ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

----- ☙ 🕮 ❧ -----

Bài tập C4.2

Giảng viên hướng dẫn: TS. Phạm Duy Hưng

Thành viên thực hiện: Nhóm 3

22022118 Phạm Văn Duy

22022103 Ngô Đức Hiếu

22022175 Nguyễn Quốc Toản

22022145 Tạ Đình Kiên

Hà Nội, tháng 3 năm 2025

Mục lục

A. SPI

I. Yêu cầu của bài toán

1. Mục tiêu của bài toán

Mục tiêu của bài toán là thiết lập giao tiếp SPI giữa hai vi điều khiển STM32F401RE, trong đó một vi điều khiển đóng vai trò Master (gửi tín hiệu điều khiển) và vi điều khiển còn lại đóng vai trò Slave (nhận tín hiệu và thực hiện hành động). Cụ thể, Master gửi dữ liệu (0xAA) để yêu cầu Slave nháy LED trên chân PB0 năm lần, sau đó Slave phản hồi dữ liệu (0xBB) để báo hiệu hoàn tất. Master sẽ tắt LED trên chân PA8 khi nhận được phản hồi.

2. Các yêu cầu cụ thể

Giao tiếp SPI: Sử dụng các dây MOSI, MISO, SCLK và NSS để truyền/nhận dữ liệu giữa Master và Slave.

Điều khiển LED:
- Master: Điều khiển LED trên chân PA8 để nháy khi chưa nhận được phản hồi từ Slave.
- Slave: Nháy LED trên chân PB0 năm lần khi nhận được dữ liệu 0xAA từ Master.

Tốc độ truyền ổn định: Cả hai vi điều khiển được cấu hình với tốc độ baud rate fPCLK/16 (~1 MHz với HSI 16 MHz) và chế độ SPI đồng bộ (Mode 0: CPOL = 0, CPHA = 0).

Phản hồi từ Slave: Slave gửi dữ liệu 0xBB sau khi hoàn thành nháy LED để thông báo cho Master.

Dễ dàng mở rộng: Hệ thống có thể được mở rộng để điều khiển nhiều Slave bằng cách quản lý thêm các chân NSS.

3. Phạm vi và giới hạn của hệ thống

- Hệ thống sử dụng bốn dây (MOSI, MISO, SCLK, NSS) và GND để kết nối giữa hai vi điều khiển.
- Chuẩn SPI hoạt động ở mức logic TTL (3.3V), không yêu cầu mạch chuyển đổi điện áp.
- Hệ thống tập trung vào giao tiếp đơn giản (8-bit dữ liệu, full-duplex), không sử dụng tính năng nâng cao như DMA hoặc kiểm tra lỗi.

II. Phần thực thi hệ thống

1. Thiết kế phần cứng

Hệ thống sử dụng hai vi điều khiển STM32F401RE với các kết nối như sau:

Kết nối SPI:

MOSI (PA7 của Master) kết nối với MOSI (PA7 của Slave).

MISO (PA6 của Master) kết nối với MISO (PA6 của Slave).

SCLK (PA5 của Master) kết nối với SCLK (PA5 của Slave).

NSS (PA4 của Master) kết nối với NSS (PA4 của Slave).

GND của cả hai vi điều khiển được nối chung để hoàn chỉnh mạch điện.

LED:

Master: LED kết nối với chân PA8 thông qua điện trở hạn dòng (~330Ω).

Slave: LED kết nối với chân PB0 thông qua điện trở hạn dòng (~330Ω).

2. Cấu hình phần mềm

Phần mềm cho vi điều khiển Master

```c
//master
#include "stm32f4xx.h"

void SystemClock_Config(void);
void SPI1_Init(void);
void GPIO_Init(void);
uint8_t SPI1_TransmitReceive(uint8_t data);

int main(void) {
    SystemClock_Config();
    GPIO_Init();
    SPI1_Init();

    uint8_t tx_data = 0xAA; // Dữ liệu gửi
    uint8_t rx_data = 0;    // Dữ liệu nhận
    uint8_t done = 0;       // Cờ hoàn tất

    while (1) {
        if (!done) {
            // Nháy LED khi chưa nhận được phản hồi
            GPIOA->ODR |= (1 << 8);  // Bật LED
            for (volatile int i = 0; i < 50000; i++);
            GPIOA->ODR &= ~(1 << 8); // Tắt LED
            for (volatile int i = 0; i < 50000; i++);

            GPIOA->ODR &= ~(1 << 4); // Kích hoạt NSS
            rx_data = SPI1_TransmitReceive(tx_data); // Gửi và nhận
            GPIOA->ODR |= (1 << 4);  // Tắt NSS

            if (rx_data == 0xBB) {
                done = 1; // Đánh dấu hoàn tất
                GPIOA->ODR &= ~(1 << 8); // Tắt LED vĩnh viễn
            }
        }
        // Khi done = 1, không làm gì nữa
    }
}
void GPIO_Init(void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    GPIOA->MODER &= ~(3 << (4 * 2) | 3 << (8 * 2));
    GPIOA->MODER |= (1 << (4 * 2) | 1 << (8 * 2));
    GPIOA->MODER &= ~(3 << (5 * 2) | 3 << (6 * 2) | 3 << (7 * 2));
    GPIOA->MODER |= (2 << (5 * 2) | 2 << (6 * 2) | 2 << (7 * 2));
    GPIOA->AFR[0] &= ~(0xF << (5 * 4) | 0xF << (6 * 4) | 0xF << (7 * 4));
    GPIOA->AFR[0] |= (5 << (5 * 4) | 5 << (6 * 4) | 5 << (7 * 4));
}

void SPI1_Init(void) {
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
    SPI1->CR1 = 0;
    SPI1->CR1 |= (1 << 2);  // Master mode
    SPI1->CR1 |= (3 << 3);  // Baud rate = fPCLK/16
    SPI1->CR1 |= (1 << 9);  // Software NSS
    SPI1->CR1 |= (1 << 8);  // SSI = 1
    SPI1->CR1 |= (1 << 6);  // Kích hoạt SPI
}

uint8_t SPI1_TransmitReceive(uint8_t data) {
    SPI1->DR = data;
    while (!(SPI1->SR & SPI_SR_TXE));
    while (!(SPI1->SR & SPI_SR_RXNE));
    return SPI1->DR;
}

void SystemClock_Config(void) {
    RCC->CR |= RCC_CR_HSION;
    while (!(RCC->CR & RCC_CR_HSIRDY));
    RCC->CFGR = 0;
}
```

Master khởi tạo giao tiếp SPI và thực hiện các tác vụ sau:

- Cấu hình SPI1:
    - Chế độ: Master, full-duplex, 8-bit dữ liệu.
    - Tốc độ clock: fPCLK/16 (~1 MHz với HSI 16 MHz).
    - CPOL = 0, CPHA = 0 (Mode 0).

- NSS: Quản lý bằng phần mềm (PA4).

- Logic hoạt động:
  - Nháy LED trên PA8 (bật/tắt với chu kỳ ~50ms) khi chưa nhận được phản hồi từ Slave.
  - Kéo NSS (PA4) xuống thấp để kích hoạt Slave, gửi dữ liệu 0xAA qua MOSI, và nhận dữ liệu từ MISO.
  - Kéo NSS lên cao sau khi truyền/nhận.
  - Nếu nhận được 0xBB từ Slave, tắt LED trên PA8 vĩnh viễn và dừng truyền dữ liệu.

Phần mềm cho vi điều khiển Slave

```c
#include "stm32f4xx.h"

void SystemClock_Config(void);
void SPI1_Init(void);
void GPIO_Init(void);

int main(void) {
    SystemClock_Config();
    GPIO_Init();
    SPI1_Init();

    uint8_t received_data = 0;

    while (1) {
        GPIOB->ODR |= (1 << 0); // LED sáng khi đợi

        while (!(SPI1->SR & SPI_SR_RXNE));
        received_data = SPI1->DR;

        if (received_data == 0xAA) {
            for (int i = 0; i < 5; i++) {
                GPIOB->ODR &= ~(1 << 0);
                for (volatile int j = 0; j < 50000; j++);
                GPIOB->ODR |= (1 << 0);
                for (volatile int j = 0; j < 50000; j++);
            }
            SPI1->DR = 0xBB;
            while (SPI1->SR & SPI_SR_BSY);
        }
    }
}

void GPIO_Init(void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOBEN;
    GPIOB->MODER &= ~(3 << (0 * 2));
    GPIOB->MODER |= (1 << (0 * 2));
    GPIOA->MODER &= ~(3 << (4 * 2) | 3 << (5 * 2) | 3 << (6 * 2) | 3 << (7 * 2));
    GPIOA->MODER |= (2 << (4 * 2) | 2 << (5 * 2) | 2 << (6 * 2) | 2 << (7 * 2));
    GPIOA->AFR[0] &= ~(0xF << (4 * 4) | 0xF << (5 * 4) | 0xF << (6 * 4) | 0xF << (7 * 4));
    GPIOA->AFR[0] |= (5 << (4 * 4) | 5 << (5 * 4) | 5 << (6 * 4) | 5 << (7 * 4));
}
```

```c
void SPI1_Init(void) {
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
    SPI1->CR1 = 0;
    SPI1->CR1 &= ~(1 << 2); // Slave mode
    SPI1->CR1 |= (1 << 6);  // Kích hoạt SPI
}

void SystemClock_Config(void) {
    RCC->CR |= RCC_CR_HSION;
    while (!(RCC->CR & RCC_CR_HSIRDY));
    RCC->CFGR = 0;
}
```

Slave nhận dữ liệu từ Master và thực hiện các tác vụ sau:

Cấu hình SPI1:
- Chế độ: Slave, full-duplex, 8-bit dữ liệu.
- CPOL = 0, CPHA = 0 (đồng bộ với Master).
- NSS: Điều khiển bởi Master qua PA4.

