同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称	操作系统应用程序开发——2048 小游戏
实验成员	冯羽芯（2251206）
日　　期	二零二五年 六 月 十 日

1、实验目的

- 学习实时操作系统应用开发，在成功移植的 μC/OS-II 操作系统之上，学习如何使用串口和 GPIO 来设计和实现应用程序，体会操作系统为应用开发提供的便利。
- 锻炼系统级软件的分析与调试能力，通过实践，培养在没有图形化界面的嵌入式环境下，分析、定位和解决从硬件初始化、系统启动到应用程序运行等各个环节问题的综合能力。
- 加强对计算机专业核心课程的掌握，融合编译原理、操作系统等底层软件知识与硬件课程，构建软硬件一体化的完整知识架构。

2、实验内容

(1) 实验环境与硬件配置

- 处理器： 13th Gen Intel(R) Core(TM) i7-13650HX，2600 Mhz
- 操作系统：Windows 11
- 开发平台：Vivado 2022.2 用于 FPGA 设计与编程、VSCode 用于代码编辑
- 仿真环境：Vivado 2022.2 自带仿真器
- 辅助工具：WSL 用于构建 Linux 环境，SSCOM5.13.1 用于实现串口通信
- 文档管理：Office Word 2021
- 开发板： NEXYS4DDRAtrix-7

(2) 实验方案

本实验在《自己动手写 CPU》第 15 章提供的内核程序源码基础上进行应用程序开发，要求进行开发板串口控制、C 语言应用程序实现和命令行图形化界面的展示，具体需要实现以下内容：

- 2048 小游戏代码实现
  - 应用程序创建与执行流程分析
  - 编写输入函数，将开发板开关与按钮状态与程序绑定
  - 编写程序主循环
- 操作系统编译
  - 编译过程分析
- 下板验证检查
  - 使用 SSCOM V5.13.1 串口/网络数据调试器，对结果进行检查
  - 改变开发板开关状态，测试程序走向是否正确

3、实验步骤

(1) 用户程序执行流程

操作系统成功编译并导出二进制文件后，尚未做任何修改，上次实验中的用户程序是在串口输出一段字符串，本次实验也就是将用户程序的部分改成 2048 小游戏。用户程序存储在 common 文件夹中，具体一点就是 openmips.c 文件中。main 函数描述了整个用户程序的执行流程，如下步骤所示：

```
void main()
{
    OSInit();                   /* μC/OS-II 初始化 */
    uart_init();                /* UART 控制器初始化 */
    gpio_init();                /* GPIO 模块初始化 */
    /* 创建用户任务 */
    OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE - 1], 0);
    OSStart();                   /* μC/OS-II 启动 */
}
```

- OSInit() 函数调用 uC/OS-II 的内核初始化函数，准备好任务管理、事件标志、信号量、内存管理等所有内核服务所需的数据结构，但此时还没有任何用户任务被创建，调度器也未启动。
- uart_init() 初始化串口控制器，是后续所有打印函数能够正常工作的前提。
- gpio_init() 初始化通用输入/输出（GPIO）模块，读取外部输入信号，在本实验中即为开发板上按钮或开关的状态输入。
- OSTaskCreate() 创建任务，main 函数的职责就是把系统引导到这个任务，然后由这个任务来展开所有其他的应用功能。
- OSStart() 启动多任务调度，CPU 的控制权被移交给 uC/OS-II。调度器会查看当前所有已创建的任务，找到优先级最高的那个，然后执行它的代码。在本操作系统中，除了 kernel 外仅有一个用户进程，故在该程序结束后由于调度，该程序再次启动上台运行，表现为游戏不断循环。

(2) 工具函数分析

1. 用户输入函数 gpio_in()

此函数的核心功能是读取 GPIO 模块所有输入引脚的当前状态，也就是用户在开发板上的输入，并将这组状态作为一个 32 位的整数返回，这个 32 位值的每一位都对应一个 GPIO 引脚的当前电平状态。用户的输入其实存储在开发板中的寄存器中，我们需要从中实时读取内容即可获取输入。具体实现如下：

```
INT32U gpio_in()                    /* 读取 GPIO 模块输入的函数 */
{
    INT32U temp = 0;
    temp = REG32(GPIO_BASE + GPIO_IN_REG);
    return temp;
}
```

其中，用户输入按键和 32 位整型的对应关系如下图：

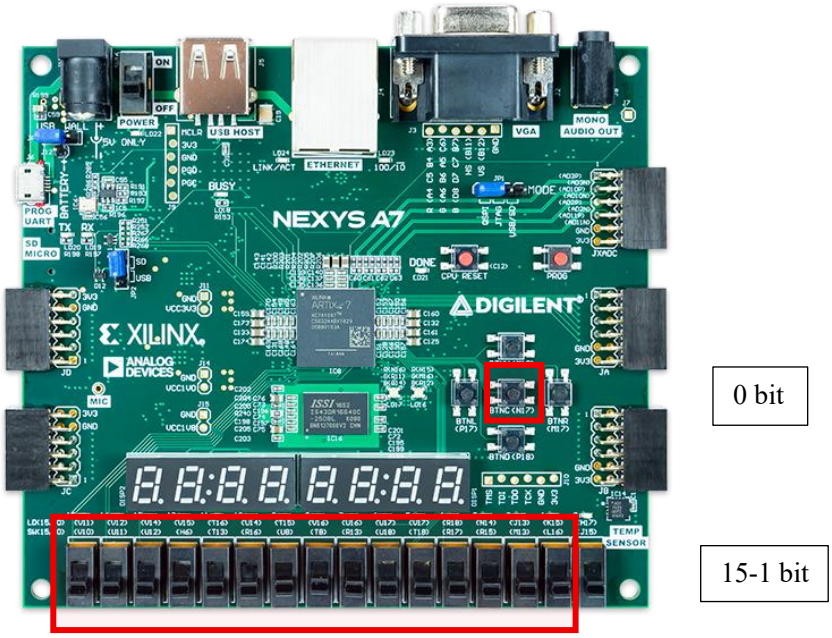

图 1：用户输入与 32 整型映射关系图

同样可以在 xdc 文件中得到验证：

```
set_property -dict {PACKAGE_PIN N17 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[0]}]
set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[1]}]
set_property -dict {PACKAGE_PIN M13 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[2]}]
set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[3]}]
set_property -dict {PACKAGE_PIN R17 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[4]}]
set_property -dict {PACKAGE_PIN T18 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[5]}]
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[6]}]
set_property -dict {PACKAGE_PIN R13 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[7]}]
```

```
set_property -dict {PACKAGE_PIN T8 IOSTANDARD LVCMOS18} [get_ports
{gpio_i[8]}]
set_property -dict {PACKAGE_PIN U8 IOSTANDARD LVCMOS18} [get_ports
{gpio_i[9]}]
set_property -dict {PACKAGE_PIN R16 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[10]}]
set_property -dict {PACKAGE_PIN T13 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[11]}]
set_property -dict {PACKAGE_PIN H6 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[12]}]
set_property -dict {PACKAGE_PIN U12 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[13]}]
set_property -dict {PACKAGE_PIN U11 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[14]}]
set_property -dict {PACKAGE_PIN V10 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[15]}]
```

