Lab1：搭建CPU实验框架

3220106039 李瀚轩 2024.3.13

一、操作方法与实验步骤

基础知识

本次实验我们基于哈佛架构搭建了CPU实验的框架，完成了七段数码管驱动的 memory mapping IO的映射，使CPU能根据指定的地址访问到七段数码管。

哈佛架构与冯诺依曼架构最大的不同在于其将数据存储和指令存储分开，将 memory 分为 instruction memory 和 data memory 两部分，使得CPU科研同时读取指令和数据，减少了访问内存的冲突，提高了CPU的性能。

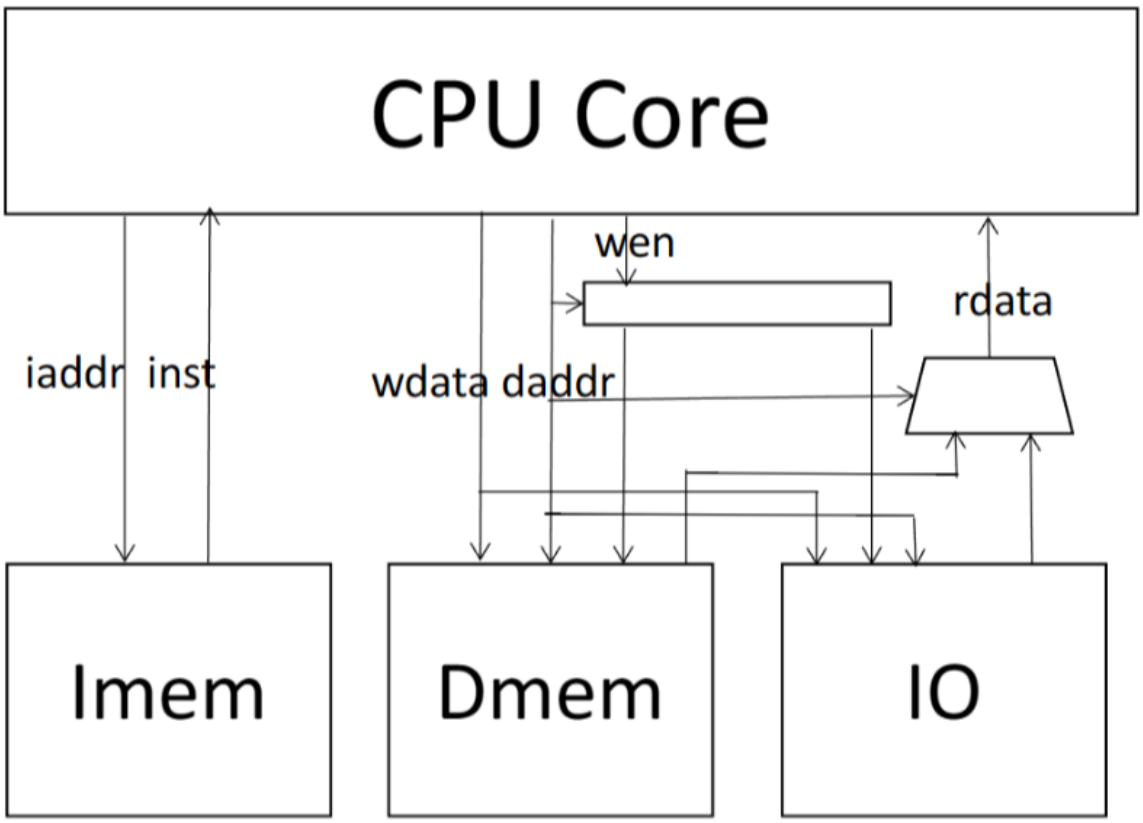
另一个基本概念是IO(input, output)， CPU与IO进行交互的形式之一是 memory mapping IO，将IO外设相关的寄存器和端口映射到某个地址上，当CPU访问某个地址时即在和对应的IO设备交互。

实验具体过程

1. 将lab1_src文件和lab0中实现的模块Add Source。
2. 实现MACtrl(memory access controller)模块。

该模块用于实现 memory mapping IO 的内存映射逻辑，根据CPU传入的访问地址来判断访问的是七段数码管驱动还是访问数据内存。原理图如下图所示，根据我的理解，该模块其实就是作为连接 CPU Core 和 Memory、IO 之间的桥梁。

