sprite (1 ball, 2 paddles) are kept a track of with coordinates set as global values. Each sprite can be created and destroyed using commands such as 'pongEngine_paddleCreate(number of paddle)'.

Once the ball is created it is required to be moved. This is carried out using the function 'pongEngine_moveBall(int angle, int speed)'. To do this code was created to make a ball path. The ball path is saved as a set of instructions in an array.

Firstly to create the set of instructions the angle of the path must be found. This is carried out by finding the angle from the centre of the ball to the outside pixels of the screen. The coordinates are input into function 'pongEngine_calcAngle' from a for loop which outputs an angle between 2 pixel co-ordinates. Once the angle has been found these coordinates are input into the function 'pongEngine_genBallPathInst(ballX, ballY, angleX, angleY)' with ballX, ballY being the current coordinates of the ball and angleX, angleY being the outside coordinates which are at the required angle. This function generates a set of instructions to an array of instructions. The set of instructions is created using Bresenham's line theorem which creates a non-visible line which the ball can follow. If the angle is not changed then the instructions will not be regenerated. Everytime the ball moves it is first destroyed (turns all pixels black) and then redrawn in its new location.

```
int pongEngine_calcAngle(int x1, int y1, int x2, int y2){
  double diff_y = y1 - y2;
  double diff_x = x1 - x2;
  double angle = atan2(diff_y, diff_x);
  float ang_d = angle * 180/PI;
  int ang_dint = ang_d;
  return ang_dint;
}
```

Figure 6: Code used to calculate angle between 2 co-ordinates

The paddles have 2 functions ('pongEngine_paddleSetYLocation(int player, int y)' and 'pongEngine_paddleSetXLocation(int player, int x)') which set the x and y value, destroys the selected paddle, redraws it, and finally stores the values into memory. These are used on initialisation, whilst in game the function 'pongEngine_paddleMove(int player, int direction, int speed)' is used. This function takes the values of player, direction, and speed. The speed set is how many pixels the paddle must be moved. Direction is either up or down.

To update the score three functions are used, one function is to add a point to a selected player which increases the value of score in a variable. Another function resets the scores to zero. Finally a refresh score function which takes the scores from the stored variables and renders the score on screen.The score values are split down into individual numbers and then sent to the seven segment displays. The numbers are also turned into a char array. Every time the score is refreshed a black rectangle is drawn over the text to make it blank and then it is re-rendered using 'pongSprites_WriteText'. An issue we encountered was random pixels appearing next to the score. The error was spotted by team member Sam Wilcock to be that a char array exit character ('\0') was missing.

3 Controls and Menus

3.1 Inputs

This subsection discusses the implementation of the PS/2 keyboard, pushbutton and slider switch inputs.

