Third Prize

Nios II Soft Core-Based Full-Color LED Music Sight Light Control System

Institution: **Harbin University of Science & Technology**

Participants: **Zhong Qiubo, Gao Junfeng, and Liu Xiaoping**

Instructor: **Dong Huaiguo**

Design Introduction

Lighting sources have evolved beyond incandescent lamps. After the launch of the China Green Lights Program, new LED lighting products have attracted wide attention in the lighting and decoration industry with their energy savings, extended life, wide application, flexible control, brilliant color, and environmental efficiency. Our design is a musical landscape lamp control system with high-level simulation software, which transfers data and MP3 files to the control system through a compact flash (CF) card. Our design also includes a series of steps including lamp installation, layout, scenario data editing, simulation, preview, and so on. The product has applications in city beautification, lighting, and music integration in public places.

Controlled by a control panel or a dedicated computer, traditional landscape lamps feature only seven-color changes, in a simple way. The speed of the dedicated computer is limited: its hardware pulse width modulation (PWM) generally has only three to six paths, and can only be expanded with software modifications. Therefore, a traditional system cannot meet the requirements of high-speed data transfer. Additionally, traditional systems cannot display a smooth gradient and the jumps are noticeable with the naked eye. Therefore, we need a powerful processor to implement a soft gradient with jumps that cannot be seen by the naked eye. Considering the cost, we adopted one controller to handle several lamps. We also implemented a packet-control mechanism to handle a large number of lamps by communication between computers.

The embedded 32-bit Nios® II soft core processor helped us create a highly integrated landscape lamp control and MP3 playing system. The computing power of the Nios II processor enables simultaneous LED lamp operation based on different scenarios and music. Further, algorithms can be developed on PC using C and can be migrated to the Nios II processor, which shortens the development cycle of the whole system. Altera's SOPC Builder can help to create and deploy users' Nios II instructions, and add customized intellectual property (IP) cores to create a more powerful system-on-a-programmable-chip

(SOPC) system. Combined with the Cyclone™ FPGA, the designed product delivers a high price/performance and promises good market prospects.

Function Description

This section provides our design's functional description.

Major System Functions

Our LED musical landscape lamp control system controls 10 LED lamps (which can be increased in number, if needed). At least 256 color changes are realized through RGB color mixing and you can change five different parameters to achieve the desired effect: static, gradient, dim, bright, and flicker. You can change the duty cycle of the PWM to control a scene made up of changes to 10 lamps, form a scenario with several scenes, and then create an animation effect by playing these scenarios continuously. Simultaneously, you can play MP3 files, creating a dynamic scenario in which light changes with the rhythm of music.

System Components

The system includes a display unit, drive unit, control uni,t and data communications unit, which are controlled by the µC/OS real-time operating system. The control unit has three tasks:

- Read lamp control data and MP3 data from the computer to the CF card memory.
- Get and analyze data from memory, and send analyzed lamp control data to the LED lamp drive unit. Then, the multi-path PWM display unit implements the LED lamp scenarios.
- Send MP3 data to the decoder for decoding and play via serial peripheral interface (SPI) SPI bus.

Figure 1 shows the system hardware design diagram (see the "Design Architecture" section). We used the FS embedded file system for data management, based on the real-time and multi-tasking features of µC/OS real-time operating system (RTOS). The drive unit is a self-customized, full color lamp-control intellectual property (IP) core, each controlling a lamp via PWM circuit. We use the lamp-control data to display scenario changes through the PWM port. The timer provides 10-ms interrupts, after which scenario-data analysis is carried out. The module drawn in dashed lines can be modified. Several Nios II control systems can be used for control, based on multi-computer communications, when you need to handle a large number of landscape lamps. Then, only one control module needs MP3 functionality, and the other modules may not need it. This functionality can be programmed in the SOPC Builder tool to save development costs.

Display Unit

The tricolor LED chip is the core component of the display unit. The LED is the most widely used lamp in electronic components, and tests have proven that three basic colors (red, green, and blue) can be mixed in different combinations to obtain other colors.

Control Unit

In the control unit, the µC/OS RTOS runs tasks by means of semaphore. The lamp control task software flow chart is shown in Figure 4 in the "Design Architecture" section.

Display Drive Principle

Generally, there are two ways to control LED brightness: by changing the current flowing through LED or by controlling the on/off period of the LED by the PWM. Controlling the LED working current allows for a wider range of LED brightness control. However, current control is difficult to realize in software; therefore, it is unsuitable for digital control. In contrast, the PWM method is widely adopted in digital circuits because it can be implemented easily in software. According to Talbot's law:

$$\overline{L} = \frac{1}{T}\int_0^T L(t)dt$$

in which, $\overline{L}$ is the visual brightness of cyclic change sensed by the eyes and T is the cycle. When brightness function $L(t)$ is a constant L, the visual brightness changes into $\overline{L} = \frac{t}{T}$ L, when PWM actually controls the working time of the LED by changing the working time in a cycle periodically to change LED brightness.

Continuously changing the LED's working time in a cycle continuously changes the LED brightness and grey scale. Dividing cycle T by n equal periods results in n grey scales of LED. To ensure that the brightness transition is not perceived by human eyes when the LED grey scale changes (i.e., no flicker), the on-and-off frequency LED should be larger than critical frequency, and the cycle should not be longer than 0.1 - 0.2 s. Tests have shown that when the LED grey scale is 256, the mix of three basic colors will not create transitions, and human eyes can perceive the color gradient. The PWM cycle of this system is 2 ms and grey scale is 256. We can generate 256 colors controlling the three basic LED colors, and use fragment delay to control the duty cycle. The basic colors are mixed according to a certain brightness ratio, which is a certain grey scale. Different grey scales correspond with different duty cycles and different LED working time cycles.

Display Drive Unit

The display drive unit design for the 10 self-customized system peripherals is shown in Figure 2. The diagram features the design of a 30-path output port with 10 PWM controllers, respectively, for scenario changes of 10 LED lamps. The PWM circuit has two caches, back and front. The control arithmetic unit sends data that needs to be stored to the back cache of The PWM. The PWM checks whether the back cache has data to be updated, if not, it continues to read the PWM value from the front cache. We check the back cache each time playing finishes, and if data is updated, we move the data from the back cache to the front cache, and play the new data. If the PWM value is 255, the output waveform is at high logic level; if PWM value is 0, the output waveform is at low logic level. If the PWM is between 0 and 255, output is made according to the relevant duty cycle based on the fixed cycle. Figure 5 shows the software design flow.

Data Communications Unit

The data communications unit transfers lamp control data and MP3 music files from the computer to the control system via the CF card. If there are several control systems, multi-computer communications with an RS-485 serial port can be used.

Performance Parameters

The design's performance parameters are as follows:

- The Nios II frequency required by the system is 85 Hz and the peripheral PWM's cycle is 2 ms, which is divided into to 256 parts. This scheme enables the lamp control data display using interrupt data processing that is performed every 10 ms.
- The system relies on the µC/OS II RTOS to handle multiple tasks and makes it possible to simultaneously execute landscape lamp scenario display and MP3 music play operations.
- The landscape lamps can support a 256-color display and five operation modes: static, gradual bright, dark, change, and flicker.
- During a gradual change, jump phenomenon cannot be observed by the naked eye. Instead, multiple colors and gentle gradual change is displayed.
- Fluent and clear MP3 play.

