McGill University

ECSE 324 - Computer Organization

Group 29

Synthetizer Laboratory Report

260741673
Karine Mellata

260653125
Roshan Ramesh

December 9, 2017

1 Introduction

The goal of this lab was to implement a 8-note digital synthesizer. This report is separated in three parts, which each assess the methodology used, challenges faced and improvements for the three I/O devices needed: Audio, Keyboard, VGA display. Throughout the lab, the software was implemented in order to abstract the most out of the main method. This was achieved by thoroughly organizing almost each functionality into respective methods. This greatly helped the process of debugging and resulted in a much more efficient and readable code.

1.1 Timers implementation

The implementation of the timers was the same for all three timers. A simple ARM subroutine called "HPS_TIMX_ISR", X being the timer number, was implemented in order to be able to set the flag variable "hps_timX_int_flag" to 1, which would mean an interrupt. These interrupts are being thrown at a rate that is proper to each I/O device, and determined by the timeout value. These flag variables were made accessible to the main C code. In order to configure all timers, a key point is that the Enable bit needed to be set to one in order to allow interrupts. The timers control the rate at which a certain part of the code runs, as displayed in the pseudo-code below.

```
while(1){ //This loop is in the main method, but displayed here for clarity
    if(interrupt_flag == 1){
        //Logic implementation
        interrupt_flag = 0;
    }
}
```

This timer is needed for a simple reason. The Cyclone V GX Nios II Processor runs at approximately 200Mhz, which is one instruction every 0.5 µs (if we assume one instruction runs in about 1 cycle). However, each I/O device can read or write data at a much slower pace (sometimes by a factor of almost 200,000 times!). This is why a timer is required to synchronize each parts of the program with its respective I/O device, and consequently avoid useless execution of code (For example, constantly trying to write to the audio FIFO while the RVALID bit is stating that there is no more space).

2 Audio - Make Waves

2.1 Description and approach

The first task was to generate a signal given a frequency. We assessed that the right sound [1]was being played using headphones. Firstly, a method called output() was implemented taking in as parameters the frequency and the time (Note that this method was changed later in the lab to take as argument the amplitude in order to change the volume). The wavetable.s, which represents an entire period of a 1Hz sine wave at a sampling rate of 48000 (48000 different values). These values were then used to generate a frequency of our choice based on the current time using these simple equations:

$$index = (f \times t)\%48000 \tag{1}$$

$$signal[t] = amplitude \times table[index] \tag{2}$$

Afterwards, a method called "make_sound()" was implemented. Note that this method is not in our current version of the code, as the current method is "make_sound_w_interrupts". Only slight improvements differentiate them. The first step of implementing this method was to initiate a timer in the main part of the C code that would control at which rate the audio samples would be written to the audio FIFO registers, and therefore at which rate the samples would be heard. This was done outside of the main infinite while loop.

2.1.1 Timer values

The timer implemented for controlling the writing of audio samples was based on the sampling rate of the audio codec. With the sampling rate being 48,000 samples/second, the simple math to find the timeout is below.

$$\frac{1}{48\,000\,\text{samples}} = 20.83\,\mu\text{s} \tag{3}$$

We therefore know that to follow the audio's pace, we must write to the FIFO approximately every 20.83 µs.

2.2 Challenges faced

The major challenge in this part was to implement the first timer. As we had access to the ARM code, and the methodology for instantiating a timer, it was still difficult to visualize how to use timers in this special case. At first, we implemented the complete first part without any timers and everything was running fine. However, the implementation with the timer did not work for a long time as we had forgotten to reset the hps_tim1_int_flag to 0 after observing an interrupt. In addition, we were not running the code to accept Exceptions for the longest time, which was a rather simple error, but took a long time to notice.

2.3 Further improvements

A further improvement that could have been made to the code would have been to store all values of a signal to an array of length 48,000. As a signal is generated for a desired frequency, it is useless to re-generate it, as it will not change if we re-press the key or let it play for more than one period. This would've greatly improved the processor usage, as it is much easier to access an element in an array than to re-calculate the entire signal.

[1] This was achieved using an online frequency generator and comparing the two sounds.

