I²C Controlled PWM Driver Design

Liukee Liu
Department of Electrical and Computer Engineering
Portland State University
Portland, Oregon, USA
liukee@pdx.edu

Saleh Esmaeil
Department of Electrical and Computer Engineering
Portland State University
Portland, Oregon, USA
esmaeil@pdx.edu

Mohammad Alshaiji
Department of Electrical and Computer Engineering
Portland State University
Portland, Oregon, USA
alshaiji@pdx.edu

Riley Cameron
Department of Electrical and Computer Engineering
Portland State University
Portland, Oregon, USA
rileycam@pdx.edu

Abstract— This report presents the design and implementation of a digital LED control subsystem for an I²C-programmable lighting device. The system includes a simulated 400 kHz internal oscillator, a parameterized clock-division module, independent 8-bit PWM channels, and a mode-selection block for flexible LED behavior. The design supports multiple output modes, including static on/off, individual brightness control, and group dimming or blinking, based on register configurations. Developed using SystemVerilog and verified through simulation in QuestaSim, the subsystem demonstrates stable timing generation, accurate duty-cycle modulation, and correct mode transitions. The resulting architecture provides a modular and scalable foundation for programmable LED control applications.

Keywords—I²C, PWM, LED driver, SystemVerilog, digital design, simulation

I. Introduction

LED drivers are commonly used in many digital systems to control lights, indicators, and basic displays. In modern designs, these drivers often use an I²C interface so the LED behavior can be changed through software. This allows the user to adjust brightness, blinking patterns, and power modes without changing the hardware. Because of this, the internal logic of an LED driver must be reliable and able to translate register values into correct timing signals for the LEDs.

In this project, we built a small I²C-controlled LED driver that includes a simulated 400 kHz oscillator, clock-division logic, four PWM channels for brightness control, and a mode-selection block for choosing how each LED behaves. The system supports different LED modes such as always off, always on, individual PWM brightness, and group blinking. A sleep mode is also included to turn off the oscillator and reduce power usage when needed.

The main goal of this work was to design these LED functions in a modular way using SystemVerilog so they can be easily tested and connected to the I²C controller. The design was verified in simulation using QuestaSim to check that the timing signals, PWM outputs, and LED modes change correctly when the registers are updated. This project demonstrates how digital control signals can be used to manage simple hardware outputs in an organized and efficient way.

II. Background

LED drivers are widely used in digital systems where multiple LEDs must be controlled with predictable timing and programmable behavior. Many modern applications require adjustable brightness, blinking patterns, and low-power operation, which makes an I²C-based register interface a practical solution. Through I²C writes and reads, a host processor can configure internal registers that determine how each LED channel operates. To support this, the internal digital logic must reliably decode bus transactions, manage register states, and generate accurate PWM timing.

This project references the structure of devices such as the PCA9632, which use I²C to control several LED channels through individual PWM generators and optional group dimming or blinking. The original device contains additional analog circuitry and hardware-specific functions, but the focus here is on reproducing the essential digital behavior needed for programmable LED control. Noncritical features such as analog output drivers, subaddress support, and auto-increment were removed to simplify the design and keep it suitable for simulation.

The simplified design maintains eight 8-bit control registers, including a mode register, four individual PWM registers, a group duty register, a group frequency register, and an output-selection register. These registers define all LED behavior and allow the system to support on/off control, individual brightness, and group dimming or blinking. The project uses SystemVerilog to implement the bit-level I²C interface, the I²C controller, the register file, the PWM generators, and the required mode-selection logic. QuestaSim was used to validate the design and confirm proper I²C operation, register updates, and LED output timing.

This work demonstrates how the digital portion of an I²C controlled LED driver can be modeled in a modular way, highlighting the use of SystemVerilog constructs for register-based hardware design and verification.

III. Architecture

This project is based on the PCA9632 part by NXP Semiconductors [1]. The high-level architecture consists of an I²C bus controller receiving data from the bus master which it uses to set the PWM mode, PWM duty cycle, and LED output state (Fig. 1).