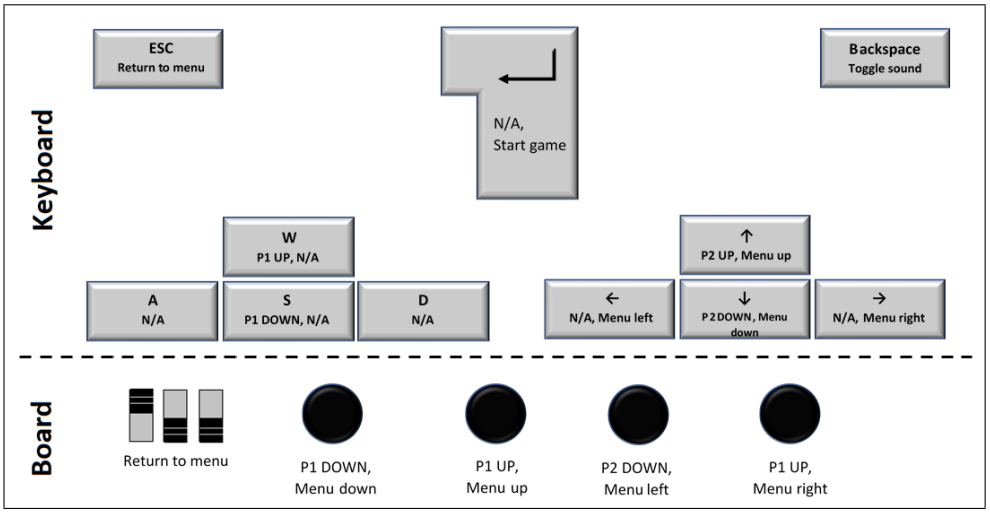

Figure 7: Input functionality

The slider buttons are arguably the simplest to implement. They are not attached to any of the built in IRQ exception handlers, and thus have to be dealt with by continuously checking their values. This is dealt with purely in the 'pongScreens.c' file as part of the gameplay screens' while loops; the pointer to the sliders' base address is defined initially, along with a buffer variable to store the previous value of the sliders as a new screen is formed, as such:

```
1  last_slider = *slider_ptr; // Ensure slider positions are saved
```

Figure 8: Save slider values

As the code for the gameplay screens are run, after the ball has been moved, the slider value is compared to the buffer, and if any of the switches have been moved, the gameplay mode variable is set to MENUS (a useful dummy definition in the header file) and the game exits to the main() while loop:

```
1      dir = arr[0];
2      vel = arr[1];
```

Figure 9: Check for exiting to menu

The use of the keyboard and the pushbutton switches allowed the use of IRQs as they have interrupt exceptions attached to them, thus avoiding continuously checking inputs unless they are flagged as having changed. In order to enable this, a custom 'DDRRomIRQ.scat' scatter file was written (see **Appendix**). For the keyboard input, a keyboardISR handler was written in line with the 'HPS_IRQ.c' format. On the keyboard interrupt flagging, the

PS/2 port interrupt is first deactivated to prevent further interrupts; the RVALID flag is checked and the PS/2 data FIFO read [1] to accept the incoming characters. The input characters are compared against some scancode definitions in the 'pongInputs.h' header [2]. Make and break codes are discarded and the key input is passed to the 'void Input(unsigned int key, unsigned int speed)' function. If the key is preceeded by a break code it is discarded to prevent bounce from single presses; the PS/2 FIFO is finally cleared by reading it until RAVAIL is decremented to 1, before the interrupt is reactivated. [1].

The pusbhutton ISR works in a similar way, by first reading the value from its interrupt register and then writing back to it to clear the interrupt flag (preventing infinite interrupts). The value is then compared in a simple state machine to ascertain which button was pressed, before the key is converted to a corresponding keyboard scancode and passed to 'void Input()'.

Additional macross which were written to help the functioning of the inputs include 'enableInputs(int enable)' which switches the input interrupts on and off; 'setInputMode(unsigned int _mode)' which sets the *mode* variable, in order for the screen functions to switch and the Input function to deal with different screens; 'int getInputMode(void)', which returns the current value of *mode*; and, most importantly, the 'Input(unsigned int key, unsigned int speed)' function, which is a state machine dealing with the inputs. The speed input serves a dual purpose. First, it was noted that it was far easier to press the keyboard keys faster than the pushbutton keys, in part due to the short typematic delay [2], and so this allows the function to penalise the paddle speed accordingly. The other use is in detecting whether the input has come from the keyboard or pushbutton for dealing with menu motions. Fig 10 below demonstrates the macro's logic.