3 Keyboard - Control Waves

3.1 Description and approach

Key	Make Code	Note	Frequency
A	0x1C	C	130.813 Hz
S	0x1B	D	146.832 Hz
D	0x23	E	164.814 Hz
F	0x2B	F	174.614 Hz
J	0x3B	G	195.998 Hz
K	0x42	A	220.000 Hz
L	0x4B	B	246.942 Hz
;	0x4C	C	261.626 Hz
8 (keypad)	0x75	N/A[2]	N/A
2 (keypad)	0x72	N/A[3]	N/A

Table 1: Respective keys mapped to their respective frequencies

The second part of the program handles the PS/2 keyboard I/O device. As seen in Table 1, the goal of this part of the program was to map each keys to a respective note (frequency). A method called "handle_keyboard" helped us achieve this goal. This method is called in the main while loop. A local variable called "key_pressed" stores the current key that has been pressed. Using the method "read_ps2_data_ASM" the current key being pressed is stored to this local variable. Only if a key has been pressed, using an IF statement, a switch statement is entered. This switch statement's cases are the make codes of all the keys observed, the break code "0xF0" and the default. The logic, as seen in Figure 1 used to implement the switch is the following:

1. If a break_code is observed, enter the "break_code" state.
2. If a make_code is observed,
   (a) If we are not currently in break code state, turn on the key.
   (b) If we are currently in break code mode, turn off the key.
3. If any other key is observed (default statement), exit the break code state that a random key would have generated (which we don't want to account for).

This part of the code is run according to the keyboard timer. After this piece of code has run, the "make_sound_w_interrupts()" method is called within this method. This method is an improved version of the make_sound(), which handles the release of the break_code.

[3]This increases the amplitude(volume) by 2.
[3]This decreases the amplitude(volume) by 2.

In other words, the signal will only be written to the FIFO if we are not currently in break_code state, as seen in Figure 1. In addition, this method takes in as parameter the amplitude desired, which is first initialized to 10. If the keys displayed in Table 1 are pressed, the amplitude decreases or increases accordingly.

3.2 Timer values

As the audio, a timer controls when the values are being read from the keyboard FIFO. However, a different timer had to be implemented as the keyboard is much slower than the audio. The keyboard's frequency is, on average, 10.9 characters/second. This simple math is displayed below to find the ideal timeout value.

$$\frac{1}{10.9\,\text{characters}} = 91\,743.11\,\mu\text{s} \tag{4}$$

In order to make sure that we are accounting for **all** keyboard values, and not missing any, the timeout value was set to 80,000.

3.3 Challenges faced

The first challenge faced in this lab was the handling of different states. As it is impossible to access the current value in the FIFO and the previous one, hence knowing if the key is being pressed or released, a notion of "memory" had to be implemented, which is why we introduced the break_code "state". This helped us control efficiently all the keys together, and cover all possible combination of keys as well. A second challenge faced was a delay. As we first implemented our code, without checking whether or not we were in break_code mode within the audio method, a delay was observed between the time we released the key and the time we heard the sound stop. After quite a bit of time spent debugging, we realized that for every time a key was pressed, the code was running for at least an entire period before stopping. This meant that our code was not taking into account a break code immediately. Hence, we changed the if statement governing the audio method to the statement displayed below.

```
if(hps_tim1_int_flag && no_break_code){
    \\Logic here
}
```

3.3.1 Further improvements

This part of the program was implemented in the most efficient way we could think of. However, further functionalities could have been implemented, such as more keys for different sounds. Another interesting improvement would have been to implement a check to know the type of the keyboard plugged in (by analyzing the I/O ports), in order to be able to handle set 1, set 2 and set 3 of make and break codes. Currently, our program only supports the PS/2 keyboard.

4 VGA - Display Waves

4.1 Description and approach

The third and last part of this program allows us to display the waves that we are generating from the keyboard on a VGA display. This functionality is given by the method "display_waves()". This method is called in the main while loop. Firstly, a global variable called "global_write" has been created to define the current signal. Every time something is being written to the audio, this global variable is set to the local variable "write" (that is associated to the generated sample), within the audio method. After, we simply use the method "VGA_draw_point_ASM" to draw the point the the screen. However, important parameters need to be passed this method. The first one is the time. This global variable is constantly being updated as the program runs, and ranges from 0 to 48000. It was then needed to divide this number by at least 150, in order to get the desired range of [0,320] and therefore not writing outside of the screen. We divided the time value by 200 to achieve a more compact sine wave. Secondly, the y value is defined by this "global_write" variable. This variable also needed scaling in order to achieve the [0,240] range. Lastly, the last parameter is the color of the sine wave. As an extra functionality, we implemented different colors for different segments of the sine wave, which is simply controlled by an if statement controlled by the time.