Logic hoạt động:
- Bật LED trên PB0 khi đang đợi dữ liệu.
- Khi nhận được dữ liệu 0xAA qua MOSI, nháy LED trên PB0 năm lần (mỗi lần bật/tắt ~50ms).
- Gửi dữ liệu 0xBB qua MISO để báo hiệu hoàn tất.
- Đợi cho đến khi giao tiếp SPI hoàn tất (kiểm tra cờ BSY).

3. Kiểm tra, đánh giá hiệu quả hệ thống và hiệu chỉnh

3.1. Kiểm tra hệ thống

- Kiểm tra phần cứng:

  Xác minh kết nối giữa các chân SPI (PA5, PA6, PA7, PA4) và GND của hai vi điều khiển, đảm bảo không có ngắn mạch hoặc đứt gãy.

  Kiểm tra mức logic trên các chân SPI (0V và 3.3V) bằng multimeter hoặc oscilloscope.

  Đảm bảo LED trên PA8 (Master) và PB0 (Slave) được kết nối đúng với điện trở hạn dòng.

- Kiểm tra phần mềm:

  Tải mã nguồn cho Master và Slave, xác nhận hoạt động bằng cách quan sát LED:

    Master: LED trên PA8 nháy liên tục trước khi nhận 0xBB, sau đó tắt vĩnh viễn.

    Slave: LED trên PB0 bật liên tục khi đợi, nháy 5 lần khi nhận 0xAA.

  Sử dụng Logic Analyzer hoặc Serial Monitor (nếu tích hợp debug) để kiểm tra dữ liệu trên MOSI (0xAA) và MISO (0xBB).

  Kiểm tra tín hiệu NSS để đảm bảo Master kích hoạt Slave đúng thời điểm.

- Kiểm tra tốc độ truyền:

  Xác nhận tốc độ clock SPI (~1 MHz) đồng bộ giữa Master và Slave.

  Quan sát tín hiệu SCLK bằng oscilloscope để đảm bảo tần số và dạng sóng ổn định.

3.2. Đánh giá hiệu quả hệ thống

- Độ ổn định của tín hiệu:

  Tín hiệu trên MOSI, MISO, SCLK và NSS ổn định, không bị nhiễu hoặc mất dữ liệu trong quá trình truyền.

  LED trên Slave nháy đúng 5 lần khi nhận 0xAA, và Master nhận chính xác 0xBB để dừng hoạt động.

- Tốc độ và độ chính xác của điều khiển LED:

  LED trên Slave phản hồi nhanh chóng (~50ms mỗi lần nháy) sau khi nhận dữ liệu.

  Master dừng nháy LED trên PA8 ngay sau khi nhận 0xBB, chứng tỏ giao tiếp full-duplex hoạt động chính xác.

- Khả năng mở rộng:

  Hệ thống có thể thêm nhiều Slave bằng cách sử dụng các chân NSS khác nhau trên Master.

3.3. Hiệu chỉnh và tối ưu hóa

- Điều chỉnh tham số SPI:

  Đảm bảo cấu hình Mode 0 (CPOL = 0, CPHA = 0) được áp dụng chính xác trên cả Master và Slave.

  Nếu dữ liệu truyền không ổn định, thử giảm tốc độ clock SPI (ví dụ: fPCLK/32 hoặc fPCLK/64).

- Cải thiện tín hiệu:

  Sử dụng dây nối ngắn (<30cm) và chất lượng cao để giảm nhiễu điện từ.

Thêm điện trở kéo lên (pull-up) trên chân NSS của Slave nếu tín hiệu NSS không ổn định.

- Kiểm tra lỗi phần cứng:

  Kiểm tra chân SPI (PA4, PA5, PA6, PA7) trên cả hai vi điều khiển để đảm bảo không bị hỏng.

  Đảm bảo nguồn điện 3.3V ổn định, tránh dao động gây lỗi giao tiếp.

- Tối ưu mã nguồn:

  Thêm kiểm tra trạng thái lỗi SPI (như OVR hoặc MODF) để xử lý các trường hợp bất thường.

  Sử dụng ngắt SPI thay vì polling trên Slave để cải thiện hiệu suất nếu cần xử lý nhiều dữ liệu hơn.

  Giảm thời gian delay trong vòng lặp nháy LED (nếu cần phản hồi nhanh hơn) bằng cách tối ưu vòng lặp for.

4. Video Demo

https://drive.google.com/file/d/1T6c5AiPjeOEQAcxsbt0FeTOXVFC_sPRv/view?usp=drivesdk

B. PPI 8255A mode 0

Mục tiêu bài tập

- Mục tiêu của bài tập là thiết kế một hệ thống nhúng sử dụng vi điều khiển STM32 để giao tiếp với vi mạch mở rộng cổng vào/ra lập trình được Intel 8255A trong chế độ hoạt động cơ bản (Mode 0). Bài tập nhằm giúp sinh viên hiểu rõ cách hoạt động của PPI 8255A, cách cấu hình các cổng vào/ra, và cơ chế truyền nhận dữ liệu song song giữa vi điều khiển và các thiết bị ngoại vi.

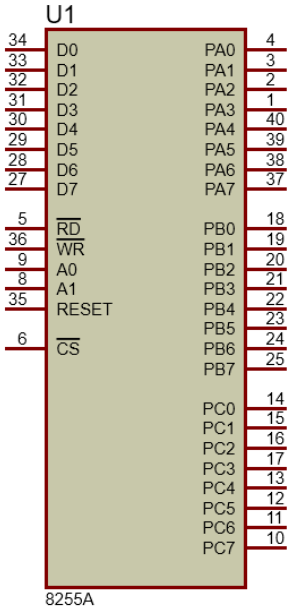
Yêu cầu bài tập

- Ghép nối STM32 với PPI 8255A thông qua bus dữ liệu D0–D7 và các chân điều khiển A0, A1, RD, WR, CS, RESET

- Cấu hình: Port A: output để điều khiển LED.

- Viết chương trình STM32 thực hiện:Ghi dữ liệu 0xAA và 0x55 luân phiên ra Port A để điều khiển LED.

1. Giới thiệu chung về 8255A và chế độ Mode 0



- Vi mạch 8255A Programmable Peripheral Interface (PPI) là một bộ giao tiếp ngoại vi lập trình được, thường được sử dụng để kết nối vi điều khiển với các thiết bị ngoại vi

như LED, bàn phím, cảm biến,... 8255A cung cấp ba cổng chính: Port A, Port B và Port C, trong đó mỗi cổng có thể được cấu hình là đầu vào (Input) hoặc đầu ra (Output), tùy vào yêu cầu của hệ thống.

- Chế độ Mode 0 là chế độ đơn giản nhất trong ba chế độ hoạt động chính của 8255A. Trong chế độ này, các cổng được cấu hình để hoạt động như các cổng vào/ra cơ bản không có bắt tay (handshaking). Đây là chế độ lý tưởng cho các ứng dụng cần giao tiếp tốc độ cao mà không yêu cầu xác nhận dữ liệu giữa vi điều khiển và thiết bị ngoại vi.

2. Nguyên lí hoạt động

a. Cấu hình chế độ Mode 0 của 8255A:

- 8255A được cấu hình ở chế độ Mode 0 – chế độ cơ bản nhất.

- Trong chế độ này, các cổng (Port A, Port B, Port C) có thể được cấu hình độc lập là đầu vào (input) hoặc đầu ra (output).

- Không yêu cầu các tín hiệu đồng bộ hay handshake phức tạp.

b. Thiết lập vai trò các cổng:

- Port A được cấu hình làm cổng xuất (output), dùng để điều khiển 8 đèn LED.

c. Giao tiếp giữa STM32 và 8255A:

- STM32 gửi dữ liệu đến 8255A thông qua **bus dữ liệu 8 bit (PB0–PB7)**.

- Các chân điều khiển **A0, A1, RD, WR, CS, RESET** được sử dụng để chọn địa chỉ port và điều khiển hoạt động ghi/đọc dữ liệu.

- Lệnh ghi dữ liệu được thực hiện bằng cách đưa dữ liệu lên bus và kích hoạt các tín hiệu điều khiển tương ứng.

d. Hoạt động chính của hệ thống:

- Chương trình lần lượt ghi hai mẫu dữ liệu 0xAA (10101010) và 0x55 (01010101) ra Port A với khoảng thời gian 1 giây giữa hai lần ghi.