在这次实验中，我们根据CPU访问的地址来判断是否访问七段数码管(当CPU访问地址为0xFE00000时访问七段数码管)，七段数码管在这里就是IO。



根据原理图以及各个端口的意义，可以写出该模块的代码：

```
module MACtrl(
 // face CPU
        input [31:0] i_iaddr, //CPU 当前访问的指令的地址。
        output [31:0] o_idata, //输出从指令内存中读取到的数据传入 CPU。
        input i_dwen, //CPU 传出的访问对应数据内存的写使能信号。
        input [31:0] i_daddr, //CPU 传出的访问数据内存的地址。
        input [31:0] i_d_idata, //CPU 传出的需要写入到数据内存的数据端口。
        output [31:0] o_d_odata, //传入 CPU 的从数据内存中读到的数据。
        // face IMEM
        output [31:0] o_iaddr, //传入 IMEM 的指令地址
        input [31:0] i_idata, //从 IMEM 中读取到的指令
        // face DMEM
        output o_dwen, //传入 DMEM 的写使能信号
        output [31:0] o_daddr, //访问数据内存的地址
        output [31:0] o_d_idata, //写入 DMEM 的数据端口
        input [31:0] i_d_odata, //从 DMEM 中读取到的数据。
        // face DispNum
        output o_drwen,
        output [31:0] o_dr_idata,
        input [31:0] i_dr_odata
    );

    assign o_iaddr = i_iaddr;
    assign o_idata = i_idata;
    assign o_daddr = i_daddr;
    assign o_d_idata = i_d_idata;
    assign o_d_odata = (i_daddr == 32'hFE000000) ? i_dr_odata : i_d_odata;
    assign o_dwen = (i_daddr == 32'hFE000000) ? 0 : i_dwen;
    assign o_drwen = (i_daddr == 32'hFE000000) ? i_dwen : 0;
    assign o_dr_idata = i_d_idata;

endmodule
```

3. 对MACtrl模块进行仿真。思路与仿真结果在实验结果与分析中给出。

4. 完成 RV32core.v 的连线。

在给的框架代码中，CPU的数据线是直接连到内存模块中的，但是现在我们需要通过MACtrl模块来实现对 CPU 和 memory 之间传输的控制，因此我们需要声明一些额外的线，来实现从 CPU 到 MACtrl 到 memory 回到 MACtrl 最后返回 CPU 的回路。明确了 MACtrl 的功能之后，可以完成连线工作，代码如下：

```
Core core(
        `Core_DBG_Arguments
        .clk(debug_clk),
        .rst(rst),
        .imem_addr(imem_addr), //从Core中输出访问指令内存的地址
        .imem_o_data(imem_o_data), // 输入
        .dmem_addr(dmem_addr),// 输出
        .dmem_o_data(dmem_o_data), // 输入
```

```
        .dmem_i_data(dmem_i_data), // 输出
        .dmem_wen(dmem_wen) // 输出
    );

    wire [31:0] imem_addr_wire;
    wire [31:0] imem_o_data_wire;
    wire [31:0] dmem_addr_wire;
    wire [31:0] dmem_i_data_wire;
    wire [31:0] dmem_o_data_wire;
    wire dmem_wen_wire;


    Mem mem(
        .i_addr(imem_addr_wire),
        .i_data(imem_o_data_wire),
        .clk(~debug_clk),
        .d_wen(dmem_wen_wire),
        .d_addr(dmem_addr_wire),
        .d_i_data(dmem_i_data_wire),
        .d_o_data(dmem_o_data_wire)
    );


    MACtrl memacc(
        //facing core
        .i_iaddr(imem_addr),
        .o_idata(imem_o_data),
        .i_dwen(dmem_wen),
        .i_daddr(dmem_addr),
        .i_d_idata(dmem_i_data),
        .o_d_odata(dmem_o_data),
        //facing IMem
        .o_iaddr(imem_addr_wire),
        .i_idata(imem_o_data_wire),
        //facing DMem
        .o_dwen(dmem_wen_wire),
        .o_daddr(dmem_addr_wire),
        .o_d_idata(dmem_i_data_wire),
        .i_d_odata(dmem_o_data_wire),
        //facing DR
        .o_drwen(dr_wen),
        .o_dr_idata(dr_i_data),
        .i_dr_odata(dr_o_data)
    );

    DispNum dr(
        .clk(led_clk),
        .rst(rst),
        .wen(dr_wen),
        .i_data(dr_i_data),
        .o_data(dr_o_data),
        .Segments(seg_ca),
        .AN(AN)
    );
```