装 订 线

2. 字符输出函数 uart_putc()与字符串输出函数 uart_print_str()

输出函数 uart_putc 是所有串口打印功能的基础，负责向 UART 发送单个字符。

```
void uart_putc(char c)            /* 通过 UART 输出字节 */
{
    unsigned char lsr;
    WAIT_FOR_THRE;              /* 等待发送 FIFO 空 */
    REG8(UART_BASE + UART_TH_REG) = c;    /* 通过 UART 输出字节 */
    if(c == '\n') {             /* 如果是换行符，那么增加一个回车符 */
        WAIT_FOR_THRE;
        REG8(UART_BASE + UART_TH_REG) = '\r';  /* 通过 UART 输出回车符 */
    }
    WAIT_FOR_XMITR;             /* 等待发送数据完毕 */
}
```

其中 WAIT_FOR_THRE 是在开头定义的宏，作用是等待 UART 的发送缓冲区变为空闲，在发送新字符前进行此等待，是为了确保上一个字符的发送流程已经启动，控制器可以接收新的字符，从而防止数据丢失。WAIT_FOR_XMITR 等待数据被完全发送出去。WAIT_FOR_THRE 只保证了数据被装入 UART 的发送缓冲区，而 WAIT_FOR_XMITR 则进一步保证了数据已经从 UART 的移位寄存器中全部发送到了串行总线上。

```
/* 循环等待，直到 UART 控制器的发送 FIFO 为空、移位寄存器为空，表示数据发送完毕 */
#define WAIT_FOR_XMITR \
        do { \
                lsr = REG8(UART_BASE + UART_LS_REG); \
        } while ((lsr & BOTH_EMPTY) != BOTH_EMPTY)
```

```
/* 循环等待，直到 UART 控制器发送 FIFO 为空，此时不一定发送完毕，但是可以接着通过
UART 控制器发送数据 */
#define WAIT_FOR_THRE \
        do { \
                lsr = REG8(UART_BASE + UART_LS_REG); \
        } while ((lsr & UART_LS_THRE) != UART_LS_THRE)
```

下面分析 uart_print_str 函数，此函数的功能依然是通过 UART 发送一个完整的字符串。但与 uart_putc 最大的不同是，它确保了在发送整个字符串的过程中，不会被其他任务或中断所打断。首先函数进入临界区，禁止中断，保存当前中断状态；连续调用 uart_putc 发送字符串中的每一个字符；最后退出临界区，恢复之前保存的中断状态。

```
void uart_print_str(char* str)    /* 通过 UART 输出字符串 */
{
    INT32U i=0;
    OS_CPU_SR cpu_sr;
    OS_ENTER_CRITICAL()         /*不希望输出字符串的过程被打断，所以进入临界区 */

    while(str[i]!=0)
    {
        uart_putc(str[i]);       /* 调用函数 uart_putc 依次输出每个字节 */
        i++;
    }

    OS_EXIT_CRITICAL()          /* 输出字符串结束，退出临界区 */
}
```

(3) 游戏函数的输入输出控制

由于用户程序是一个无限的循环，每一次循环都会读开发板上按键状态进行重新读取，由于 2048 小游戏需要按步进行，每次选择一个方向进行输入，故需要着重考虑用户的确认逻辑。本实验采用 N17 作为确认按键，只有在确认按键为高电平时，才会进行下一步游戏内容的输出，否则将会一直“等待”用户输入，即不进入游戏分支。代码实现方面，只有在按下 N17 按钮后，输入一个高电平信息，32 位整型的最低位置为 1，将其左移 31 位，填补 31 个 0，也就是只判断最高位是否为 1，若是，则置 ready 标志位为 1，只有当 ready 为真时才会进入游戏逻辑。

```
void TaskStart (void *pdata)
{
    INT32U count = 0;
    INT32U data;
    pdata = pdata;
    OSInitTick();       /* 在用户任务中初始化定时器、允许时钟中断 */
```

```
    for (;;) {            /* 一般而言，任务都是一个永不结束的循环 */
        data = gpio_in();
        INT32U ready = data << 31;

        if (ready) {
            ...
        }
        count = count + 1;
    }
}
```

(4) 2048 棋盘数据结构与输入输出

1. 2048 棋盘数据结构

2048 棋盘是使用一个二维整型数组来实现的：

```
#define BOARD_SIZE 4
int board[BOARD_SIZE][BOARD_SIZE];
```

这种数据结构简单明了，能直接映射 4x4 的游戏棋盘，便于进行逻辑判断和数值运算。将 board 定义为全局变量，使得项目中的任何函数都可以直接访问和修改它，这简化了函数之间的数据传递；数组中每个位置的整数值代表了棋盘上对应格子的数字，0 代表该格子是空的。