A combination of the self-defined IP core and the Nios II processor greatly accelerates operation and processing. Also, using the Nios II soft core, you can set the cycle of PWM at 2 ms and enable simultaneous operations of MP3 play and landscape lamp scenario display.

Design Architecture

Figures 1 and 2 show the hardware design. Figures 3 through 7 show the project software flows.

Figure 1. Hardware Design

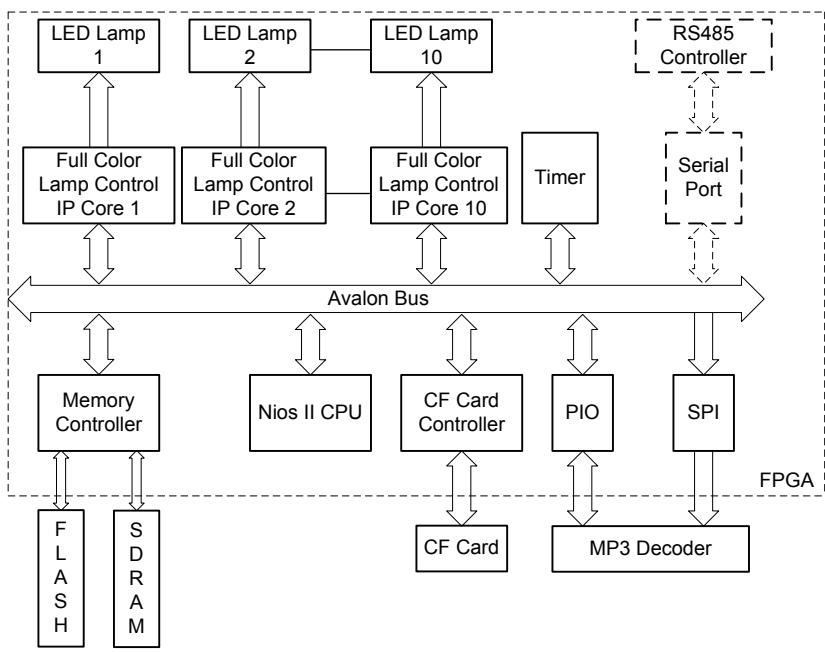


Figure 2. Full Color Lamp Control IP Core Hardware Design

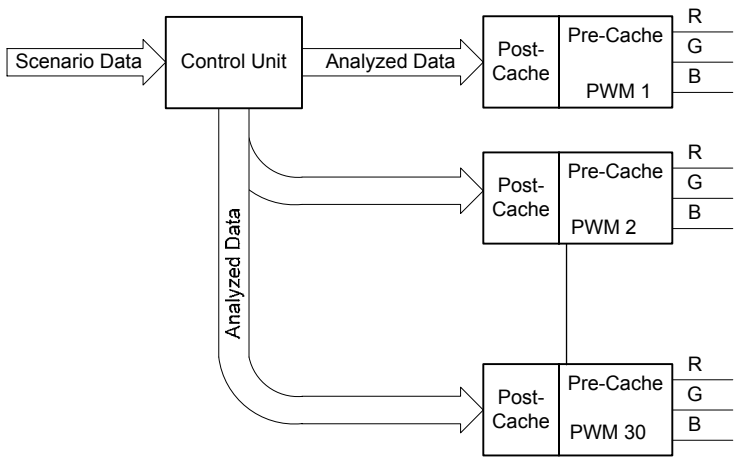


Figure 3. Timed Interruption Software Design Flow

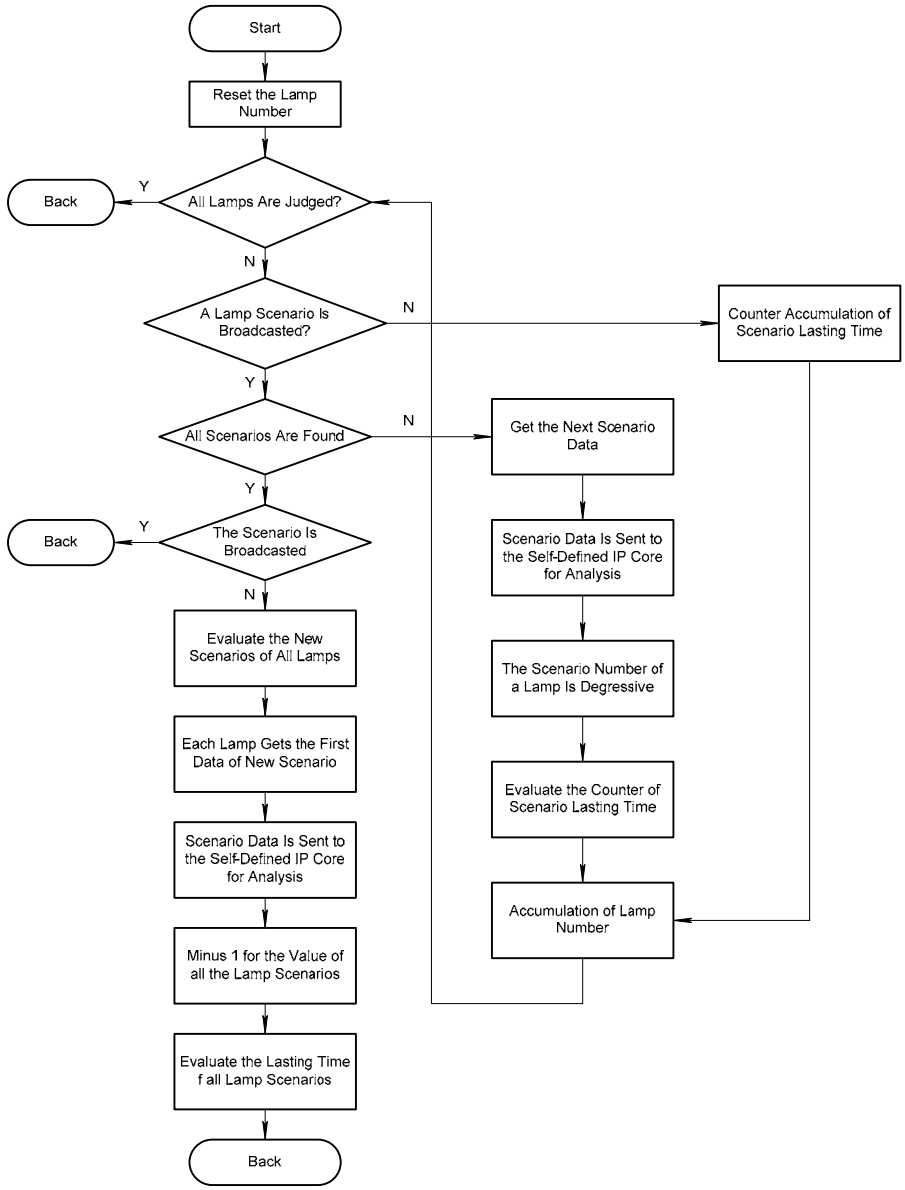


Figure 4. Software Design Flow Diagram

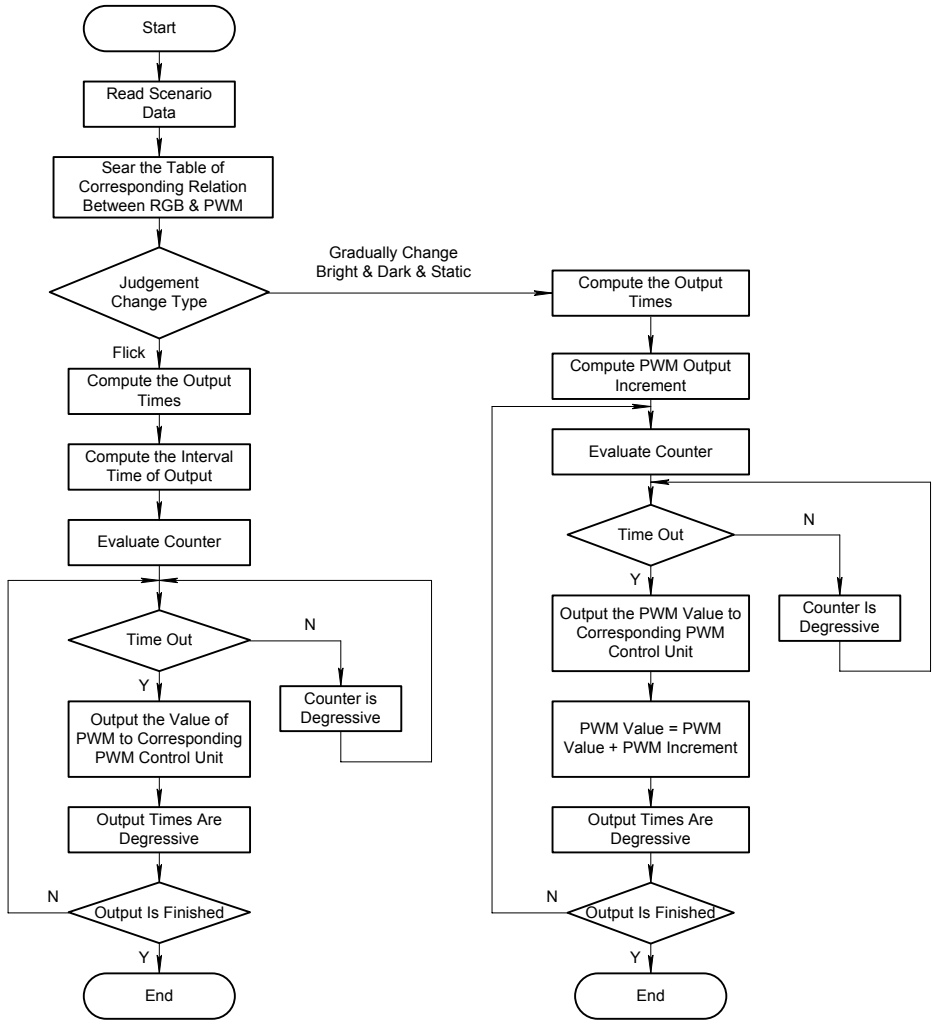


Figure 5. Peripheral PWM Software Design Flow Diagram

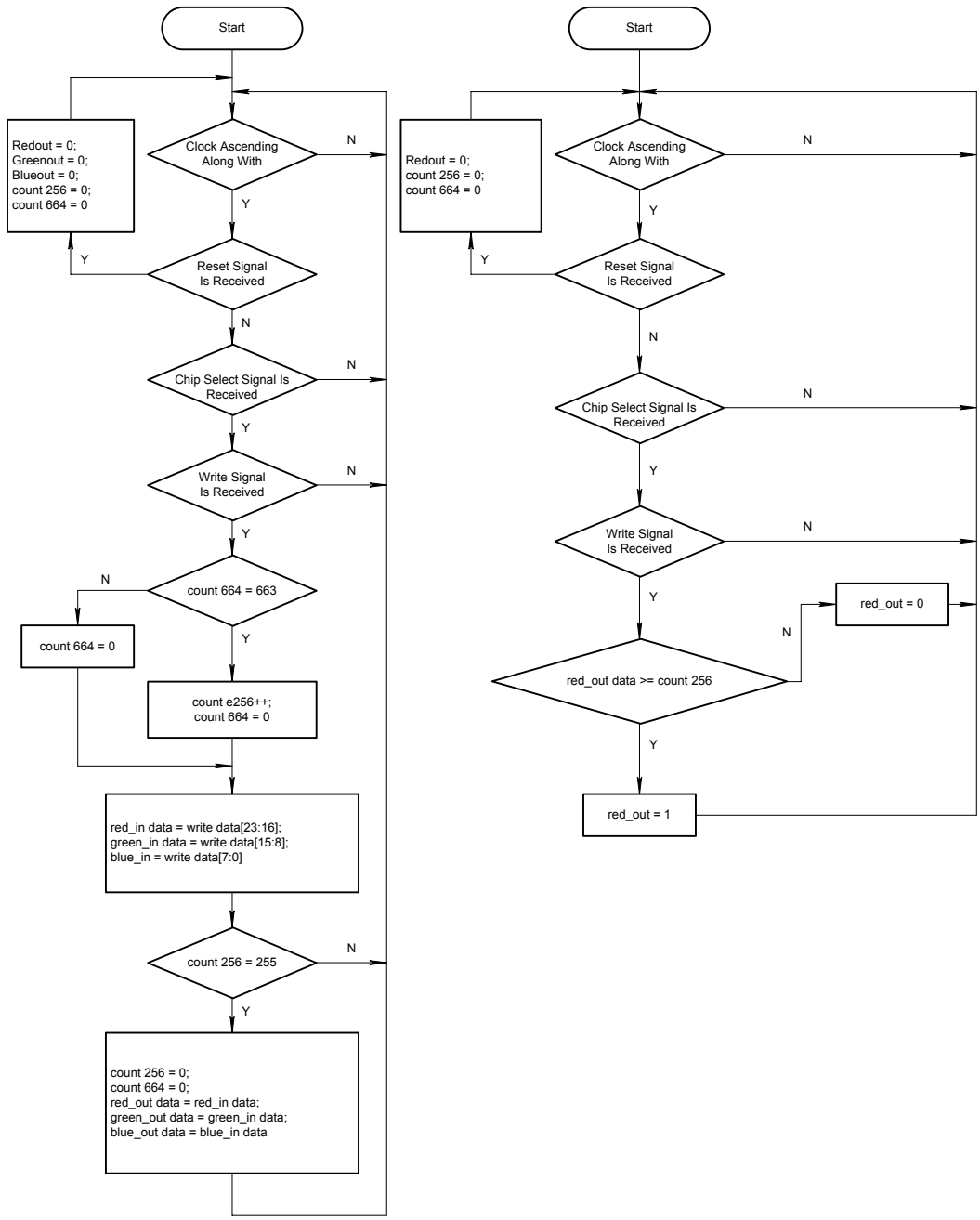


Figure 6. MP3 Design Flow Diagram

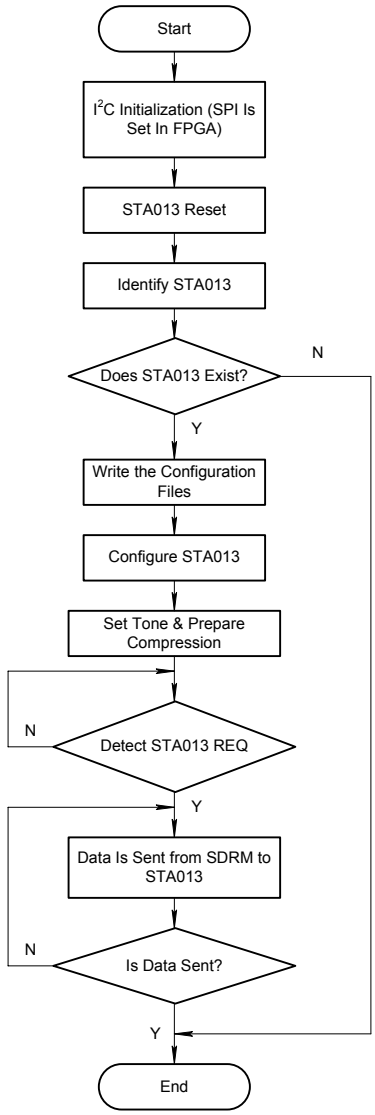
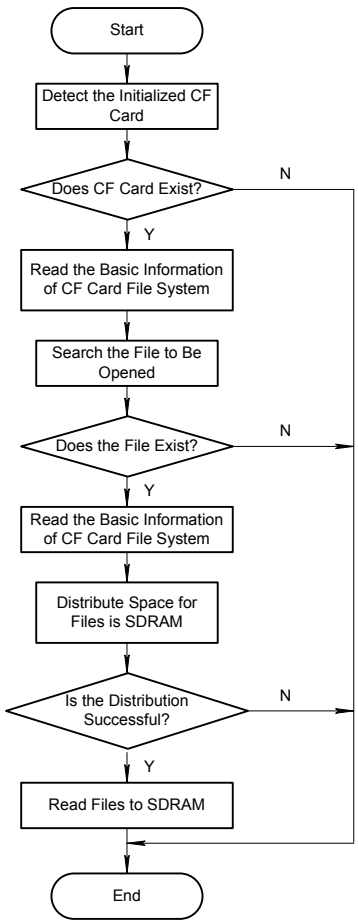


Figure 7. CF Card Read Software Design Diagram



Design Methodology

