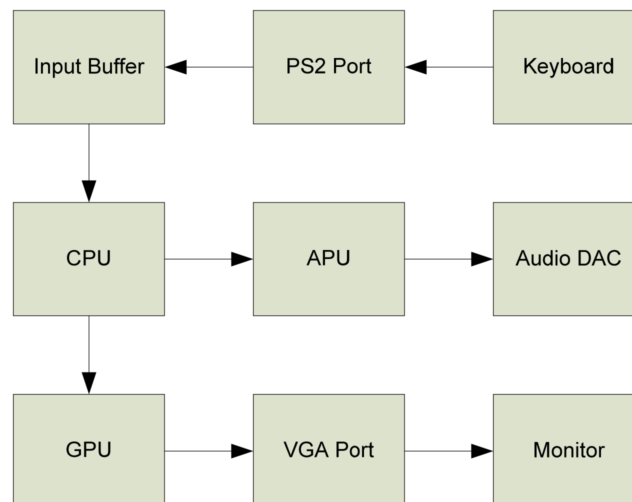
ECE 576: Lab 3

NiosII with μC/OS Audio Generator

Idan Beck, ib54@cornell.edu

Introduction

In this lab an Audio Synthesizer interface was implemented. This included 4 wavetable synthesizers and a random noise source that could be added together. The sources could be used as duration based sources or as signal generators. This was quite useful since the Audio Synthesis module could provide as either a general purpose synthesizer or a signal generator. The design included a VGA interface for the purpose of setting up, calibrating and displaying the status of the system. Also a console was designed for this lab so that input could be obtained from an input buffer that was also designed for this lab. The input buffer implements a circular FIFO buffer that stores the converted ASCII codes from the inputted scan codes from a PS2 keyboard attached to the DE2 board. An overall system diagram is shown below.



Design

Hardware

The main hardware aspects of the design developed in this lab was the NiosII system implemented using SOPC builder, the input buffer, and the APU which included the DDS synthesis of the wave table oscillators and noise source. The GPU design was described and explained in lab 2 so please note this for reference.

NiosII System

The system designed for this lab required a number of interfaces for the APU, GPU, and the input buffer. The APU was mainly an output port, the input buffer was mostly an input port, and the GPU was mostly an output port. The overall system is shown with the different reference addresses. Also SDRAM was used for the program memory as can be seen in the below screen capture:

Use	Module Name	Description	Input Clock	Base	End	IRQ
✓	cpu_0	Nios II Processor - Altera ...	clk			
	instruction_master	Master port				
	data_master	Master port		IRQ 0	IRQ 31	
	jtag_debug_module	Slave port		0x00800000	0x008007FF	
✓	sdram	SDRAM Controller	clk	0x00000000	0x007FFFFF	
✓	jtag_uart	JTAG UART	clk	0x00800870	0x00800877	0
✓	GpuInterface	PIO (Parallel I/O)	clk	0x00800800	0x0080080F	
✓	GpuComplete	PIO (Parallel I/O)	clk	0x00800810	0x0080081F	
✓	GpuValid	PIO (Parallel I/O)	clk	0x00800820	0x0080082F	
✓	PS2BufferEmpty_N	PIO (Parallel I/O)	clk	0x00800830	0x0080083F	1
✓	PS2Valid	PIO (Parallel I/O)	clk	0x00800840	0x0080084F	
✓	PS2ASCIIOut	PIO (Parallel I/O)	clk	0x00800850	0x0080085F	
✓	PS2Complete	PIO (Parallel I/O)	clk	0x00800860	0x0080086F	
✓	APUValid	PIO (Parallel I/O)	clk	0x00800880	0x0080088F	
✓	APUOp	PIO (Parallel I/O)	clk	0x00800890	0x0080089F	
✓	APUData	PIO (Parallel I/O)	clk	0x008008A0	0x008008AF	
✓	timer_0	Interval timer	clk	0x008008C0	0x008008DF	2

Timer_0 is used by the operating system to generate the correct intervals. Notice that the input of the PS2BufferEmpty_N has an associated IRQ. Although this was never needed to be implemented in this lab in a truly multitasking environment this could be very useful to interrupt the OS for user input.

