同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称	μC/OS-II 操作系统移植
实验成员	冯羽芯（2251206）
日　　期	二零二五年 五 月 十九 日

1、实验目的

- 在 NexyS4DDRAtrix-7 开发板上实现 μC/OS-II 操作系统的移植
- 为后续系统上的程序开发奠定基础
- 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
- 开发平台：Vivado 2022.2
- 仿真环境：Vivado 2022.2 自带仿真器
- 文档管理：Office Word 2021
- 开发板： NEXYS4DDRAtrix-7

(2) 实验方案

参考《自己动手写 CPU》，在上一次实验的 MIPS 89 条指令动态流水线 CPU 的基础上，实现以下内容：

- Wishbone 总线
  - 为 OpenMIPS 增加 Wishbone 总线接口
  - 增加 Wishbone 总线
  - 增加 GPIO
  - 增加 UART 控制器
  - 增加 Flash 控制器
  - 增加 SDRAM 控制器
  - 实现完整的 SOPC
- 操作系统 μC/OS-II 的移植
  - 在 Ubuntu 上建立交叉编译环境
  - 对 μC/OS-II 系统进行改写与编译
- 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

3、实验步骤

(1) 系统设计总框图

在本次实验中，我们的 CPU 仍然采用哈佛结构，将程序指令存储和数据存储分开。在上一个实验中，指令存储器 ROM 和数据存储器 RAM 位于 FPGA 内部，由于测试程序和数据都较少，故内部的空间可以驾驭。但本次实验涉及操作系统的移植，程序体量大、传输数据多，仍在 FPGA 内部实验存储器不再合理。故我们使用在 FPGA 外部的 Flash 作为指令存储器，使用 SDRAM 作为数据存储器，所以我们需要添加 Flash 控制器和 SDRAM 控制器。同时，由于需要进行串口调试以对系统的正确性进行检查，我们还需要添加 UART 控制器和 GPIO 控制器。

总的来看，为了更好更方便地接入、移除和管理设备，我们需要增加 Wishbone 总线，OpenMIPS CPU 作为主设备，其指令总线和数据总线分别占据 Wishbone 总线的两个主设备接口。SDRAM、Flash、UART 和 GPIO 的控制器分别作为从设备连接至总线。

下图展示了改前和改后系统的总结构：

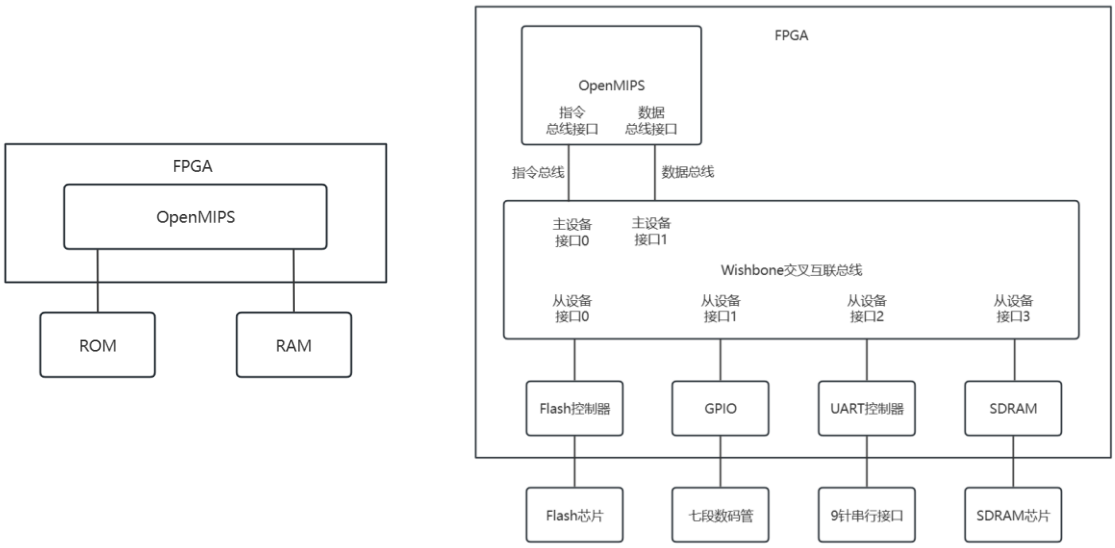


图 1：实验 1 与实验 2 的系统总结构框架图

(2) 修改 OpenMIPS 处理器

1. 添加 Wishbone 总线接口

Wishbone 有多种连接方式：点对点、数据流、共享总线、交叉互联等。图中输出信号的名称使用“_O”结束，输入信号的名称使用“_I”结束。此外，所有的信号都是高电平有效。