We adopted a design methodology that blends nicely with the self-defined peripherals option in the SOPC Builder tool. The system displays the landscape lamp scenario and plays MP3 files under the semaphore control mode of µC/OS-II RTOS in the Nios II integrated development environment (IDE) after download. Our design comprises two modules: hardware and software.

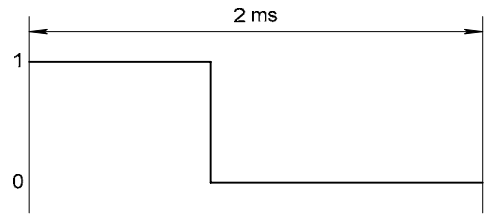
Hardware Design

We extended the system hardware by adding the STA013 MP3 decoder and D/A converter for playing MP3 files on the Altera® Cyclone II EP2C3. We implemented the lamp control on the UP3 development board to promote the application of Cyclone II FPGA, which is the most cost-effective device that offers the best price-performance ratio among competing devices. We implemented all system functions on the EP2C3 device. Additionally, we applied the ULN2803 power drive to control the voltages used in the LED lamp display (see the Appendix for the circuit schematic).

Full-Color Lamp Control IP Core Design

We used Verilog HDL to design the self-defined peripheral full-color lamp control IP core's control unit, which implements a 10-lamp, self-defined IP core controller with 30 PWM circuits. The software design flow is shown in Figure 5. The cycle of PWM is set to 2 ms, as shown in Figure 8.