- Dữ liệu này sẽ được truyền đến các chân PA0–PA7 của 8255, nơi đã kết nối với 8 LED.

- Các LED sẽ sáng/tắt tương ứng với từng bit của dữ liệu, tạo hiệu ứng nhấp nháy xen kẽ.

3. Thực thi hệ thống

- Chuẩn bị linh kiện

+ STM32F401RE

+ 8255A Programmable Peripheral Interface

+ LED x 8 cái

+ Điện trở x 8 cái

- Trong bài thực hành này, vi điều khiển STM32F401RE được kết nối với vi mạch mở rộng vào/ra 8255A để thực hiện giao tiếp song song ở chế độ Mode 0. Việc kết nối được thực hiện theo các nhóm tín hiệu chính như sau:

a. Kết nối bus dữ liệu D0–D7

- Bus dữ liệu 8 bit giữa STM32 và 8255A được thiết lập thông qua các chân PB0 đến PB7 của STM32. Các chân này đảm nhiệm vai trò truyền dữ liệu 8 bit đến 8255A hoặc đọc dữ liệu từ các port đầu vào. Cụ thể:

STM32 (GPIOB)	8255A	Tín hiệu
PB0	D0	Dữ liệu bit 0
PB1	D1	Dữ liệu bit 1
PB2	D2	Dữ liệu bit 2
PB3	D3	Dữ liệu bit 3
PB4	D4	Dữ liệu bit 4
PB5	D5	Dữ liệu bit 5
PB6	D6	Dữ liệu bit 6
PB7	D7	Dữ liệu bit 7

b. Kết nối các tín hiệu điều khiển

- Các tín hiệu điều khiển giữa STM32 và 8255A được thiết lập qua các chân PC0 đến PC5 của STM32. Chúng có vai trò lựa chọn port, ghi hoặc đọc dữ liệu, và reset chip:

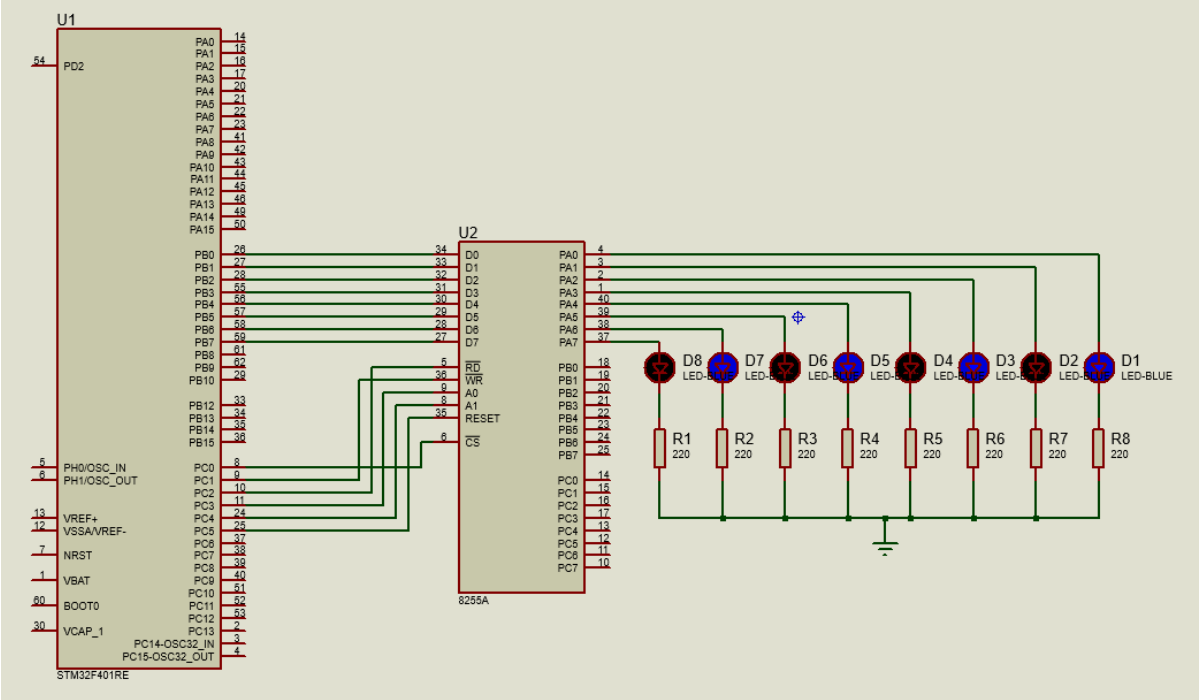
STM32 (GPIOC)	8255A	Chức năng
PC0	CS	Chip Select – cho phép giao tiếp với 8255
PC1	WR	Ghi dữ liệu từ STM32 vào 8255
PC2	RD	Đọc dữ liệu từ 8255 về STM32
PC3	A0	Địa chỉ bit thấp – chọn port hoặc thanh ghi
PC4	A1	Địa chỉ bit cao – kết hợp với A0 để chọn đúng cổng
PC5	RESET	Reset lại cấu hình của 8255A

c. Kết nối Port A của 8255A với LED

- Cổng A của 8255A được cấu hình làm cổng xuất (output) và được sử dụng để điều khiển 8 đèn LED. Mỗi chân từ PA0 đến PA7 được nối với một LED thông qua điện trở hạn dòng 220Ω. Đầu còn lại của LED được nối về mass (GND). Cụ thể

8255A	Thiết bị nối	Ghi chú
PA0–PA7	8 LED qua trở 220Ω	LED sáng/tắt theo từng bit dữ liệu gửi từ STM32

Sơ đồ kết nối phần cứng hoàn thiện:



d. Cấu hình phần mềm

```c
#include "stm32f4xx.h"

void SystemClock_Config(void);
void GPIO_Init(void);
void PPI_Init(void);
void PPI_Write(uint8_t address, uint8_t data);
uint8_t PPI_Read(uint8_t address);
void Error_Handler(void);

int main(void) {
    SystemCoreClock = 16000000;
    SystemClock_Config();
    GPIO_Init();
    PPI_Init();

    while (1) {
        PPI_Write(0x00, 0xAA);
        for (volatile uint32_t i = 0; i < 1000000; i++);
        PPI_Write(0x00, 0x55);
        for (volatile uint32_t i = 0; i < 1000000; i++);
        PPI_Read(0x01);
    }
}

void SystemClock_Config(void) {
    RCC->CR |= RCC_CR_HSEON;
    while (!(RCC->CR & RCC_CR_HSERDY));

    RCC->PLLCFGR &= ~(0x3FFF);
    RCC->PLLCFGR |= (4 << 0) | (72 << 6) | (0 << 16) | RCC_PLLCFGR_PLLSRC_HSE;
    RCC->CR |= RCC_CR_PLLON;
    while (!(RCC->CR & RCC_CR_PLLRDY));

    FLASH->ACR &= ~FLASH_ACR_LATENCY;
    FLASH->ACR |= FLASH_ACR_LATENCY_2WS;

    RCC->CFGR &= ~(RCC_CFGR_SW | RCC_CFGR_HPRE | RCC_CFGR_PPRE1 | RCC_CFGR_PPRE2);
```

```
38      RCC->CFGR |= RCC_CFGR_SW_PLL | RCC_CFGR_HPRE_DIV1 | RCC_CFGR_PPRE1_DIV2 | RCC_CFGR_PPRE2_DIV1;
39      while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
40
41
42  oid GPIO_Init(void) {
43      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN | RCC_AHB1ENR_GPIOCEN;
44
45      GPIOB->MODER &= ~(0xFFFF);
46      GPIOB->MODER |= 0x5555;
47      GPIOB->OTYPER &= ~(0xFF);
48      GPIOB->OSPEEDR &= ~(0xFFFF);
49      GPIOB->PUPDR &= ~(0xFFFF);
50
51      GPIOC->MODER &= ~(0xFFFF);
52      GPIOC->MODER |= 0x5555;
53      GPIOC->OTYPER &= ~(0x3F);
54      GPIOC->OSPEEDR &= ~(0xFFFF);
55      GPIOC->PUPDR &= ~(0xFFFF);
56
57      GPIOC->ODR |= GPIO_ODR_OD0 | GPIO_ODR_OD1 | GPIO_ODR_OD2;
58      GPIOC->ODR &= ~(GPIO_ODR_OD3 | GPIO_ODR_OD4 | GPIO_ODR_OD5);
59
60
61  oid PPI_Init(void) {
62      GPIOC->ODR |= GPIO_ODR_OD5;
63      for (volatile uint32_t i = 0; i < 1000; i++);
64      GPIOC->ODR &= ~GPIO_ODR_OD5;
65
66      PPI_Write(0x03, 0x80);
67
68
69  oid PPI_Write(uint8_t address, uint8_t data) {
70      if (address & 0x01) {
71          GPIOC->ODR |= GPIO_ODR_OD3;
72      } else {
73          GPIOC->ODR &= ~GPIO_ODR_OD3;

74      }
75      if (address & 0x02) {
76          GPIOC->ODR |= GPIO_ODR_OD4;
77      } else {
78          GPIOC->ODR &= ~GPIO_ODR_OD4;
79      }
80
81      GPIOB->ODR &= ~0xFF;
82      GPIOB->ODR |= data;
83
84      GPIOC->ODR &= ~GPIO_ODR_OD0;
85      GPIOC->ODR &= ~GPIO_ODR_OD1;
86      __NOP();
87      GPIOC->ODR |= GPIO_ODR_OD1;
88      GPIOC->ODR |= GPIO_ODR_OD0;
89  }
90
91  uint8_t PPI_Read(uint8_t address) {
92      uint8_t data;
93
94      GPIOB->MODER &= ~(0xFFFF);
95
96      if (address & 0x01) {
97          GPIOC->ODR |= GPIO_ODR_OD3;
98      } else {
99          GPIOC->ODR &= ~GPIO_ODR_OD3;
100     }
101     if (address & 0x02) {
102         GPIOC->ODR |= GPIO_ODR_OD4;
103     } else {
104         GPIOC->ODR &= ~GPIO_ODR_OD4;
105     }
106
107     GPIOC->ODR &= ~GPIO_ODR_OD0;
108     GPIOC->ODR &= ~GPIO_ODR_OD2;
```

```
    __NOP();
    data = GPIOB->IDR & 0xFF;
    GPIOC->ODR |= GPIO_ODR_OD2;
    GPIOC->ODR |= GPIO_ODR_OD0;

    GPIOB->MODER &= ~(0xFFFF);
    GPIOB->MODER |= 0x5555;

    return data;
}

void Error_Handler(void) {
    while (1) {}
}
```