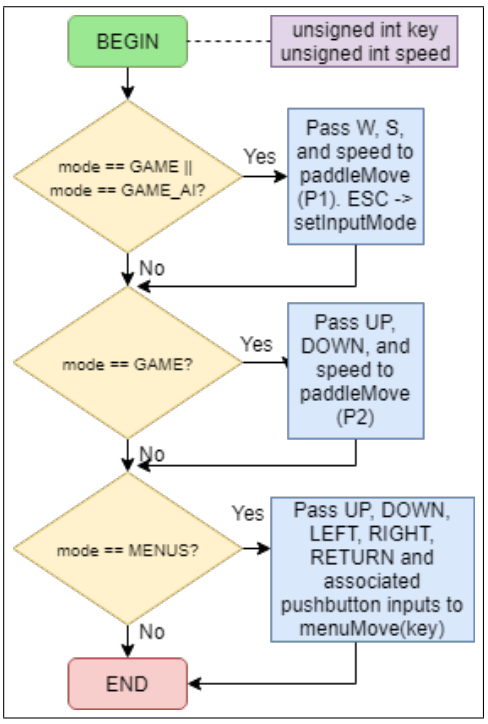

Figure 10: The Input() macro flow

3.2 Menus & Screen Navigation

In this subsection, the menu and screen driver is discussed, along with how they are interfaced with from the main program loop.

There are 3 main types of screen to be reached within the pong game: a start-up screen, loaded once the game is switched on; a menu system, which needed to contain important settings for the game and be reachable from the game; and a game-play screen, where the pong engine is utilised. The screen functions are called from the main() source. After the various libraries have been initialised, the start screen is called, and then the program enters an infinite loop which continuously checks the *mode* variable from pongInputs. Different functions are then called depending on the mode - either MENUS, GAME, or GAME_AI.

The start-up screen was simplest to implement, as it is non-interactive and runs on a timed basis; this meant that it could utilise purely blocking functions, including sound generation. At this point, the inputs are not enabled, so as not to produce unwanted effects for keypresses. A sound is played and a greeting displayed over a colour fill; this gave the added benefit at the time of debugging of being a useful tester for display and sound functions. This screen displays for a total of 4 seconds – 2 for the sounds to play in addition to an extra 2 second sleep.

```
1  void startScreen(){
2    // Clear screen and set input mode
3    Displays_fillColour(_RED);  // Fill background
4    setInputMode(MENUS);     // Ensure input mode set to menus
5
6    // Write greeting and refresh
7    pongSprites_writeText(60, 120, 1, "Welcome to armPONG", 0xFFFF);
8    Displays_forceRefresh();
9    ResetWDT();
10
11   startSound();          // Play startup sound
12   usleep(1000000); ResetWDT();  // Sleep for 2s
13   usleep(1000000); ResetWDT();
14   last_slider = *slider_ptr; // Ensure slider positions are saved
15 }
```

Figure 11: Code used to run start-up screen.

Figure 12: Start-up screen

The menu system contained both blocking and non-blocking elements, including input and timer ISRs, sounds and animation, making it more complex. The menu is initialised simply, by using the 'pongSprites_writeText' function to produce a list of items, with the currently selected menu item (dictated by variable *menuSelector*) highlighted in blue, in addition to clearing the 7-segment displays. The setting values for the game mode, paddle size and volume are displayed next to the text, where the mode can be either 2-player or 1 player (AI) mode, and the paddle size and volume are an integer from 1 to 9. The 'pongSprites_renderBall' macro was also used to provide a pointer to the current menu item. While the menu *mode* is set to MENUS, the 'gameMenu(void)' function simply loops, waiting. The motion of the menu and its effects are passed on to the 'menuMove(unsigned int direction)' macro, which the input interrupts interact with through 'Input'; this function stores the previous values of *menuSelector* and each menu option, and if there is a change, it shifts the ball pointer and colour highlighting of the text. It does this simply by writing over the previous text and ball items in the default colours (background white for the ball, black for the text), and then writing over the newly selected locations.

Figure 13: Menus, showing choice of item, option, and the animation which runs on START

The first option, Mode, dictates the game mode, and works simply by changing the *gameMode* variable which is passed to 'setInputMode' on starting a game. Second is the paddle size, which changes the paddle size with 'pongSprites_changePaddleSize'. This function was built especially, and works on the formula $Length_y(px) = 60 + (size - 5) \times 10$ - hence the default option of 5 (out of 10) gives a paddle height of 60 pixels.

The volume setting was slightly more difficult, as it was decided that changing the volume should play a tone at the current amplitude defined by this setting. It was found that the blocking nature of sounds combined with the fact that 'menuMove' is called by interrupts caused the sounds to play forever; hence this option only changes the volume (using 'setVolume') and then the screen's main loop plays sounds if it sees a change in this setting.