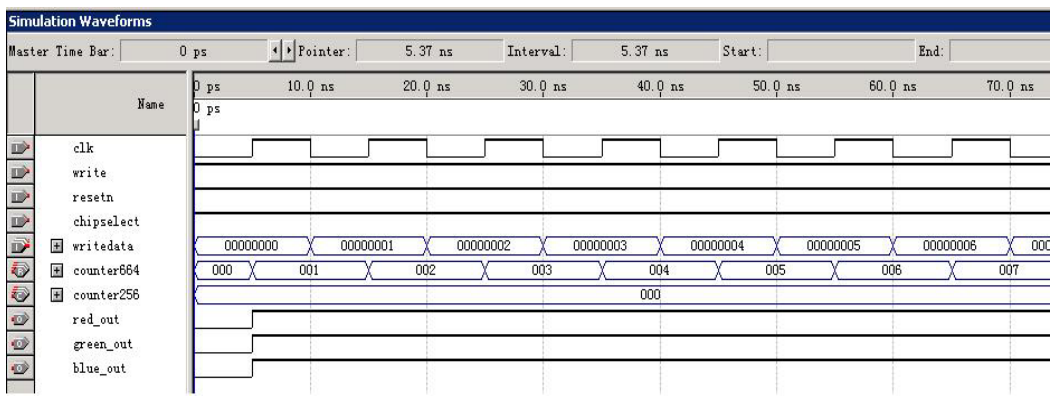
Figure 8. PWM Timing Diagram



Function Simulation

After PWM design, we carried out functional simulation as shown in Figure 9.

Figure 9. PWM Functional Simulation



The simulation variables in the oscilloscope display are described as follows:

- `clk`: clock signal.
- `resetn`: PWM reset signal.
- `chipselect`: PWM chip select signal.
- `write`: PWM write signal.
- `writedata`: data written to PWM.
- `red_out`: red corresponding output signal in PWM.
- `green_out`: green corresponding output signal in PWM.
- `blue_out`: blue corresponding output signal in PWM.

- `counter256`: one cycle is divided into 256 parts for computing the duty ratio of each PWM cycle.
- `counter664`: each part comprises 664 clock cycles for computing whether the count is over or not.

MP3 Design

We used the I²C bus to control the STA013 device. In this way, we were able to transfer MP3 data from SDRAM to STA013 through SPI, which is set using the SOPC Builder tool as shown in Figure 10. We added four PIO interfaces in SOPC Builder to connect with the SDA, SCL, DATA_REQ, and RESET pins. The PIO connected with SDA is set as a bidirectional port.