Input Buffer

The input buffer consists of a PS2 state machine which spits out the scan code of the last pressed key. If the pressed key is shift the PS2 module outputs this but does not trigger a valid key. The scan code and shift enable are then fed to a scan code decoder which interprets them and outputs an ASCII character. This was done in pure logic since due to the shift enable introduces some logic required. This may be fixed in later versions but since this was such a small number of logic elements I did not further investigate it.

There are two state machines in the buffer, one for reading and the other for writing. When the key is valid as signaled by the PS2 module in the input buffer the ASCII key is written by the buffer to the write position in the circular buffer and the write position incremented. When the read and write positions of the circular buffer differ the PS2 input buffer will lower the empty buffer signal. This will signal to the environment that there exists a character in the buffer waiting to be read.

To read the buffer the environment sets the iValid line. This will read out a character into the output data on oASCIIout. Once this is registered as a valid output the oComplete flag is set to tell the environment that the output is valid. Once the environment is done reading the character it will lower the valid flag thereby resetting the read state machine.

APU – Audio Generation

The APU is rather straightforward. It uses the AUDIO_DAC and I2C_AV_CONFIG modules for the control and drive of the Audio DAC. The audio dac also required an audio control clock which was already being generated in the VGA module for us in the VGA_Audio_PLL. Instead of using a third PLL the signal was wired through the GPU module through the top module all the way down to the APU module. This signal is called iAUDCtrlClk and can be tracked through the design.

The APU uses the I2C_AV_CONFIG to initialize the Audio DAC using 10 commands. Those commands are shown in the below table with their functionalities:

Command	Op-Data	Use
0x001A;	00-1A	Sets register 0 to the left line gain value of 1A
0x021A	02-1A	Sets register 1 to the right line gain value of 1A
0x047B	04-7B	Sets register 2 to the left headphone volume value of 7B
0x067B	06-7B	Sets register 3 to the right headphone volume value of 7B
0x08F8	08-F8	Sets register 4 to turn on the ADC DAC by turning the sidetone volume to FF, turning on the sidetone, the DAC, and turning on bypass.
0x0A06	0A-06	Sets register 5 to set the digital path by turning on the DAC soft mute and setting the de-emphasis of 48KHz to 2.
0x0C00	0C-00	This turns on the power for all on, setting register 6 to the value of 00
0x0E01	0E-01	Sets register 7 to the format of the input to the DAC. Sets the bit length to 16 bits and MSB left.
0x1002	10-02	Sets register 8 to normal mode at 384f_s oversampling
0x1201	12-01	Sets register 9 to activate the codec

Much like the design of the GPU you will notice that the APU takes in instructions in the form of 16 bit wide operation instructions. The exact definitions of the operations are detailed in the APU.h file. The interface was made to be 16 bits wide as to ensure compatibility for future uses with the GPU interface. However, only 8 bits of this interface is ever used. The OP code is used to both set the target and operation for the data line which is 32 bits wide. Essentially the APU looks for an iValid and when this occurs it will read the Op and manipulate the internal registers appropriately with the input data.

Other than the details of the APU interface the APU is rather self standing and requires no other manipulation for operation. Inside the APU reside four wave table oscillators. The output of the APU to the audio DAC is the addition of all of the outputs of the wave table oscillators and the noise source.

The wave table oscillators are implemented using Direct Digital Synthesis with a look up 16 bit 2's compliment LUT designed using matlab. Each of the different DDS units has it's own independent frequency, duration and On variables which can each be manipulated individually or with the ALL specification in the op code.

The DDS unit's increment is then set with the frequency based on the following multiply:

$$Increment = \frac{2^{32}}{47.04\ KHz} F_{out}$$

$$Increment \approx 91304 \cdot F_{out}$$

This value was close enough to ensure for a good enough accuracy for the outputted waves. The way that wave duration was implemented was using a timing variable which would count up to the WaveDuration variable. Once the two matched the time would be reset and the WaveOn would be set low. Since the wave would only free run when the WaveOn variable was set high then this would halt the wave. A special clause was included, however, to allow for signal generation properties. If the wave duration was zero then the prior was skipped and the wave was allowed to free run forever or until the WaveOn variable was de-asserted.