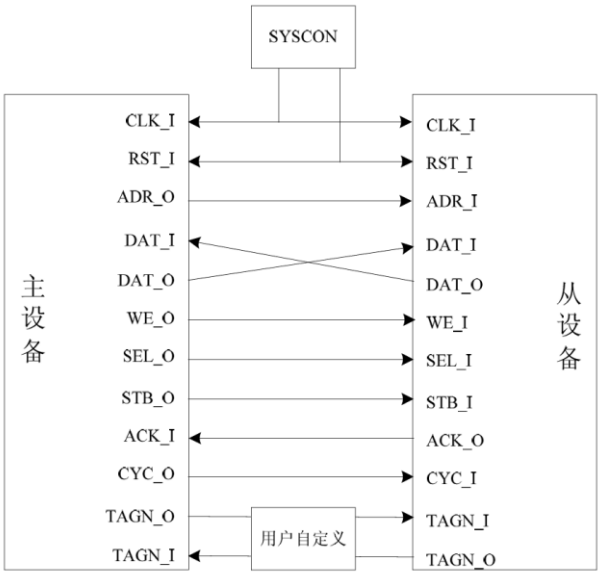


图 2：Wishbone 总线规范点对点连接方式

其中，具体的信号定义如下：

接口名称	接口含义
CLK_I、RST_I	时钟信号、复位信号
DAT_O/DAT_I	数据总线，数据可以由主设备传送给从设备，也可以由从设备传送给主设备
ADR_O/ADR_I	地址总线，地址由主设备传送给从设备
WE_O/WE_I	写使能信号，由主设备传送给从设备，代表当前进行的是写操作还是读操作，1 代表写操作，0 代表读操作
SEL_O/SEL_I	数据总线选择信号，用于标识当前操作中，数据总线上哪些比特是有效的，以总线粒度为单位。SEL_O/SEL_I 的宽度为数据总线宽度除以数据总线粒度
CYC_O/CYC_I	总线周期信号，CYC_O/CYC_I 有效代表一个主设备请求总线使用权或者正在占有总线，但是不一定正在进行总线操作
STB_O/STB_I	选通信号。选通信号有效代表主设备发起一次总线操作
ACK_O/ACK_I	主从设备间的操作结束信号，表示成功
TAGN_O/TAGN_I	用户可以利用标签信号传递自定义的信息

在添加接口之前，必须先了解 Wishbone 总线单次读写的过程。

• 单次读操作

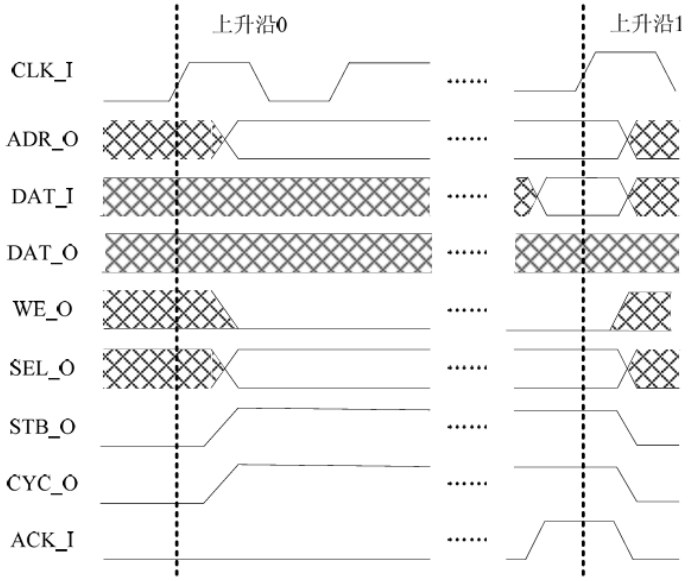


图 3：Wishbone 总线单次读操作时主设备的信号（不考虑 TAGN_O/TAGN_I）

在时钟上升沿 0，主设备将地址信号 ADR_O、适当的 SEL_O 放到总线上，随后将 WE_O 置低，表示读操作。

将 CYC_O、STB_O 置高表示一次总线操作开始。

在时钟上升沿 1 到达之前，从设备检测到主设备发起的操作，将适当的数据放到主设备的输入接口 DAT_I，同时将主设备的输入 ACK_I 置高，作为对主设备 STB_O 的响应。

从设备可以在设置 ACK_I 有效之前，插入任意数量的等待状态。在时钟上升沿 1，主设备发现 ACK_I 信号为高，于是采样 DAT_I 信号，作为读取到的数据，并将 CYC_O 和 STB_O 置低，表示操作完成。

从设备检测到 STB_O 置低后，将主设备的输入 ACK_I 也置低。

• 单次写操作

在时钟上升沿 0，主设备将地址信号 ADR_O、数据信号 DAT_O 放到总线上，将 WE_O 置高，表示写操作。

将适当的 SEL_O 放到总线上，以告诉从设备 DAT_O 中哪些字节是有效的。将 CYC_O、STB_O 置高，表示一次总线操作开始。

在时钟上升沿 1 到达之前，从设备检测到主设备发起的操作，于是锁存 DAT_O 的数据， 同时将主设备的输入 ACK_I 置高，作为对主设备 STB_O 的响应。从设备可以在设置 ACK_I 有效之前，插入任意数量的等待状态。

在时钟上升沿 1，主设备发现 ACK_I 信号为高，于是将 STB_O 和 CYC_O 置低，表示操作完成。从设备检测到 STB_O 置低后，将主设备的输入信号 ACK_I 也置低。