2. 游戏面板打印

由于没有 C 的标准库函数，故许多已有方法不能使用，本输出函数中采用手动右对齐的方式调整棋盘格式，使之更加美观。

遍历棋盘的每一行在每一行数字的上方打印一个分隔线上边框，使棋盘结构更清晰，函数末尾的同一行代码则构成了整个棋盘的下边框。

如果单元格为空，直接将 buf 填充为 5 个空格；

如果单元格中含有数字，则从右到左填充 buf 数组，从而实现右对齐。

```
void print_board(void) {
    int i, j, k, num;
    char buf[8]; // 足够存储一个 5 位整数 + 空格 + 字符串结束符

    for (i = 0; i < BOARD_SIZE; i++) {
        // 输出上边框
        uart_print_str("|--------|--------|--------|--------|\n");
        // 输出数字和左右边框
        uart_print_str("|");
```

```
        for (j = 0; j < BOARD_SIZE; j++) {
            num = board[i][j];

            // 将整数转换为字符串
            if (num == 0) {
                buf[0] = ' ';
                buf[1] = ' ';
                buf[2] = ' ';
                buf[3] = ' ';
                buf[4] = ' ';
                buf[5] = '\0';
            } else {
                for (k = 4; k >= 0; k--) {
                    buf[k] = (num % 10) + '0';
                    num /= 10;
                    if (num == 0) {
                        // 填充空格
                        while (k > 0) {
                            k--;
                            buf[k] = ' ';
                        }
                        break;
                    }
                }
                buf[5] = '\0';
            }

            uart_print_str("  ");
            uart_print_str(buf);
            uart_print_str(" |");
        }
        uart_print_str("\n");
    }
    uart_print_str("|--------|--------|--------|--------|\n");
}
```

3. 用户的方向输入

用户在 2048 游戏中，每一步抉择需要选择一个方向，将数字移动以尽可能合并，本实验中，使用开发板上的四个开关作为输入，分别是 R17、R15、M13、L16，开关上拨代表方向选择，每次只能选择一个方向，也就是能且仅能上拨一个开关。

由于 GPIO 的最低位是确认按钮，故读出的开关状态需右移一位，并先与 0b1111 相与，排除陌生位干扰，再判断是否与目标状态相同，从而使状态具有独立性。

```
            INT32U choice = data >> 1;
            INT32U up = (choice & 0x0000000F) == 0x00000008;
            INT32U down = (choice & 0x0000000F) == 0x00000004;
            INT32U left = (choice & 0x0000000F) == 0x00000002;
            INT32U right = (choice & 0x0000000F) == 0x00000001;
```

(5) 2048 主逻辑实现

1. 游戏循环

以下为游戏主逻辑的代码：

```
void  TaskStart (void *pdata)
{
    INT32U count = 0;
    INT32U data;
    INT32U moved;
    pdata = pdata;
    OSInitTick();       /* 在用户任务中初始化定时器、允许时钟中断 */

    init_board();
    uart_print_str("Welcome to 2048!\n");
    print_board();
    uart_print_str("Use buttons to move left, right, up, down.\n");

    for (;;) {            /* 一般而言，任务都是一个永不结束的循环 */
        data = gpio_in();
        INT32U ready = data << 31; // 也就是只判断这一位，因为移出来的都是 0
        INT32U choice = data >> 1;

        if (ready) {
            moved = 0;

            INT32U up = (choice & 0x0000000F) == 0x00000008;
            INT32U down = (choice & 0x0000000F) == 0x00000004;
            INT32U left = (choice & 0x0000000F) == 0x00000002;
            INT32U right = (choice & 0x0000000F) == 0x00000001;

            if (up) {
                uart_print_str("Your choice is: up\n");
                moved = move_up();
            }
            else if (down) {
                uart_print_str("Your choice is: down\n");
                moved = move_down();
```

```
            }
            else if (left) {
                uart_print_str("Your choice is: left\n");
                moved = move_left();
            }
            else if (right) {
                uart_print_str("Your choice is: right\n");
                moved = move_right();
            }
            else {
                uart_print_str("Invalid move!\n");
                continue;
            }

            if (moved) {
                add_new_tile();
            }

            print_board();

            if (is_game_over()) {
                uart_print_str("Game Over!\n");
                break; // 结束游戏
            } else {
                uart_print_str("Enter move...\n");
            }
        }
        count = count + 1;
    }
}
```

下图展示主逻辑的流程图：

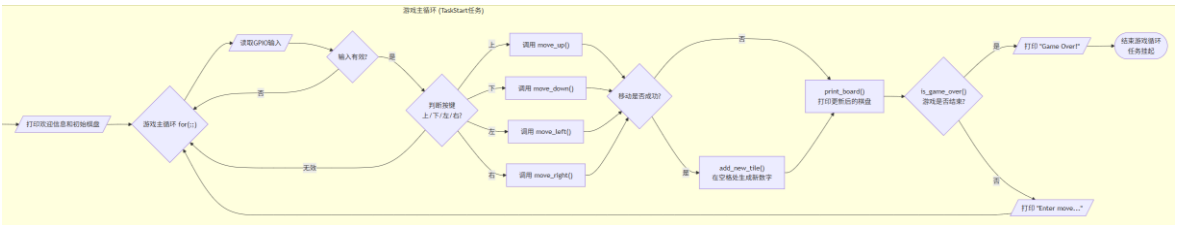

图 2：游戏主逻辑流程图

首先初始化游戏棋盘，并在棋盘上随机生成两个初始数字块。在打印出欢迎信息和初始棋盘后，程序便进入一个无限循环的游戏主流程。在这个循环中，系统持续监听来自 GPIO 的硬件按键输入。一旦检测到有效的方向指令，程序会调用相应的移动函数来滑动和合并数字方块。如果这次移动成功地改变了棋盘布局，系统将在一个随机的空位上生成一个新的数字（2 或 4）。每次

操作后，更新后的棋盘会被重新打印到终端。紧接着，程序会检查游戏是否结束，即判断棋盘是否已满且没有任何可以合并的相邻方块。如果游戏可以继续，则提示玩家进行下一步操作；如果游戏结束，则显示“Game Over”信息并终止循环。

2. 棋盘初始化函数

每当一局新游戏开始时，都需要调用这个函数来创建一个全新的初始棋盘。首先清空整个棋盘，并放置两个随机的初始数字块，在棋盘上所有空格中，随机挑选一个位置，并在这个位置上生成一个新的数字，90% 的概率是 2，10% 的概率是 4。

```
void add_new_tile(void) {
    int i, j;
    int empty_tiles = 0;
    for (i = 0; i < BOARD_SIZE; i++) {
        for (j = 0; j < BOARD_SIZE; j++) {
            if (board[i][j] == 0) {
                empty_tiles++;
            }
        }
    }
    if (empty_tiles == 0) return;

    int pos = my_rand() % empty_tiles;
    int count = 0;
    for (i = 0; i < BOARD_SIZE; i++) {
        for (j = 0; j < BOARD_SIZE; j++) {
            if (board[i][j] == 0) {
                if (count == pos) {
                    board[i][j] = (my_rand() % 10 == 0) ? 4 : 2;
                    return;
                }
                count++;
            }
        }
    }
}

void init_board(void) {
    int i, j;
    for (i = 0; i < BOARD_SIZE; i++) {
        for (j = 0; j < BOARD_SIZE; j++) {
            board[i][j] = 0;
        }
```

```
    }
    add_new_tile();
    add_new_tile();
}
```