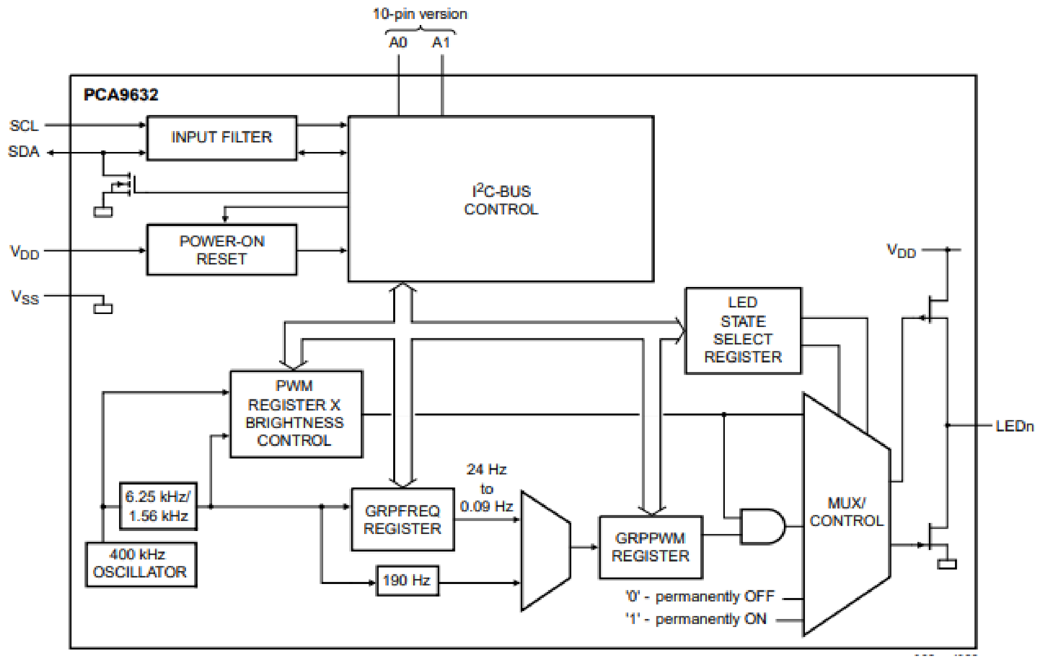


Fig. 1. PCA9632 Block Diagram

These are the specific simplifications that were made for this project:

1. Analog portions: input filter, power-on reset, & LED output FETs (no configuration for open-drain vs totem pole)
2. Address inputs A0 and A1 (static address is used instead: 0x40)
3. Subaddresses and all-call address for I²C
4. Auto-increment for I²C writes
5. Output change configuration (LEDs update on I²C STOP vs ACK)
6. Software reset

After eliminating these features from the scope, the number of required registers drops from 13 to 8. This is more efficient since it fully utilizes a 3-bit address space. Each register is 1-byte. Table I. lists the control registers implemented.

TABLE I. I²C Controlled PWM Driver Design - Control Registers

Reg Addr	Name	Type	Function
0x00	MODE	R/W	Mode register
0x01	PWM0	R/W	Brightness control LED0
0x02	PWM1	R/W	Brightness control LED1
0x03	PWM2	R/W	Brightness control LED2
0x04	PWM3	R/W	Brightness control LED3
0x05	GRPPWM	R/W	Group duty cycle control
0x06	GRPFREQ	R/W	Group frequency
0x07	LEDOUT	R/W	LED output state

The MODE register contains general control flags which are listed in Table II. PWM0-3 registers are used for individual brightness control of each LED channel. They allow the duty cycle to be set in the range of 0% (0) to 99.6% (255). The GRPPWM and GRPFREQ registers are used for global dimming and blinking control. The LEDOUT register selects which internal signal drives each LED channel.

TABLE II. I²C Controlled PWM Driver Design - Mode Register Bitfields

Bit	Symbol	Type	Value	Description
7	AI2	R	0	Register Auto-Increment disabled
			1	Register Auto-Increment enabled
6	AI1	R	0/1	Auto-Increment bit 1
5	AI0	R	0/1	Auto-Increment bit 0
4	SLEEP	R/W	0	Normal mode
			1	Low power mode - oscillator off
3	DMBLNK	R/W	0	Group control = dimming
			1	Group control = blinking
2	INVRT	R/W	0	Output logic state not inverted
			1	Output logic state inverted
1	OCH	R/W	0	Outputs change on STOP command
			1	Outputs change on ACK
0	–	–	0/1	reserved

The MODE register implements some signals that are unused: AI2, AI1, AI0, and OCH. These were included in the architecture so that they could be implemented if the core features were finished with time to spare, however this was not the case. Setting the SLEEP bit turns off the LED outputs and stops activity of all modules while maintaining the register states. The DMBLNK bit controls whether group mode (when selected by LEDOUT) is used to dim all LEDs by a given factor or blink all LEDs at a given period and duty-cycle. Setting the INVRT bit inverts the output logic signals.

Fig. 2 depicts the block diagram of the design after making the specified simplifications. The diagram also shows how the higher level modules are defined, each block represents a module from the SystemVerilog codebase. The ports for the top-level of the design are the reset signal, the SCL and SDA I²C bus lines, and the four LED output channels.

