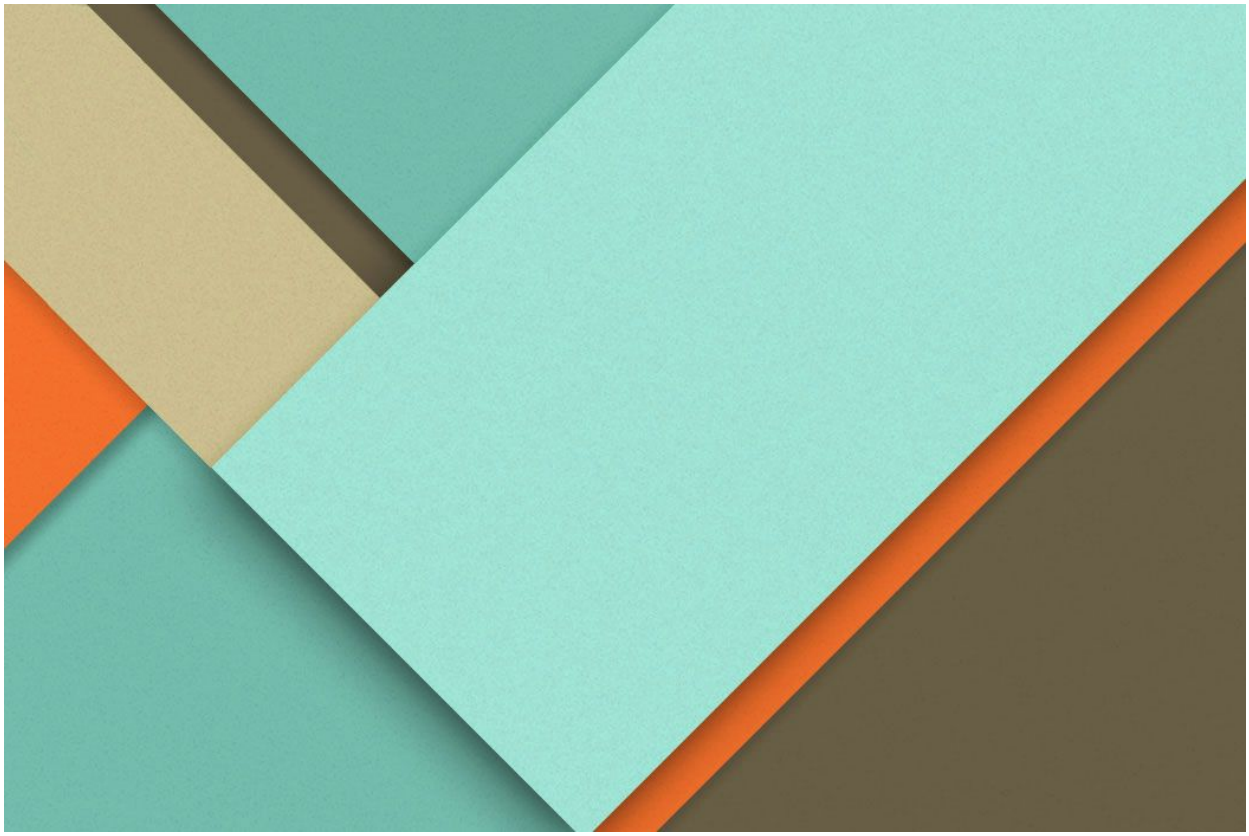

Space Invaders Lab 5 Report

10.29.2015

Yazan Halawa
Adam Hastings

Chapter 3: Game Audio

This Chapter will discuss the process with which we integrated sound into our Space Invaders game.

Section 3.1 WAV files and the AC97

This section will first discuss the way we got the game sounds in WAV format and converted them to data we can read and use in our program. After that we will discuss how the AC97 sound chip works and how we used it in this lab. Finally it will discuss which events produce sounds in our game and it all works integrated.

SubSection 3.1.a: WAV File Conversion

Looking online, we managed to find the game sounds in files of WAV format. However, those files were not readable to us and were impossible to use in our game. To get around this we use a python script, implementing the WAVE library, to parse those files and convert them to .c files that we can read and use in our game. The data we managed to extract from the WAV files was the the actual data of the sound wave in an array, the rate at which to play the sound (which was the same for all files), and the number of frames producing each sound. Once we acquired those, it was just a matter of writing those data from the array to the sound chip at the corresponding rate.

