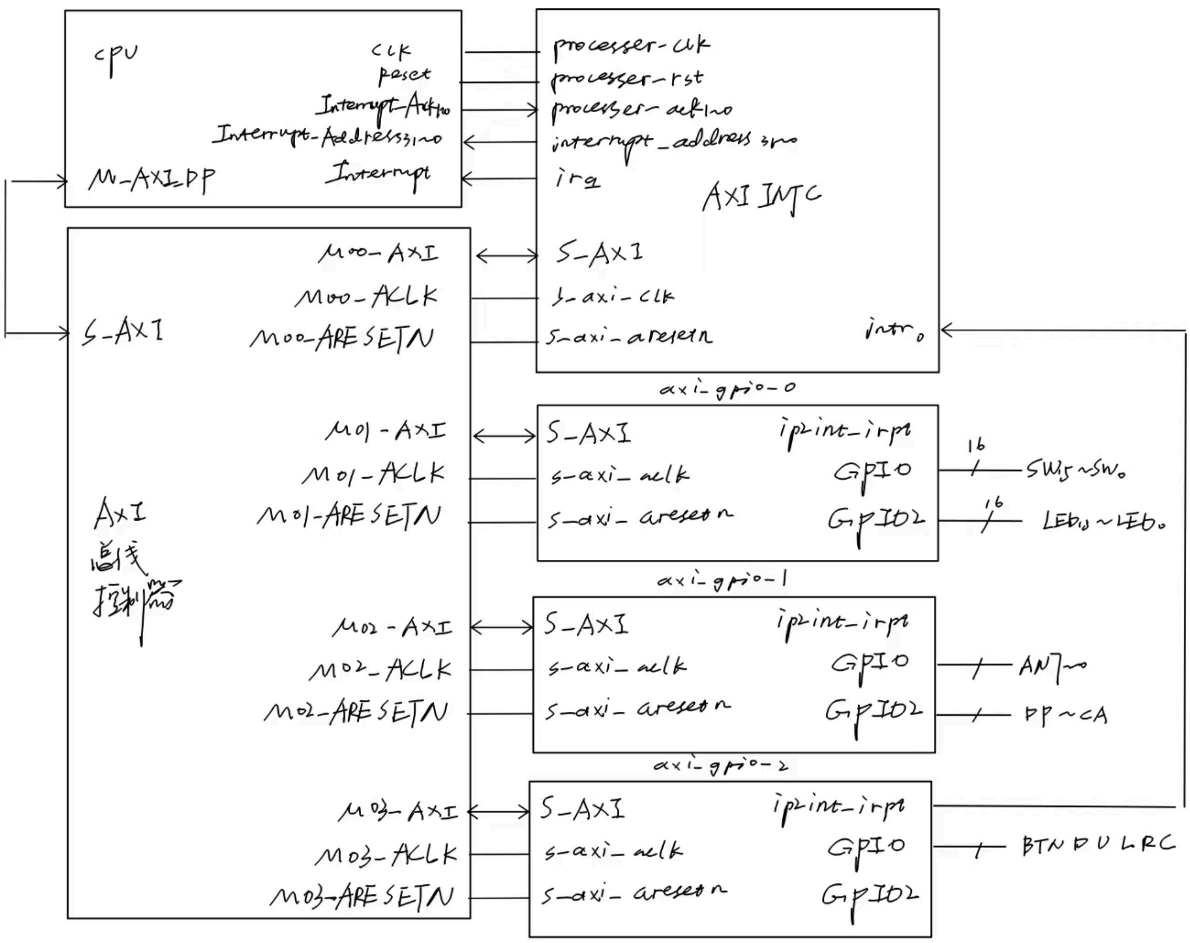
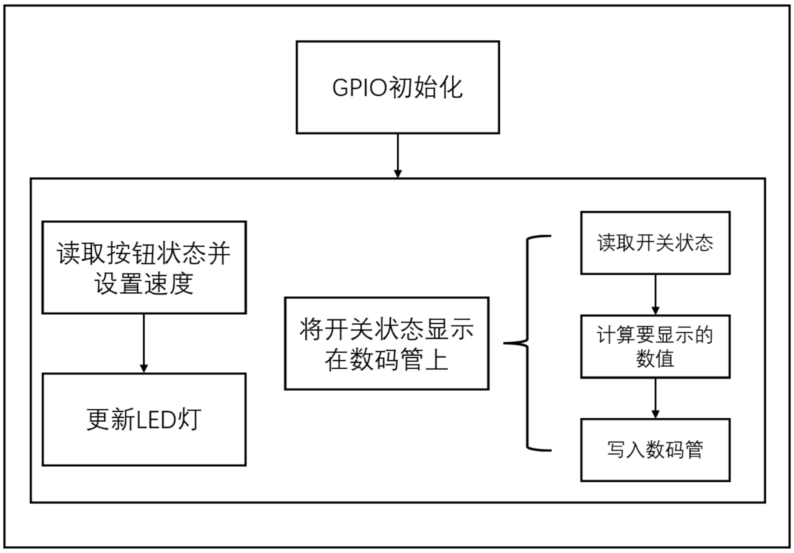
MicroBlaze_GPIO_MIMO