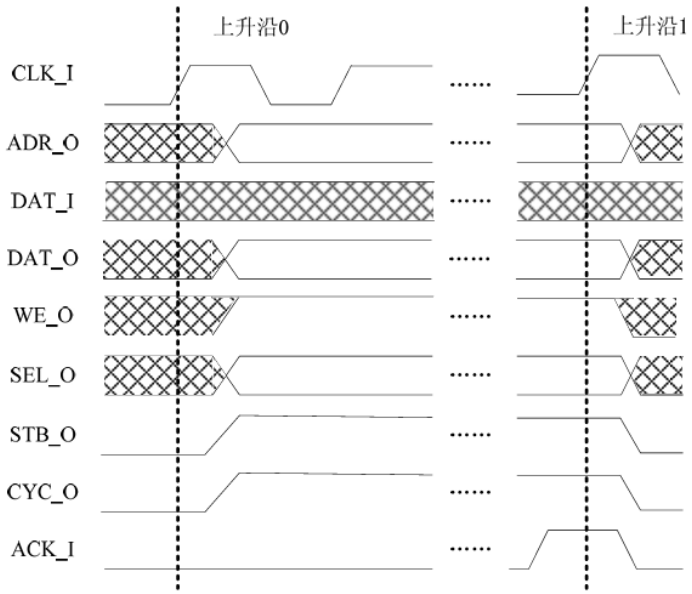


图 4：Wishbone 总线单次写操作时主设备的信号（不考虑 TAGN_O/TAGN_I）

随后编写 Wishbone 总线接口模块。

- **功能与设计：**使用有限状态机实现 Wishbone 总线接口模块。共有三个状态：
  - **空闲状态 WB_IDLE：**复位的时候进入空闲状态 WB_IDLE。当处于空闲状态 WB_IDLE 时，如果处理器发出了访问请求，且当前没有处于流水线清除过程中，那么会进入总线忙状态 WB_BUSY，开始访问总线。但是，如果处于流水线清除过程中，那么本次的总线访问当然会无效，所以不必进入 WB_BUSY 状态。
  - **总线忙状态 WB_BUSY：**当处于总线忙状态 WB_BUSY 时，如果收到 Wishbone 总线的响应，表示本次访问结束，此时需要判断流水线是否处于暂停状态。
    - 如果没有处于暂停状态，那么将访问到的数据送入处理器，进入空闲状态 WB_IDLE，等待下一次访问请求。
    - 如果处于暂停状态，那么将访问到的数据暂时保存起来，同时进入等待暂停结束状态 WB_WAIT_FOR_STALL。当流水线暂停结束时，再将访问到的数据送入处理器，并且进入空闲状态 WB_IDLE，等待下一次访问请求。

    当处于总线忙状态 WB_BUSY 时，如果发生了异常，那么会清除流水线，此时将直接取消此次 Wishbone 总线访问，并且回到状态 WB_IDLE。
  - **等待暂停结束状态 WB_WAIT_FOR_STALL**

下面作出上述状态机的状态转换示意图。

图 5：Wishbone 总线接口模块的状态机

• **模块设计**

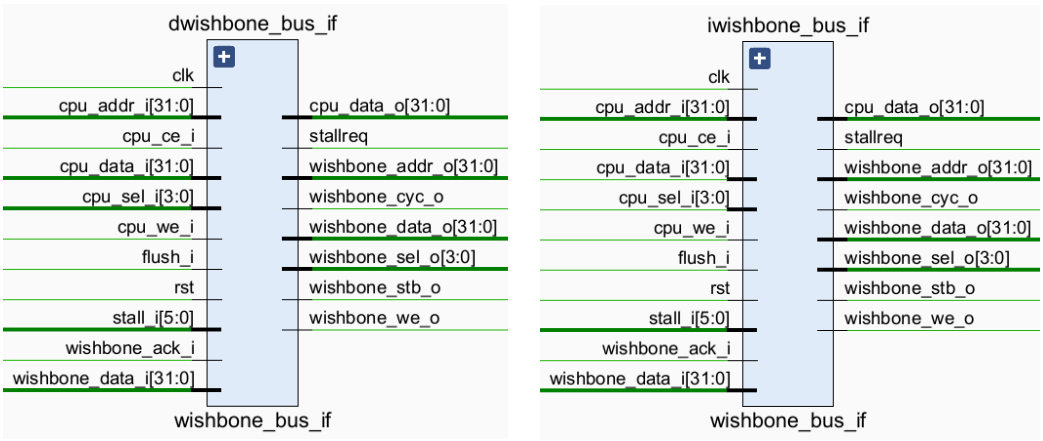


图 6：Wishbone 总线接口模块设计图

• **接口定义**

```
module wishbone_bus_if(
    input  wire         clk,
    input  wire         rst,

    // 来自 ctrl 模块
    input  wire[5:0]              stall_i,
    input  wire                   flush_i,     // 处理异常，需要清空流水线

    // CPU 侧的接口
    input  wire                   cpu_ce_i,    // 来自处理器的访问请求信号
    input  wire[`RegBus]          cpu_data_i,  // 来自处理器的数据
```


装 订 线

```
    input  wire[`RegBus]          cpu_addr_i,
    input  wire                   cpu_we_i,
    input  wire[3:0]              cpu_sel_i,
    output reg[`RegBus]           cpu_data_o,  // 输出到处理器的数据

    // Wishbone 侧的接口
    input  wire[`RegBus]          wishbone_data_i,  // Wishbone 总线输入的数据
    input  wire                   wishbone_ack_i,
    output reg[`RegBus]           wishbone_addr_o,
    output reg[`RegBus]           wishbone_data_o,
    output reg                    wishbone_we_o,
    output reg[3:0]               wishbone_sel_o,
    output reg                    wishbone_stb_o,   // Wishbone 总线选通信号
    output reg                    wishbone_cyc_o,   // Wishbone 总线周期信号

    output reg                    stallreq
);
```

2. ctrl.v

- **修改部分 1：**这里要注意，上个实验中实现的是适配 MARS 的 MIPS-CPU，MARS 中的程序起始地址为 32'h00400000，这里挂在总线上后，需要修改跳转地址，如下代码所示。

```
case (excepttype_i)
    32'h00000001: begin                 // 中断
        new_pc <= 32'h00000020;
    end
    32'h00000008: begin                 // 系统调用异常 syscall
        new_pc <= 32'h00000040;         // 中断例程地址
    end
    32'h0000000a: begin                 // 无效指令异常
        new_pc <= 32'h00000040;
    end
    32'h0000000d: begin                 // 自陷异常
        new_pc <= 32'h00000040;
    end
    32'h0000000c: begin                 // 溢出异常
        new_pc <= 32'h00000040;
    end
    32'h0000000e: begin                 // 异常返回指令 eret
        new_pc <= cp0_epc_i;
    end
    default: begin
    end
endcase
```