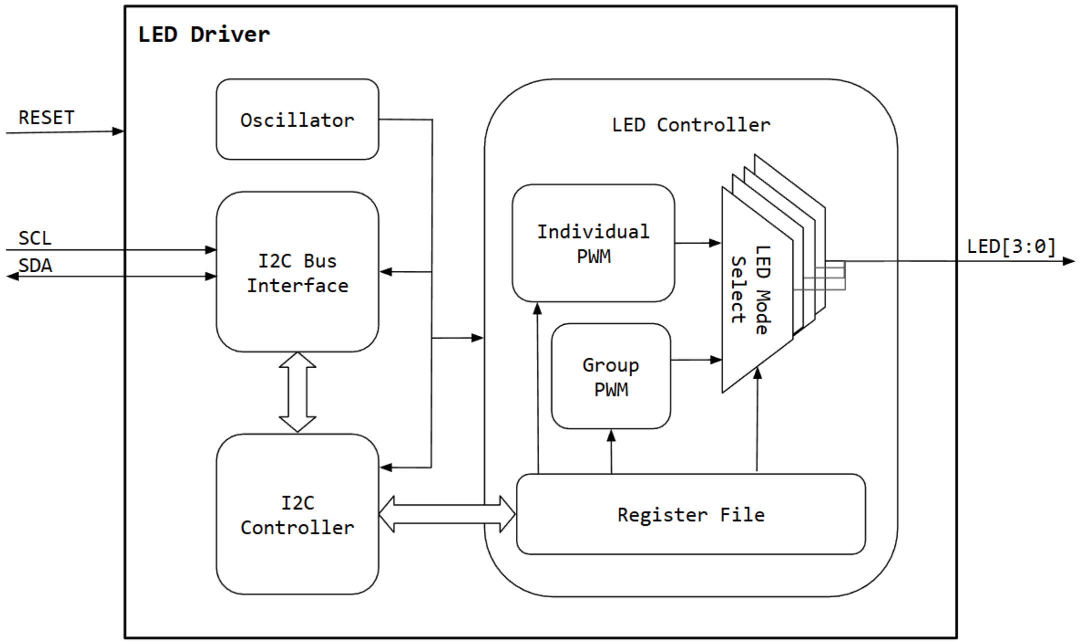


Fig. 2. I²C Controlled PWM Driver Design - Block Diagram

A. Oscillator (*oscillator_400K.sv*)

The Oscillator module takes the reset and sleep signals as input, and outputs a simulated 400 KHz clock signal using timing delays. When the reset signal is high the clock stops, and when the reset signal falls back to low the clock restarts. This is a simulated replacement for the actual hardware clock and is not synthesizable.

B. I²C Bus Interface (*i2c_bus_interface.sv*)

The I²C bus interface handles all low-level communication on the SDA and SCL lines. It detects START and STOP conditions, shifts one bit on every SCL rising edge, and assembles the incoming bits into an 8-bit byte. When a full byte is received, the interface outputs it along with a valid pulse for the I²C controller.

For outgoing data, the interface loads a byte from the controller and shifts it out one bit at a time on SCL falling edges, driving SDA in open-drain mode. After transmission, it raises a ready signal to request the next byte.

The design keeps the bit-level and byte-level logic separated on purpose. The bus interface focuses only on timing, edges, and bit shifting, while the I²C controller handles address decoding and register operations. This separation results in a simpler structure that is easier to verify and matches how real I²C devices organize their internal logic.

C. I²C Controller (*i2c_controller.sv*)

The i2c_controller module acts as the central protocol management layer, bridging the gap between the physical bit-level signals handled by the i2c_bus_interface and the internal register file of the system. Designed as a Finite State Machine (FSM), it abstracts the serial nature of the I2C protocol into byte-level transactions.

The controller operates using four distinct states. In the CTRL_IDLE state, the system waits for a START condition to be detected on the bus. Upon receiving a start signal, it transitions to CTRL_ADDR, where it latches the incoming 7-bit device address and the Read/Write bit. If the address matches the device ID, the state machine moves to CTRL_REG, where the subsequent byte is interpreted as the register pointer (reg_addr_ptr). Finally, the machine enters the CTRL_DATA state to handle the data payload. In this state, the controller either drives data onto the bus from the selected register or writes received data into the register file, depending on the transaction type.

D. LED Controller (*led_controller.sv*)

The LED Controller block is a wrapper for instantiating and connecting all of the PWM control logic. It also contains the byte array representing the register file, and utilizes the *bus_if* interface to service reads and writes from the I²C Controller. The LED controller contains the following modules:

1) Individual PWM (*individual_pwm_block.sv*)

The Individual PWM block generates a separate brightness control signal for each LED. It uses four independent 8-bit PWM engines, one per LED, where each engine compares a free-running counter against the corresponding PWM register (PWM0–PWM3). This allows every LED to have its own brightness level based on the duty cycle written through the I²C interface.

2) Group PWM (*group_pwm.sv*)

The Group PWM block is responsible for generating a single PWM signal that will be overlaid on the individual signals (ANDed with) when group mode is selected. It has two modes, dimming and blinking, depending on the DMBLNK bit. It uses the values from the GRPFREQ and GRPPWM registers to create the output signal (Fig. 3).

In dimming mode, the PWM block inside of Group PWM takes the GRPPWM value as its duty cycle and a 50 KHz, divided down from 400 KHz, signal as its clock. The final PWM output is a 195 Hz signal due to the inherit division created in the PWM block (50 KHz / 256 = ~195 Hz). The GRPFREQ register is not used.

In blinking mode, the PWM also uses GRPPWM as the duty cycle, but the clock input is changed to a variable frequency in the range of 6.25 KHz to 24 Hz that is determined by the GRPFREQ register. The final PWM output is in the range of 24 Hz to 0.09 Hz which is slow enough to create a visible blinking effect.