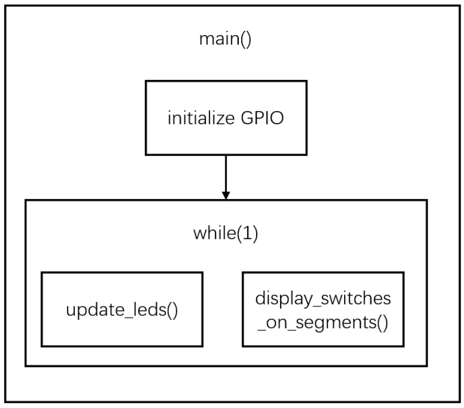
班级：**提高2301班**
姓名：**张禹阳**
学号：**U202314270**

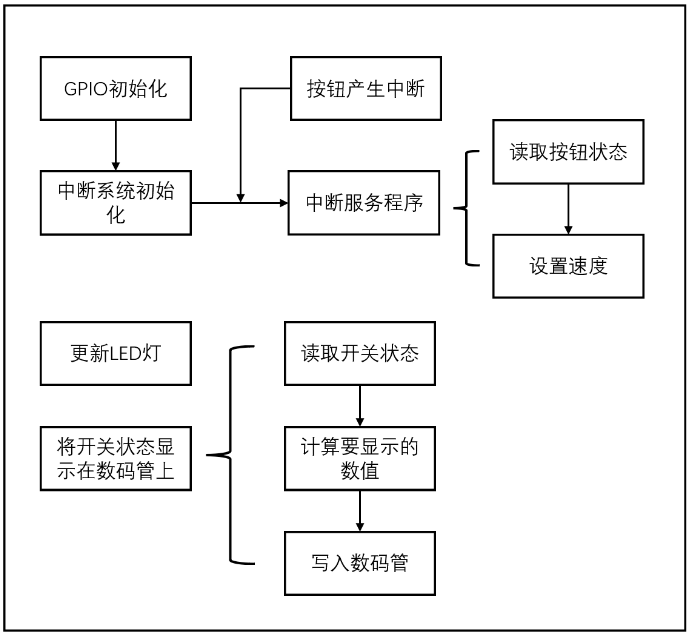
接口电路设计

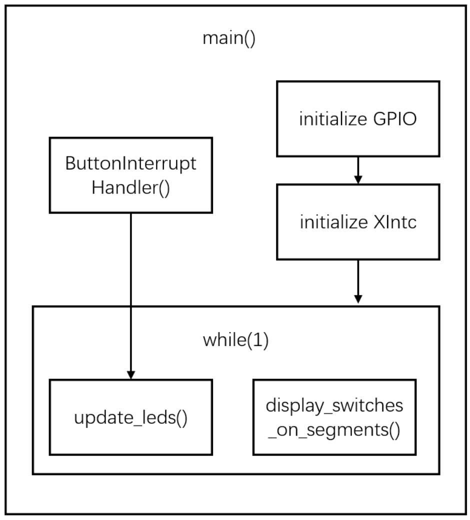


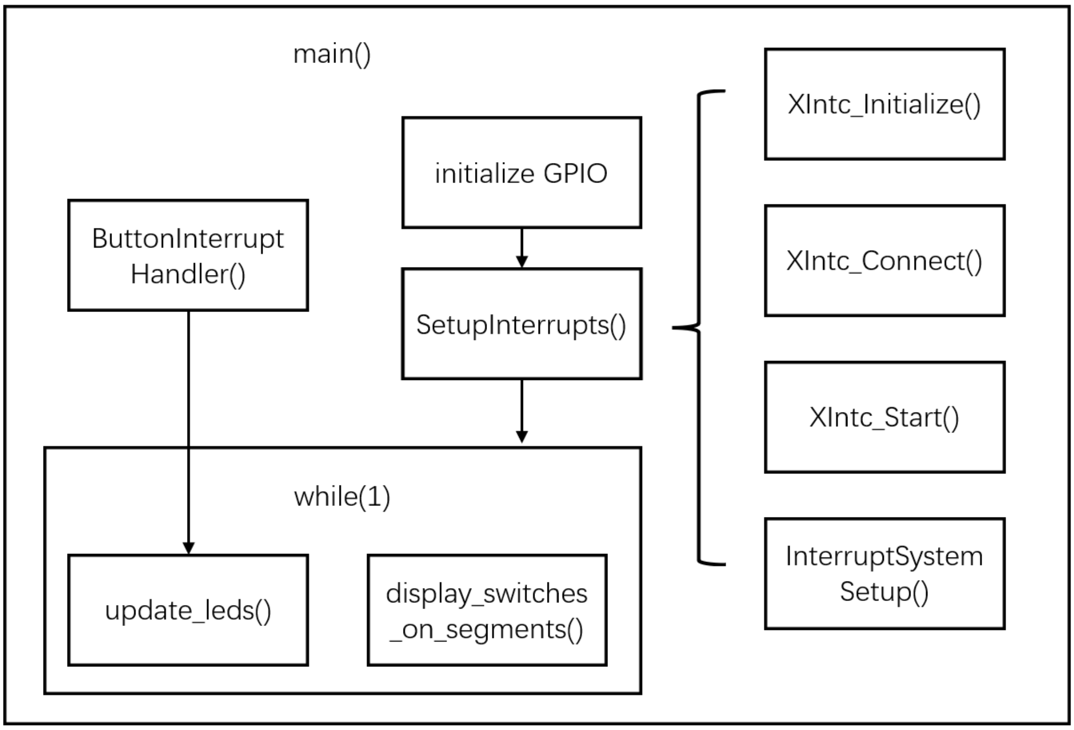
程序设计


GPIO初始化
读取按钮状态并设置速度
更新LED灯
将开关状态显示在数码管上
读取开关状态
计算要显示的数值
写入数码管


main()
initialize GPIO
while(1)
update_leds()
display_switches_on_segments()


GPIO初始化
按钮产生中断
中断系统初始化
中断服务程序
读取按钮状态
设置速度
更新LED灯
读取开关状态
将开关状态显示在数码管上
计算要显示的数值
写入数码管


main()
initialize GPIO
ButtonInterrupt Handler()
initialize XIntc
while(1)
update_leds()
display_switches_on_segments()



`program_control.c`

```c
#include "xgpio.h"
#include "xparameters.h"
#include "xil_printf.h"

#define LED_CHANNEL 1
#define SWITCH_CHANNEL 2
#define BUTTON_CHANNEL 3
#define SEGMENT_CHANNEL 4

XGpio Gpio;

void delay(int delay_ms)
{
    for (int i = 0; i < delay_ms * 1000; i++);
}

void update_leds(int speed)
{
    static int led_state = 0x01;
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led_state);