SubSection 3.1.b: AC97 Operation

The AC97 is an audio codec standard developed by Intel Architecture labs in 1997. It was the main piece of hardware that we used to enable sound with our project. It is essentially a FIFO that was given sound data values. We would write only 128 bytes at a time to prevent overflow. Once an entire sound was written to the FIFO, (and subsequently played through the speaker) we would clear the FIFO and the re-enable interrupts.

SubSection 3.1.c: Sound Triggering

There are several events that produce sound in the game:

- Aliens Marching

- Tank Firing bullets
- Tank being hit
- Alien being hit
- Saucer flying over
- Saucer being hit

The way we implemented is we had several flags that we would set whenever we want to output a sound. Then we would check for those flags inside the fifo interrupt. We check the flags using if statements ordered by priority. This means that if the aliens are marching, and a saucer flies over then the sound of the aliens marching is stopped and the sound of the saucer flying over starts because it has higher priority and is checked first in the if statements block. The events are ordered by priority like this:

1. Saucer being hit (Highest priority)
2. Saucer flying over
3. Tank being hit
4. Tank shooting
5. Alien dying
6. Aliens marching (Lowest priority)

Bug Report

We had issues whenever we call the clear FIFOs function. If the FIFOs were writing data at the point we cleared them then that would cause the buzzing to start happening. Once we rearranged them and called clear FIFOs at the right times, we had issues where the sound would not run after we clear the FIFOs. To solve this we re-enabled the FIFO interrupt each time after we clear FIFOs and that seemed to solve that problem.