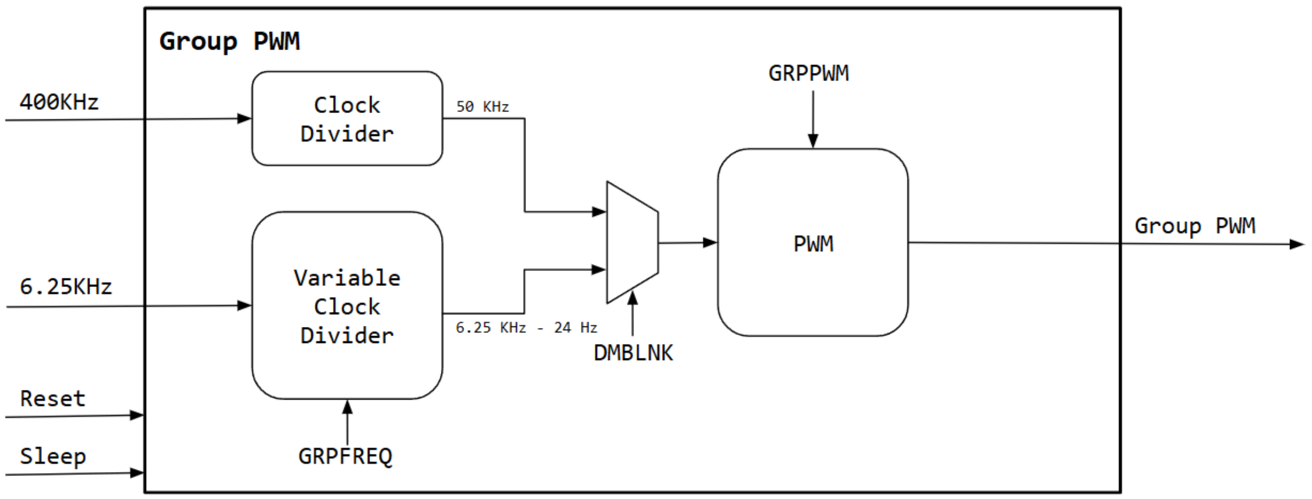


Fig. 3. Group PWM Control Module - Block Diagram

3) LED Mode Select (*led_mode_select.sv*)

The LED Mode Select mux selects which control signal drives each LED based on the LEDOUT register. For every LED, the mux chooses one of four modes: always OFF, always ON, individual PWM, or group PWM. This logic ensures the LED output matches the mode configured by the I²C controller.

IV. Verification

Testbenches were created for every module so that correct functionality could be ensured before integrating modules into the next level of hierarchy. For the top level module, located in *led_driver.sv*, the goal of the testbench was to evaluate the design from the perspective of the end-user. It programs the LED driver in all possible modes and tests edgecase scenarios. For a design like this it is not possible to create an exhaustive test since there are too many possible state combinations. Instead the test is broken down into several feature-specific tests: on/off, individual mode, group dimming mode, group blinking mode, sleep, inverted output, and register readback.

In practice, the *led_driver_tb.sv* defines tasks for reading from and writing to the control registers over the I²C bus. There is also a task defined for each of the features above that performs the necessary register programming to check different scenarios within that feature. The results of each scenario are confirmed by analyzing the transcript, waveform, and the LED PWM Viewer. One example from each feature test is shown below.

A. On & Off

After resetting the DUT, the registers are all zeroed out which puts the LEDs into the OFF state. Writing 0x55 to the LEDOUT register sets all states to ON (Fig. 4).

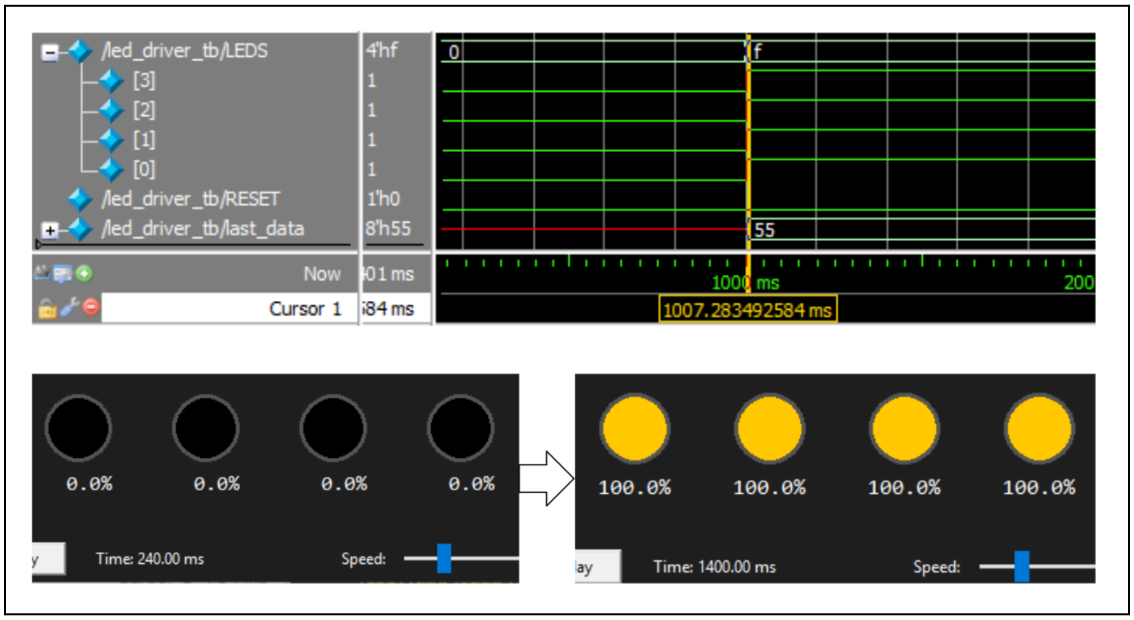


Fig. 4. Testbench results of On/Off test

B. Individual Mode

Writing values to each of the PWMx registers (in this case: 0x40, 0x80, 0xC0, & 0xFF) then writing 0xAA to the LEDOUT register sets all LED outputs to individual mode (Fig. 5).