    led_state = (led_state << 1)|((led_state >> 15) & 0x1);
    delay(speed);
}
```

```
void display_switches_on_segments()
{
    u16 switch_state = XGpio_DiscreteRead(&Gpio, SWITCH_CHANNEL);
    for (int i = 0; i < 4; i++)
    {
        u8 digit = (switch_state >> (i*4)) & 0xF;
        XGpio_DiscreteWrite(&Gpio, SEGMENT_CHANNEL, digit);
        delay(2);
    }
}

int main()
{
    int speed = 100;
    XGpio_Initialize(&Gpio, XPAR_AXI_GPIO_0_DEVICE_ID);

    XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x0);
    XGpio_SetDataDirection(&Gpio, SWITCH_CHANNEL, 0xFFFF);
    XGpio_SetDataDirection(&Gpio, BUTTON_CHANNEL, 0xFFFF);
    XGpio_SetDataDirection(&Gpio, SEGMENT_CHANNEL, 0x0);

    while(1)
    {
        int buttons = XGpio_DiscreteRead(&Gpio, BUTTON_CHANNEL);

        if (buttons & 0x01) speed += 10;
        if (buttons & 0x02) speed -= 10;
        if (speed < 10) speed = 10;

        update_leds(speed);
        display_switches_on_segments();
    }

    return 0;
}
```