3. 移动合并函数

上下左右的移动合并函数的核心思想是相同的，都是遍历棋盘上的每一个数字块，并尝试将它沿着指定方向移动到底，直到：

- 遇到边界
- 遇到另一个不同的数字块
- 遇到一个相同的数字块

下以向左滑动为例，进行详细分析：

外层循环逐行处理棋盘；

中层循环从左到右依次“捡起”每一个非零数字块。这个顺序很关键，它保证了最左边的合并会先发生；

内层循环“捡起”一个数字块后，这个 while 循环就负责让它一直向左“滑行”。

- 如果左边是空格，就交换位置，滑块的坐标 k 也减一，继续向左滑。
- 如果左边是相同数字，则合并，然后用 break 结束这个滑块的使命。
- 如果左边是不同数字，则被挡住，同样用 break 结束滑动。

```
int move_left(void) {
    int i, j, k;
    int moved = 0;
    for (i = 0; i < BOARD_SIZE; i++) {
        for (j = 1; j < BOARD_SIZE; j++) {
            if (board[i][j] != 0) {
                k = j; // Start from the current position
                while (k > 0) {
                    if (board[i][k - 1] == 0) {
                        // Move to the left
                        board[i][k - 1] = board[i][k];
                        board[i][k] = 0;
                        moved = 1;
                        k--; // Continue moving to the left
                    } else if (board[i][k - 1] == board[i][k]) {
                        // Merge with the left tile
                        board[i][k - 1] *= 2;
                        board[i][k] = 0;
                        moved = 1;
```

```
                    break; // Stop moving this tile
                } else {
                    // Blocked by a different tile
                    break; // Stop moving this tile
                }
            }
        }
    }
    return moved;
}
```

4. 游戏结束判断函数

此函数的核心功能是检查当前棋盘状态下，玩家是否还能进行任何有效的移动。如果不能，则游戏结束。函数通过返回一个整数来表示结果，0 表示游戏未结束，玩家还可以继续操作；1 表示游戏已结束，玩家失败。

函数通过一个双重 for 循环遍历棋盘上的每一个格子，并对每个格子进行三种可能性检查。只要有任何一种可能性成立，就意味着游戏还能继续：

- 当前格子为空
- 当前格子和它下方的格子数字相同
- 当前格子和它右方的格子数字相同

即寻找一个空格，或者一对可合并的相邻块。

```
int is_game_over(void) {
    int i, j;
    for (i = 0; i < BOARD_SIZE; i++) {
        for (j = 0; j < BOARD_SIZE; j++) {
            if (board[i][j] == 0) return 0;
            if (i < BOARD_SIZE - 1 && board[i][j] == board[i + 1][j])
                return 0;
            if (j < BOARD_SIZE - 1 && board[i][j] == board[i][j + 1])
                return 0;
        }
    }
    return 1;
}
```

(6) 随机函数实现

由于无法使用库函数 rand()，本模块采用手动设置种子的方式来生成伪随机数。

该函数实现的是一个经典的线性同余生成器（Linear Congruential Generator, LCG）算法。LCG

是一种简单而快速的伪随机数生成算法，非常适合在不方便使用标准库 rand() 函数的嵌入式环境中使用。此函数的功能是生成一个伪随机的、范围在 0 到 32767 (0x7FFF) 之间的无符号整数。

static 局部变量的特点是它只在程序第一次执行到这里时被初始化一次。在后续的函数调用中，my_seed 会一直保持它上一次被修改后的值。这就实现了“状态保持”，使得每次调用 my_rand 都能在上一次的基础上生成一个新的随机数。

函数体中，我们对新生成的 32 位种子进行右移 16 位的操作。在 LCG 算法中，通常认为高位的随机性要优于低位。因此，通过右移操作，我们舍弃了随机性较差的低 16 位，而保留了质量更高的高 16 位。

```
unsigned int my_rand(void) {
    static INT32U my_seed = 123456789; // 可以用时间或其它方式初始化
    my_seed = (1664525 * my_seed + 1013904223); // 32 位乘法和加法
    return (my_seed >> 16) & 0x7FFF; // 取高 15 位，并确保结果为正
}
```

装

(7). 交叉编译

1. 项目建立

本部分所有用到的文件全部来自《自己动手写 CPU》。

将合并二进制文件的程序 BinMerge.exe 放在 ucosii_OpenMIPS 目录下。

将链接脚本文件 ram.ld 放在 ucosii_OpenMIPS 目录下，在其中定义了很多的 Section，这些都是在编译的时候会生成的 Section。此外，单独声明一个 vectors Section，占用低 0x80 字节的空间，用来存放异常处理例程入口地址，其余的可执行程序放在地址 0x80 以上的空间。

订

新建 config.mk、Makefile 文件，放在 ucosii_OpenMIPS 目录下。

线

同样地，在每个子文件夹中都建立 Makefile，最终项目的文件结构如下：

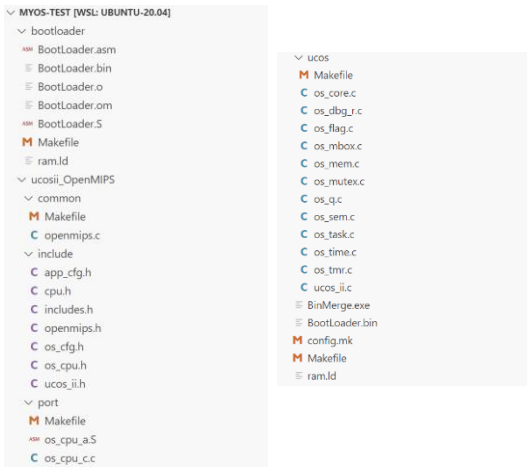

图 3：整个项目的文件架构

2. 修改 config.mk

调整编译器和链接器的 -G 选项，设为 0，不要将任何变量放入 small-data section，从而完全绕过 64KB 的限制。修改完毕后，先进行 `make clean` 清除缓存文件。