5. 修改 data.mem 文件，将第二行最后四位数据改成学号后四位。

二、实验结果与分析

1. 首先我们通过仿真来验证 MACtrl 模块的正确性。

大体思路就是改变CPU访问的地址，检查数据内存和七段数码管驱动的使能信号是否正确地，以及传回CPU的从内存中读到的数据是否正确。仿真代码如下：

```
module MACtrl_sim(

    );
    reg [31:0] i_iaddr;
    reg i_dwen;
    reg [31:0] i_daddr;
    reg [31:0] i_d_idata;
    reg [31:0] i_idata;
    reg [31:0] i_d_odata;
    reg [31:0] i_dr_odata;
    wire [31:0] o_idata;
    wire [31:0] o_d_odata;
    wire [31:0] o_iaddr;
    wire o_dwen;
    wire [31:0] o_daddr;
    wire [31:0] o_d_idata;
    wire o_drwen;
    wire [31:0] o_dr_idata;

    MACtrl m0 (
        .i_iaddr(i_iaddr),
        .o_idata(o_idata),
        .i_dwen(i_dwen),
        .i_daddr(i_daddr),
        .i_d_idata(i_d_idata),
        .o_d_odata(o_d_odata),
        .o_iaddr(o_iaddr),
        .i_idata(i_idata),
        .o_dwen(o_dwen),
        .o_daddr(o_daddr),
        .o_d_idata(o_d_idata),
        .i_d_odata(i_d_odata),
        .o_drwen(o_drwen),
        .o_dr_idata(o_dr_idata),
        .i_dr_odata(i_dr_odata)
    );
    initial begin
        i_daddr <= 32'hFE000000;
        i_d_idata <= 32'h1;
        i_d_odata <= 32'h2;
```

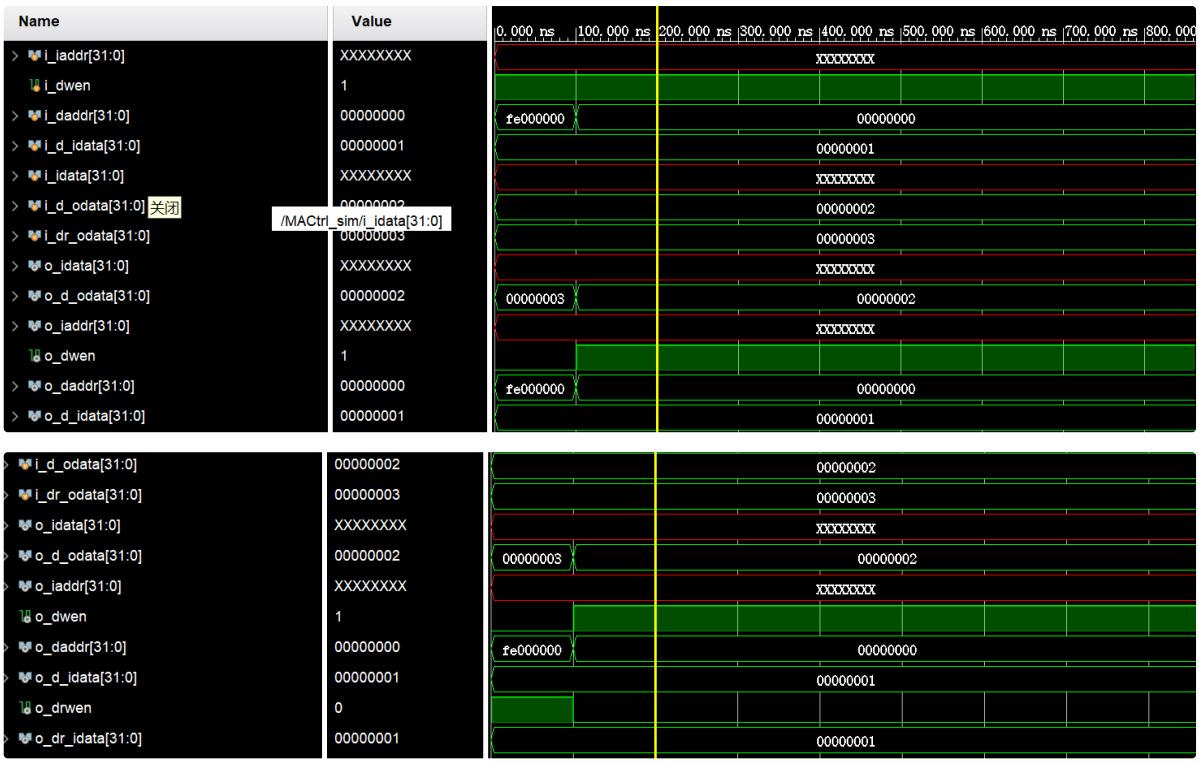
```
        i_dr_odata <= 32'h3;
        i_dwen <= 1;
        #100;
        i_daddr <= 32'h00000000;
        #100;
    end

endmodule

// 我的这个markdown主题，在代码块中会把非阻塞赋值的字符自动转化成小于等于号/(ToT)/~~
```