The noise source was not implemented with a look up table since this would not have been random in any means. The final version utilized a 32 bit shift register with taps taken at the 32nd, 31st, 30th, and 10th bits OR-ed together. This combination was taken from an online source found through Wikipedia for Linear Feedback Shift Registers. This noise source was then run through a low pass filter with a variable cutoff frequency as designated by the internal state of the APU. This frequency could be adjusted through the audio interface as well.

To generate the random numbers the LFSR was shifted each time step. The MSB of the LFSR was then used as x(n), and the previous value of NoiseOut as y(n-1) in the following equation:

$$y(n) = x(n) + \beta(y(n-1) - x(n))$$

Where β is a number between [1,0] which represents a direct mapping to the normalized frequency between 0.05 and .5 the Nyquist frequency. The value of β was calculated in software and then passed to the APU to get rid of complicated border cases. This way the value is considered valid as soon as a new frequency is set to the APU for the noise source.

Software

The software designed for lab consisted of the PS2 and APU APIs as well as an implementation of these APIs through a Console application that enabled the operation of a the APU API manipulations.

PS2 API

This API was rather lightweight. It simply interfaced with the input buffer and allowed the user to read an individual character or read in the whole buffer that was yet to be read. The two functions are shown below:

```
int GetPS2Char(char *pcCharOut);
```

This function would output pcCharOut as the next character in the input buffer FIFO. The return value would be 1 if success or 0 if the buffer is empty.

```
int GetPS2Buffer(char *pszBuffer, int *cchBuffer);
```

This function will output a buffer of length cchBuffer. The memory location is allocated by the function however the caller must free the memory. If the buffer is empty then the function returns 0, otherwise it will return 1.

The Console

The console API consisted of three functions. The first function, InitializeConsole, is trivial. All it does is set up the console graphics and display some text on the screen using the GPU API. The UpdateConsole function is where all of the input occurs. This function displays the caret and makes it blink so that the user can see where the whitespaces occur. Next the function checks the input buffer through the PS2 API to check whether any input has been entered since the last time it was entered. It then checks to see what kind of input it is and other than the case of the entry of an Enter command it affects the console's internal buffer szConsoleInput accordingly. Also it will draw the characters onto the screen so that the user may see what it is that they are typing.

In the case that the user has inputted an Enter command the function will append a whitespace to the buffer and write over the characters in black to erase them from the screen. It saves the length of the buffer to a variable to be used within the tokenizer function and then resets the cchConsoleInput variable so that the next time the buffer is written to it is done so at the beginning of the buffer. This is better in performance than resetting the whole array which would be a waste of cycles. Then the function will return with the value KEY_ENTER which allows a program to know when to run the tokenizer function GetTokens.

GetTokens is a tokenizer function which utilizes the standard IO function strtok() function. This function will step through the console's internal buffer and spit out a range of tokens and a range of their respective lengths. The lengths is never truly used since strcmp() is used to compare the strings, however this could be useful for things in the future so was included in the functionality as well. This function is rather straightforward but tends to be a little buggy at times. There were a few bugs in the interface between GetTokens and UpdateConsole, however the emphasis of the lab was not on these and since the input buffer was being transferred correctly these bugs were deemed acceptable and will be returned to when it is more crucial.

APU API

The APU API is large because of the fact that each of the functions was repeated for each of the different oscillator sources. In hindsight this should have been more of a table based process however this will be regressed in the future. Each oscillator source has its own set of functions:

SetWaveduration – This sets the wave's duration, updates the value shown on the screen, and sends it to the APU.

SetWaveFrequency – This sets the wave's frequency, updates the values shown on the screen and sends the updated value to the APU. For the noise source this would convert the frequency to the normalized frequency by dividing it by the Nyquist frequency then multiplying it by 0x10000 and subtracting this from the 0x10000. The 0x10000 value represented a value of 1.0 in the signed multiply. This would then mean that we rescaled the input frequency to the normalized frequency and then to the correct β for the low pass filter. This saves a great deal of hardware complexity.

SetWaveOn – This function simply turned the wave on and off. In the case of the duration not being equal to 0 when the wave would outlive the duration given to it the on would still appear on the screen since the APU had no way of letting the CPU know that it was done. If the APU is ever made into a fully fledged synthesizer this would be an important feature but was not needed for our purposes.

SetWaveOff – This function would turn the wave off and remove the On from the screen for the appropriate wave.