`interrupt_handler.c`

```
#include "xgpio.h"
#include "xintc.h"
#include "xparameters.h"

#define LED_CHANNEL 1
#define SWITCH_CHANNEL 2
#define BUTTON_CHANNEL 3
#define SEGMENT_CHANNEL 4

XGpio Gpio;
```

```c
XIntc Intc;

volatile int speed = 100;

void update_leds(int speed)
{
    static int led_state = 0x01;
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led_state);
    led_state = (led_state << 1)|((led_state >> 15) & 0x1);
    delay(speed);
}

void display_switches_on_segments()
{
    u16 switch_state = XGpio_DiscreteRead(&Gpio, SWITCH_CHANNEL);
    for (int i = 0; i < 4; i++)
    {
        u8 digit = (switch_state >> (i*4)) & 0xF;
        XGpio_DiscreteWrite(&Gpio, SEGMENT_CHANNEL, digit);
        delay(2);
    }
}

void ButtonInterruptHandler(void *CallbackRef)
{
    int buttons = XGpio_DiscreteRead(&Gpio, BUTTON_CHANNEL);

    if (buttons & 0x01) speed += 10;
    if (buttons & 0x02) speed -= 10;
    if (speed < 10) speed = 10;

    XGpio_InterruptClear(&Gpio, 0xFFFFFFFF);
}

int main()
{
    XGpio_Initialize(&Gpio, XPAR_AXI_GPIO_0_DEVICE_ID);
    XIntc_Initialize(&Intc, XPAR_INTC_0_DEVICE_ID);

    XGpio_SelectDataDirection(&Gpio, LED_CHANNEL, 0x0);
    XGpio_SelectDataDirection(&Gpio, BUTTON_CHANNEL, 0xFFFF);
    XGpio_InterruptEnable(&Gpio, 0xFFFFFFFF);
    XGpio_InterruptGlobalEnable(&Gpio);

    XIntc_Connect(&Intc, XPAR_INTC_0_GPIO_0_VEC_ID, ButtonInterruptHandler,
&Gpio);
    XIntc_Start(&Intc, XIN_REAL_MODE);

    while (1)
    {
```

```
        update_leds(speed);
        display_switches_on_segments();
    }

    return 0;
}
```

fast_interrupt.c

```
#include "xgpio.h"
#include "xparameters.h"
#include "xintc.h"
#include "xil_exception.h"

#define LED_CHANNEL 1
#define SWITCH_CHANNEL 2
#define BUTTON_CHANNEL 3
#define SEGMENT_CHANNEL 4

XGpio Gpio;
XIntc Intc;

volatile int speed = 100;
volatile int interrupt_flag = 0;

void delay(int delay_ms) {
    for (volatile int i = 0; i < delay_ms * 1000; i++);
}

void update_leds(int speed) {
    static int led_state = 0x01;
    XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led_state);
    led_state = (led_state << 1) | ((led_state >> 15) & 0x1);
    delay(speed);
}

void display_switches_on_segments()
{
    u16 switch_state = XGpio_DiscreteRead(&Gpio, SWITCH_CHANNEL);
    for (int i = 0; i < 4; i++)
    {
        u8 digit = (switch_state >> (i*4)) & 0xF;
        XGpio_DiscreteWrite(&Gpio, SEGMENT_CHANNEL, digit);
        delay(2);
    }
}

void ButtonInterruptHandler(void *CallbackRef) {
```

```c
    int buttons = XGpio_DiscreteRead(&Gpio, BUTTON_CHANNEL);

    if (buttons & 0x01) speed += 10;
    if (buttons & 0x02) speed -= 10;
    if (speed < 10) speed = 10;

    XGpio_InterruptClear(&Gpio, 0xFFFFFFFF);
    interrupt_flag = 1;
}

int InterruptSystemSetup(XIntc *IntcInstancePtr) {
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                 (Xil_ExceptionHandler)XIntc_InterruptHandler,
                                 IntcInstancePtr);

    Xil_ExceptionEnable();
    return XST_SUCCESS;
}

int SetupInterrupts() {
    int Status;

    Status = XIntc_Initialize(&Intc, XPAR_INTC_0_DEVICE_ID);
    if (Status != XST_SUCCESS) return XST_FAILURE;
    Status = XIntc_Connect(&Intc, XPAR_INTC_0_GPIO_0_VEC_ID,
                           (XInterruptHandler)ButtonInterruptHandler, &Gpio);
    if (Status != XST_SUCCESS) return XST_FAILURE;
    Status = XIntc_Start(&Intc, XIN_REAL_MODE);
    if (Status != XST_SUCCESS) return XST_FAILURE;

    XGpio_InterruptEnable(&Gpio, 0xFFFFFFFF);
    XGpio_InterruptGlobalEnable(&Gpio);

    InterruptSystemSetup(&Intc);

    return XST_SUCCESS;
}

int main() {
    int Status;

    Status = XGpio_Initialize(&Gpio, XPAR_AXI_GPIO_0_DEVICE_ID);
    if (Status != XST_SUCCESS) return XST_FAILURE;

    XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x0);
    XGpio_SetDataDirection(&Gpio, SWITCH_CHANNEL, 0xFFFF);
    XGpio_SetDataDirection(&Gpio, BUTTON_CHANNEL, 0xFFFF);

    Status = SetupInterrupts();
    if (Status != XST_SUCCESS) return XST_FAILURE;
```

```c
    while (1) {
        if (interrupt_flag) {
            xil_printf("Speed updated: %d ms\n", speed);
            interrupt_flag = 0;
        }

        update_leds(speed);
        display_switches_on_segments();
    }

    return 0;
}
```