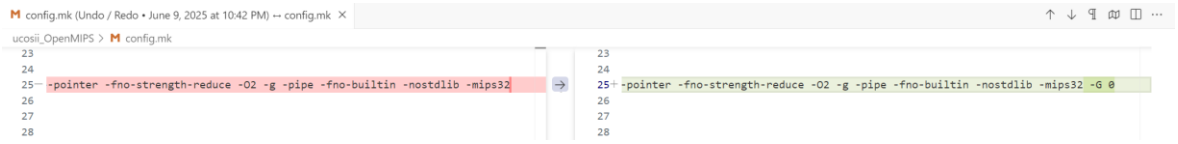

图 4：更改 config.mk

```
chenovo@LAPTOP-MGRUHPKO:~/MyOS/ucosii_OpenMIPS$ make clean
find . -type f \
        \( -name 'core' -o -name '*.bak' -o -name '*~' \
        -o -name '*.o'  -o -name '*.tmp' -o -name '*.hex' \
        -o -name 'OS.bin' -o -name 'ucosii.bin' -o -name '*.srec' \
        -o -name '*.mem' -o -name '*.img' -o -name '*.out' \
        -o -name '*.aux' -o -name '*.log' -o -name '*.data' \) -print \
        | xargs rm -f
rm -f System.map
```

图 5：清除缓存

3. 修改 BinMerge.exe 的权限

由于 Ubuntu 的权限问题，此处需要更改 BinMerge.exe 的执行权限之后进行 `make all`，否则会在中途由于访问权限不足导致编译链接中断。

```
./BinMerge.exe -f ucosii.bin -o OS.bin
make: execvp: ./BinMerge.exe: Permission denied
make: *** [Makefile:33: OS.bin] Error 127
chenovo@LAPTOP-MGRUHPKO:~/MyOS-test/ucosii_OpenMIPS$ chmod +x BinMerge.exe
```

图 6：更改 BinMerge.exe 的权限

最终再次执行 `make all` 指令，得到 OS.bin 文件。

4、实验结果

(1) 串口通讯与可视化界面

在下板载入 bit 流的那一步之前，我们需要配置 Flash，将刚刚编译得到的 OS.bin 文件载入。右键开发板，选择 Add Configuration Memory Device。

