Lab 5: Audio I/O and Wave Generation

For ECSE 324: Computer Organization

Report Written by Group 03

Antoine Wang (260766084)

Alfred Wang (260771591)

Due Dec 2nd at 5PM

McGill University

Audio I/O

The first section of this lab aims to generate a alternating square wave by using the audio implementation on the DEC-So1 Computer. The essence of generate an audible output is identically same as other I/O devices such as PS/2 keyboard and VGA display described in previous labs.

Thus, in the assembly code, the program checks the status register (called FIFO_SPACE located at 0xFF203044). The information in this register keeps track of the FIFO buffers of the audio by having two 8-bit numbers. Bit 24-31 are called WSLC which shows how much space is left in the left buffer. Bit 16-23 keeps track of how much space is left in the right buffer. Both bit patterns will appear to be zero if the buffer is full. Thus, by making sure WSLC and WSRC is non-zero, the program makes sure the writing command is valid. The checking uses ANDS command which update the Z flag, then BEQ command will handle the case if WSLC or WSRC is zero. R5 and R6 holds two check patterns, 0xFF000000 and 0x00FF0000, corresponding to bit 24-31 and 16-23. The code episode is shown in figure 1.

```
ANDS R5, R1              // check bit 24-31, if wslc equals to zero, update flag
BEQ FULL                 // if equals to zero, meaning left fifo full, branch to FULL
ANDS R6, R1              // check bit 23-16, if wsrc equals to zero, update flag
BEQ FULL                 // if equals to zero, meaning right fifo full, branch to FULL
```

Figure 1: Check if buffers are full

Then, figure 2 and figure 3 demonstrates what to do in two conditions. If both left and right buffers have space the rest is simply a writing command using STR (Figure 2). If one of the buffers is unavailable, it branches to FULL which return 0 and skip the storing procedure. The writing fails.

```
// If both left and right has space, starting the write operation
STR R0, [R2]             // Store R0, the input parameter into both left and right
STR R0, [R3]
MOV R0, #1               // return R0 with 1, indicating a successful return
BX LR
```

Figure 2: A successful writing, storing bit pattern and return 1

```
// If either left and right is full, the writing fails.
// return a zero and end subroutine
FULL:
    MOV R0, #0           // return R0 with 0
    BX LR
```

Figure 3: A failed writing, return 0 directly

By testing the subroutine, a C loop is implemented to see if a wave can be generated in this way. The loop is basically alternating the input pattern when sufficient “Highs” and “Lows” are generated. The loop also has the function of checking whether the writing is successful. If the FIFO buffer malfunctions, it will halt the increment of the counter and keep trying to push values into the buffer.

```
while (1) {
    if (write_in_FIFO_ASM(signal)) {        // if a write is successful, increment the counter
        counter++;                          // if write fails, try writing again
    }
    if (counter >= 240) {
        counter = 0;
        if (signal == 0x00FFFFFF)           // every time half a cycle of high signal (240 samples) completes
            signal = 0x00000000;            // Switch to low signal
        else
            signal = 0x00FFFFFF;
    }
}
```

Figure 4: Code block of the C implementation testing the Assembly audio routine

A possible improvement is that there is no clock monitoring the actual frequency of the wave output. A simple calculation, which divides the natural frequency 48000Hz into 480 samples per output period to generate the 100Hz square, is not precise enough since the timing also depends on the delay of the code execution. In later section there will be usage of HPS_timers.

Make Waves

In this section audio.s.o file will be used directly, thus please refer to the API document to check the definition of new subroutines which writes to the audio I/O. One significant difference is that the subroutine responsible for writing, "audio_write_data_ASM", now takes two inputs addressing for left and right buffers instead of one like in previous section.

A wavetable.s file stores a long series of values which convert a sine wave from analog to digit. The two equations below are used to index the wave table, so the value of the sine wave is approximated in the discrete time model.

$$index = (frequency * time)\ \%\ 48000 \quad \textbf{(1)}$$

$$signal[time] = A * table[index] \quad \textbf{(2)}$$

Thus, by using both equation (1) and (2), the standard sine wave can be compressed or elongated based on the required frequency since frequency dictates index, which is used to pinpoint a specific sine value from the sine table. The actual C implementation is straightforward. Figure 5 shows how the sine index is calculated is transcribed into code. Figure 6 illustrates the linear interpolation of the sine wave if the actual index in between two integers.

```
int freqInt = (int) (freq*t);               // integral part of the wave index
double freqFractional = (freq*t) - freqInt; // fractional part of the wave index
int index = freqInt % 48000;                //  make sure the range of the index is within 48000
```

Figure 5: Finding and partitioning the sine index

```
double signal = (1.0 - freqFractional) * sine[index] + freqFractional * sine[index + 1];
```

Figure 6: Generating the sine signal at a particular moment using indexing and linear interpolation.