其中我将 `i_d_odata`（从 DMEM 中读取到的数据）和 `i_dr_odata`（从七段数码管驱动中读取到的数据）赋给不同的初值，便于根据仿真结果中 `o_d_odata`（传入 CPU 的从数据内存中读到的数据）的值判断是否正确访问对应的模块。将 `i_dwen` 赋值为 1 ，便于验证使能信号的正确性。

`i_daddr` 是CPU访问的地址，首先置为 `0xFE000000` ，此时访问七段数码管驱动。预期结果是 `o_drwen` 为 1 ，即七段数码管被使能， `o_dwen` 为 0， 即数据内存不被使能， `o_d_odata` 值为 `i_dr_odata` 的值，代表访问到七段数码管。

在100ns之后，我将 `i_daddr` 赋值为0，实际上只要不是 `0xFE000000` 即可。预期结果是 `o_drwen` 为 0 ，即七段数码管被禁止， `o_dwen` 为 1， 即数据内存被使能， `o_d_odata` 值为 `i_d_odata` 的值，代表访问到数据内存。

仿真波形如下所示，符合预期。



2. 验证上板结果。

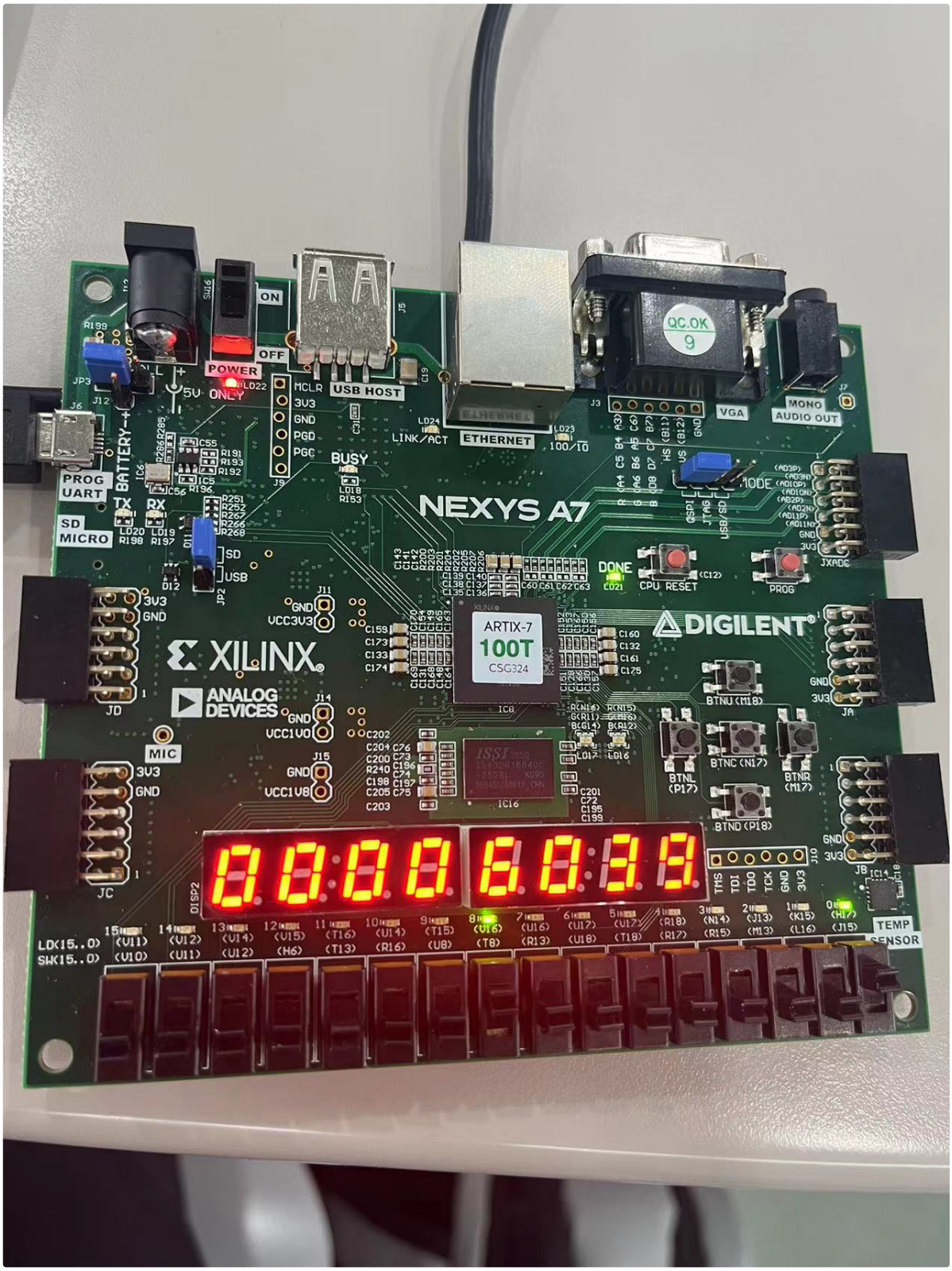
将生成的比特流上板，进行单步调试。每按一次会输出 Core 中 32 个寄存器的值，以及当前 Core 的 PC 值，指令的十六进制编码，访问内存的地址和数据。