• **修改部分 2：**由于修改后取指是从 Flash 芯片中取、访存是通过 SDRAM 芯片，拿到数据的周期都超过一个时钟周期，故需要对流水线 CPU 进行暂停，指令与数据模块各有一个流水线暂停请求信号 `stallreq`，都输出到 CTRL 模块，分别表示取指阶段请求流水线暂停、访存阶段请求流水线暂停，所以要修改 CTRL 模块，添加部分接口。

```
module ctrl(
    ...
    input wire          stallreq_from_if,   // 来自取指阶段的暂停请求
    input wire          stallreq_from_mem,  // 来自访存阶段的暂停请求

    ...
);

always @ (*) begin
    if(rst == `RstEnable) begin
        ...
    end
    else if(excepttype_i != `ZeroWord) begin  // 不为 0，表示发生异常
        ...
    end
    else if(stallreq_from_mem == `Stop) begin
        stall <= 6'b011111;
        flush <= 1'b0;
    end
    else if(stallreq_from_ex == `Stop) begin
        ...
    end
    else if(stallreq_from_id == `Stop) begin
        ...
    end
    else if(stallreq_from_if == `Stop) begin
        stall <= 6'b000111;  // 译码阶段也暂停，保持了转移指令与延迟槽指令在流水线中的相对位置，从而能够正确识别出延迟槽指令
        flush <= 1'b0;       // 否则，填充的空指令被误认为是延迟槽指令
    end
    else begin
        ...
    end
end
endmodule
```

3. openmips.v

- **修改部分：**

在取指阶段添加了 Wishbone 总线接口模块，使得 PC 模块给出的指令存储器访问信号不再直接连接外部指令存储器，而是经过 Wishbone 总线接口模块转化为 Wishbone 总线接口信号。

由于指令存储器是只读的，并且指令宽度固定为 32 位，所以取指阶段添加的 Wishbone 总线接口模块的输入 `cpu_data_i` 直接设置为 `32'h00000000`；`cpu_we_i` 固定为 `1'0`，表示始终是读操作；`cpu_sel_i` 固定为 `4'b1111`。

添加 Wishbone 总线接口后，指令会存储在 FPGA 芯片外部的 Flash 中，导致取指时间多于 1 个时钟周期。在指令没有取到时，需要暂停流水线，所以在取指阶段添加的 Wishbone 接口模块有一个输出接口 `stalllreq`，连接到 CTRL 模块新增加的输入接口 `stallreq_from_if`，该信号表示取指阶段是否请求流水线暂停。

在访存阶段也添加了 Wishbone 总线接口模块，使得 MEM 模块对数据存储器的访问信号不再直接连接外部数据存储器，而是经过 Wishbone 总线接口模块转化为 Wishbone 总线接口信号。

添加 Wishbone 总线接口后，数据会存储在 FPGA 芯片外部的 SDRAM 中，导致访问数据的时间多于 1 个时钟周期。在数据没有访问到时，需要暂停流水线，所以在访存阶段添加的 Wishbone 接口模块也有一个输出接口 `stalllreq`，连接到 CTRL 模块新增加的输入接口 `stallreq_from_mem`，该信号表示访存阶段是否请求流水线暂停。

- **模块设计**

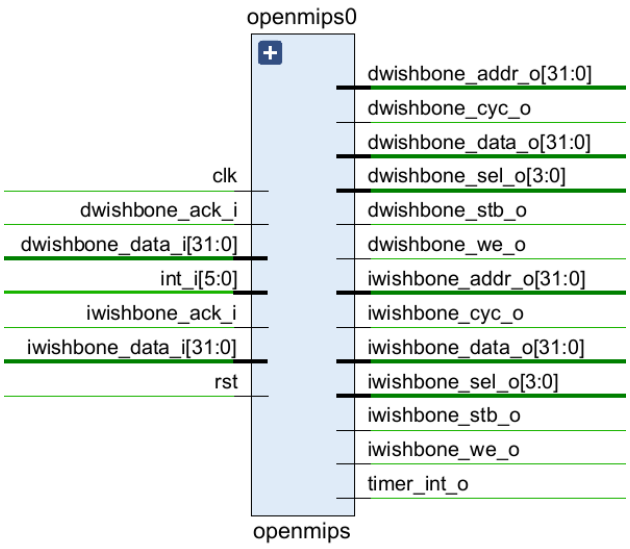


图 7：修改后 openmips 模块设计图

- **接口定义**

```
module openmips(
    input  wire                 clk,
    input  wire                 rst,

    input wire[5:0]             int_i,       // 6 个外部硬件中断输入

    // 指令 wishbone 总线
    input wire[`RegBus]           iwishbone_data_i,
    input wire                    iwishbone_ack_i,
    output wire[`RegBus]          iwishbone_addr_o,
    output wire[`RegBus]          iwishbone_data_o,
    output wire                   iwishbone_we_o,
    output wire[3:0]              iwishbone_sel_o,
    output wire                   iwishbone_stb_o,
    output wire                   iwishbone_cyc_o,

    // 数据 wishbone 总线
    input wire[`RegBus]           dwishbone_data_i,
    input wire                    dwishbone_ack_i,
    output wire[`RegBus]          dwishbone_addr_o,
    output wire[`RegBus]          dwishbone_data_o,
    output wire                   dwishbone_we_o,
    output wire[3:0]              dwishbone_sel_o,
    output wire                   dwishbone_stb_o,
    output wire                   dwishbone_cyc_o,

    output wire                   timer_int_o  // 是否有定时中断发生
);
```

4. pc_reg.v

- **修改部分：**Flash 对应的是从地址 0x30000000 开始的 256M 字节空间，所以需要修改 OpenMIPS 处理器，使得其在复位结束后从地址 0x30000000 处开始取指。只需要修改取指阶段的 PC 模块即可

```
    if (ce == `ChipDisable) begin
        pc <= 32'h30000000;       // 指令存储器禁用的时候，PC 为 0
    end
```

(3) 增加互联总线模块

由于本次实验中，至少有两台主设备（OpenMIPS 的数据总线和指令总线），和四台从设备，并且指令存储器和数据存储器需要同时访问，因此更倾向于使用交叉互联的方式实现。代码来自《自己动手写 CPU》的附件。