Giải thích:

a. Hàm main

```
int main(void) {
    SystemCoreClock = 16000000;
    SystemClock_Config();
    GPIO_Init();
    PPI_Init();

    while (1) {
        PPI_Write(0x00, 0xAA);
        for (volatile uint32_t i = 0; i < 1000000; i++);
        PPI_Write(0x00, 0x55);
        for (volatile uint32_t i = 0; i < 1000000; i++);
        PPI_Read(0x01);
    }
}
```

· SystemCoreClock = 16000000: Đặt tần số hệ thống ban đầu (16 MHz, cần khớp với cấu hình thực tế).

· Gọi SystemClock_Config(), GPIO_Init(), PPI_Init() để cấu hình clock, GPIO, và PPI.

· Vòng lặp while(1):

- Ghi giá trị 0xAA (10101010) vào địa chỉ 0x00 (Port A) để điều khiển LED, chờ ~1 giây (vòng lặp for đơn giản).
- Ghi giá trị 0x55 (01010101) vào địa chỉ 0x00, chờ ~1 giây.
- Đọc từ địa chỉ 0x01 (Port B) nhưng không dùng kết quả.

b. Hàm SystemClock_Config()

```
25⊖void SystemClock_Config(void) {
26      RCC->CR |= RCC_CR_HSEON;
27      while (!(RCC->CR & RCC_CR_HSERDY));
28
29      RCC->PLLCFGR &= ~(0x3FFF);
30      RCC->PLLCFGR |= (4 << 0) | (72 << 6) | (0 << 16) | RCC_PLLCFGR_PLLSRC_HSE;
31      RCC->CR |= RCC_CR_PLLON;
32      while (!(RCC->CR & RCC_CR_PLLRDY));
33
34      FLASH->ACR &= ~FLASH_ACR_LATENCY;
35      FLASH->ACR |= FLASH_ACR_LATENCY_2WS;
36
37      RCC->CFGR &= ~(RCC_CFGR_SW | RCC_CFGR_HPRE | RCC_CFGR_PPRE1 | RCC_CFGR_PPRE2);
38      RCC->CFGR |= RCC_CFGR_SW_PLL | RCC_CFGR_HPRE_DIV1 | RCC_CFGR_PPRE1_DIV2 | RCC_CFGR_PPRE2_DIV1
39      while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
40  }
```

· **Mục đích**: Cấu hình clock hệ thống sử dụng HSE và PLL.

· **Giải thích**:

- Bật HSE: RCC->CR |= RCC_CR_HSEON đặt bit để kích hoạt dao động ngoài, chờ sẵn sàng (RCC_CR_HSERDY).
- Cấu hình PLL:
  - RCC->PLLCFGR &= ~(0x3FFF) xóa các bit PLLM, PLLN, PLLP.
  - |= (4 << 0) | (72 << 6) | (0 << 16) | RCC_PLLCFGR_PLLSRC_HSE đặt PLLM=4, PLLN=72, PLLP=2, nguồn HSE.
- Bật PLL: RCC->CR |= RCC_CR_PLLON, chờ sẵn sàng (RCC_CR_PLLRDY).
- Cấu hình Flash: FLASH->ACR &= ~FLASH_ACR_LATENCY; |= FLASH_ACR_LATENCY_2WS đặt độ trễ 2 chu kỳ chờ.
- Chọn clock hệ thống:
  - RCC->CFGR &= ~... xóa các bit chọn nguồn clock và bộ chia.
  - |= RCC_CFGR_SW_PLL | ... chọn PLL làm SYSCLK, AHB=/1, APB1=/2, APB2=/1.
  - Chờ PLL được chọn (RCC_CFGR_SWS_PLL).

c. Hàm GPIO_Init()

```
42⊖void GPIO_Init(void) {
43      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN | RCC_AHB1ENR_GPIOCEN;
44
45      GPIOB->MODER &= ~(0xFFFF);
46      GPIOB->MODER |= 0x5555;
47      GPIOB->OTYPER &= ~(0xFF);
48      GPIOB->OSPEEDR &= ~(0xFFFF);
49      GPIOB->PUPDR &= ~(0xFFFF);
50
51      GPIOC->MODER &= ~(0xFFFF);
52      GPIOC->MODER |= 0x5555;
53      GPIOC->OTYPER &= ~(0x3F);
54      GPIOC->OSPEEDR &= ~(0xFFFF);
55      GPIOC->PUPDR &= ~(0xFFFF);
56
57      GPIOC->ODR |= GPIO_ODR_OD0 | GPIO_ODR_OD1 | GPIO_ODR_OD2;
58      GPIOC->ODR &= ~(GPIO_ODR_OD3 | GPIO_ODR_OD4 | GPIO_ODR_OD5);
59  }
```

· **Mục đích**: Cấu hình GPIOB và GPIOC làm output để điều khiển PPI.

· **Giải thích**:

- Bật clock: RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN | RCC_AHB1ENR_GPIOCEN kích hoạt GPIOB và GPIOC.
- Cấu hình GPIOB (PB0-PB7):
  - GPIOB->MODER &= ~(0xFFFF); |= 0x5555 đặt PB0-PB7 làm output (01 cho mỗi chân).
  - GPIOB->OTYPER &= ~(0xFF) đặt push-pull.
  - GPIOB->OSPEEDR &= ~(0xFFFF) đặt tốc độ thấp.
  - GPIOB->PUPDR &= ~(0xFFFF) tắt pull-up/down.
- Cấu hình GPIOC (PC0-PC5): Tương tự GPIOB, đặt PC0-PC5 làm output.
- Trạng thái ban đầu:
  - GPIOC->ODR |= GPIO_ODR_OD0 | GPIO_ODR_OD1 | GPIO_ODR_OD2 đặt CS=1, WR=1, RD=1.
  - GPIOC->ODR &= ~(GPIO_ODR_OD3 | GPIO_ODR_OD4 | GPIO_ODR_OD5) đặt A0=0, A1=0, RESET=0.

d. Hàm PPI_Init()

```
61 void PPI_Init(void) {
62     GPIOC->ODR |= GPIO_ODR_OD5;
63     for (volatile uint32_t i = 0; i < 1000; i++);
64     GPIOC->ODR &= ~GPIO_ODR_OD5;
65
66     PPI_Write(0x03, 0x80);
67 }
```

· **Mục đích**: Khởi tạo PPI (8255) bằng cách reset và cấu hình các cổng.