观察寄存器的值随PC值变化的现象，都与实验文档里的值相符，可以验证上板正确。

这里给出 `PC = 0x0000003C` 和 `PC = 0X0000004C` 时的现象，由于截图太多，不一一给出，正确性已在验收时得到验证。

其中 `PC = 0x0000003C` 时，能看到七段数码管显示 `0000xxxx`，`xxxx` 为学号后四位，同时可以观察到 `x14` 变为 `0xFE000000` 。这里对应的就是在MACtrl中实现的功能，访问七段数码管驱动。如下图所示。

```
x01=0x12345678
x02=0x2468ACF0
x03=0x12345678
x04=0x00000002
x05=0x48D159E0
x06=0x00000001
x07=0x00000001
x08=0x12345678
x09=0x048D159E
x10=0x048D159E
x11=0x1234567A
x12=0x00000000
x13=0x00006039
x14=0xFE000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
PC     =0x0000003C
```



当 `PC = 0x0000004C` 时，`x16` 变为 `0x12345678` 随后不断地单步调试，PC 的值保持 `0x0000004C` 不变。如下图所示。

```
x04=0x00000002
x05=0x48D159E0
x06=0x00000001
x07=0x00000001
x08=0x12345678
x09=0x048D159E
x10=0x048D159E
x11=0x1234567A
x12=0x00000000
x13=0x00006039
x14=0xFE000000
x15=0x00006039
x16=0x12345678
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
PC     =0x0000004C
INST   =0x0000006F
MEMADDR=0x00000000
MEMDATA=0x00000000
```

三、思考题与心得

思考题

> 在 MACtrl 模块的实验原理图中，为什么 Core 传出的访问数据内存或者七段数码管的数据（wdata）直接同时接入两个模块中，而不需要像写使能（wen）一样需要根据 daddr 的值判断其传入到哪个模块?

倘若数据内存没有被使能，即使能信号为0，那么数据内存就不会被访问，七段数码管驱动同理。因此可以直接同时接入两个模块。而使能信号就必须根据 daddr 的值判断，以使能将要访问的模块，禁止另一个模块。

心得

本次实验刚开始的时候，由于我对哈佛架构、内存、IO这些概念非常陌生，不太理解本次实验的逻辑，虽然根据MACtrl各个端口的意义可以正确实现该模块，但是我不太明白这个模块的具体作用。

让我逐渐理解 memory mapping IO 的作用以及 MACtrl 存在的意义的是完成 RV32core.v 连线时遇到的困难。刚开始时，我并没有声明额外的线，直接将同一个变量从CPU连到MACtrl,再连到memory。在 implmentation 的时候报错，经过查询之后发现是因为同一根线连到了多个端口。这时候我再回看实验原理以及 MACtrl 的原理图，我意识到 MACtrl 其实充当了 CPU 和 Memory 之间联系的桥梁，CPU 传输给 MACtrl ， MACtrl 根据 CPU 传入的东西判断访问数据内存还是IO，并进行访问，最后将返回的数据传回 CPU。

对原理和作用清晰之后，整个连线就比较简单了，但是我在第一次上板进行单步调试时发现寄存器的值一直为0?这令我百思不得其解，一直以为是数据内存的那条通路出现了问题，仔细检查了好几遍，最后发现是有一条额外的线我没有声明就传入端口了(;´Д`) 。很奇怪的是生成比特流文件过程中这个情况不会报错，我上网也没有查到其中具体的机制。重新声明之后，结果就正确啦。

总而言之，本次实验我学到了很多，了解了哈佛架构，memory mapping IO ， CPU 和内存、IO之间的通信等系统相关知识，收获很多！！