• **接口定义**

```
module wb_conmax_top(
    clk_i, rst_i,

    // Master 0 Interface
    m0_data_i, m0_data_o, m0_addr_i, m0_sel_i, m0_we_i, m0_cyc_i,
    m0_stb_i, m0_ack_o, m0_err_o, m0_rty_o,

    // Master 1 Interface
    m1_data_i, m1_data_o, m1_addr_i, m1_sel_i, m1_we_i, m1_cyc_i,
    m1_stb_i, m1_ack_o, m1_err_o, m1_rty_o,

    ...
    // Master 7 Interface
    m7_data_i, m7_data_o, m7_addr_i, m7_sel_i, m7_we_i, m7_cyc_i,
    m7_stb_i, m7_ack_o, m7_err_o, m7_rty_o,

    // Slave 0 Interface
    s0_data_i, s0_data_o, s0_addr_o, s0_sel_o, s0_we_o, s0_cyc_o,
    s0_stb_o, s0_ack_i, s0_err_i, s0_rty_i,

    // Slave 1 Interface
    s1_data_i, s1_data_o, s1_addr_o, s1_sel_o, s1_we_o, s1_cyc_o,
    s1_stb_o, s1_ack_i, s1_err_i, s1_rty_i,

    // Slave 2 Interface
    s2_data_i, s2_data_o, s2_addr_o, s2_sel_o, s2_we_o, s2_cyc_o,
    s2_stb_o, s2_ack_i, s2_err_i, s2_rty_i,

    ...
    // Slave 15 Interface
    s15_data_i, s15_data_o, s15_addr_o, s15_sel_o, s15_we_o, s15_cyc_o,
    s15_stb_o, s15_ack_i, s15_err_i, s15_rty_i
);
```

• **地址分配**

OpenMIPS 具有分开的指令、数据接口，所以占用 WB_CONMAX 两个主设备接口，其

中数据接口连接到主设备接口 0，指令接口连接到主设备接口 1。

SDRAM 控制器连接到从设备接口 0、UART 控制器连接到从设备接口 1、GPIO 连接到从设备接口 2、Flash 控制器连接到从设备接口 3。上述各个外设的寻址空间如下表所示。

名称	寻址空间
SDRAM	0x00000000-0x0FFFFFFF
UART	0x10000000-0x1FFFFFFF
GPIO	0x20000000-0x2FFFFFFF
Flash	0x30000000-0x3FFFFFFF

(4) 增加 GPIO 模块

代码来自《自己动手写 CPU》的附件。

• **功能**：GPIO（General Purpose Input Output）是以位为单位进行数字输入输出的 I/O 接口，作为单纯的通用输入/输出 I/O，输入时从外部读取输入信号，输出时将写入的值输出到外部。处理器通过 GPIO 可以与各种设备相连接，如 LED、开关、七段数码管等。

• **模块设计**

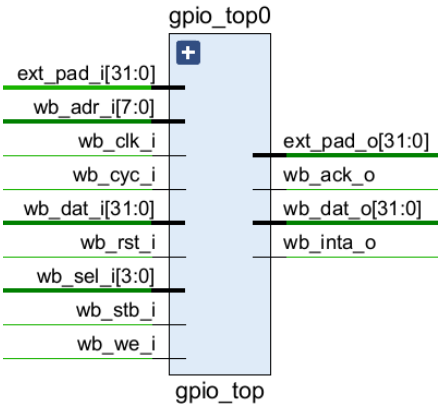


图 8：GPIO 模块设计图

• **接口定义**

```
module gpio_top(
  // WISHBONE Interface
  wb_clk_i, wb_rst_i, wb_cyc_i, wb_adr_i, wb_dat_i, wb_sel_i, wb_we_i,
wb_stb_i,
  wb_dat_o, wb_ack_o, wb_err_o, wb_inta_o,

`ifdef GPIO_AUX_IMPLEMENT
```

```
  // Auxiliary inputs interface
  aux_i,
`endif //  GPIO_AUX_IMPLEMENT

  // External GPIO Interface
  ext_pad_i, ext_pad_o, ext_padoe_o
`ifdef GPIO_CLKPAD
  , clk_pad_i
`endif
);
```

其中，本次实验 GPIO 将控制七段数码管，GPIO 中的主要寄存器如下：

寄存器名称	地址	宽度	访问方式	作用描述
RGPIO_IN	Base + 0x0	1~32	只读	输入到 GPIO 的信号
RGPIO_OUT	Base + 0x4	1~32	可读可写	GPIO 输出的信号
RGPIO_OE	Base + 0x8	1~32	可读可写	GPIO 输出接口使能信号
RGPIO_INTE	Base + 0xC	1~32	可读可写	中断使能信号

(5) 增加 UART 模块

代码来自《自己动手写 CPU》的附件。

• **功能：**UART 即通用异步收发器（Universal Asynchronous Receiver/Transmitter），是广泛使用的串行数据传输协议。它的功能是将并行的数据转变为串行的数据发送或者将接收到的串行数据转变为并行数据。

• **模块设计**

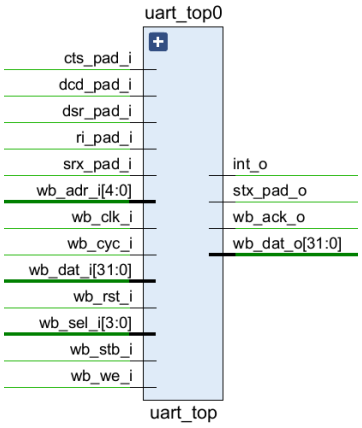


图 9：UART 模块设计图

• 接口定义

```
module uart_top (
    wb_clk_i,

    // Wishbone signals
    wb_rst_i, wb_adr_i, wb_dat_i, wb_dat_o, wb_we_i, wb_stb_i, wb_cyc_i,
wb_ack_o, wb_sel_i,
    int_o, // interrupt request

    // UART signals
    // serial input/output
    stx_pad_o, srx_pad_i,

    // modem signals
    rts_pad_o, cts_pad_i, dtr_pad_o, dsr_pad_i, ri_pad_i, dcd_pad_i
`ifdef UART_HAS_BAUDRATE_OUTPUT
    , baud_o
`endif
    );
```