· **Giải thích**:

- Reset PPI:
  - GPIOC->ODR |= GPIO_ODR_OD5 đặt RESET=1.
  - Chờ ngắn (vòng lặp for).
  - GPIOC->ODR &= ~GPIO_ODR_OD5 đặt RESET=0.
- Gọi PPI_Write(0x03, 0x80) để ghi giá trị 0x80 vào thanh ghi điều khiển (địa chỉ 0x03), cấu hình Port A là output, Port B là input, Port C là output.

e. Hàm PPI_Write(uint8_t address, uint8_t data)

```
69  void PPI_Write(uint8_t address, uint8_t data) {
70      if (address & 0x01) {
71          GPIOC->ODR |= GPIO_ODR_OD3;
72      } else {
73          GPIOC->ODR &= ~GPIO_ODR_OD3;
74      }
75      if (address & 0x02) {
76          GPIOC->ODR |= GPIO_ODR_OD4;
77      } else {
78          GPIOC->ODR &= ~GPIO_ODR_OD4;
79      }
80
81      GPIOB->ODR &= ~0xFF;
82      GPIOB->ODR |= data;
83
84      GPIOC->ODR &= ~GPIO_ODR_OD0;
85      GPIOC->ODR &= ~GPIO_ODR_OD1;
86      __NOP();
87      GPIOC->ODR |= GPIO_ODR_OD1;
88      GPIOC->ODR |= GPIO_ODR_OD0;
89  }
90
91  uint8_t PPI_Read(uint8_t address) {
92      uint8_t data;
93
94      GPIOB->MODER &= ~(0xFFFF);
95
96      if (address & 0x01) {
97          GPIOC->ODR |= GPIO_ODR_OD3;
98      } else {
99          GPIOC->ODR &= ~GPIO_ODR_OD3;
100     }

101     if (address & 0x02) {
102         GPIOC->ODR |= GPIO_ODR_OD4;
103     } else {
104         GPIOC->ODR &= ~GPIO_ODR_OD4;
105     }
106
107     GPIOC->ODR &= ~GPIO_ODR_OD0;
108     GPIOC->ODR &= ~GPIO_ODR_OD2;
109     __NOP();
110     data = GPIOB->IDR & 0xFF;
111     GPIOC->ODR |= GPIO_ODR_OD2;
112     GPIOC->ODR |= GPIO_ODR_OD0;
113
114     GPIOB->MODER &= ~(0xFFFF);
115     GPIOB->MODER |= 0x5555;
116
117     return data;
118 }
119
```

PPI Write:

· **Mục đích**: Ghi dữ liệu vào một cổng hoặc thanh ghi điều khiển của PPI.

· **Giải thích**:

- Đặt địa chỉ (A0, A1):
  - Nếu address & 0x01, GPIOC->ODR |= GPIO_ODR_OD3 đặt A0=1, ngược lại &= đặt A0=0.
  - Nếu address & 0x02, GPIOC->ODR |= GPIO_ODR_OD4 đặt A1=1, ngược lại &= đặt A1=0.
- Ghi dữ liệu:
  - GPIOB->ODR &= ~0xFF xóa 8 bit thấp của GPIOB.
  - GPIOB->ODR |= data ghi giá trị data vào 8 bit thấp.
- Chu kỳ ghi:
  - GPIOC->ODR &= ~GPIO_ODR_OD0 đặt CS=0.
  - GPIOC->ODR &= ~GPIO_ODR_OD1 đặt WR=0.
  - __NOP() tạo độ trễ ngắn.
  - GPIOC->ODR |= GPIO_ODR_OD1 đặt WR=1.
  - GPIOC->ODR |= GPIO_ODR_OD0 đặt CS=1.

PPI READ:

· **Mục đích**: Đọc dữ liệu từ một cổng của PPI.

· **Giải thích**:

- Chuyển GPIOB sang input: GPIOB->MODER &= ~(0xFFFF) đặt PB0-PB7 thành input.
- Đặt địa chỉ (A0, A1): Tương tự PPI_Write, dùng |= và &= để đặt A0, A1 dựa trên address.
- Chu kỳ đọc:
  - GPIOC->ODR &= ~GPIO_ODR_OD0 đặt CS=0.
  - GPIOC->ODR &= ~GPIO_ODR_OD2 đặt RD=0.
  - __NOP() tạo độ trễ ngắn.
  - data = GPIOB->IDR & 0xFF đọc 8 bit thấp từ GPIOB.
  - GPIOC->ODR |= GPIO_ODR_OD2 đặt RD=1.
  - GPIOC->ODR |= GPIO_ODR_OD0 đặt CS=1.

- Khôi phục output:
    - GPIOB->MODER &= ~(0xFFFF); |= 0x5555 đặt lại PB0-PB7 thành output.
- Trả về data.

Video :
https://drive.google.com/file/d/1z_u_ky5DNPCDkOQsitFkYCsxgkpjthLl/view?usp=sharing

4. Kiểm tra, đánh giá hiệu quả hệ thống và hiệu chỉnh

a. Kiểm tra và đánh giá hệ thống

- Các kết nối giữa vi điều khiển STM32F401RE và vi mạch mở rộng 8255A đã được thiết lập chính xác theo sơ đồ nguyên lý:

+ Bus dữ liệu D0–D7 của 8255A nối với các chân PB0–PB7 của STM32.

+ Các tín hiệu điều khiển (CS, WR, RD, A0, A1, RESET) nối tương ứng với các chân PC0–PC5.

- Các LED được nối với Port A của 8255A thông qua điện trở hạn dòng 220Ω để đảm bảo an toàn cho linh kiện.

- Quá trình mô phỏng trên Proteus cho thấy không có hiện tượng chập, sai điện áp hoặc xung đột tín hiệu.

- Toàn bộ hệ thống sử dụng nguồn 5V và được nối chung GND để đảm bảo mạch hoạt động ổn định.

- Chương trình STM32 đã thực hiện ghi lệnh điều khiển 0x80 vào địa chỉ 0x03 để thiết lập chế độ Mode 0, với Port A là đầu ra.

- Dữ liệu mẫu 0xAA và 0x55 được ghi liên tục vào Port A trong vòng lặp while(1). LED phản hồi chính xác

- Các mức logic tại Port A được kiểm tra bằng Logic Probe cho thấy dữ liệu được ghi ra đúng theo giá trị đã lập trình.

- Dữ liệu ghi ra Port A được LED phản hồi ngay lập tức trong mô phỏng, không có độ trễ đáng kể.

- Mỗi bit dữ liệu điều khiển chính xác một LED tương ứng (D0 điều khiển LED0, ..., D7 điều khiển LED7), đảm bảo tính nhất quán trong điều khiển đầu ra.

- **Kết luận:** Hệ thống ghép nối STM32F401RE với 8255A ở chế độ Mode 0 đã hoạt động đúng như mong đợi. Việc ghi dữ liệu từ vi điều khiển đến Port A của 8255A được thực hiện chính xác, các LED phản hồi tương ứng theo từng bit dữ liệu. Giao tiếp giữa STM32 và 8255A ổn định, không xảy ra lỗi logic hay xung đột tín hiệu trong mô phỏng. Tốc độ phản hồi nhanh và trạng thái LED thay đổi đúng thời điểm cho thấy hệ thống có độ đồng bộ tốt. Qua đó có thể khẳng định hệ thống thiết kế đạt yêu cầu về kỹ thuật và chức năng theo đề bài đặt ra.

b. Hiệu chỉnh hệ thống

- Trong quá trình thiết kế và mô phỏng hệ thống STM32F401RE giao tiếp với 8255A ở chế độ Mode 0, một số hiệu chỉnh đã được thực hiện nhằm đảm bảo mạch hoạt động ổn định, dữ liệu được truyền đúng và các thiết bị ngoại vi (như LED) phản hồi chính xác.

Hiệu chỉnh cấu hình

- Kiểm tra và xác nhận lệnh điều khiển (Control Word) đã được gửi chính xác đến thanh ghi điều khiển của 8255A.

- Trong chế độ Mode 0, giá trị 0x80 được sử dụng để cấu hình:

+ Port A là đầu ra (Output).

+ Port B là đầu vào (Input) – không sử dụng trong bài này.

+ Port C là đầu ra (Output).

- Đảm bảo rằng dữ liệu được ghi đến đúng địa chỉ của Port A (A1 = 0, A0 = 0), và địa chỉ thanh ghi điều khiển là 0x03 (A1 = 1, A0 = 1).

- Nếu LED không sáng hoặc không đúng mẫu dữ liệu, cần kiểm tra lại trình tự thiết lập A0, A1 và kích hoạt các tín hiệu CS và WR để tạo chu kỳ ghi hợp lệ.

Hiệu chỉnh tín hiệu phần cứng

- LED được nối với Port A thông qua điện trở hạn dòng 220Ω. Trong trường hợp LED sáng yếu, có thể giảm điện trở xuống 220Ω để tăng độ sáng, tuy nhiên cần đảm bảo dòng không vượt quá giới hạn cho phép của LED (thường không quá 20mA).

- Các dây điều khiển như CS, WR, RD, RESET cũng cần được nối chính xác đến các chân tương ứng của STM32 (thường là PC0–PC5), và được điều khiển đúng mức logic.

Hiệu chỉnh logic mô phỏng

- Trong mô phỏng Proteus, nếu LED không phản hồi dữ liệu ngay lập tức, cần chèn thêm **lệnh trễ ngắn (**__NOP() **hoặc** delay_ms()**)** giữa các lần ghi liên tiếp để đảm bảo 8255A có đủ thời gian nhận dữ liệu.

- Kiểm tra các kết nối trong sơ đồ Proteus để đảm bảo không có dây nối sai hoặc ngắt mạch.

C. PPI 8255A mode 1.

I. Yêu cầu của bài toán

1. Mục tiêu của bài toán

Mục tiêu của bài toán là thiết lập giao tiếp sử dụng vi mạch PPI 8255A ở Mode 1 (Output) trên Port A để điều khiển các LED dựa trên trạng thái của các công tắc (switch). Hệ thống được mô phỏng trên phần mềm Proteus, trong đó các công tắc được sử dụng để gửi tín hiệu điều khiển, và các LED được kết nối với Port A để hiển thị trạng thái đầu ra.