Figure 10. SPI Setting

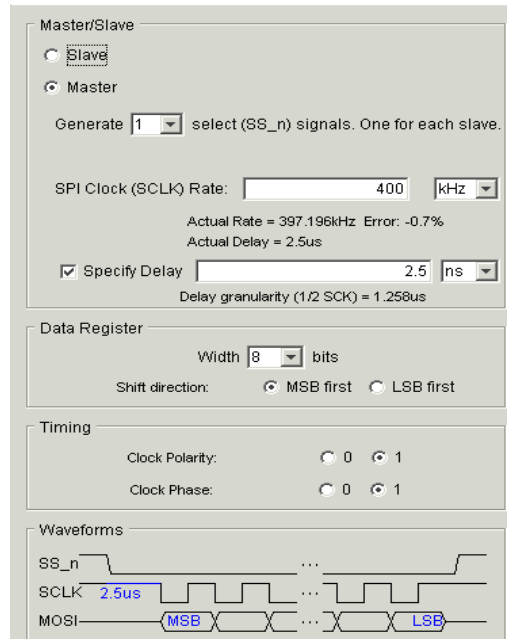
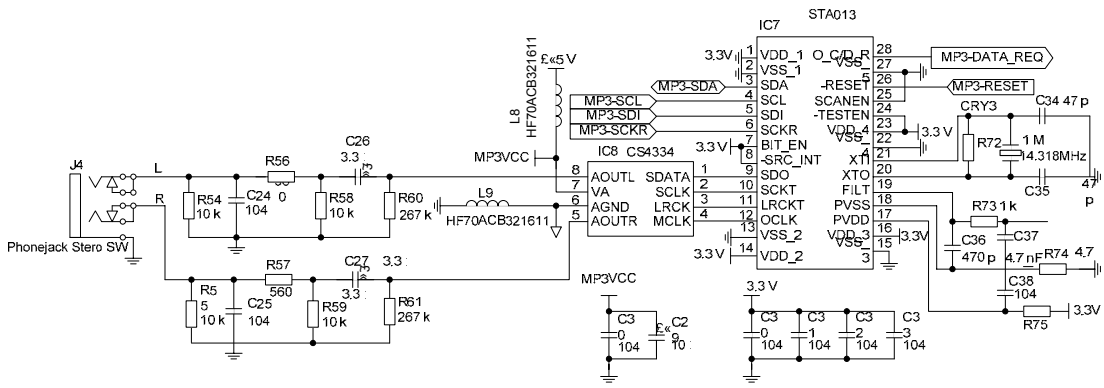


Figure 11 shows the MP3 decoding circuit.

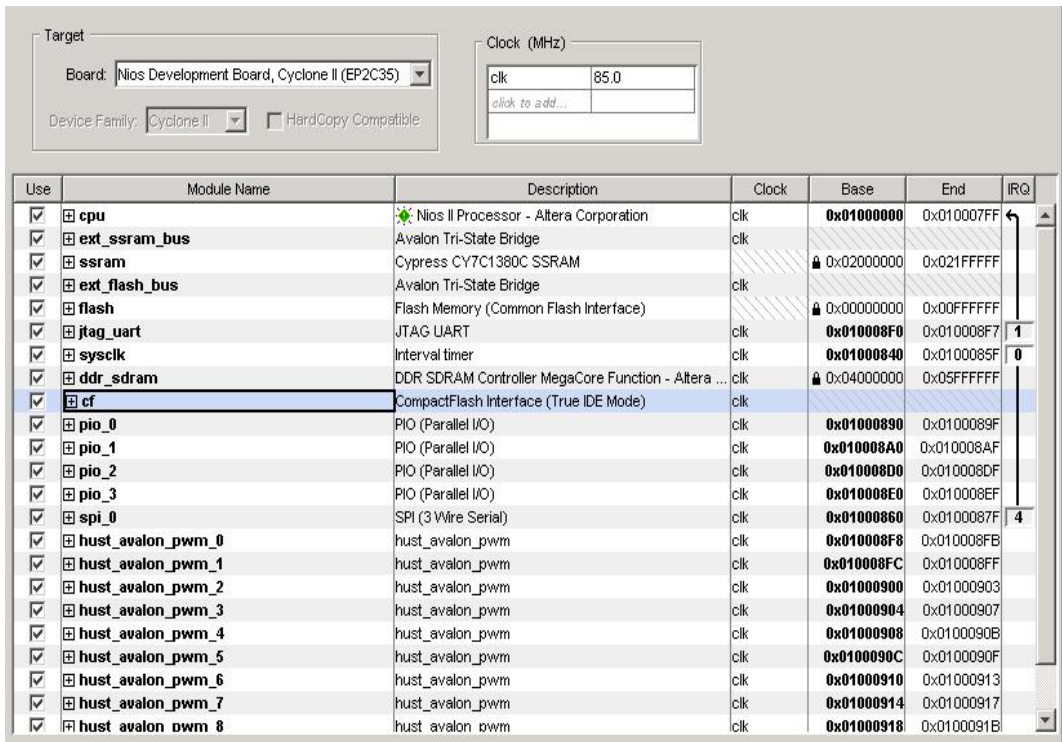
Figure 11. MP3 Decoding Circuit Schematic



SOPC Builder Configuration

After finishing the IP core design and simulating, we used the SOPC Builder tool to configure the whole system. The settings are shown in Figure 12.

Figure 12. SOPC Builder Settings



Compiler

After configuring all the required system parts, the SOPC Builder tool assigns pin definitions with the Quartus® II development tool and then compiles. See Figures 13 through 17, which show the Compiler output.