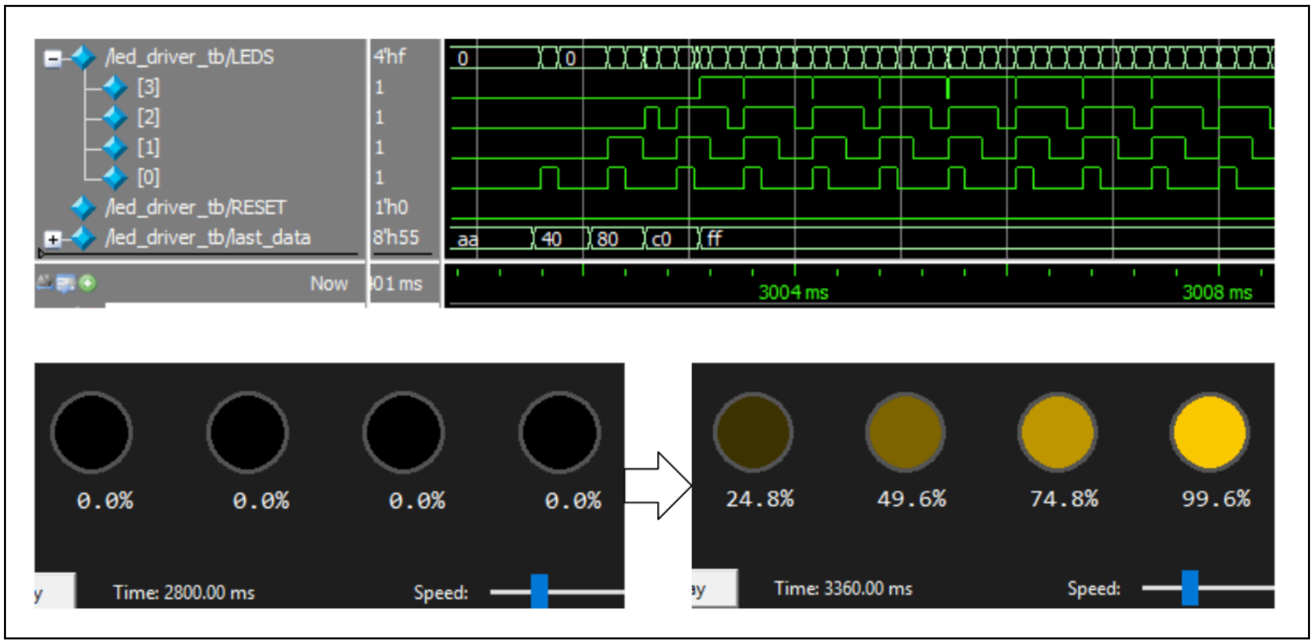


Fig. 5. Testbench results of Individual Mode test

C. Group Dimming Mode

Clearing the DMBLNK bit, setting the GRPPWM to 0xC0 (75%), and writing 0xFF to the LEDOUT register sets all LED outputs to group mode. The 75% dimming filter is applied on top of all the individual PWM signals (Fig. 6).

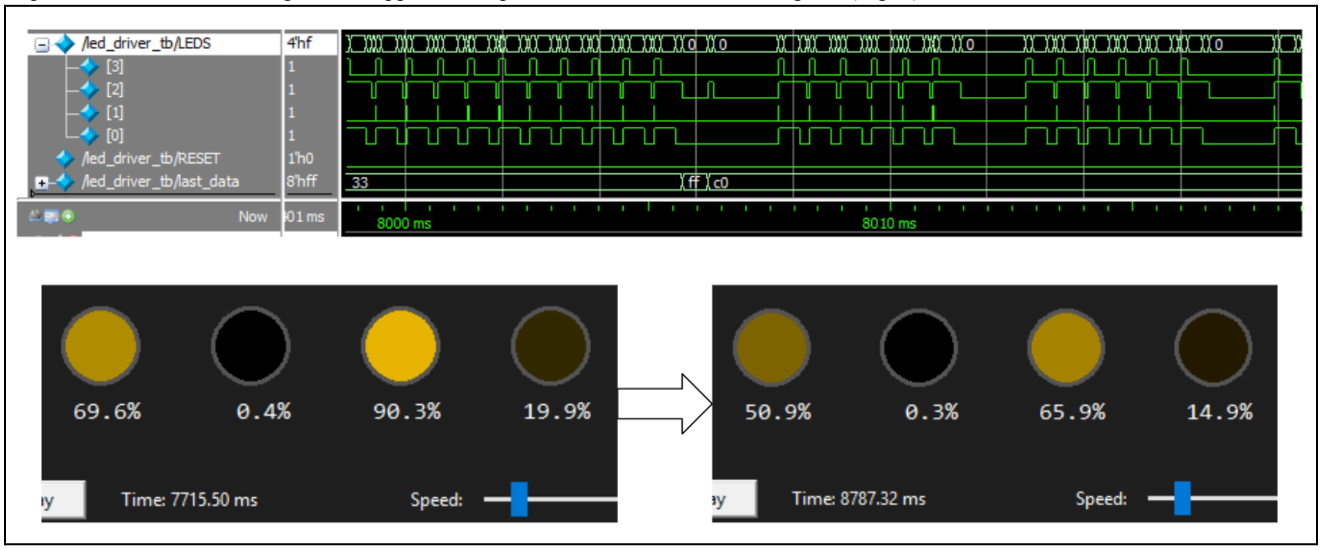


Fig. 6. Testbench results of Group Dimming Mode test

D. Group Blinking Mode

Setting the DMBLNK bit, the GRPPWM register to 0x80 (50%), the GRPFREQ register to 0x10 (1.52 Hz), and the LEDOUT register to 0xFF sets all LED outputs to group blinking mode. The blinking filter is applied on top of all the individual PWM signals, creating a toggling output once every ~328ms (Fig. 7).