After getting the signal from the sine table, the following to do is to write it to the audio at a certain speed. Figure 7 shows a timer, TIM0, which is used to ensure the sampling frequency is about 48000Hz.

```
// Config a timer to set samplong time in to 1/48000Hz = 20.8*10(-6)s
hps_tim.tim = TIM0;
hps_tim.timeout = 21; //1/48000 = 20.8
hps_tim.LD_en = 1;
hps_tim.INT_en = 1;
hps_tim.enable = 1;
HPS_TIM_config_ASM(&hps_tim);
```

Figure 7: Timer implementation of a 48000Hz sampling rate

Lastly, a while loop keeps writing the signal to the audio buffer and forming a wave. It writes when an interruption is generated at the timer, meaning the timer finishes one period. The program then calculates the corresponding value at a specific frequency from the sine table and writes to the audio (Figure 8).

```
while(1){
    if(hps_tim0_int_flag == 1) {
        double signalSum = getSample(130.813, time);        // Set a test note (note c) and calculate the corresponding output from wave table
        audio_write_data_ASM(signalSum, signalSum);         // Write the wave to the audio output
        hps_tim0_int_flag = 0;                              // clear interrupt timer for next counting cycle
        time++;
        }
```

Figure 8: Generating the sine wave to the audio

Control Waves

The task of this part is to make it possible for user to play the synth using a PS/2 keyboard. The basic ideal of implementation is to map each note to a key on the keyboard, then record the status of the keys (whether the key is pressed or released) using an array, and finally generate the output sound signal according to the key status array.

An array named “keyPressed” is used to hold the status of the key. It is initialized with 0, which corresponds to none of the key is pressed. As the sample code shown below, if “A” key is pressed, the first element in the status array will be changed. If the original status is “0” (key not pressed), the status will be changed to “1” (key pressed), and vice versa (Figure 9).

```
153             case 0x1C:                  //case "A" (A is pressed)
154               if(keyPressed[0]==1){     //if pressed (1), change it to not pressed (0)
155                 keyPressed[0]=0;
156               }else{                    //if not pressed (0), change it to pressed (1)
157                 keyPressed[0]=1;
158               }
159               break;
```

Figure 9: One sample switch case changing the status of the key pressed

A press-and-release action of a key will generate 3 hexadecimal signals from the keyboard. For example, if “A” is pressed and released, the signal sequentially generated would be 0x1C, 0xF0, and 0x1C. In order to handle the repeated signals and 0xF0, we add an int variable called “pressed” as shown below. Whenever the keyboard input is 0xF0, “pressed” will be set to 1, and the next input will be ignored (Figure 10 and 11).

```
148         if (read_ps2_data_ASM(&value)) {    //if there is an input from the keyboard
149           if(pressed==0){         //if the previous key is released
150             switch (value){
```

Figure 10: The “pressed” flag oversees the entire condition of the switch loop. If the flag is 1, the switch loop will be skipped, and the key status will remain the same until the next input.

```
226             case 0xF0:      //this is used to handle the press and release action
227               pressed = 1;  //if 0xF0 is detected from keyboard, it means that the key is pressed and going to be released
228                             //then we ignore the next input from the keyboard by setting 'pressed' to 1
229               break;
230             } // end of case
231           } else{
232             pressed = 0;
233           }//end of pressed
```

Figure 11: After finished handling the input, an extra case will handle the 0xF0 signal generated by the PS/2 keyboard.

Volume control is implemented by using “-” and “+” keys on the keyboard, and an int variable called “amplitude”. As the code shown below, when “+” is pressed and if the volume is smaller than the maximum threshold, “amplitude” will then be increased by 1. The amplitude will then be multiplied to the signal generated, so that the volume can be increased. Reducing volume is just the opposite implementation of increasing volume (Figure 12 and 13).

```
219             case 0x55:                          // if "+" is pressed on the keyboard
220               if(amplitude<max_vol) amplitude++;  // if the volumn is smaller than max volume, increase by 1
221               break;
```

Figure 12: Responding to the pressing of “+” key to increment the amplitude multiplexer

```
236         //generate the signal at this t based on what keys were pressed
237         signalSum = amplitude * generateSignal(keyPressed, t, frequencies);
```

Figure 13: By multiplying a integer to the input signal, the wave is enlarged in amplitude

In order to synthesize the notes into one signal, we create a method call “generateSignal”. This method uses a for loop to get all key status from the array mentioned above. Each time the key is sensed to be pressed, it will generate the corresponding signal and add it to the total signal, which in called “data” in the method. After going through the entire array, the total signal “data” will be returned, which is the final synthesized signal.

Furthermore, since the sampling frequency is 48000 Hz, the calculated period is 1/48000 seconds, which is around 21 microseconds. Since we are using TIM0, which is a microsecond timer, we then set the timeout of the timer to 21. For more specific information about the timer please refer to the previous section at Figure 7.