• UART 的数据传输

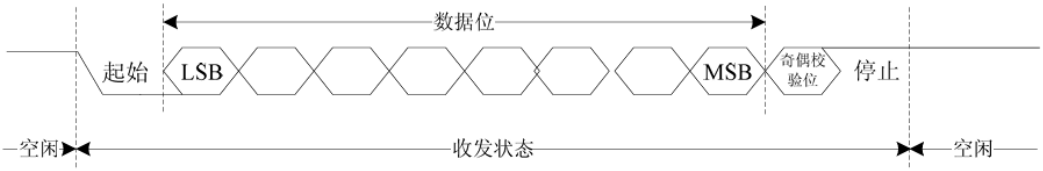


图 10：UART 的数据传输过程

UART 在传输的时候，将待传输数据的每个字符一位一位地传输，依次传输起始位、数据位、奇偶校验位、停止位。UART 常用的波特率有 9600 baud、19200 baud、38400 baud 等，本实验中，我们采用 9600 的波特率。

- 起始位：先发出一个低电平信号，也就是逻辑“0”，表示传输的开始。
- 数据位：紧接着起始位之后的是数据位。数据位的个数可以是 4、5、6、7、8 等，构成一个字符，从字符的最低位开始传送。
- 奇偶校验位：数据位之后是奇偶校验位。数据位加上这一位后，使得“1”的个数为偶数（偶校验）或奇数（奇校验），以此来判断数据传送的正确与否。
- 停止位：是一个字符数据的结束标志，可以是 1 位、1.5 位、2 位的高电平信号。

• **UART 的数据接收**

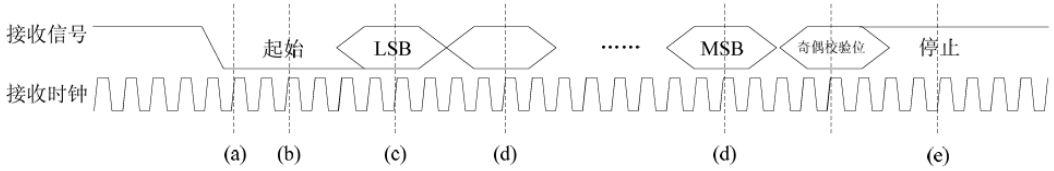


图 11：UART 的数据接收过程

UART 的数据接收部分采用比波特率高的采样频率实现。实际使用中，一般使用比波特率高 16 倍的接收时钟进行采样。

- 当接收信号由高电平变为低电平时，表示检测到起始位。
- 检测到起始位后，在接下来的第 2 个时钟周期检查接收信号，如果保持为低电平，说明确实是起始位，开始接收数据。否则认为起始位检测错误，将其忽略。
- 确定是起始位后，等待 4 个时钟周期检查接收信号，得到的值就是接收到的第一个 bit，也就是 LSB。
- 之后每隔 4 个时钟周期检查接收信号，依次得到传送过来的数据位、奇偶校验位。

• 分频系数寄存器

两个分频系数寄存器形成一个 16bit 的分频系数，其值需要依据系统时钟、波特率进行计算，计算方法如下。

分频系数 = 系统时钟 / （16 倍的波特率）

使用上式的结果设置分频系数寄存器，而且设置的时候，要先写高字节，也就是将分频系数的高 8 位写入寄存器 Divisor Latch Byte 2，再写低字节，也就是将分频系数的低 8 位写入寄存器 Divisor Latch Byte 1。

(6) 增加 Flash 模块

本模块未采用《自己动手写 CPU》中的代码，书中设计的 Flash 控制器针对的是 S29AL032D70TFI04 芯片，本次实验的 Nexys4 DDR 对应的 Flash 芯片是板载的 SPI-FLASH 模块，具体型号是 S25FL128S 芯片，如下图所示。

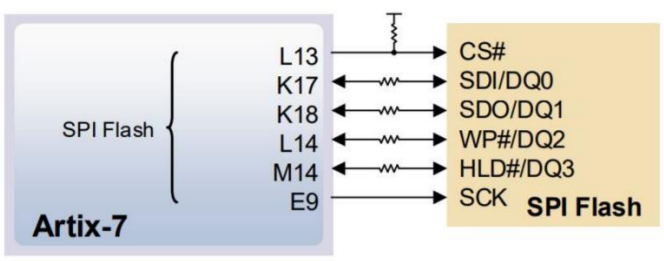


图 12：S25FL128S 芯片对应接口图

装 订 线

• **功能：**存储指令，由于本实验需要实现操作系统的移植，启动操作系统需要 Bootloader，以及操作系统中的内核指令，这些指令需要存储在 Flash 中，后续进行取指。