Our next big issue was prioritizing sounds. This took us some time to figure out but once we used flags that seemed to solve half the problem. Our next issue was switching between sound and an another causes some weird clicking. To solve this we created a global flag which kind of works like a mutex. So we would not run any sounds until one sound is complete (that way we would not have to clear FIFOs or anything because that would just work). Obviously we created one exception of that for the Saucer hit sound because it actually has to stop the Saucer sound.

```
/*****************************************************************************
* interrupts.c
*****************************************************************************/



// This sets the flags of all sound data to 0, with the exception of the saucerHighFlag.
// This prevents other sounds from being written while another sound is being played,
// which in turn prevents FIFO overflow. Note that the saucerHighFlag is not cleared
// because it is of highest priority.
void clearAllSounds(){
		setExplosionFlag(0);
		setMarchingFlag(0);
		setMarchingFlag2(0);
		setMarchingFlag3(0);
		setMarchingFlag4(0);
		setSaucerFlag(0);
		setInvaderKilledFlag(0);
		setPingFlag(0);
}

// Global variables used in the fifo_writer function.
// They hold the necessary data to iterate through an
// array of sound data.
int* pointInArray;
int indexInArray = 0;

// This function writes data to the XAC97.
void fifo_writer(int* array, int frameNums){
		int diff;
		if (indexInArray == 0){				// This means that the function will begin playing a new sound
			pointInArray = array;			// The pointInArray iterator is set to the beginning of the sound array
			setSwitchContext(false);		// Disable the ability to assign new sound flags
		}
		if (indexInArray + XAC97_FIFO_BUFFER_SIZE >= frameNums){
// This means we've reached the end of a sound data array
			diff = indexInArray + XAC97_FIFO_BUFFER_SIZE - frameNums;
// This is the number of array locations to the end of the array
			XAC97_PlayAudio(XPAR_AXI_AC97_0_BASEADDR, pointInArray, pointInArray+diff);		// Play the audio
			pointInArray = array;
								// Reset the point in the array
			indexInArray = 0;
								// Reset the index in the array
			setSwitchContext(true);
								// Allow other sound flags to be set
			clearAllSounds();
								// Clear all sounds (except for the saucerHighPitch)
			XAC97_ClearFifos(XPAR_AXI_AC97_0_BASEADDR);
				// Clear the FIFO of all data
			XAC97_mSetControl(XPAR_AXI_AC97_0_BASEADDR, AC97_ENABLE_IN_FIFO_INTERRUPT);		// Re-enable interrupts
		} else {
			XAC97_PlayAudio(XPAR_AXI_AC97_0_BASEADDR, pointInArray, pointInArray+XAC97_FIFO_BUFFER_SIZE);// This means we're in the
middle of a sound data array
```

```
                pointInArray += XAC97_FIFO_BUFFER_SIZE;
                                                // Increment the pointInArray by how much was written to FIFO
                indexInArray += XAC97_FIFO_BUFFER_SIZE;
                                                // Increment the indexInArray by how much was written to FIFO
        }
}

// Global variables used in the fifo_writer2 function.
// They hold the necessary data to iterate through an
// array of sound data.
int* pointInArray2;
int indexInArray2 = 0;

// This function writes data to the XAC97.
// This function is only called by the saucerHighPitch
// It's only difference with fifo_writer() is that it disables
// the saucerHighFlag. This ensures that the saucerHighPitch
// will not loop indefinitely and allows other sounds to play.
void fifo_writer2(int* array, int frameNums){
        int diff;
        if (indexInArray2 == 0){                        // This means that the function will begin playing a new sound
                pointInArray2 = array;                          // The pointInArray2 iterator is set to the beginning of the sound array
                setSwitchContext(false);                // Disable the ability to assign new sound flags
        }
        if (indexInArray2 + XAC97_FIFO_BUFFER_SIZE >= frameNums){
// This means we've reached the end of a sound data array
                diff = indexInArray2 + XAC97_FIFO_BUFFER_SIZE - frameNums;
// This is the number of array locations to the end of the array
                XAC97_PlayAudio(XPAR_AXI_AC97_0_BASEADDR, pointInArray2, pointInArray2+diff);           // Play the audio
                pointInArray2 = array;
                                                // Reset the point in the array
                indexInArray2 = 0;
                                                // Reset the index in the array
                setSwitchContext(true);
                                                // Allow other sound flags to be set
                clearAllSounds();
                                                // Clear all sounds (except for the saucerHighPitch)
                setSaucerHighFlag(0);
                                                // Clear the saucerHighPitchSound
                XAC97_ClearFifos(XPAR_AXI_AC97_0_BASEADDR);
                        // Clear the FIFO of all data
                XAC97_mSetControl(XPAR_AXI_AC97_0_BASEADDR, AC97_ENABLE_IN_FIFO_INTERRUPT);             // Re-enable interrupts
        } else {
                XAC97_PlayAudio(XPAR_AXI_AC97_0_BASEADDR, pointInArray2, pointInArray2+XAC97_FIFO_BUFFER_SIZE); // This means we're in
the middle of a sound data array
                pointInArray2 += XAC97_FIFO_BUFFER_SIZE;
                                                // Increment the pointInArray2 by how much was written to FIFO
                indexInArray2 += XAC97_FIFO_BUFFER_SIZE;
