Real-time LED Music Visualizer

Jisoo Kim, Jiewen Sun, Pierre Karashchuk, Baihong Jin

***Abstract*—FlexPRET [9] is a processor platform for mixed-criticality systems, which can provide accurate timing control guarantees at the architecture level. In this course project, we utilize FlexPRET as a Real-time Unit (RTU) and build a real-time LED music visualizer that can drive seven LED strips and output audio signal synchronously.**

I. Introduction

An emerging research trend in real-time embedded systems is executing multiple software tasks with mixed criticality on a single hardware platform concurrently, while still ensuring the timing guarantees. FlexPRET [9] is a processor platform for mixed-criticality systems.

In this project, we aim to build a real-time LED music visualizer, which requires precise timing control on multiple tasks. Due to the lack of timing guarantees provided by conventional processors, achieving this on a conventional processor is not easy, which may involve using complicated hardware mechanisms such as timed interrupts and timing libraries written by professional people. Different from conventional processors, FlexPRET is built to enable accurate and precise timing control at the architecture level. Moreover, the compiler for FlexPRET architecture provides user-friendly interfaces, so that programmers can control timing in an easy and explicit way. As a result, we chose FlexPRET as the platform to implement a real-time led music visualizer.

II. Problem Definition

The desired functions of our system include generating sound by toggling a GPIO pin or driving a MIDI, and driving 7 LED strips that represent different notes. In particular, When the music is playing, the corresponding LED strip will be glowing with the music. They need to be sychronized very well.

III. Background

A. myRIO

We choose myRIO-1950 [4] as our hardware platform to implement the music visualizer. The National Instruments myRIO-1950 is an embedded microcontroller with multiprocessor architecture. The reconfigurable processor on myRIO is the Xilinx Artix-7 field-programmable gate array (FPGA).

We want to express our sincere appreciation towards our Professors Edward Lee and Alberto Sangiovanni-Vincentelli and our mentor Michael Zimmer for their valuable guidance in this project.

B. FlexPRET

FlexPRET is a 32-bit, 5-stage, fine-grained multithreaded processor with software-controlled, flexible thread scheduling, designed for mixed-criticality systems. [9] It uses a classical RISC 5-stage pipeline. Zimmer *et al* extended the RISC-V ISA to support timing instructions that enables more user-friendly timing control instructions in high-level languages such as C.

Under fine-grained multithreading, the processor switches between different hardware threads on each clock cycle. In single-threaded processors, a *context switch* is needed to switch between different tasks while maintaining spatial isolation. To achieve fine-grained multithreading, FlexPRET was built with extra hardware resources to allow each thread to maintain its own state. Using fine-grained multithreading with flexible scheduling and timing instructions, it allows each task to make a trade-off between hardware-based isolation and efficient processor utilization.

Several Berkeley researchers have developed a prototype RTU based on the open-source Berkeley RISC-V architecture and realized it as a soft core [8] on the myRIO platform. The processor is written in Chisel, which generates both Verilog code and C++ simulator for various configurations.

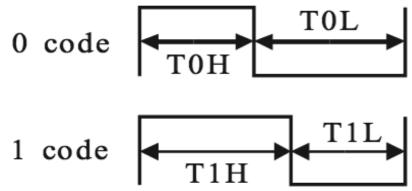
Figure 1 is a piece of example code showing the software-controlled thread scheduling. Under the hard active scheduling configuration, three threads are executed in a fixed order. The interval between the execution of consecutive instructions in the same thread is three clock cycles.

```
// Run sync_comm() and duty_comm() on different hardware threads.

#include "flexpret_threads.h"
#include "flexpret_timing.h"
#include "flexpret_io.h"

#define PERIOD 10000
#define HIGH1 7500
#define HIGH0 2500

void sync_comm()
{
}

void duty_comm()
{
}

int main() {
    hwthread_start(1, sync_comm, NULL);
    hwthread_start(2, duty_comm, NULL);
    // Round Robin
    set_slots(SLOT_T0, SLOT_T1, SLOT_T2, SLOT_D,
        SLOT_D, SLOT_D, SLOT_D,SLOT_D);
    // All Hard + Active
    set_tmodes_4(TMODE_HZ, TMODE_HA, TMODE_HA, TMODE_HA);
    while((hwthread_done(1) & hwthread_done(2)) == 0);
    return 1;
}
```

Fig. 1. An example code showing the software-comtrolled thread scheduling

C. NeoPixels

NeoPixel [1] is Adafruits brand for individually addressable RGB color strips. It is based on the WS2812 driver [6], using a single-wire control protocol. The signal is self-clocked, in which high and low bits are differentiated using different duty cycles of a square wave at a fixed frequency of 880Hz.