• **模块设计**

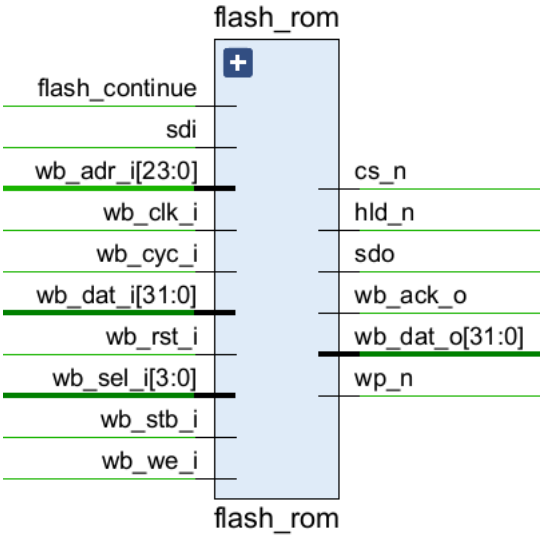


图 13：Flash 模块设计图

• **接口定义**

```
module flash_rom(
    // Wishbone 总线接口
    input wire wb_clk_i,        // Wishbone 时钟
    input wire wb_rst_i,        // Wishbone 复位
    input wire wb_cyc_i,        // Wishbone 总线周期有效
    input wire wb_stb_i,        // Wishbone 选通信号
    input wire wb_we_i,         // Wishbone 写使能
    input wire [3:0] wb_sel_i,  // Wishbone 字节选择
    input wire [23:0] wb_adr_i, // Wishbone 地址
    input wire [31:0] wb_dat_i, // Wishbone 写数据
    output reg [31:0] wb_dat_o, // Wishbone 读数据
    output reg        wb_ack_o, // Wishbone 应答

    input wire flash_continue, // Flash 继续操作信号

    // SPI Flash 接口
    output reg cs_n,   // 片选信号，低有效
    input  sdi,        // SPI 数据输入（从 Flash 到 FPGA）
    output reg sdo,    // SPI 数据输出（从 FPGA 到 Flash）
    output reg wp_n,   // 写保护，低有效
    output reg hld_n   // 保持信号，低有效
    );
```

该 Flash 采用 SPI 协议，SPI 协议是一种串行总线，即数据用一根线一位一位的传，这里需要用的的有 sck，cs_n，si，so 四根信号线

- sck 是时钟线，用来做信号的同步，频率在工作范围内随意
- cs_n 是片选信号，低电平有效，故当 cs_n 信号被拉低的时候，芯片开始工作
- si 是输入到主设备的信号线，在 Flash 芯片读取数据时起作用
- so 是输入到 Flash 的信号线，在 Flash 芯片写入数据时起作用

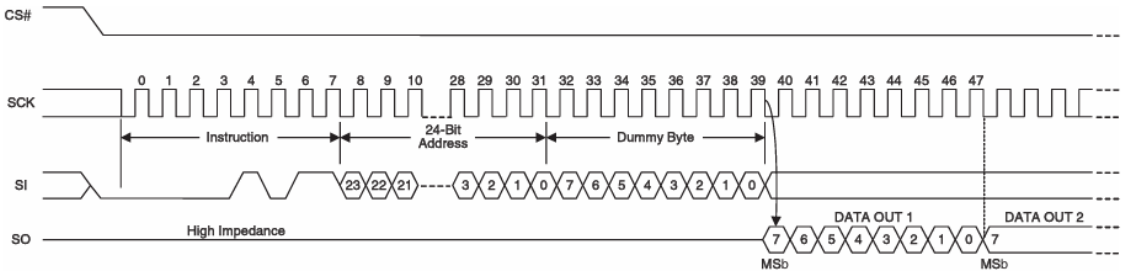


图 14：Flash 快速读取时序图

查阅 S25FL128S 官方文档可以得知 24-bit 地址的读取时序图，我们在本次实验中实现的也是 24 位地址的 Flash 模块。尽管 OpenMIPS-CPU 中的地址线是 32 位的，但是后续交叉编译得到的操作系统二进制文件的总大小为 31KB 左右，远没有达到 2 的 24 次方。

从图上看，我们要想读取数据，需要做的是先将 cs_n 拉低，再保持 sck 在工作范围内，查阅文档可知，READ 指令最大工作频率是 50MHz，故 sck 可选的频率在（0，50M]之间，由于 Nexys4 DDR 的板载晶振频率是 100MHz，这里用 50MHz 会比较好分频。

Table 13 AC characteristics (Single die package, $V_{IO} = V_{CC}$ 2.7 V to 3.6 V)

Symbol	Parameter	Min	Typ	Max	Unit
$F_{SCK, R}$	SCK clock frequency for READ and 4READ instructions	DC	–	50	MHz
$F_{SCK, C}$	SCK clock frequency for single commands as shown in Table 48[29]	DC	–	133	MHz
$F_{SCK, C}$	SCK clock frequency for the following dual and Quad commands: DOR, 4DOR, QOR, 4QOR, DIOR, 4DIOR, QIOR, 4QIOR	DC	–	104	MHz
$F_{SCK, QPP}$	SCK clock frequency for the QPP, 4QPP commands	DC	–	80	MHz

图 15：S25FL128S 官方文档对 SCK 频率的限制

然后通过 so 信号线一位一位的输入指令 READ(03H)，读取地址（24 位），最后 Flash 芯片就会通过 si 信号线一位一位的输出数据。

下面作出 Flash 模块的状态机：

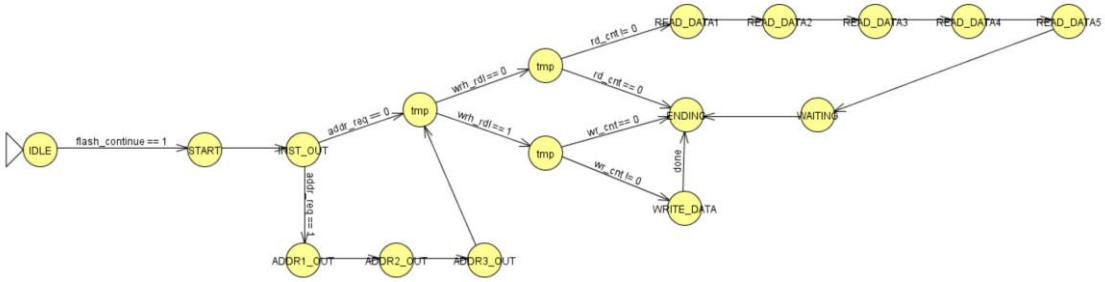


图 16：Flash 模块的状态机

各状态的转换关系解释如下：

• **IDLE (空闲状态)** 该状态是系统的初始状态，在 IDLE 状态下，会清零 wait_count 计数器。若 flash_continue 信号为高电平，状态机将转换到 START 状态以启动一次新的 Flash 操作；若 flash_continue 信号为低电平，则状态机将保持在 IDLE 状态。