In order to reset the score, a function was added to pongEngine, which simply sets the scores for each player to zero. Finally, the start button resets the score if *gameMode* has changed, and then sets the input mode, allowing the menu loop to end and return to main(). An animation runs on pressing START, which moves the ball across the start text, erasing it.

It was initially found that if inputs were pressed whilst the menu items were changing, a bug occurred where the display did not accurately depict the current menu items selected. This was remedied by disabling the inputs and re-enabling either side of 'menuMove', using 'enableInputs'.

4 Audio Output

This section explains the method used to implement sounds in the final design. In order to include sounds, the DE1-SoC WM8731 and I2C drivers were utilised in combination with an example code. *[Reference T. Carpenter's work from lab 4??]*. It was noted initially that the code was slow to perform due to the float arithmetic involved in calculating the sinusoid of an angle, which caused the FIFO buffer of the audio device to overflow. We realised therefore that there were 2 real ways of approaching the problem: either, to utilise the onboard FPU of the DE1-SoC, or to create a faster approximation of the $\sin\theta$ function.

The code below demonstrates the fast sin function that was devised. The values of $\sin\theta$ from $0°$ to $90°$ were first calculated in Matlab, multiplied by 10^5 and rounded to whole numbers. These results were then copied into a lookup table within the 'lookupSin(unsigned int degree)' macro. The storing of the numbers as fixed integers as opposed to floating point numbers served to save on memory usage.

```
static unsigned int lookup[91] = { 0, 1745, 3490, 5234, 6976, 8716, 10453, 12187, 13917,
           15643, 17365, 19081, 20791, 22495, 24192, 25882, 27564,
           ... etc. ...
  if (degree <= 90){
    return (float) lookup[degree]/100000;
  }
  else if ((degree > 90) && (degree <= 180)){
    return (float) lookup[180-degree]/100000;
  }
  else if ((degree > 180) && (degree <= 270)){
    return (float) -lookup[degree-180]/100000;
  }
  else if ((degree > 270) && (degree <= 360)){
    return (float) -lookup[360-degree]/100000;
  } else return 0;
}
```

Figure 14: Code used produce fast sin lookup results for whole degrees.

By utilising the characteristics of the $y = \sin\theta$ function over the period $0°$ to $180°$, where [y] is mirrored about $x = 90°$, and the characteristic that the function is mirrored about $y = 0$ at $x = 180°$, it was then possible to find an approximation of $\sin\theta$ for any whole positive integer θ up to $360°$. Within the Sound macro, the float input to 'lookupSin', indicating the phase, was wrapped to a maximum of 360°and cast to an integer type. The maximum error in this lookup function at whole integers was approximately 4.9×10^{-6}.

The 'Sound(int _freq, float _duration)' macro first checks if sounds are flagged as on (via the *SoundOn* variable). If so, a timer ISR is created to run for the period length. The phase is then incremented by a set amount determined by the frequency argument. The sounds are pushed to the WM8731 FIFOs and played in a while loop which checks the *sound* variable. When the timer ISR triggers, *sound* is changed to 0 and the Sound function finishes. It was initially noted that during gameplay, key presses caused gaps in sounds, so the 'Sound' macro temporarily disables inputs for the duration it runs. Variable volume was implemented by adding a *VOLUME* variable, which can be changed globally using the 'setVolume(int _volume)' macro. The amplitude sent to the WM8731 is set as $2^{23+VOLUME}$, by bitshifting the value 2^{23} left VOLUME bits. In order to make sound generation more intuitive, an additional header file was created containing the frequencies of musical notes from A3 to G6, and then sound macros were created for the various components of the game, including paddle collisions, scoring, and the start sound.

5 Game Physics

6 Problems Encountered

7 Conclusion

8 Appendix

References

[1] Altera, *DE1-SoC Computer System with Nios II*. Altera, 2014.

[2] Altium. Ps2 keyboard scan codes.