2. Các yêu cầu cụ thể

Giao tiếp với PPI 8255A: Sử dụng PPI 8255A ở Mode 1 (Output) trên Port A để xuất dữ liệu điều khiển LED.

- Điều khiển LED:

  Các LED được kết nối với các chân của Port A (PA0 đến PA7).

  Trạng thái LED (bật/tắt) được điều khiển bởi dữ liệu đầu ra từ Port A, dựa trên trạng thái của các công tắc.

- Sử dụng công tắc:

  Các công tắc được kết nối để mô phỏng tín hiệu điều khiển
  Công tắc có thể được kết nối với Port B hoặc Port C để đọc trạng thái, sau đó sử dụng để ghi dữ liệu ra Port A.

Mô phỏng trên Proteus: Hệ thống được thiết kế và kiểm tra trong môi trường Proteus, đảm bảo hoạt động ổn định và trực quan.

Dễ dàng mở rộng: Hệ thống cho phép mở rộng để điều khiển thêm các LED hoặc tích hợp các thiết bị khác bằng cách sử dụng Port B hoặc Port C.

3. Phạm vi và giới hạn của hệ thống

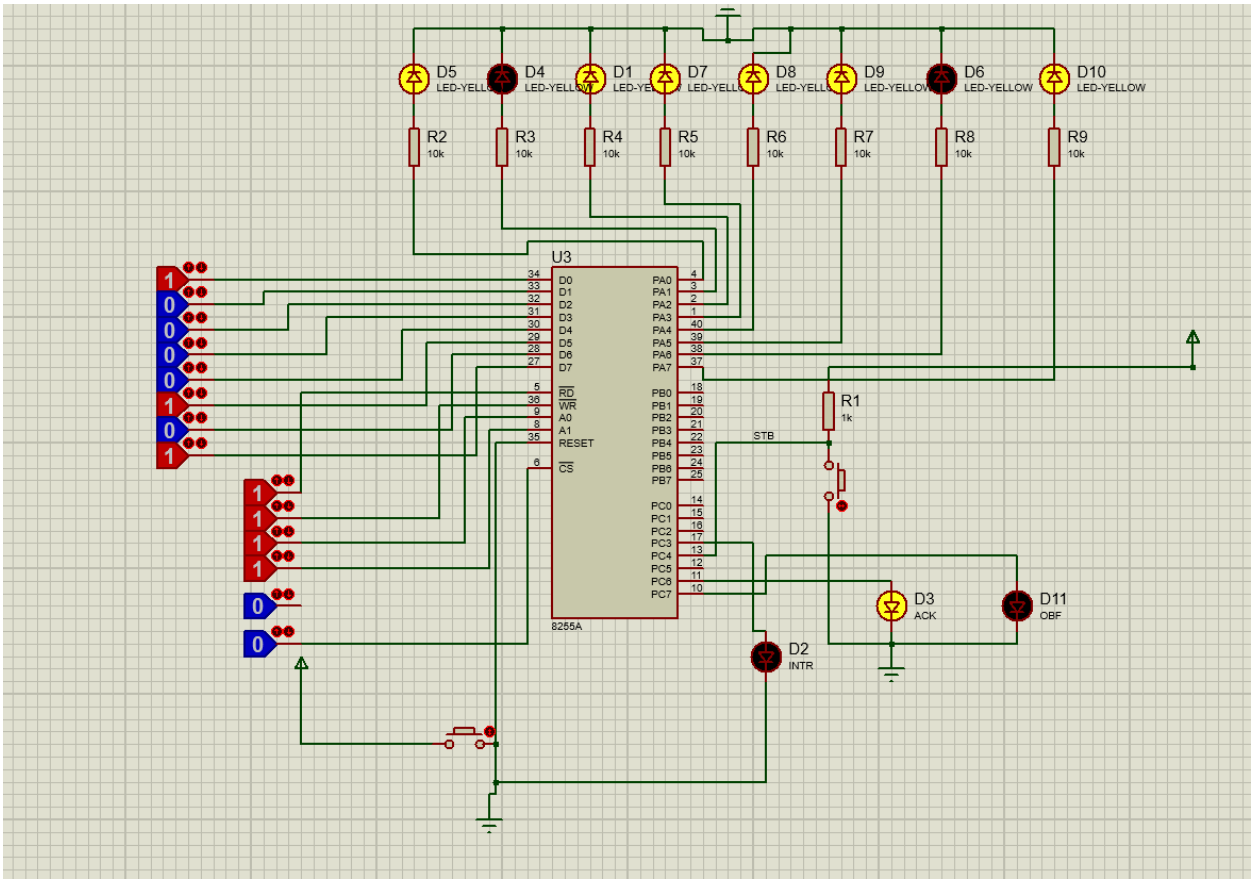
Hệ thống sử dụng PPI 8255A ở Mode 1 (Output) trên Port A, với các chân điều khiển (STB, IBF, INTR) được cấu hình phù hợp.

Chỉ sử dụng các công tắc và LED để mô phỏng giao tiếp đơn giản, không yêu cầu giao tiếp với vi điều khiển hoặc thiết bị phức tạp.

Mô phỏng trên Proteus tập trung vào việc kiểm tra trạng thái logic (0/1) của các chân Port A để điều khiển LED.

II. Phần thực thi hệ thống

1. Thiết kế phần cứng



Hệ thống được thiết kế trên Proteus với các thành phần sau:

- Vi mạch PPI 8255A:

  Port A (PA0–PA7) được cấu hình làm cổng đầu ra (Output) ở Mode 1.

  Port B hoặc Port C (tùy thiết kế) được cấu hình làm cổng đầu vào để đọc trạng thái công tắc.

  Các chân điều khiển (RD, WR, CS, A0, A1, RESET) được kết nối với mạch logic mô phỏng để đọc/ghi dữ liệu.

- Kết nối LED:

Tám LED được kết nối với các chân PA0–PA7 của Port A thông qua điện trở hạn dòng (~330Ω).

Anode của LED nối với Port A, cathode nối với GND (hoặc ngược lại tùy thiết kế).

- Kết nối công tắc:

  Tám công tắc được kết nối với (D0–D7) để mô phỏng tín hiệu đầu vào.

  Mỗi công tắc được nối với nguồn 5V qua điện trở kéo lên (pull-up, ~10kΩ) hoặc GND qua điện trở kéo xuống (pull-down) để tạo trạng thái logic rõ ràng (1 hoặc 0).

Nguồn điện:

PPI 8255A được cấp nguồn 5V, với GND chung cho tất cả các thành phần.

2. Cấu hình phần mềm

Hệ thống được mô phỏng trên Proteus, sử dụng logic điều khiển đơn giản để cấu hình và vận hành PPI 8255A.

- Cấu hình PPI 8255A
  Control Word:

  Ghi vào thanh ghi điều khiển (Control Register) để cấu hình Mode 1 Output cho Port A.

  Ghi giá trị 0x88 vào Control Register:

  - Bit 7 = 1: Chọn chế độ cấu hình.
  - Bit 6–5 = 00: Mode 0 (cơ bản, nhưng sẽ điều chỉnh cho Mode 1 sau).
  - Bit 4 = 0: Port A là Output.
  - Bit 3 = X: Không sử dụng Port C upper.
  - Bit 2 = 00: Mode 0 cho Port B (hoặc Mode 1 nếu cần).
  - Bit 1 = 1: Port B là Input (để đọc công tắc).
  - Bit 0 = X: Không sử dụng Port C lower.

Để kích hoạt Mode 1 Output cho Port A, sử dụng Bit Set/Reset để cấu hình các chân điều khiển như OBF và INTR.

Quy trình hoạt động:

Đọc trạng thái công tắc từ Port B (hoặc Port C).

Ghi dữ liệu tương ứng vào Port A để điều khiển LED.

- Logic điều khiển trong Proteus

Đọc công tắc:

Khi công tắc trên (PD0–PD7) thay đổi trạng thái, dữ liệu được đọc
Ví dụ: Nếu PD0 = 1, PD1 = 0, dữ liệu đọc được là 0x01.

Ghi ra Port A:

Dữ liệu được sao chép trực tiếp vào Port A.

Ví dụ: Ghi 0x01 vào Port A sẽ bật LED trên PA0 và tắt các LED khác.

Chân điều khiển Mode 1:

OBF (Output Buffer Full): Chân PC7 được giám sát để xác nhận dữ liệu đã được ghi ra Port A.

ACK (Acknowledge): Chân PC6 có thể được mô phỏng bằng logic để xác nhận dữ liệu đã được đọc bởi thiết bị ngoại vi (LED).

INTR (Interrupt): Chân PC3 có thể được sử dụng để báo hiệu hoàn tất truyền dữ liệu.

3. Kiểm tra, đánh giá hiệu quả hệ thống và hiệu chỉnh

3.1. Kiểm tra hệ thống

- Kiểm tra phần cứng (trong Proteus):

Xác minh kết nối giữa PPI 8255A, LED (Port A), và công tắc.