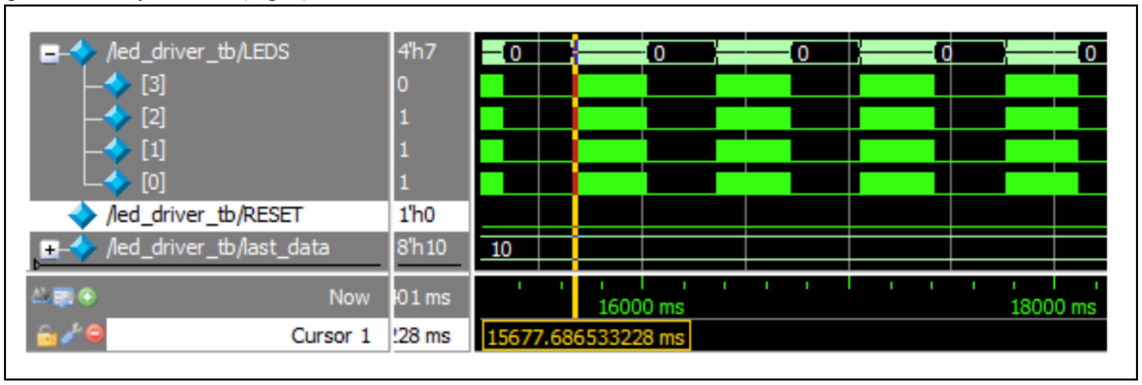


Fig. 7. Testbench results of Group Blinking Mode test

E. Sleep Mode

Sleep mode is entered by setting the SLEEP bit in the MODE register. It causes all four outputs to drop to zero, but maintains the control register states when it wakes up (Fig. 8).

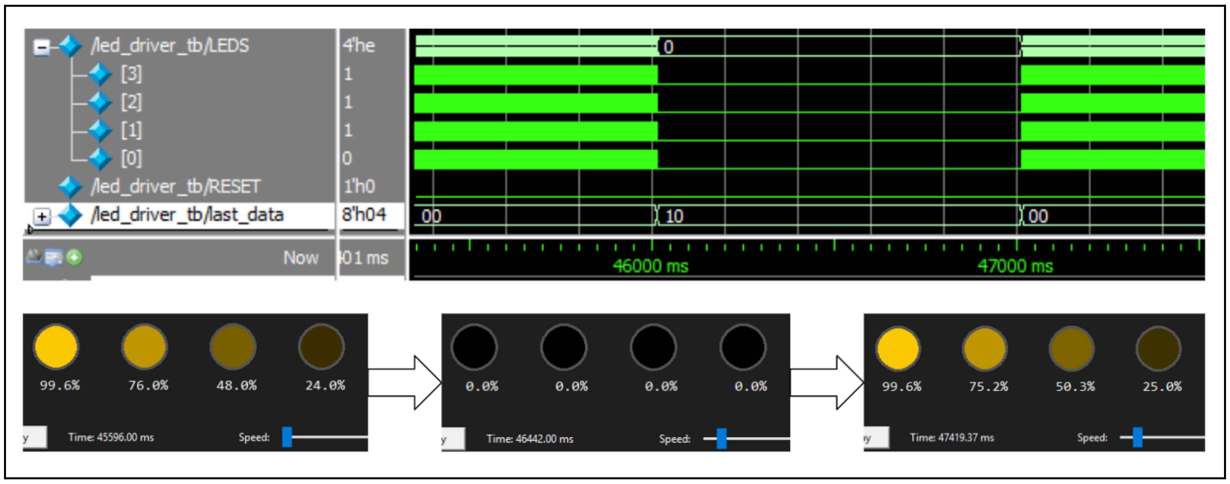


Fig. 8. Testbench results of Sleep Mode test

F. Inverted Output

Inverted mode is entered by setting the INVRT bit in the MODE register. It causes all four outputs to invert themselves regardless of which LEDOUT mode they are in (Fig. 9).

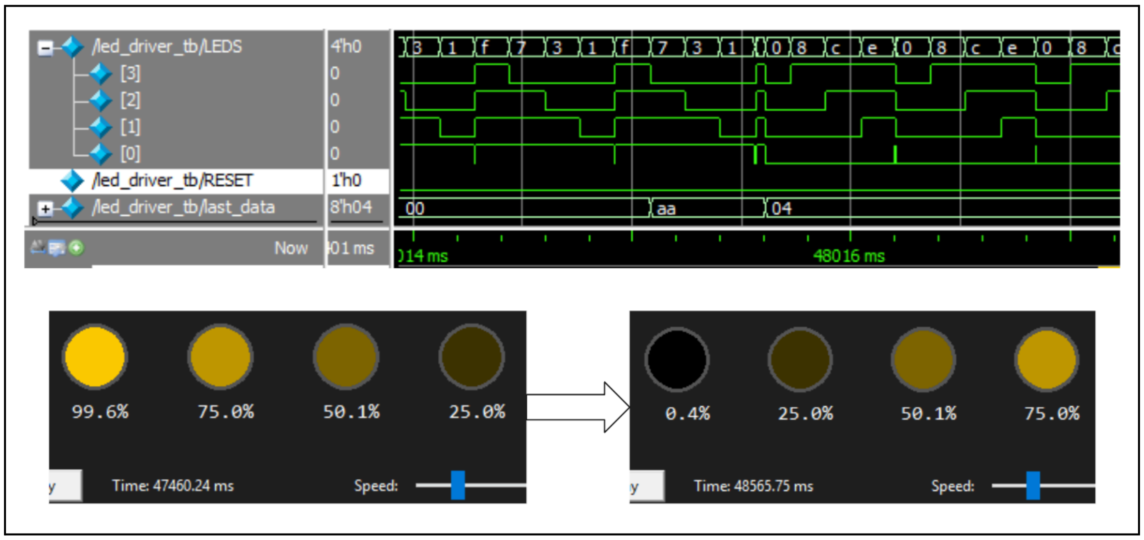
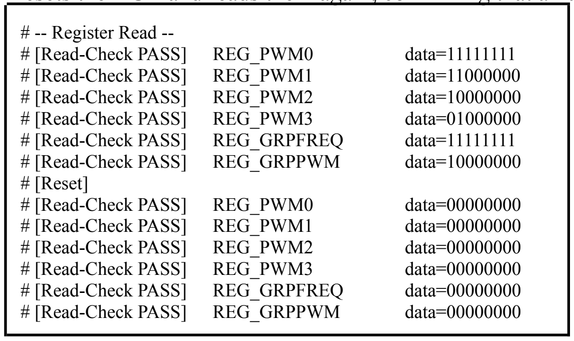