三种控制方式分析

1. 程序控制方式

程序控制方式通过主循环轮询外设状态，执行控制逻辑。所有任务均在主循环中完成

优点

- 实现简单：无需中断机制，代码逻辑清晰，易于调试和移植
- 硬件要求低：不需要中断控制器支持，适用于简单的硬件架构

缺点

- 实时性差：主循环的执行时间较长，外设数据更新和按键响应存在延迟
- CPU 资源不足：主循环占用大量 CPU 时间，无法高效处理其他任务

可能出现的实验现象及原因

1. LED 走马灯响应延迟或卡顿
    - 原因：主循环中包含多个任务（如检查按键、更新 LED、刷新数码管），每个任务所需时间累积，导致 LED 更新不及时
2. 按键按下后反应迟缓
    - 原因：主循环只有轮询到按键时才会处理，无法实时响应按键操作

改进措施

- 优化主循环
    - 使用任务分时调度机制，将 LED 更新、按键检测、数码管显示的任务分散到不同的时间片中，减轻主循环的负担
- 引入中断机制

    - 对于按键操作，可引入中断来代替轮询，减少主循环的任务量，提高响应速度

2. 普通中断方式

普通中断方式通过中断处理按键事件，主循环仅负责外设的更新逻辑

优点

- 实时性较好：按键事件由中断触发处理，响应速度快
- 主循环压力减轻：主循环不再轮询按键状态，可以专注于外设更新

缺点

- 中断延迟：中断服务程序（ISR）运行时间较长，可能影响后续中断的处理
- 复杂性增加：需要配置中断控制器和优先级，调试较困难

可能出现的实验现象及原因

1. LED 更新速度异常
    - 原因：中断服务程序执行时间过长，导致主循环中的 LED 更新被延迟
2. 按键长时间无响应
    - 原因：中断优先级设置不当，低优先级的按键中断被高优先级任务占用时间

改进措施

- 缩短 ISR 执行时间：
    - 在 ISR 中仅完成必要的操作（如设置标志位），将复杂的逻辑移到主循环中完成
- 优化中断优先级：
    - 为重要的外设（如按键）分配更高的中断优先级，避免被次要任务阻塞

3. 快速中断方式

快速中断方式将中断处理逻辑简化为直接操作标志位，尽量减少中断处理时间，主循环完成主要的任务

优点

- 实时性最佳：中断服务程序非常短，响应速度快
- 系统稳定性高：由于 ISR 执行时间短，不容易阻塞其他任务
- 高效利用 CPU：主循环处理大部分逻辑，避免了中断的复杂性

缺点

- 不适合复杂中断逻辑：快速中断机制无法处理需要精确时间控制的复杂任务
- 需要硬件支持：对中断控制器的响应速度和优先级管理提出较高要求

可能出现的实验现象及原因

1. 数码管刷新闪烁
    - 原因：主循环中任务较多（如 LED 更新、按键处理），导致数码管刷新频率下降
2. 任务间冲突
    - 原因：主循环中任务执行时间过长，无法及时处理中断标志位

改进措施

- 增加硬件支持：
    - 配置硬件计时器，用于定时刷新数码管或更新 LED，减少主循环的任务
- 优化中断处理：
    - 结合普通中断方式，将部分复杂的逻辑从主循环中分离出来