Đảm bảo các điện trở hạn dòng (~330Ω) cho LED và điện trở kéo lên/xuống (~10kΩ) cho công tắc được cấu hình đúng.

Kiểm tra mức logic trên các chân Port A (0V hoặc 5V) khi ghi dữ liệu.

- Kiểm tra phần mềm (logic mô phỏng):

  Ghi giá trị Control Word (0x88 hoặc tương tự) vào thanh ghi điều khiển để cấu hình Mode 1 Output cho Port A.

  Thay đổi trạng thái công tắc trên Port B (hoặc Port C) và quan sát trạng thái LED trên Port A:

  Kiểm tra các chân điều khiển Mode 1 (OBF, ACK, INTR) bằng Probe trong Proteus để đảm bảo hoạt động đúng:

    OBF chuyển sang mức thấp khi dữ liệu được ghi vào Port A.

    INTR chuyển sang mức cao khi giao tiếp hoàn tất.

- Kiểm tra tính đồng bộ:

  Đảm bảo dữ liệu từ công tắc được truyền đến LED mà không có độ trễ đáng kể trong mô phỏng.

  Kiểm tra tính nhất quán: Mỗi công tắc điều khiển chính xác một LED tương ứng.

3.2. Đánh giá hiệu quả hệ thống

- Độ ổn định của tín hiệu:

  Các chân Port A xuất dữ liệu ổn định, LED bật/tắt chính xác theo trạng thái công tắc.

  Không có hiện tượng nhiễu logic trong mô phỏng (ví dụ: LED sáng/tắt không đúng).

- Tốc độ và độ chính xác của điều khiển LED:

  LED phản hồi ngay lập tức khi công tắc thay đổi trạng thái, phù hợp với tốc độ mô phỏng của Proteus.

  Dữ liệu từ Port B được sao chép chính xác sang Port A, đảm bảo mỗi công tắc điều khiển đúng LED tương ứng.

- Khả năng mở rộng:

Hệ thống có thể mở rộng bằng cách sử dụng Port B hoặc Port C để điều khiển thêm các thiết bị hoặc tích hợp các chế độ khác của 8255A (Mode 0 hoặc Mode 2).

3.3. Hiệu chỉnh và tối ưu hóa

Điều chỉnh cấu hình 8255A:

Kiểm tra lại Control Word để đảm bảo Port A được cấu hình chính xác ở Mode 1 Output.

Nếu LED không phản hồi đúng, kiểm tra trạng thái các chân điều khiển (OBF, ACK) và điều chỉnh logic mô phỏng.

Cải thiện tín hiệu:

Đảm bảo các điện trở kéo lên/xuống cho công tắc được thiết lập đúng để tránh trạng thái logic không xác định.

Nếu LED không sáng đủ, giảm giá trị điện trở hạn dòng (ví dụ: từ 330Ω xuống 220Ω, nhưng đảm bảo dòng điện an toàn).

Kiểm tra lỗi mô phỏng:

Kiểm tra các kết nối trong sơ đồ Proteus để đảm bảo không có dây nối sai hoặc ngắt mạch.

Nếu Proteus chạy chậm, giảm tần số mô phỏng hoặc tối ưu sơ đồ bằng cách loại bỏ các thành phần không cần thiết.

Tối ưu logic điều khiển:

Thêm thời gian trễ giả lập để mô phỏng thực tế hơn quá trình đọc/ghi dữ liệu.

Sử dụng các chân INTR để mô phỏng ngắt, cải thiện khả năng phản hồi nếu tích hợp với vi điều khiển trong tương lai.

4. Video demo

https://drive.google.com/file/d/1FTC4-b9vtGq5hZm1HajwCURIu2mfmR1Y/view?usp=drive_link

D. PPI 8255A mode 2

I. Yêu cầu của bài toán

1. Mục tiêu của bài toán

Mục tiêu của bài toán là thiết lập giao tiếp sử dụng vi mạch PPI 8255A ở Mode 2 (Bidirectional Bus) trên Port A để thực hiện truyền/nhận dữ liệu hai chiều. Hệ thống được mô phỏng trên phần mềm Proteus, trong đó Port A vừa xuất dữ liệu để điều khiển LED (dựa trên trạng thái công tắc) vừa nhận dữ liệu từ thiết bị ngoại vi giả lập để điều khiển LED khác. Các tín hiệu điều khiển như OBF, IBF, và INTR được quan sát để xác minh giao tiếp handshake.

2. Các yêu cầu cụ thể

Giao tiếp với PPI 8255A: Sử dụng Mode 2 trên Port A để thực hiện giao tiếp hai chiều.

Điều khiển LED:

Gửi dữ liệu (Output): Port A điều khiển 8 LED (PA0–PA7) dựa trên trạng thái trong bus số liệu.

Nhận dữ liệu (Input): bus số liệu nhận dữ liệu từ PORT A và điều khiển các led ở PORTB tương

Quan sát tín hiệu điều khiển:

Sử dụng LED để quan sát các tín hiệu OBF (Output Buffer Full), IBF (Input Buffer Full), và INTR (Interrupt) trên Port C.

Mô phỏng trên Proteus: Hệ thống được thiết kế và kiểm tra trong Proteus, đảm bảo giao tiếp hai chiều hoạt động ổn định.

Dễ dàng mở rộng: Hệ thống cho phép tích hợp thêm thiết bị ngoại vi hoặc điều khiển thêm LED bằng cách sử dụng Port B và Port C.

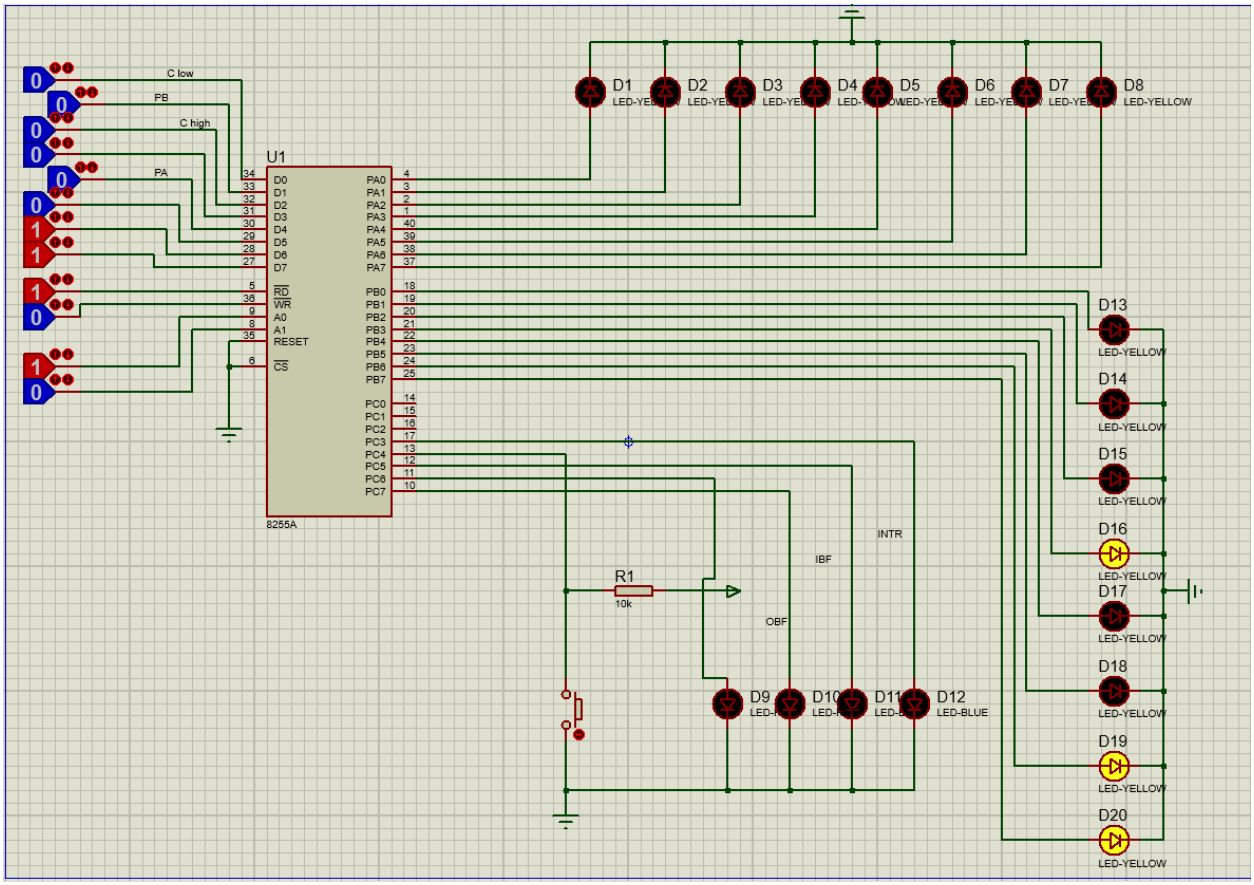
3. Phạm vi và giới hạn của hệ thống

Hệ thống sử dụng Mode 2 trên Port A, với Port B ở Mode 0 (Input) và Port C hỗ trợ tín hiệu điều khiển.