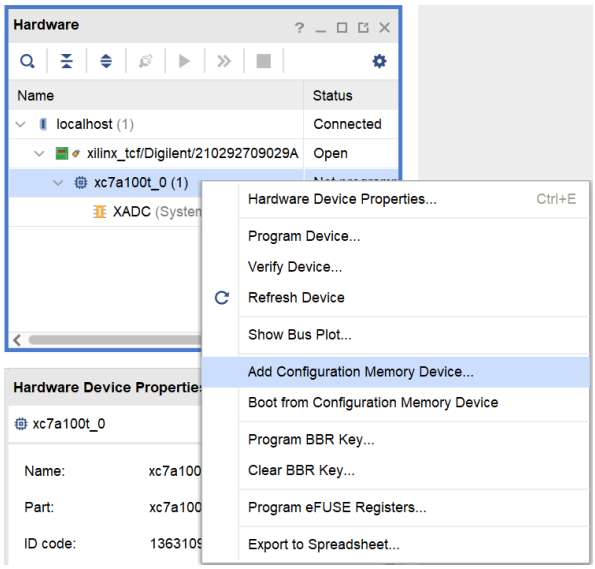

图 7：配置 SPI-Flash

选中 s25fl128sxxxxxx0-spi-x1_x2_x4。

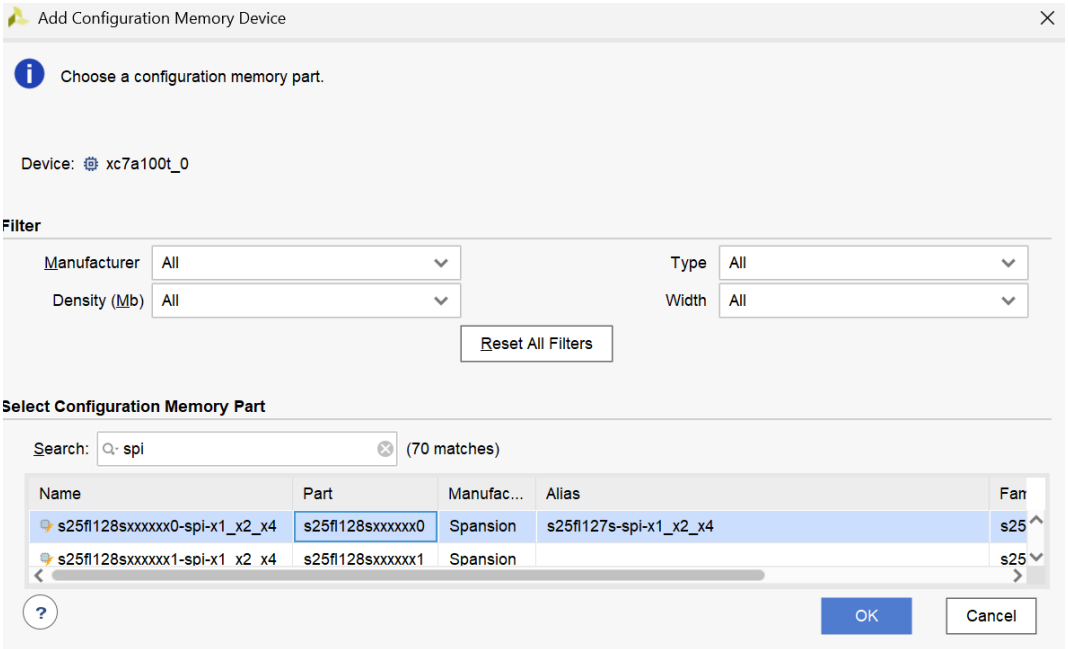

图 8：配置 SPI-Flash

选择编译得到的 bin 二进制文件下板。

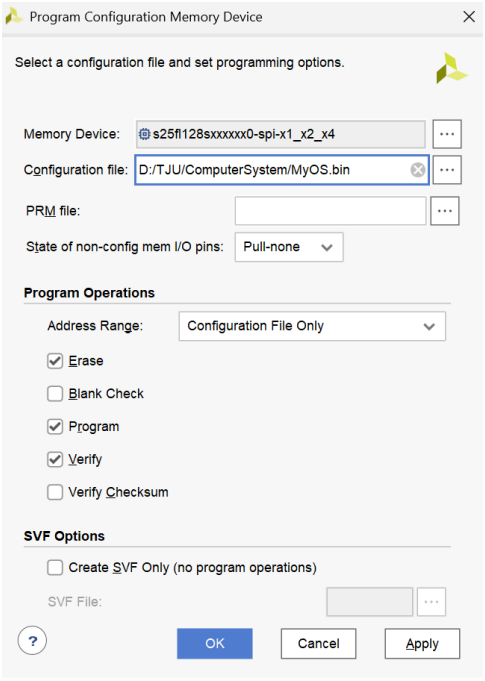

图 9：配置 SPI-Flash

下板后打开 SSCOM V5.13.1 串口/网络数据调试器（必应搜索下载即可），选择对应的端口号，我这里是 COM5 USB Serial Port，调整波特率为 9600，取消加时间戳和分包显示，点击打开串口，将开发版右侧第一个开关上拨，即可看到 2048 游戏成功启动。

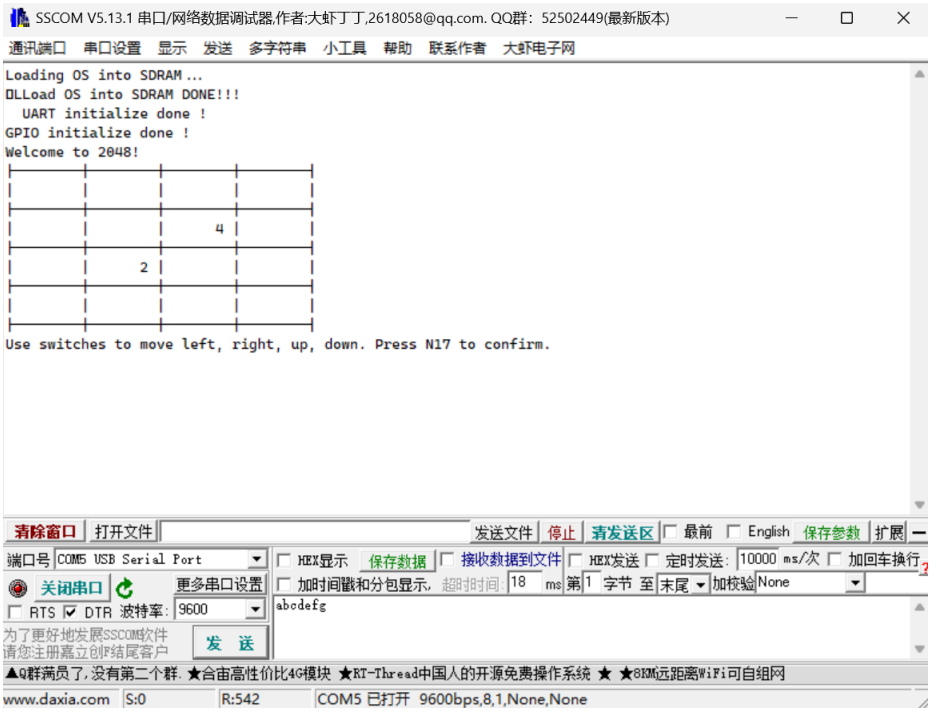

图 10：2048 成功显示

装 订 线

(2) 应用程序流程

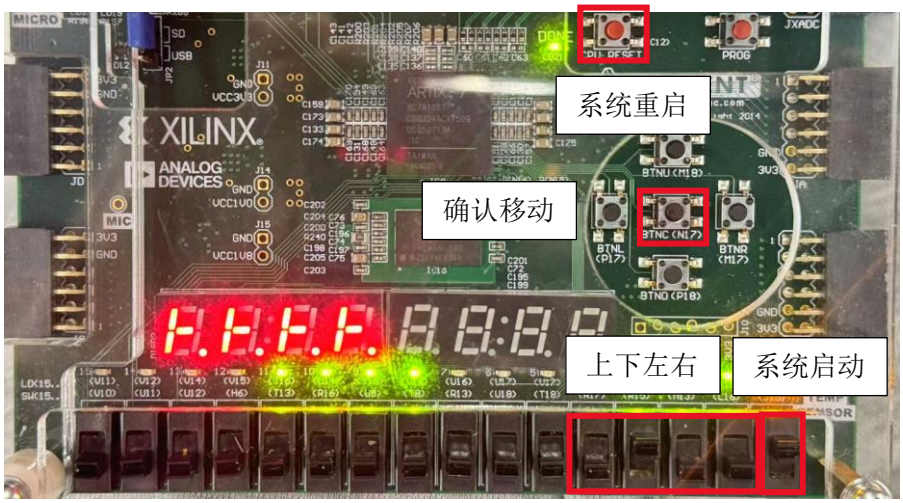

图 11：下板结果演示

开发板下方的右侧五个开关分别为：上下左右控制开关和系统使能开关。中间黑色按钮为确认移动按钮、上方红色按钮是重置键。图中选择移动方向是下方，确认后，串口给出新的界面：