Data transfer time (TH+TL=1.25μs±300ns)

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	

Fig. 2. The timing specification of NeoPixels

D. MIDI

Musical Instrument Digital Interface (MIDI) is a technical standard that describes a protocol and a digital interface, which allows a wide variety of electronic musical instruments and devices to connect and communicate with one another. [3]

IV. Modeling and Analysis

A. Timing Analysis

Since the Flexpret architecture gives us strong guarantees about timing and scheduling, we are able to probe the feasibility of implementing the Neopixel driver under various circumstances. In particular, we're interested in how precisely we can control the timing under different clock frequencies and for different numbers of threads running concurrently.

To do this, we'll analyze the `delay_until` instruction from FlexPRET semantics [9]. This instruction delays the execution of a thread for some amount of time. How precise can we make this delay? That is, if we specify a delay, how long will it really take until the next instruction is executed?

Suppose we have N threads and a clock period of P. The answer lies in FlexPRET's interleaving schedule and pipeline. Once the delay has passed, the thread is scheduled again. It will take between 1 and N cycles until the next instruction is fetched, and 3 more cycles until it is executed (see Figure 3).

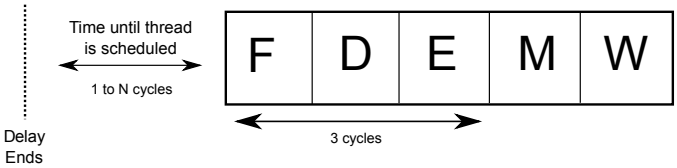


Fig. 3. Analysis of `delay_until` instruction

In total, the instruction will be executed between 4 and $N+3$ cycles after the delay ends. If we set the delay to be 4 cycles shorter than we want, the interval becomes 0 to $N-1$ cycles. If we want to further maximize the precision, we can set the delay to be even shorter by $\lfloor (N-1)/2 \rfloor$ cycles. Now the next instruction will execute between $\lfloor (N-1)/2 \rfloor$ cycles before the expected delay and $\lceil (N-1)/2 \rceil$ cycles after, giving us a final precision of $\lceil (N-1)/2 \rceil$ cycles. In terms of time, this is:

$$\text{precision} = P\lceil (N-1)/2 \rceil$$

.

Given our clock period of $P = 10$ ns and desired precision of 150ns (to match Neopixel specification [6]), we could in principle run up to 31 concurrent threads that all control LEDs!

In fact, we are only running 2 threads (one for sound, one for LEDs). Under these conditions, we can control our output up to a precision of 10ns, the clock period!

Certainly, the FlexPRET goes above and beyond in meeting our requirements.

V. Implementation

A. Workflow

Figure 4 shows the workflow of our project. To be more specific, we wrote our code in C, and compile it into `.elf` file using the extended RISC-V compiler. Then we can do simulation with the C++ simulator generated during the compilation. After that, we use LabVIEW FPGA to deploy the FlexPRET specification with our software code overlaid into the memory part in the `.bmm` file onto the FPGA on the myRIO board. We hook up the general output of FlexPRET to the Connector A on myRIO, which is connected to the NeoPixel LEDs, speaker or MIDI-to-USB cable.

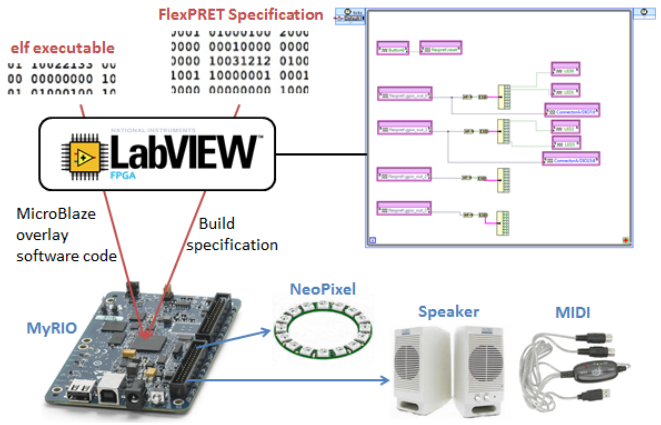


Fig. 4. Work flow of the system

B. Hardware

Figure 5 shows the hardware of our system. myRIO and speaker use external power source. Since the voltage output of general-purpose DIO is 3.3V, we also use the 3.3V power output on the board to power NeoPixel, which makes them compatible. Our LED strips are designed to be in the length of 16 LEDs, which is short, and 3.3V is enough to run it.