                                                // Increment the indexInArray2 by how much was written to FIFO
        }
}


// FIFO Interrupt Handler which runs the sound through the XAC 97 sound chip
```

```
// It will check each flag in descending prioriy until it finds a raised flag.
// If a flag is found, then it will write the corresponding sound data to the
// XAC97 FIFO. All flags will use the fifo_writer() function, with the sole
// exception of the saucerHighFlag, which uses the fifo_writer2() function.
// Reasons for this are explained in the fifo_writer2() function.
void fifo_interrupt_handler(){
		if (getSaucerHighFlag()) {
			fifo_writer2(ufoHighSoundData, ufoHighSoundFrames);
		}
		else if (getSaucerFlag()){
			fifo_writer(ufoSoundData, ufoSoundFrames);
		}
		else if (getExplosionFlag()){
			fifo_writer(explosionSoundData, explosionSoundFrames);
		}
		else if (getInvaderKilledFlag()){
			fifo_writer(invaderkilledSoundData, invaderkilledSoundFrames);
		}
		else if (getPingFlag()){
			fifo_writer(shootSoundData, shootSoundFrames);
		}
		else if (getMarchingFlag()){
			fifo_writer(fastinvader1SoundData, fastinvader1SoundFrames);
		}
		else if (getMarchingFlag2()){
			fifo_writer(fastinvader2Data, fastinvader2Frames);
		}
		else if (getMarchingFlag3()){
			fifo_writer(fastinvader3Data, fastinvader3Frames);
		}
		else if (getMarchingFlag4()){
			fifo_writer(fastinvader4Data, fastinvader4Frames);
		}
}

// Main interrupt handler, queries the interrupt controller to see what peripheral
// fired the interrupt and then dispatches the corresponding interrupt handler.
// This routine acks the interrupt at the controller level but the peripheral
// interrupt must be ack'd by the dispatched interrupt handler.
void interrupt_handler_dispatcher(void* ptr) {
		int intc_status = XIntc_GetIntrStatus(XPAR_INTC_0_BASEADDR);
		// Check the FIT interrupt first.
		if (intc_status & XPAR_FIT_TIMER_0_INTERRUPT_MASK){
			XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_FIT_TIMER_0_INTERRUPT_MASK);
			timer_interrupt_handler();
		}
		// Check the push buttons.
		if (intc_status & XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK){
			XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK);
			pb_interrupt_handler();
		}

		// Check the FIFO-IN in the XAC97
		if (intc_status & XPAR_AXI_AC97_0_INTERRUPT_MASK){
```

```
                XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_AXI_AC97_0_INTERRUPT_MASK);
                fifo_interrupt_handler();
        }
}

/*****************************************************************************
* globals.c
*****************************************************************************/

// Macros for numbers
#define MAX_MARCHING_STAGE 4
#define VOL_MAX 3
#define VOL_MUTE 0

// Flags used for the XAC97 Sounds.
// When a flag is raised, the corresponding
// sound data will be written to the XAC97 FIFO.
static bool marchingFlag = false;
static bool marchingFlag2 = false;
static bool marchingFlag3 = false;
static bool marchingFlag4 = false;
static int marchingStage = MAX_MARCHING_STAGE;
static bool saucerFlag = false;
static bool saucerHighFlag = false;
static bool explosionFlag = false;
static bool pingFlag = false;
static bool invaderKilledFlag = false;

static int volLevel = VOL_MAX;                          // Volume level of the sound.
static bool switchContext = true;       // Bool that allows flags to be raised when true.


// switchContext getter
bool canSwitchContext(){
        return switchContext;
}

// switchContext setter
void setSwitchContext(bool newVal){
        switchContext = newVal;
}

// volLevel getter
int getVolLevel(){
        return volLevel;
}

// volLevel setter
void setVolLevel(int newVal){
        if (newVal > VOL_MAX)
                newVal = VOL_MAX;
        if (newVal <= VOL_MUTE)
                newVal = VOL_MUTE;
        volLevel = newVal;
```

```
}

// marchinFlag getters and setters
bool getMarchingFlag(){
        return marchingFlag;
}

void setMarchingFlag(bool newVal){
        if (canSwitchContext())
                marchingFlag = newVal;
}

// marchingFlag2 getters and setters
bool getMarchingFlag2(){
        return marchingFlag2;
}

void setMarchingFlag2(bool newVal){
        if (canSwitchContext())
                marchingFlag2 = newVal;
}

// marchingFlag3 getters and setters
bool getMarchingFlag3(){
        return marchingFlag3;
}

void setMarchingFlag3(bool newVal){
        if (canSwitchContext())
                marchingFlag3 = newVal;
}

// marchingFlag4 getters and setters
bool getMarchingFlag4(){
        return marchingFlag4;
}

void setMarchingFlag4(bool newVal){
        if (canSwitchContext())
                marchingFlag4 = newVal;
}

// saucerFlag getters and setters
bool getSaucerFlag(){
        return saucerFlag;
}

void setSaucerFlag(bool newVal){
        if (canSwitchContext())
                saucerFlag = newVal;
}

// saucerHighFlag getters and setters
bool getSaucerHighFlag(){
```

```
		return saucerHighFlag;
}

void setSaucerHighFlag(bool newVal){
		saucerHighFlag = newVal;
}

// explosionFlag getters and setters
bool getExplosionFlag(){
		return explosionFlag;
}

void setExplosionFlag(bool newVal){
		if (canSwitchContext())
				explosionFlag = newVal;
}

// pingFlag getters and setters
bool getPingFlag(){
		return pingFlag;
}

void setPingFlag(bool newVal){
		if (canSwitchContext())
				pingFlag = newVal;
}

// invaderKilledFlag getters and setters
bool getInvaderKilledFlag(){
		return invaderKilledFlag;
}

void setInvaderKilledFlag(bool newVal){
		if (canSwitchContext())
				invaderKilledFlag = newVal;
}
```