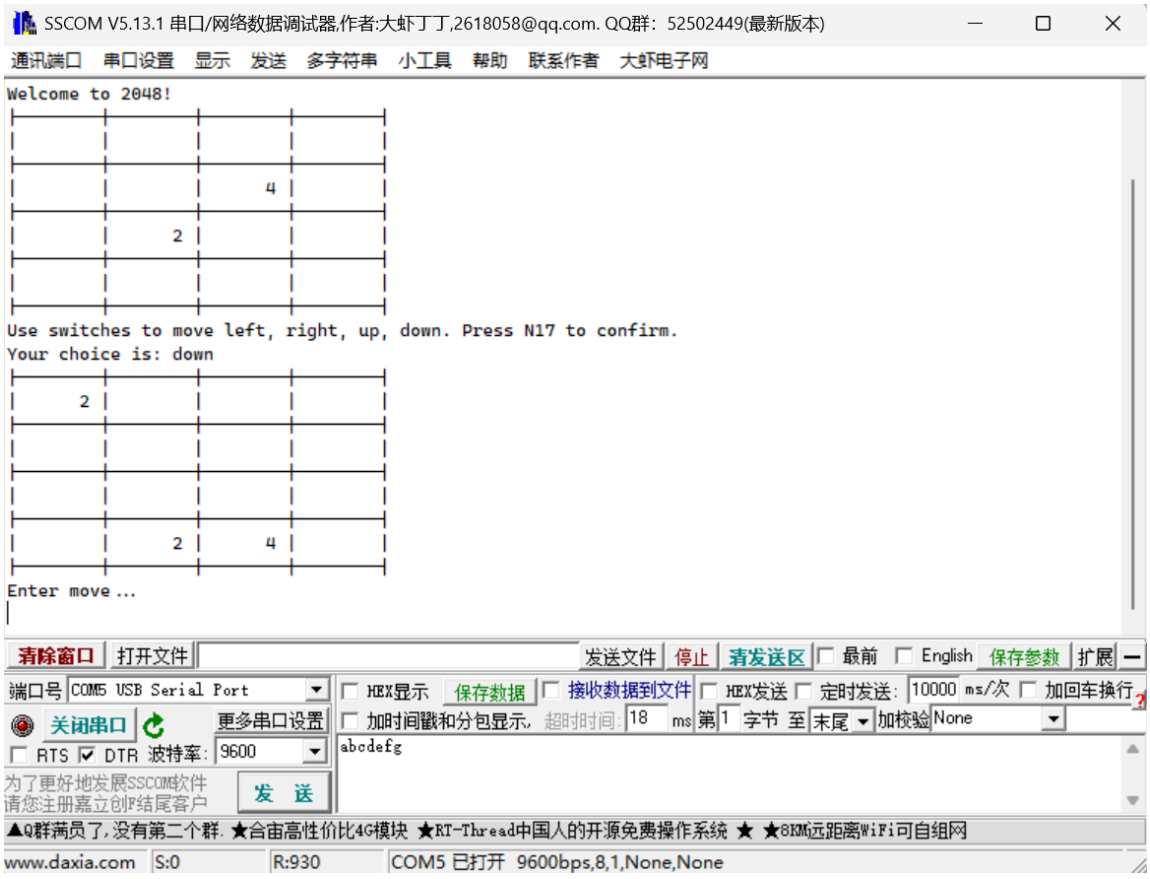

图 12：下移示意图

下面展示数字生成、移动与合并的情况：可以看到右移后两个 2 合并成了 4，两个 4 合并成

了 8，并在一个随机的位置生成了一个数字 2。

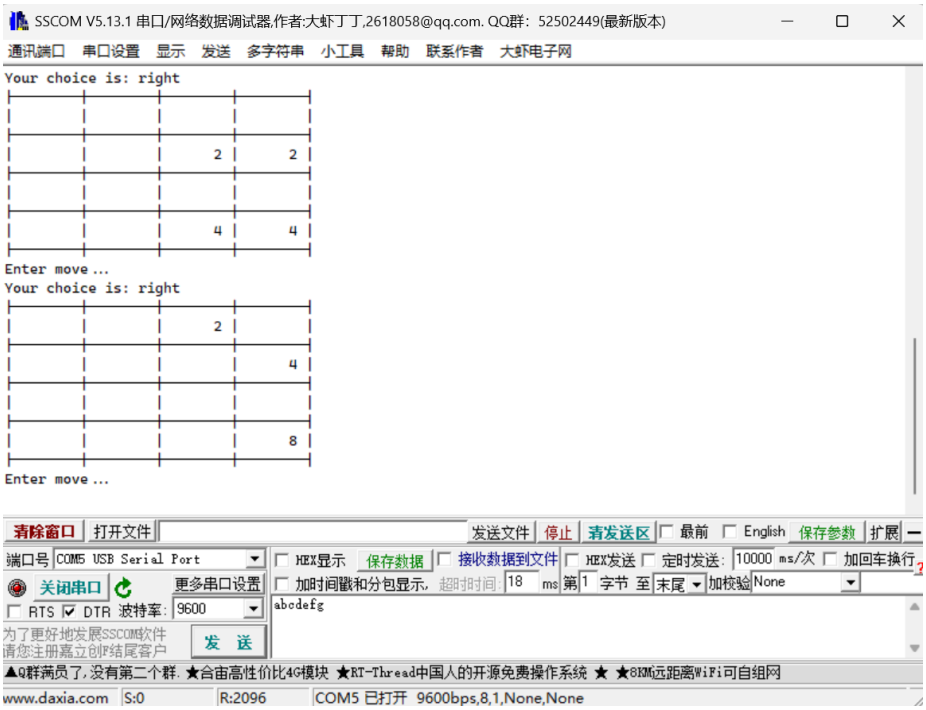

图 13：合并示意图

如果存在非法的输入，例如用户同时上拨多个方向开关，或未输入任何方向就按下确认键，则串口会给出 Invalid move!的提示：

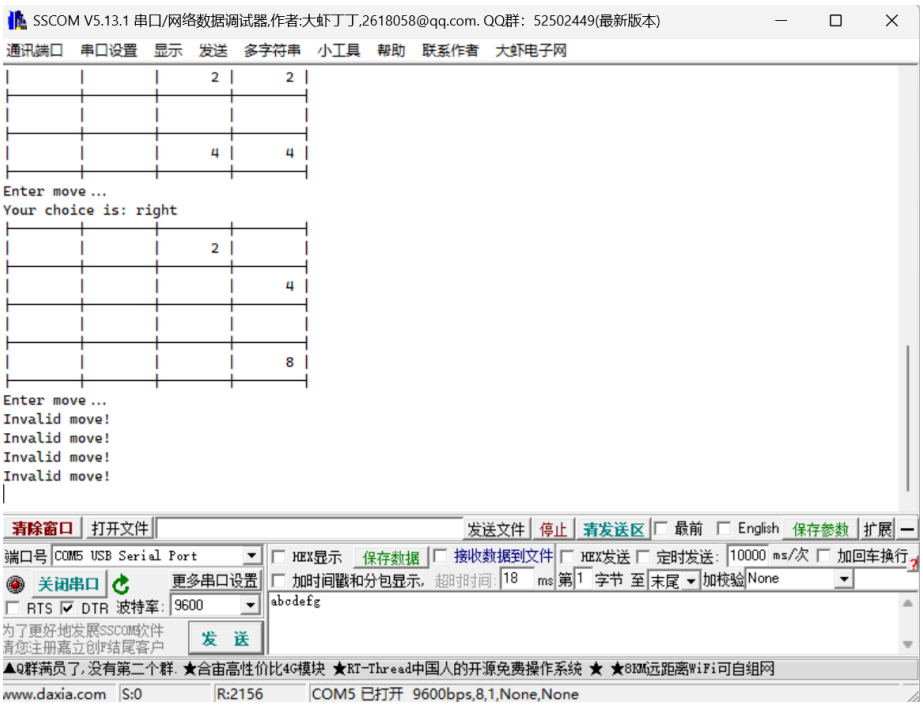

图 14：非法移动示意图

若游戏失败，即已经没有可以进行的任何方向，则会提示游戏结束，并重新开始一次新游戏：

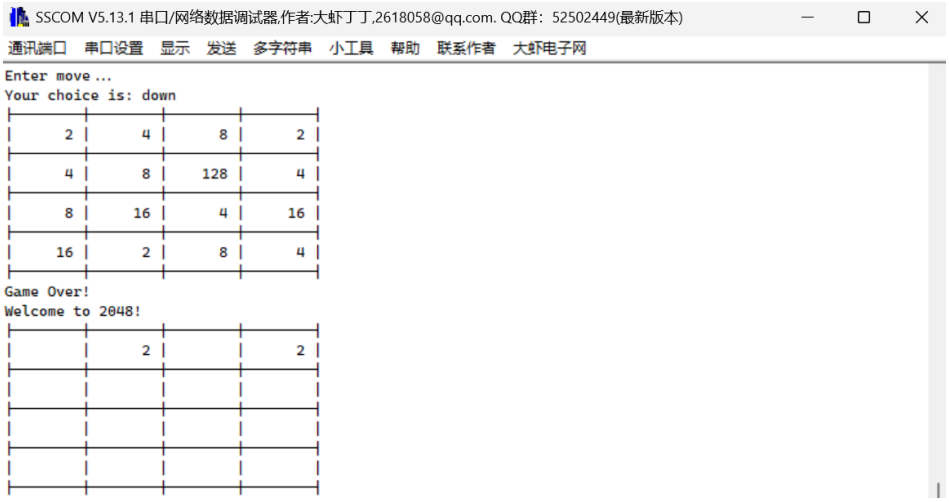

图 15：游戏结束示意图

5、实验总结

(1) 遇到的问题

编译报错 /opt/mips-4.3/bin/../lib/gcc/mips-sde-elf/4.3.2/../../../../mips-sde-elf/bin/ld: small-data section exceeds 64KB; lower small-data size limit (see option -G)

在 MIPS 架构中，为了提高访问全局变量和静态变量的效率，编译器会将一部分“小”变量存放在一个特殊的内存区域，称为 small-data section（通常是 .sdata 和 .sbss 段）。

这个区域的特点是可以通过一个专用的全局指针寄存器加上一个较小的偏移量来快速访问，而不需要复杂的地址计算。但它的缺点是大小有限，通常被限制在 64KB。而我项目中所有的“小”全局变量和静态变量的总大小超过了 64KB 的限制，导致链接器无法将它们全部放入 small-data section。

解决方法是调整编译器和链接器的 -G 选项。这个选项告诉工具链，多大的变量可以被当作“小变量”放入 small-data section。

错误信息提示“lower small-data size limit”，意思是我需要减小 -G 选项后面的数值。通过减小这个值，可以让更少的变量被认为是“小变量”，从而将它们从拥挤的 small-data section 移到普通的 .data 和 .bss section。虽然访问这些普通段的变量效率会略微降低，但这可以彻底解决链接错误。故我最后采用较为直接的办法，将 -G 的值设为 0，即在 config.mk 文件的 CFLAGS 后加上这一限制 -G 0。

(2) 心得体会