TABLE I
HARDWARE SPECIFICATION

Hardware Platform	myRIO-1950(Xilinx Zynq Z-7010[7])
LED Strip	Adafruit NeoPixel Digital RGB - 60 LEDs / 1m [1]
Speaker	Altec Lansing ACS90 [2]

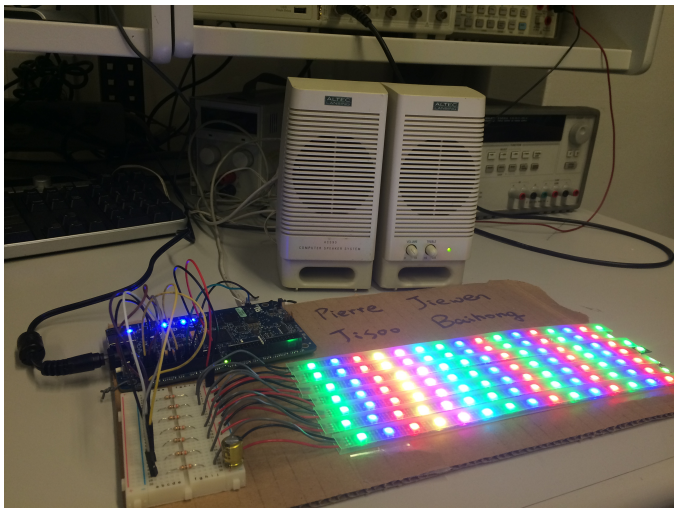


Fig. 5. Hardware of the system

C. Sound Driver

Our first approach to generating sound was to generate square waves with different frequencies by toggling a GPIO pin (GPIO 1), which was directly connected to speakers (Figure 5). This was an intuitive and only possible approach to produce sound directly from FlexPRET, since it was not feasible to process the sound on FlexPRET to make it nicer due to the lack of available memory and limited operations. Figure 6 shows the waveform of the sample sound wave.

In order to produce a square wave with a certain frequency, we used `periodic_delay` instruction from FlexPRET semantics, which delays the execution of a thread for certain period of time in nanoseconds. This instruction guarantees precise timing, so we were able to generate a square wave with a precise and consistent frequency.

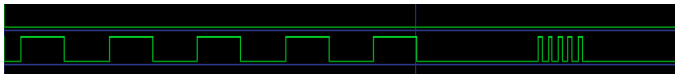


Fig. 6. Square wave with different frequencies

The downside of this approach was that the quality of the sound was not pleasing. To improve the quality of sound, we decided to implement a MIDI driver on FlexPRET so the sound signal can be transmitted to computer and synthesized. We followed MIDI 1.0 specification. We had to modify the sound generating part of our c program to send MIDI signal bits to GPIO 1 instead of toggling it.

To play or stop a note, we need to send a sequence of 3 bytes: the first byte speficies note on/off and MIDI channel, the second byte specifies the pitch, and the third byte speicifies the velocity, which usually gets translated into volume. Each byte is sent in the order of least significant bit first, and concatenated with start and stop bit, which are always 0 and 1 respectively. All the bits must be sent with the fixed rate of 31.25 kbit/s [3]. Since it requires a precise timing, we used `periodic_delay` instruction to acheive the bitrate. The last four bits of the first byte represents MIDI channel, which ranges from 1 to 16. MIDI channel 10 is reserved for only percussion instruments, so we chose to use channel 3 for our purpose.

For our song Fight For California, there was no harmonics, so we sequentially sent the set of 6 bytes for each note, where the first three bytes were for playing the note and the last three bytes were for stopping the note. We used `periodic_delay` between the two sets in order to play a note for a certain period of time.

Here is the example sequence of bytes in order to play note D4 on MIDI channel 3: `0x93` (note on channel 3) `0x62` (pitch D4) `0x3E` (with velocity 62) `0x83` (note off) `0x62 0x3E`. The waveform of the first byte is in Figure 7.

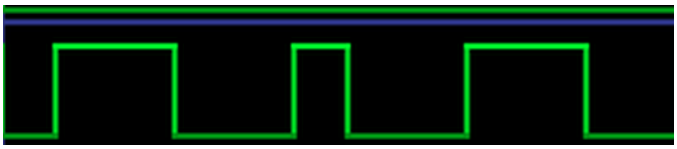


Fig. 7. Wave form of 0x93 (0 1100 1001 1) including framing bits, in the order of LSB

D. Pattern Generator

For our project, each song is represented as a sequence of notes and their durations (in second). Using our Python script, we convert the sequence into several arrays where each array contains information for notes and their duration in different format. These arrays are used in our main C program that generates the sound and drives the LED strips. For square wave generation, we used the arrays that represent each note in terms of its period (in nanosecond) divided by two, and the duration of each note in terms of number of cycles. For MIDI, we used the arrays that represent each note with corresponding byte representation and duration in terms of nanoseconds.

This method was employed because FlexPRET doesn't support multiplication and division yet, as well as floating point. Therefore, precalculation of everything was needed beforehand. The example format of a song pattern and the output arrays are in Figure 8.