Figure 13. Compiler Analysis Report

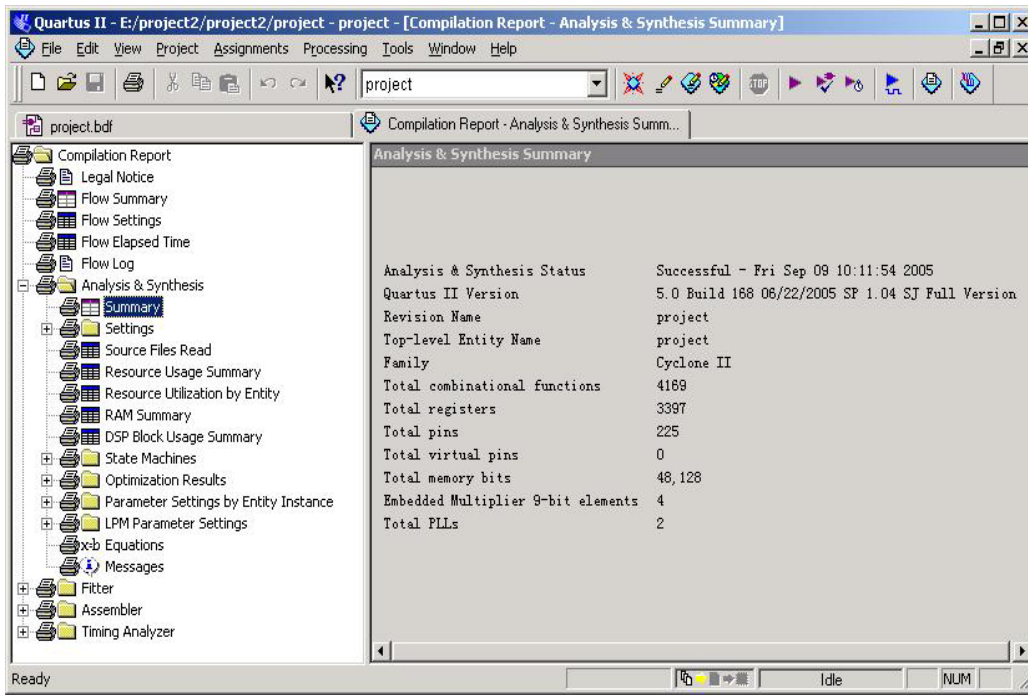


Figure 14. Assembler Report

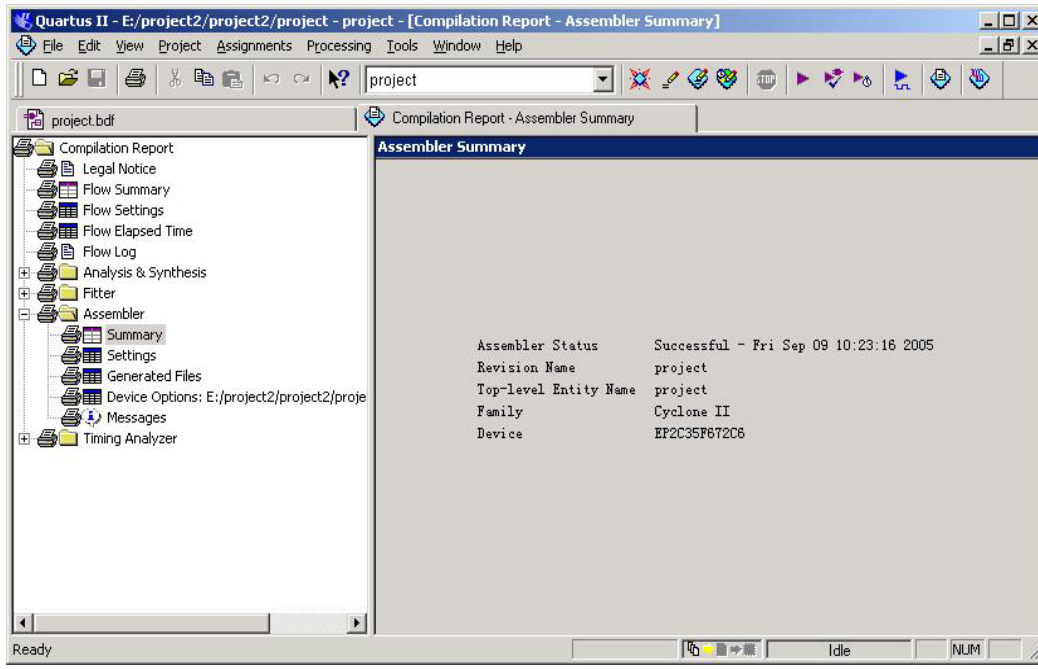


Figure 15. Fitter Report

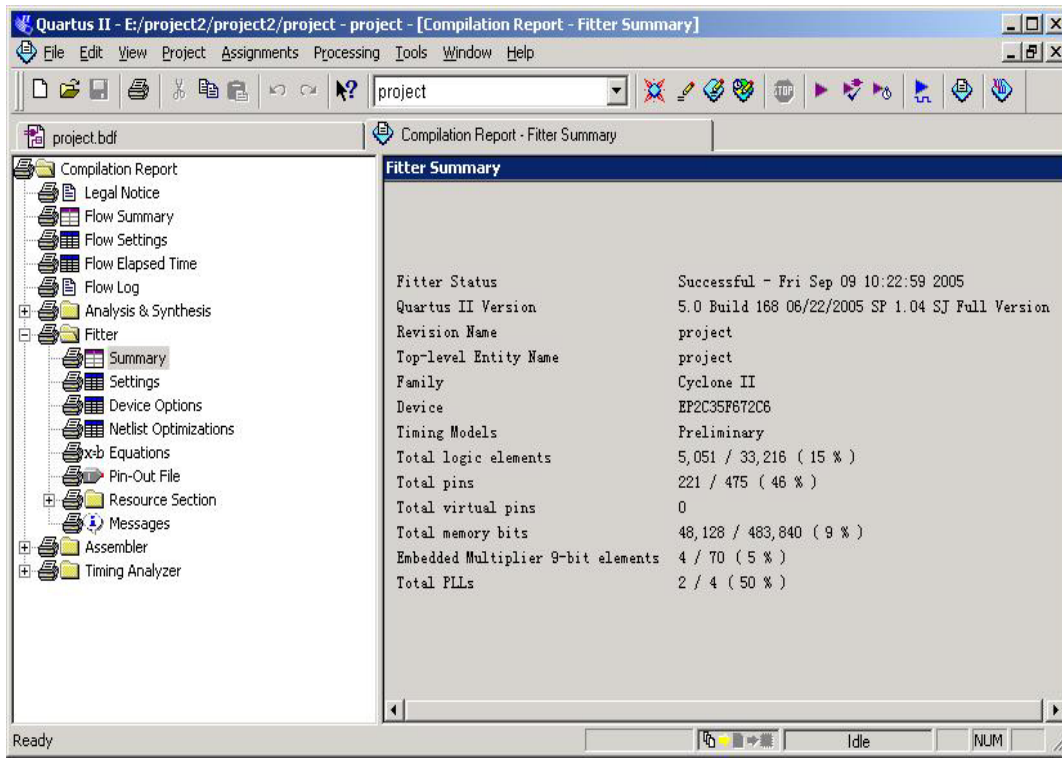


Figure 16. Flow Report

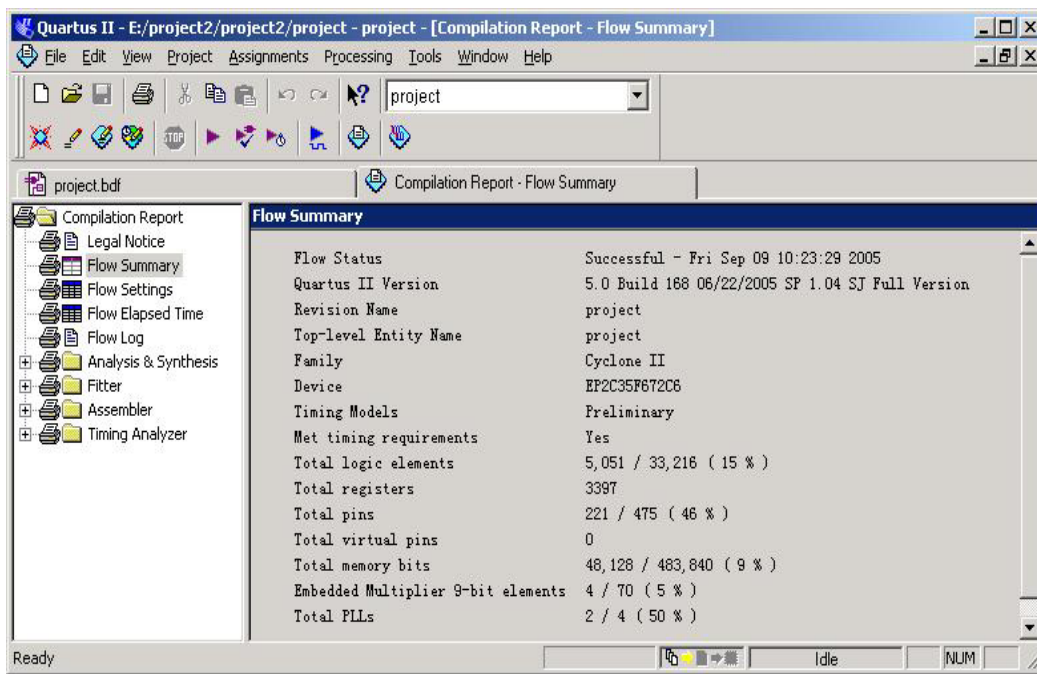
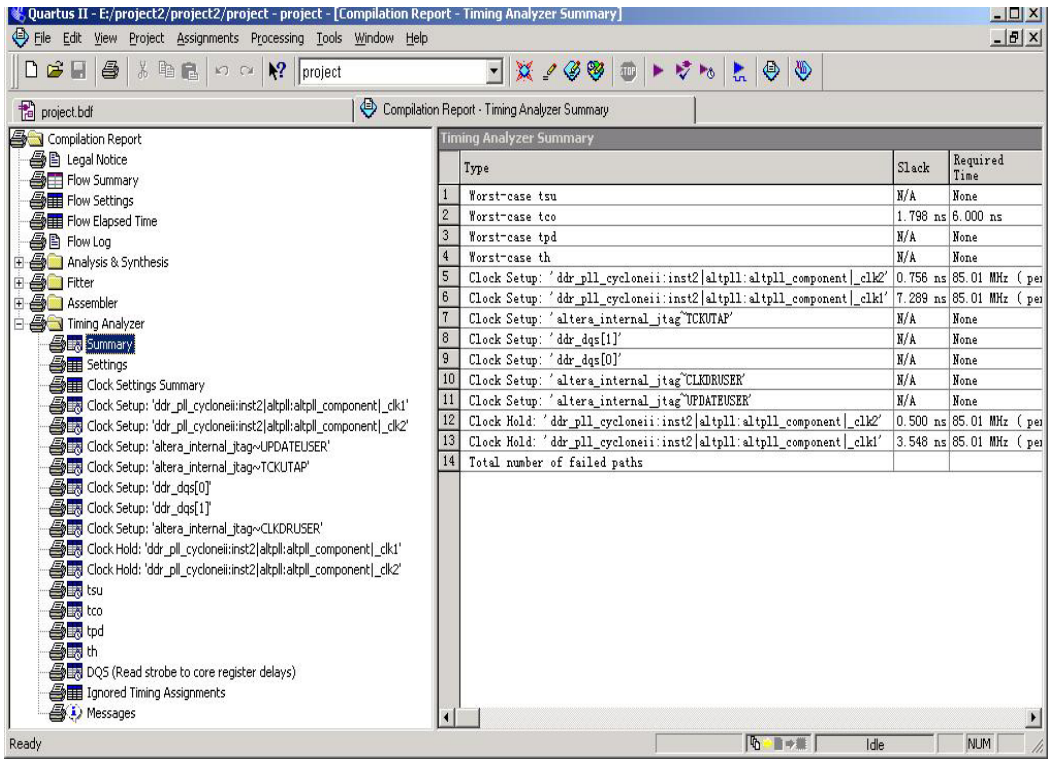