通过在 OpenMIPS 平台上移植操作系统并运行 2048 游戏这一综合性实验，我对嵌入式系统开发，特别是其与通用软件开发的区别，有了更为深刻和具象的理解。

与在 PC 上编写 C 程序最显著的不同，在于嵌入式开发需要我们时刻与硬件打交道。通用 C 语言开发中，我们习惯于使用 printf、scanf 等标准库函数，这些函数背后是操作系统对硬件细节的高度封装。而在此次实验中，我们必须亲手实现这一切：比如，直接的内存映射 I/O。我们不再调用抽象的库函数，而是通过 REG8、REG32 等宏，直接读写 UART_BASE、GPIO_BASE 等内存地址来控制硬件。

此外，在高级语言的开发中，我们很少用到位运算。但是在这里，无论是初始化 UART 时设置数据格式，还是从 GPIO 读取按键状态时使用位掩码，都离不开精细的位运算。由于嵌入式环境资源受限，我们只能手动实现 print_board 中的整数到字符串转换，以及一个基于线性同余算法的 my_rand 伪随机数生成器。

与此同时，引入了 μC/OS-II 实时操作系统，程序不再是一个 main 函数从头跑到尾的线性流程，而是演变成了一系列并发执行的任务。一旦有了多任务，就必须考虑共享资源的访问冲突问题。在 uart_print_str 函数中，采用了 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 进行临界区保护，也正体现了操作系统课程学习到的原子操作这一知识点。

总而言之，这次实验是一次完美的理论与实践的结合。它将计算机体系结构、操作系统原理和 C 语言融会贯通，让我不再仅仅是知道操作系统可以运行多任务，而是亲手创建和调度了这些任务，加深了我对嵌入式系统“软硬件协同工作”本质的认识。这次经历为我后续更深入的系统级开发学习打下了坚实的基础。

这也是最后一次使用这块开发板了吧！

也终于知道了为什么在数字逻辑课上老师说有些同学的板子启动流水灯被刷掉了。

从开始的数字逻辑点灯和写子系统、到计算机组成原理搓 31 条和 54 条 CPU、再到计算机系统结构写静态和动态流水线 CPU，最后到计算机系统实验的 89 条动态流水线 CPU、操作系统移植和小程序开发，这块开发板承载了太多太多的回忆，有痛苦、有迷茫，但最多的还是成就感，毕竟是硬件之路上不多的陪伴者，我永远记得凌晨三点的通达馆，凌晨五点的寝室，人来人往，春去夏至，但板子一直都在身边！

参考资料

[1] 秦国锋, 王力生, 陆有军, 郭玉臣. 计算机系统结构实验指导, 清华大学出版社,2019.

[2] 雷思磊. 自己动手写 CPU, 电子工业出版社,2014.

[3] 张晨曦, 王志英等. 计算机体系结构, 高等教育出版社,2014.