Fig. 9. Testbench results of Inverted Output test

G. Register Readback

Register readback is performed by sending an I²C read request along with a byte containing the requested data's address. The DUT will then respond by reading the data out of its control registers and sending it back to the testbench (bus master) on the I²C bus. The test reads the PWM and frequency registers then resets the DUT and reads them again, confirming that all are now zeroed out (Fig. 10).



```
# -- Register Read --
# [Read-Check PASS]    REG_PWM0        data=11111111
# [Read-Check PASS]    REG_PWM1        data=11000000
# [Read-Check PASS]    REG_PWM2        data=10000000
# [Read-Check PASS]    REG_PWM3        data=01000000
# [Read-Check PASS]    REG_GRPFREQ     data=11111111
# [Read-Check PASS]    REG_GRPPWM      data=10000000
# [Reset]
# [Read-Check PASS]    REG_PWM0        data=00000000
# [Read-Check PASS]    REG_PWM1        data=00000000
# [Read-Check PASS]    REG_PWM2        data=00000000
# [Read-Check PASS]    REG_PWM3        data=00000000
# [Read-Check PASS]    REG_GRPFREQ     data=00000000
# [Read-Check PASS]    REG_GRPPWM      data=00000000
```

Fig. 10. Testbench results of Register Readback test.

V. Challenges

A. False STOP Condition During Readback:

- Issue: Read transactions for even-numbered addresses failed intermittently, returning high-impedance states due to the RTL incorrectly detecting a STOP condition mid-transaction.
- Resolution: Traced the error to a race condition where SDA changed state while SCL was high. Implemented a "bus parking" strategy in the testbench to force SCL low before releasing SDA, preventing invalid stop detection.

B. Simulation Time Precision Mismatch:

- Issue: Delays defined as fractional real numbers were being rounded down to zero by the simulator, causing simultaneous signal transitions and protocol violations.
- Resolution: Enforced a timescale directive in the testbench to prevent rounding errors and ensure accurate execution of microsecond-level delays.

C. Excessive Simulation Overhead**:**

- Issue: Full system verification generated unmanageable waveform files (>6GB) due to high-speed clock toggling during long-duration blink tests, which hindered the debugging process.
- Resolution**:** Optimized the simulation strategy by selectively enabling waveform dumping only during critical read/write phases and temporarily reducing wait delays in the testbench.

D. Finite State Machine Rigidity:

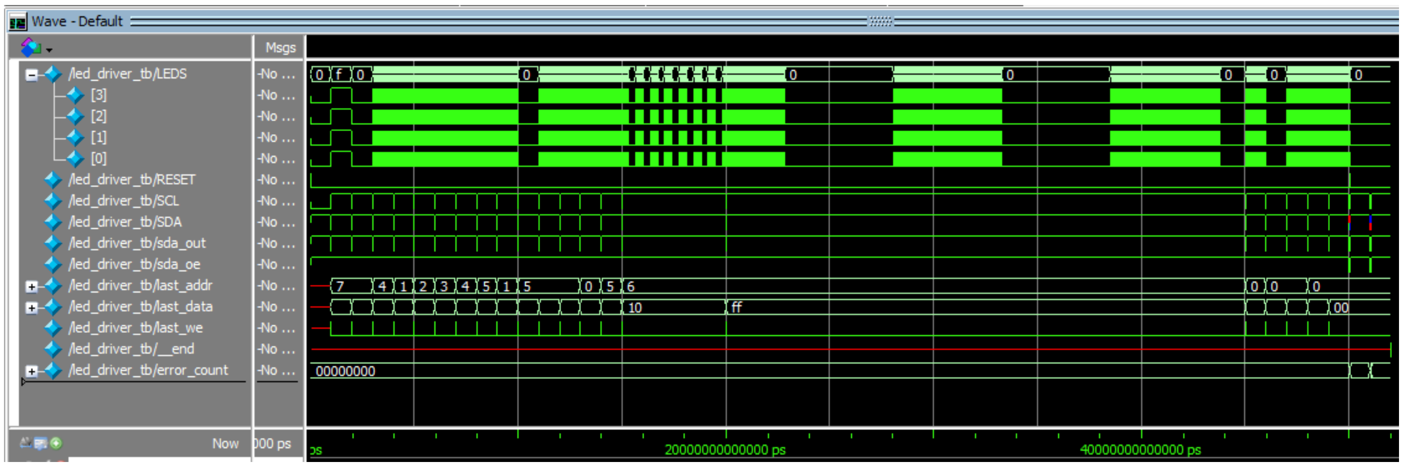
- Issue: The controller's state machine was designed with a rigid transition path that assumes every write transaction must be followed by a register pointer update, complicating standard current-address read operations.
- Resolution: Designed the system architecture to enforce a combined format protocol, requiring the master to always set the register address via a Write transaction before initiating a Read to ensure deterministic pointer management.