Figure 17. Timing Analyzer Report



Software Design

The basic design of the software module is to send out the play and data read tasks accurately and in a timely manner using the µC/OSII RTOS.

The first task is to read the CF card and send its data and MP3 files to SDRAM for the next two tasks. In our design, we used the Quartus II version 5.0 compact flash core as the interface between the CF card and the Nios II processor. This routine uses two pointers, *MP3data and *pwmdat, to assign space for data and MP3 files on the SDRAM. We designed a small file allocation table (FAT) file system for CF card reading. This system:

- Does not support long file names.
- Does not support the FAT12 file format.
- Sets data and MP3 files in the root directory of the CF card.
- Does not support writing into the CF card (the data in CF card can be written from by PC with reader/writer).

We defined three data structures: BPB, file directory entries, and FAT. The specific definitions are as follows:

```
typedef struct
{
    unsigned char  Type;          // file format type
    unsigned char  StartLBA;      //BPB start sector
```

```
    unsigned char  SectorsPerCluster;     //sectors per cluster
    unsigned char  LShift;              //shift number of SectorsPerCluster
    unsigned short  SectorsBeforeFAT;   // reserved FAT number
    unsigned char       FATs;               //FAT number
    unsigned short  FAT16RootEntries;    // number of root directory entries
    unsigned long  TotalSectors;          //total sector number
    unsigned short  SectorsPerFAT;      //sector number per FAT
    unsigned long  FAT32RootStartCluster; //start cluster of root directory when the
file format is FAT32

} FS_TBPB;
typedef struct
{
    unsigned long  StartLBA;   //start sector of file allocation table
    unsigned char  LShift;     // shift number of file format type
    unsigned long  DataStartLBA; //start sector of data area
} FS_TFAT;
typedef struct
{
    unsigned char  Attrib;   //file attributes
    unsigned long  StartCluster;   //file start cluster
    unsigned long  StartLBA;     //file start sector
    unsigned long  CurrentCluster;   //current cluster
    unsigned long  CurrentLBA;    //current LBA
    unsigned long  Offset;         //system reserved
    unsigned long  Length;        //file length
} FS_TFile;
```

Refer back to Figure 6 for a detailed software flow.

At system initialization, we invoke CF card initialization function IDE_initialize() to determine whether the CF card exists or not. If the CF card exists, we read the basic information of the FAT file system, such as the file format the CF card has adopted, start sector of root directory, and data area. We invoke the FS_SearchFile (char *FName, FS_TFile *R, unsigned char dir) function to search the file to be read and then assign a buffer for the file with a pointer. Because SDRAM has enough space, the file data can be totally read into SDRAM, which is the file size in SDRAM. One sector is read each time until all data is moved into SDRAM. The key to FAT file system design is to get data of the next cluster after reading the current one. In this design, we defined the function, FS_GetNextCluster(unsigned long Cluster). We read the whole cluster chain into an array when opening a file. Although this routine occupies some space on the SDRAM, the search of cluster in future will not read the FAT table. This is because the function slows down system speed.

The second task is to display the scenario file and to receive the scenario data of different lamps as well as search the PWM values R, G, B binary-coded according to Table 1. This task judges the changing modes, such as gradual change, bright, dark, and static, in the same control mode. The flicker mode is handled differently.

Formatting of scenario data comprises five bytes: the last byte indicates the address information and the first four bytes are shown as follows:

Front color	Back color	D15 D14 D13 D12 D11 D10 D9 D8	D7 D6 D5 D4 D3 D2 D1 D0

Front color and back color separately occupy one byte; D15 in the third byte is the marker bit of FLICK, following two situations that may occur in terms of D15's value:

- D15=0, gradual change, bright and dark as well as static, D14……..D0 indicate the lamp on lasting time.

- D15=1, flick mode, D14……..D9 indicate the flicker time, D8……D0 indicate the lamp flicker lasting time.

For gradual change, bright, and dark as well as static, the data increment PWM_D sent to PWM per cycle is computed using the following formula:

```
PWM_D =(PWM_BACK_COLOR - PWM_FRONT_COLOR)/(LASTING_TIME/10ms)
PWM_FRONT_COLOR  PWM value of front color
PWM_BACK_COLOR   PWM value of back color
LASTING_TIME       lasting time of scenario
```

10 ms is the cycle period time of the PWM.

Interrupt time scenario is TIMES = LASTING_TIME /10ms, in which the increment of static mode is 0. Accordingly, based on the principle that PWM_D sends TIMES to PWM per 10 ms, we can achieve the control of gradual change, bright and dark as well as static.

FLICK (flicker), the lasting time of such a scenario can be obtained in terms of the following formula LASTING_TIME:

```
LASTING_TIME=FLICK_TIMES*FLICK_TIME
FLICK_TIMES    flicker times
FLICK_TIME     flicker lasting time
```

Thus, this routine delivers PWM_FRONT_COLOR and PWM_BACK_COLOR by turns to PWM using FLICK_TIME as the interval, and after delivering FLICK_TIMES, ends the control of flicker function.