E. Neopixel Driver

Since FlexPRET is a research architecture, we had to implement our own Neopixel library. The specification for the LEDs suggests that we need to drive the GPIO pin with 150ns precision [6], though empirical tests found that we can get away with 215ns of precision [5]. In order to show the reliability of FlexPRET, we opted to meet the 150ns precision.

As the timing analysis suggests, we used the `delay_until` instruction to meet the requirement. We would set the gpio pin high, then delay for some time,

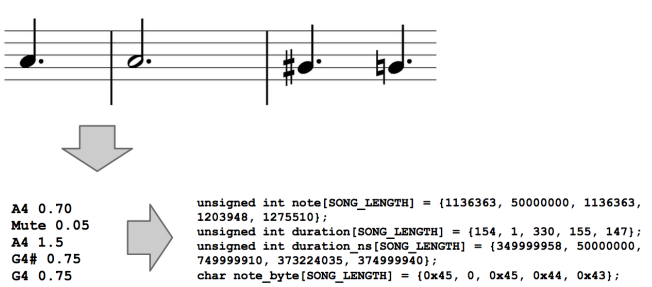


Fig. 8. Converting four notes to sequence, and then arrays

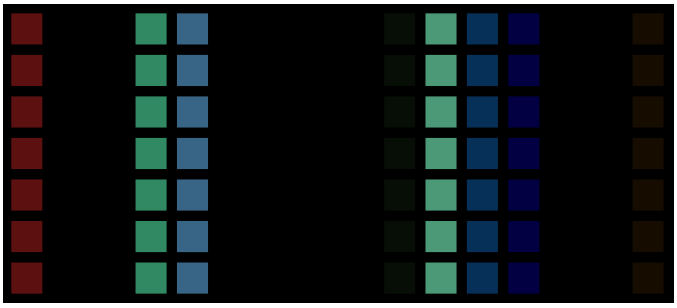


Fig. 9. LED Simulator built in pygame to test out patterns

depending on the bit (0 has smaller delay). Then, we would set the GPIO pin low, and delay again. Using the FlexPRET simulator [8], we found that this approach gave us cycle accurate timing with 1 thread as predicted, while being off by a few cycles for 2 threads.

F. LED & Music Synchronization

With the LED drivers written, we moved on to generating pleasing patterns on our small (7x16) LED matrix, synchronized to the music.

In order to perform the synchronization, we simply had the same delays between notes for LEDs and for the music. The semantics of FlexPRET guaranteed that the LEDs and the music would not become offset.

To test synchronization with the song, we first tried a simple pattern: light one LED strip at a time, corresponding to the note played.

We found that it indeed worked well when generating square waves and toggling LEDs. They were perfectly synchronized. However, when generating MIDI output, the synthesizer on the computer had a small delay, so we had to artifically introduce a startup delay to our LEDs.

G. LED Patterns

Once we got the basic patterns down, we moved on to generate more complex patterns, which would change between notes. In order to prototype these patterns more efficiently, we created a basic LED simulator using python and pygame (see Figure 9). Using this simulator, we found that identifying notes with columns instead of rows/strips gave better results. In addition, fading these columns out, instead of simply turning them off looked cooler. So we implemented these changes in the real architecture and got our final result.

VI. Summary

In this project, we used FlexPRET, an academic processor architecture, as a real-time unit to control NeoPixel LEDs and output square-wave/MIDI audio signals. FlexPRET, unlike conventional processors, utilizes the inherent timing accuracy from the hardware (cycles) and provides the users an easy-to-use interface for accurate timing control. Since FlexPRET offers reliable real-time guarantees, we analyzed its performance limit based on the timing constraints. The results prove that FlexPRET is capable of perform accurate concurrent timing control over multiple threads in an efficient way, and thus is very appropriate for timing-critical embedded system applications.

References

[1] Adafruit. *Adafruit NeoPixel Uberguide*.
[2] Altec Lansing. *Altec Lansing Computer Speaker System ACS-90*.
[3] MIDI Manufacturers Association et al. *The complete MIDI 1.0 detailed specification: incorporating all recommended practices*. MIDI Manufacturers Association, 1996.
[4] National Instruments, Austin, Texas. *USER GUIDE AND SPECIFICATIONS for NI myRIO-1950*.
[5] Tim. Light_ws2812 library v2.0 - part i: Understanding the WS2812. https://cpldcpu.wordpress.com/2014/01/14/light_ws2812-library-v2-0-part-i-understanding-the-ws2812/.
[6] Worldsemi. *WS2812 Datasheet*.
[7] Xilinx. *Zynq-7000 All Programmable SoC Overview*.
[8] Michael Zimmer. Flexpret. https://github.com/pretis/flexpret, 2014.
[9] Michael Zimmer, David Broman, Christopher Shaver, and Edward A Lee. Flexpret: A processor platform for mixed-criticality systems. Technical report, DTIC Document, 2013.