SetWaveADSR – This function was written for the future when the APU module is converted into a real synthesizer. ADSR stands for Attack Delay Sustain and Release. This function would display the amplitude envelope on the screen scaled appropriately and would then set the APU values for the wave. This was never carried out in hardware however although the registers were coded but there is no way to set them and they do not affect the wave itself either.

The APU also had a function InitializeAudioSynth, this function initialized all of the oscillators to some general starting parameters, displayed and displayed the displays for the different oscillators. It would also initialize the APU interface to 0 so that no strange conditions would occur in the beginning of the program.

Putting it all together: OS

The µC/OS operating system was used for this lab. In all reality due to the non-blocking nature of the console's implementation of the PS2 API this could have well been a single threaded application. However, the multiple threads provide a powerful way to expand on this program with additional components such as a display that outputs the audio level on an oscilloscope like display. There was no time to implement this however but may be revisited in the future.

The operating system consists of two threads: audio_synth and console. The audio_synth thread is used to initialize the APU and was originally intended to be used to implement an oscilloscope like display which would output the wave from to the VGA display. This was never implemented however will likely be revisited in the future. This thread runs and then sleeps for 10 ms to allow the more important thread to continue to operate.

The second thread, console, is where most of the work occurs. This thread initializes the console and updates it every 10ms. Essentially it polls the input buffer using the UpdateConsole function and when this function returns INPUT_ENTER this thread parses through the output spew and figures out what the user is trying to do with their commands. It then appropriately converts the input parameters to correct values and dispatches these values to the appropriate APU API for the wave specified. One of the most convenient features of this was that it was possible to use the 'all' command

to manipulate all of the synthesizers in unison which allowed for easier debugging as well as more convenient shut off of an annoying tone.

Setting up the OS was done in main. On entry the program creates the OS message queue and the two tasks with their respective priorities and stacks. Then it stars the OS which should never return.

Testing

Hardware

Hardware testing was done for each of the different modules independently and then bundled together with a CPU to then fully test the module using a program to make sure that all border cases were working correctly. For example, the PS2 buffer was first implemented by outputting the PS2 spew onto the HEX displays, then the value of the ASCII and then finally was integrated into the input buffer module which was then tested through the implementation of the PS2 and Console APIs.

The APU module was tested by first manually entering operations into it by hand with constant value for the data input. Once this seemed to be working the APU API was developed and it was used to make sure that all of the functionality was working for the sine wave table. Once this was working it was extended to all of the different oscillator sources. The noise source was done somewhat independently since it was not a DDS unit but most of the functionality was the same as the other DDS units so very little was done differently for the noise source.

Software

Since the VGA was used to implement the interface to the APU it was very easy to debug the software using the GPU API that I developed for the last lab. Also because the output was audio it was possible to listen as well as use the oscilloscope to tell whether the correct functionality was being provided through the software. The software was rather minimal in this lab other than interfaces to the hardware. Since the hardware was first tested it was possible to find discrepancy between the software and hardware and using the VGA display it was very easy to debug this.

Results and Analysis

Results

The resulting system was one which could take user input to manipulate an APU that had 4 wave table oscillators and a noise source that could be added together to get a variety of interesting waveforms. The interface was very friendly except for a few input flaws in the tokenizer which resulted in some strange outputs which the program could not correctly identify. This was usually not a big problem so it was not addressed in this lab.

Analysis

The resulting wave forms were tested for their accuracy within frequency. The DDS units exhibited a negligible difference in the designated frequency and the actual frequency as shown on the oscilloscope. The noise source cutoff frequency was increased by 10 and an approximate 3 times increase in its amplitude was shown which meant that the low pass filter was correctly isolating lower frequencies as it was lowered.

Conclusion

This lab was a very interesting lab with the manipulation of multiple DDS units through a CPU. It very much reminded me of a synthesizer which could eventually be plugged into a MIDI based keyboard and then used to create very interesting sounds on the fly. Although I never got to the point of providing much musical sound modeling my project exhibited some extra functionality such as a keyboard interface and all wired through to a VGA display which could show the user the status of the system. Eventually I wish to return to this project and implement the Karplus-Strong algorithm as well as ADSR envelopes and filters but since this was not crucial for this lab this will have to wait until later.