Table 1. Look-up Table of RGB Binary Value & Corresponding PWM Value

Value	R	PWM	G	PWM	B	PWM
	000	0	000	0	00	0
	001	36	001	36	01	85
	010	72	010	72	10	170
	011	108	011	108	11	255
	100	144	100	144		
	101	180	101	180		
	110	216	110	216		
	111	255	111	255		

When the 10-ms interrupt is received, the processed scenario data is delivered to the self-defined peripherals for display (the design flow is shown in Figure 3). All lamps are judged in the interrupt cycle to determine whether the system needs to play a scenario completely. If the present scenario is totally played out, data for the next scenario is collected and delivered to the control unit of the self-defined IP core for analysis and processing. If the scenario is still incomplete, the interrupt routine returns.

The third task is to play MP3 format music. To fulfill this task, tone quality has to be taken into account. It is interesting to observe how landscape lamps appear to change with anamorphic music, and therefore we adopted a secure hardware based decoding solution. We have used the STA013 decoding chip and the CS4334 D/A converter. Refer back to Figure 6 for the detailed design flow.

When the task is activated, it first initializes the I^2C bus, and then invokes the sta_Init() function to initialize STA013. This initialization includes resetting STA013, verifying ST013, and writing the configuration files, which are loaded in STA013_UpdateData[] array. The following operation configures STA013 and set tone as well as prepares data for compression. We start by invoking the decode control function sta_SendToDecoder (unsigned short len, unsigned char MP3_data[]) for

decoding. When the DATA_REQ pin of STA013 is high, it indicates that STA013 needs new MP3 data to compress and play. By querying the sixth bit of the status register in SPI core we judge whether status register TXDATA requires new data (or whether previous data was delivered to STA013); if this bit is low, we write new MP3 data to TXDATA. The received data from STA013 is decoded and played.

The difficult problems in the above routines are in the decoding time sequence setting and phase-locked loop (PLL) configuration. The data input/output accords a certain standard of time sequence. For instance, here we set the SPI frequency clock to 400 kHz so that the music can be played smoothly. If this frequency is too high or too low, it will affect the tone quality and music rhythm. An improper setting can even cause cacophony. The PLL may impact the operating clock of the on-chip components. Therefore, we had to be careful with the PLL setting, because a wrong setting of PLL may generate sampling drift and consequently cause anamorphic music.

Design Features

The system uses the Nios II soft core combined with an FPGA to control LED lamps. At least 256 lamp colors can be displayed in our system with full dynamic effects based on five changing modes: static, gradual changing, bright, dark, and flicker. Simultaneously, we can change the lamps' colors along with MP3 music rhythm. Our system can be used in applications that integrate decoration lamps with music in public places. Because of the nearly 200 MIPS capacity of the Nios II soft core, no color leap appears in the gradually changing LED color. By deploying the user-defined peripherals, the system can quickly perform data analysis of lamp control, and allows for easy expansion of peripherals. Using the SOPC Builder tool, it was easy for us to delete and add the MP3 expansion circuitry and the user-defined peripherals. By taking full advantage of the FPGA, we were able to develop PWM IP core, expand multi-PWM circuits in peripherals based on the design requirement. After optimization of a design, the system' logic units are much reduced when compared with the purely traditional embedded, bus-based designs.

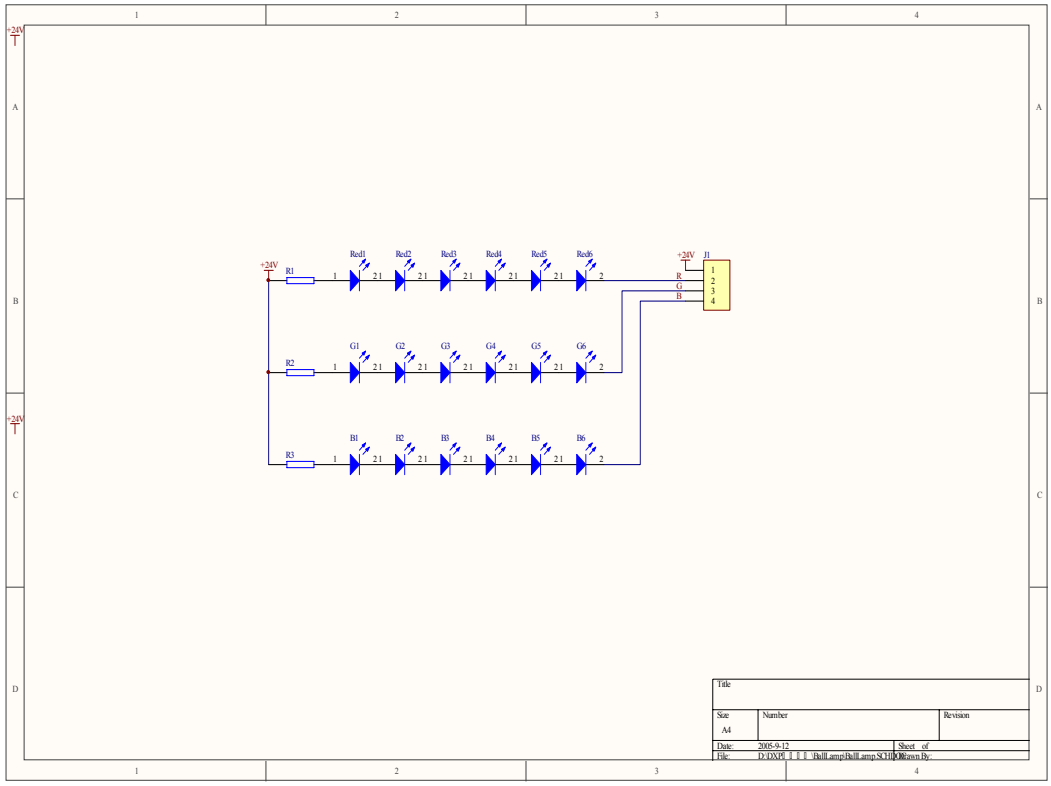
Conclusion

With more than two months of learning, we have been able to appreciate the Quartus II tool's powerful design functions and flexibility. The system provided us with many common IP cores in SOPC Builder, which helped in our design work and enabled us to add our self-defined IP cores and commands to meet the customer specific requirements. This approach made our design more flexible, especially the self-defined commands, which when added to existing 256 colors, are sufficient to meet most customer requirements. Additionally, the Quartus II software provided the functions from the start of the design to completion. These functions are easy to handle in the GUI. As for software development, the Quartus II software also integrates the Nios II IDE. We were able to finish the program design and download the final design using the Nios II IDE GUI.

Additionally, when we compared Quartus II version 4.2 and Quartus II version 5.0, we noticed that with Quartus II version 5.0 we can save system compilation time. Previously, even a small design modification needed the whole system to be recompiled. However, the Quartus II software version 5.0 provides optimized compilation, which only compiles the modified parts each time. As for the system design, we know about the advantages of FPGA and soft core design methods, especially during product development. With these methods, we can shorten the development cycle, reduce development risk, and get the early-to-market advantage. The Quartus II tool provided us with abundant materials for development, which are easy to understand, and each user reference emphasized a design principle by illustrating it with diagrams and code samples. By studying these materials, we were able to develop our own systems easily and wrote programs based on our requirements.

Appendix

Schematic Circuit of LED Lamp



Schematic Circuit of LED Drive