Giao tiếp hai chiều được mô phỏng bằng cách sử dụng công tắc và mạch logic để giả lập thiết bị ngoại vi.

II. Phần thực thi hệ thống

1. Thiết kế phần cứng



Hệ thống được thiết kế trên Proteus với các thành phần sau:

- Vi mạch PPI 8255A:

  Port A (PA0–PA7): Kết nối với 8 LED (D1–D8, màu vàng) để xuất dữ liệu.

  Port B (PB0–PB7): Kết nối với 8 công tắc để nhập dữ liệu.

  Port C:

  - PC7 (OBF): Kết nối với LED D10 (LED_RED) để quan sát tín hiệu OBF.

- PC5 (INTR): Kết nối với LED D12 (LED_BLUE) để quan sát tín hiệu ngắt.
- PC4 (IBF): Kết nối với LED D12 (LED_RED) để quan sát tín hiệu IBF.

Chân điều khiển (RD, WR, CS, A0, A1, RESET) được kết nối với mạch logic mô phỏng để đọc/ghi dữ liệu.

- Kết nối LED:

  8 LED trên Port A, mỗi LED có điện trở hạn dòng (~220Ω), anode nối với Port A, cathode nối GND.

  LED quan sát tín hiệu (OBF, IBF, INTR) được nối với PC7, PC4, PC5 qua điện trở.

- Nguồn điện:

  Cấp nguồn 5V cho 8255A, với GND chung cho tất cả các thành phần.

2. Cấu hình phần mềm

Cấu hình PPI 8255A

- Control Word:

  Cấu hình Mode 2 cho Port A, Mode 0 cho Port B (Input), và Port C lower (PC0–PC2) làm Output:

  - Bit 7 = 1: Chế độ cấu hình.
  - Bit 6–5 = 10: Mode 2 cho Port A.
  - Bit 4 = X: Không cần thiết.
  - Bit 3 = 0: Port C upper tự động điều khiển bởi Mode 2.
  - Bit 2 = 0: Mode 0 cho Port B.
  - Bit 1 = 1: Port B là Input.
  - Bit 0 = 0: Port C lower là Output.

  Giá trị Control Word: 0x90.

- Ghi Control Word:

  Ghi giá trị 0x90 vào thanh ghi điều khiển (địa chỉ: A1 = 1, A0 = 1) bằng mạch logic hoặc công cụ mô phỏng trong Proteus.

Logic điều khiển trong Proteus

- Gửi dữ liệu (Output qua Port A):

  Đọc trạng thái công tắc từ Port B (PB0–PB7).

  Ví dụ: Nếu PB0 = 1, PB1 = 1, dữ liệu đọc được là 0x03.

  Ghi dữ liệu 0x03 vào Port A → LED D1 và D2 sáng, các LED còn lại (D3–D8) tắt.

  Quan sát OBF (PC7): LED D10 (OBF) tắt khi dữ liệu được ghi, sáng lại khi ACK (PC6) được kích hoạt.

- Nhận dữ liệu (Input qua Port A):

  Mô phỏng thiết bị ngoại vi bằng cách đặt dữ liệu lên Port A (ví dụ: 0x05, tức PA0 = 1, PA2 = 1).

  Kích hoạt STB (PC4) để báo dữ liệu sẵn sàng → IBF (PC4) sáng (LED D12 đỏ).

  Đọc dữ liệu từ Port A → Dữ liệu 0x05 được sử dụng để điều khiển PC0–PC2:

  - PC0 = 1 (D9 sáng), PC1 = 0 (D10 tắt), PC2 = 1 (D11 sáng).

  Quan sát INTR (PC5): LED D12 (màu xanh) sáng khi dữ liệu được đọc.

- Mô phỏng tín hiệu điều khiển:

  Sử dụng công tắc hoặc mạch logic để mô phỏng ACK (PC6) và STB (PC4).

  ACK: Kéo xuống mức thấp để xác nhận dữ liệu đã được nhận → OBF trở lại mức cao.

  STB: Kéo xuống mức thấp để báo dữ liệu sẵn sàng trên Port A → IBF chuyển sang mức cao.

3. Kiểm tra, đánh giá hiệu quả hệ thống và hiệu chỉnh

3.1. Kiểm tra hệ thống

- Kiểm tra phần cứng:

  Xác minh kết nối giữa 8255A, LED (Port A), và tín hiệu vào ra theo cả 2 chiều

  Đảm bảo các điện trở hạn dòng (~220Ω) cho LED và điện trở kéo xuống (~10kΩ) cho công tắc được thiết lập đúng.

  Kiểm tra mức logic trên các chân Port A (0V hoặc 5V) khi ghi/đọc dữ liệu.

- Kiểm tra logic điều khiển:

  Ghi Control Word (0x90) để cấu hình Mode 2.

  Gửi dữ liệu:

  - Nhấn công tắc PB0 và PB1 → LED D1 và D2 sáng (Port A = 0x03).
  - Quan sát OBF (PC7): LED D10 tắt khi ghi dữ liệu, sáng lại khi kích hoạt ACK.

  Nhận dữ liệu:

  - Đặt dữ liệu 0x05 lên Port A, kích hoạt STB → IBF (LED D12 đỏ) sáng.
  - Đọc dữ liệu → LED D9 và D11 (PC0, PC2) sáng, IBF tắt.
  - INTR (LED D12 xanh) sáng khi dữ liệu được đọc.

- Kiểm tra tín hiệu điều khiển:

  Đảm bảo OBF, IBF, và INTR hoạt động đúng:

  - OBF thấp khi ghi dữ liệu, cao khi ACK được kích hoạt.
  - IBF cao khi dữ liệu được đặt lên Port A, thấp sau khi đọc.
  - INTR cao khi hoàn tất gửi/nhận dữ liệu.

3.2. Đánh giá hiệu quả hệ thống

- Giao tiếp hai chiều:

Port A gửi dữ liệu thành công: LED D1–D8 phản hồi chính xác với trạng thái công tắc trên Port B.

Port A nhận dữ liệu thành công: LED D9–D11 trên PC0–PC2 phản hồi đúng với dữ liệu giả lập (0x05).

- Tín hiệu điều khiển:

  OBF, IBF, và INTR hoạt động ổn định, đảm bảo giao tiếp handshake giữa 8255A và thiết bị ngoại vi giả lập.

- Tốc độ và độ chính xác:

  LED trên Port A và Port C phản hồi ngay lập tức khi dữ liệu được gửi/nhận, phù hợp với tốc độ mô phỏng.

  Không có hiện tượng mất dữ liệu hoặc nhiễu trong mô phỏng.

3.3. Hiệu chỉnh và tối ưu hóa

- Điều chỉnh cấu hình 8255A:

  Đảm bảo Control Word (0x90) được ghi đúng vào thanh ghi điều khiển.

  Nếu LED không phản hồi, kiểm tra trạng thái OBF/IBF và điều chỉnh logic mô phỏng ACK/STB.

- Cải thiện tín hiệu:

  Đảm bảo các công tắc trên Port B có điện trở kéo xuống để tránh trạng thái logic không xác định.

- Kiểm tra lỗi mô phỏng:

  Kiểm tra các kết nối trong sơ đồ Proteus để đảm bảo không có dây nối sai hoặc ngắt mạch.

  Nếu Proteus chạy chậm, giảm tần số mô phỏng hoặc loại bỏ thành phần không cần thiết.

- Tối ưu logic điều khiển:

  Thêm thời gian trễ giả lập (nếu cần) để mô phỏng thực tế hơn quá trình đọc/ghi dữ liệu.

Sử dụng INTR để mô phỏng ngắt, cải thiện khả năng phản hồi nếu tích hợp với vi điều khiển trong tương lai.

4. Video Demo

https://drive.google.com/file/d/13Gk_g6N8WNYp0_V3beAya0nPDD0J5iF0/view?usp=drive_link

Bảng đánh giá thành viên

Thành viên	Công việc	Đóng góp
22022118 Phạm Văn Duy	Nghiên cứu lí thuyết, viết code và mô phỏng **bài C: PPI 8255A Mode 1**; hỗ trợ viết tài liệu chung; viết báo cáo Word cho bài C.	25%
2202103 Ngô Đức Hiếu	Nghiên cứu lí thuyết, viết code và mô phỏng **bài A: SPI**; hỗ trợ các bài khác; viết báo cáo Word cho bài B.	25%
2202217 Nguyễn Quốc Toản	Nghiên cứu lí thuyết, viết code và mô phỏng **bài B: PPI 8255A Mode 0**; hỗ trợ viết tài liệu chung; viết báo cáo Word cho bài C.	25%
2202145 Tạ Đình Kiên	Nghiên cứu lí thuyết, viết code và mô phỏng **bài D: PPI 8255A Mode 2**; hỗ trợ kiểm tra tổng hợp; viết báo cáo Word cho bài D.	25%