VI. Appendices

A. Transcript

```
# [Reset]
#
# === Starting Testcases ===
#
#
# -- LEDs On (No PWM) --
# [Write-Check PASS]    REG_LEDOUT      addr=7    data=01010101
#
# -- Individual PWM --
# [Write-Check PASS]    REG_LEDOUT      addr=7    data=10101010
# [Write-Check PASS]    REG_PWM0        addr=1    data=01000000
# [Write-Check PASS]    REG_PWM1        addr=2    data=10000000
# [Write-Check PASS]    REG_PWM2        addr=3    data=11000000
# [Write-Check PASS]    REG_PWM3        addr=4    data=11111111
# [Write-Check PASS]    REG_PWM0        addr=1    data=10110010
# [Write-Check PASS]    REG_PWM1        addr=2    data=00000001
# [Write-Check PASS]    REG_PWM2        addr=3    data=11100111
# [Write-Check PASS]    REG_PWM3        addr=4    data=00110011
#
# -- Group Dimming --
# [Write-Check PASS]    REG_LEDOUT      addr=7    data=11111111
# [Write-Check PASS]    REG_GRPPWM      addr=5    data=11000000
# [Write-Check PASS]    REG_PWM3        addr=4    data=01000000
# [Write-Check PASS]    REG_PWM2        addr=3    data=10000000
# [Write-Check PASS]    REG_PWM1        addr=2    data=11000000
# [Write-Check PASS]    REG_PWM0        addr=1    data=11111111
# [Write-Check PASS]    REG_GRPPWM      addr=5    data=00000000
# [Write-Check PASS]    REG_GRPPWM      addr=5    data=10000000
# [Write-Check PASS]    REG_GRPPWM      addr=5    data=00110000
#
# -- Group Blinking --
# [Write-Check PASS]    REG_LEDOUT      addr=7    data=11111111
# [Write-Check PASS]    REG_MODE        addr=0    data=00001000
# [Write-Check PASS]    REG_GRPPWM      addr=5    data=10000000
# [Write-Check PASS]    REG_GRPFREQ     addr=6    data=00010000
# [Write-Check PASS]    REG_GRPFREQ     addr=6    data=11111111
# [Write-Check PASS]    REG_MODE        addr=0    data=00000000
#
# -- Sleep --
# [Write-Check PASS]    REG_LEDOUT      addr=7    data=10101010
# [Write-Check PASS]   REG_MODE          addr=0    data=00010000
# [Write-Check PASS]   REG_MODE          addr=0    data=00000000
#
# -- Invert Output --
# [Write-Check PASS]   REG_LEDOUT        addr=7    data=10101010
# [Write-Check PASS]   REG_MODE          addr=0    data=00000100
# [Write-Check PASS]   REG_MODE          addr=0    data=00000000
#
# -- Register Read --
# [Read-Check PASS]    REG_PWM0          data=11111111
# [Read-Check PASS]    REG_PWM1          data=11000000
# [Read-Check PASS]    REG_PWM2          data=10000000
# [Read-Check PASS]    REG_PWM3          data=01000000
# [Read-Check PASS]    REG_GRPFREQ       data=11111111
# [Read-Check PASS]    REG_GRPPWM        data=10000000
# [Reset]
# [Read-Check PASS]    REG_PWM0          data=00000000
# [Read-Check PASS]    REG_PWM1          data=00000000
# [Read-Check PASS]    REG_PWM2          data=00000000
# [Read-Check PASS]    REG_PWM3          data=00000000
# [Read-Check PASS]    REG_GRPFREQ       data=00000000
# [Read-Check PASS]    REG_GRPPWM        data=00000000
#
# === ALL TESTS PASSED ===
```

B. Waveform



C. LED PWM Viewer Video

https://drive.google.com/file/d/1cur0XTjT6hOeXpAsOVk7cibnPJeYMFsJ/view?usp=sharing

VII. Conclusion

This project provided a valuable learning experience in understanding how digital hardware systems are designed, organized, and controlled at the register level. By implementing a simplified I²C-controlled LED driver, we gained practical skills in SystemVerilog, modular design, clock generation, and PWM-based signal control. Working with the internal oscillator, clock dividers, PWM engines, and mode-selection logic helped us understand how different hardware components interact to produce correct real-time output behavior.

The project also improved our ability to write structured hardware code and to verify functionality using simulation tools such as QuestaSim. Although the design achieves the required functionality, there are still opportunities to expand it by adding more SystemVerilog constructs, improving parameterization, and exploring more advanced low-power or timing features. Overall, this work strengthened our understanding of digital design concepts and demonstrated how communication protocols, timing logic, and output control can be combined to build a complete hardware subsystem.

References

[1] NXP Semiconductors, "PCA9632 4-bit Fm+ I²C-bus low power LED driver - Product data sheet, Rev. 6, 21 September 2021." https://www.nxp.com/docs/en/data-sheet/PCA9632.pdf

[2] Wikipedia Contributors. "I2C." Wikipedia, Wikimedia Foundation, https://en.wikipedia.org/wiki/I2C