4.2 Timer values

The logic behind the timer controller the VGA display follows the same logic as the two previous timers. The VGA controller is slower than the audio, but faster than the keyboard. It has a frequency of 60 frames per second. The simple math is displayed below.

$$\frac{1}{60\,\text{frames}} = 16\,666.66\,\mu\text{s} \tag{5}$$

However, this timeout value was not used as we realized this was, in practice, much too slow. As the audio was sampling at its own pace, and the keyboard sampling at its own pace also, the resulting VGA writer would not always have something to write (even with efficient timers, it is impossible to always successfully write to the audio FIFO or always read a key_pressed exactly when required). Therefore, we had to account for this, and made the timeout value much smaller. We experimented with different timeout values, and a timeout value of [4000,7000] seemed like the most efficient at displaying waves.

4.3 Challenges faced

One of the challenges faced was not in the implementation of this display, but in the logic behind it. As our display is continuously writing the the VGA, we can see a continuous wave "advancing" across the screen, and we can see real time changes in the wave (hence, in one frame we can see multiple frequency, similar to a time-line). However, this added functionality had its disadvantages. It made it extremely hard to have one timeout value that would clearly display all frequencies altogether. This part took a lot of trial and error, as a timeout values being too high would result in two sine waves in a double helix structure.

4.4 Further improvements

An improvement for this program would have been to have different timers for different frequencies. As this is not the most efficient implementation, and adds a great deal of complexity, we decided not to go with it. However, this would have greatly improved the quality and clarity of each sine waves, as some timeout values outputted really clear sine waves, and other timeout values outputted a different set of really clear sine waves. In addition, it would have been more efficient to write a black point to the VGA display rather than clearing all buffers.

5 Appendix

5.1

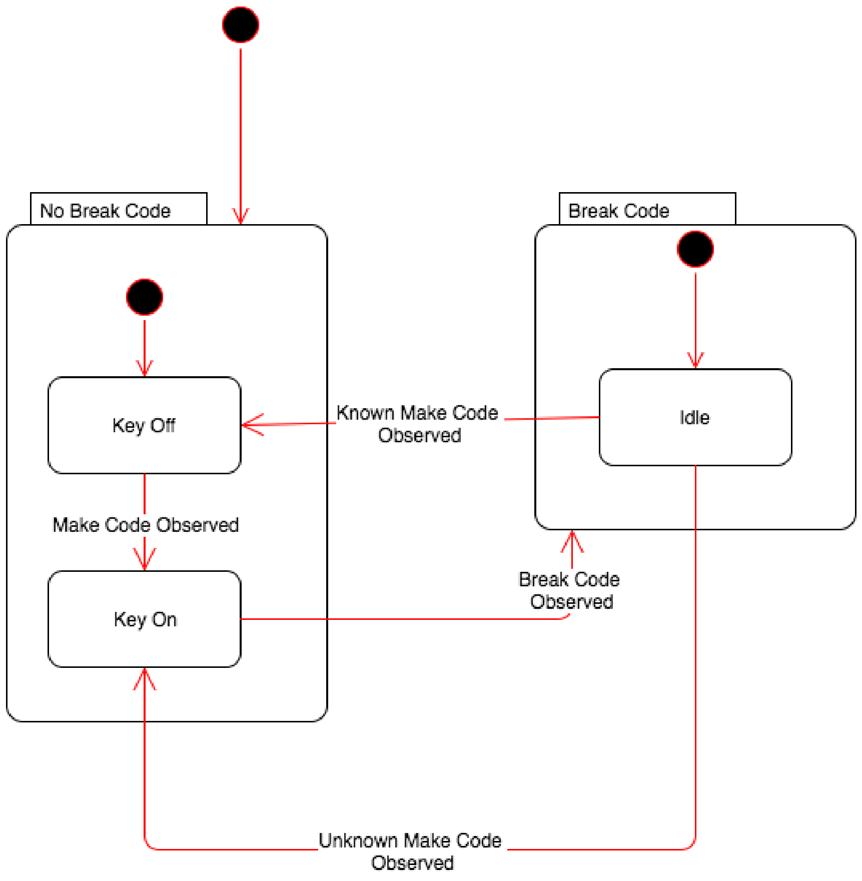

Figure 1: Finite State Machine for one key pressed [4]

[4]Note that the composite states are not necessary, they are only represented for clarity.