• **START (开始状态)** 此状态由 IDLE 状态在 flash_continue 为高时进入。在 START 状态，模块会执行 SPI 操作的初始化，包括保存 Wishbone 总线传入的 24 位地址 wb_adr_i 到内部 addr 寄存器，使能 SPI 时钟 (sck_en 置高)，拉低 SPI 片选信号 (cs_n_d[0] 置低以选中 Flash 芯片)，并将读计数器 read_count 加 1。完成这些准备工作后，状态机自动转换到 INST_OUT 状态。

• **INST_OUT (指令输出状态)** 该状态从 START 状态转换而来。在 INST_OUT 状态期间，模块通过 SPI 接口串行输出存储在 instruction 寄存器中的 8 位操作指令，此串行输出过程由 sdo_count 计数器控制，共持续 16 个 wb_clk_i 时钟周期（对应产生 8 个 SPI 时钟 sck 周期）。

指令发送完成后，若 addr_req (地址请求) 标志为高，表示操作需要地址，则状态转换到 ADDR1_OUT；

若 addr_req 为低，则根据 wrh_rdl (写/读标志) 和相应的字节数计数器 (wr_cnt 或 rd_cnt) 判断：

若是写操作 (wrh_rdl 为高) 且有数据要写 (wr_cnt 非零)，则进入 WRITE_DATA 状态；

若是写操作但无数据可写 (wr_cnt 为零)，则直接进入 ENDING 状态；

若是读操作 (wrh_rdl 为低) 且有数据要读 (rd_cnt 非零)，则进入 READ_DATA1 状态；

若是读操作但无数据可读 (rd_cnt 为零)，则也直接进入 ENDING 状态。

• **ADDR1_OUT (地址高位输出状态)** 此状态由 INST_OUT 状态在需要发送地址时进入。

在 ADDR1_OUT 状态，模块串行输出 24 位地址 addr 中的最高 8 位 (即 addr[23:16])。该过程同样由 sdo_count 控制，持续 16 个 wb_clk_i 周期。地址高 8 位发送完成后，状态机转换到 ADDR2_OUT 状态。

• **ADDR2_OUT (地址中位输出状态)** 该状态从 ADDR1_OUT 状态转换而来。在 ADDR2_OUT 状态，模块继续串行输出地址 addr 的中间 8 位 (即 addr[15:8])。此过程由 sdo_count 控制，持续 16 个 wb_clk_i 周期。地址中 8 位发送完成后，状态机转换到 ADDR3_OUT 状态。

• **ADDR3_OUT (地址低位输出状态)** 此状态由 ADDR2_OUT 状态转换而来。在 ADDR3_OUT 状态，模块串行输出地址 addr 的最低 8 位 (即 addr[7:0])，并清零页计数器 page_count。此过程由 sdo_count 控制，持续 16 个 wb_clk_i 周期。地址全部发送完成后，状态转换逻辑与 INST_OUT 状态在 addr_req 为低时的逻辑相似：

根据 wrh_rdl 和相应的字节数计数器 (wr_cnt 或 rd_cnt) 判断，若为写操作且有数据，则进入 WRITE_DATA；

若为写操作无数据，则进入 ENDING；若为读操作且有数据，则进入 READ_DATA1；若为读操作无数据，则进入 ENDING。

• **WRITE_DATA (写数据状态)** 该状态可由 INST_OUT、ADDR3_OUT 或 WRITE_DATA 自身 (当写入多个字节时) 进入。在 WRITE_DATA 状态，模块串行输出一个字节的数据（代码中固定为 8'h5A，实际应用中通常来自 wb_dat_i 或其他数据源）。此过程由 sdo_count 控制，持续 16 个 wb_clk_i 周期，同时 page_count 会增加以跟踪已写字节数。若当前字节写入后 page_count 仍小于 (wr_cnt - 1)，表示还有数据需要写入，状态机将再次进入 WRITE_DATA 状态以处理下一个字节；若 page_count 大于等于 (wr_cnt - 1)，表示所有指定字节均已写入，状态机转换到 ENDING 状态。

• **READ_DATA1 (读数据字节 1 状态)** 此状态可由 INST_OUT 或 ADDR3_OUT 在确定为读操作且有数据要读时进入。在 READ_DATA1 状态，模块通过 SPI 接口串行接收一个字节的数据，并存入临时的 datain 寄存器。此过程由 sdo_count 控制，持续 16 个 wb_clk_i 周期，同时 page_count 增加。第一个字节接收完成后，状态机转换到 READ_DATA2 状态。

• **READ_DATA2/3/4 (读数据字节 2/3/4 状态)** 同理。

• **READ_DATA5 (读数据字节 5 状态)** 此状态从 READ_DATA4 转换而来。在 READ_DATA5 状态，模块串行接收第五个字节的数据（通常对应 32 位读操作的最后一个字节，如果仅读 4 字节，则此状态用于接收第 4 字节，代码中读取 5 个字节似乎是为了对齐或是一个特定设计），存入 read_data[7:0] (同时也更新 datain)。此过程由 sdo_count 控制，持续

16 个 wb_clk_i 周期，page_count 增加。数据全部接收完毕后，状态机转换到 WAITING 状态。（注：根据代码 rd_cnt <= 16'd4;，通常应只读 4 字节，READ_DATA5 可能对应第 4 字节，或者这里的注释可以理解为“读数据，直到读满 rd_cnt 个字节”，而当前状态机结构是固定读 5 个字节到 read_data，并将最后 4 个字节作为有效输出）。

• **WAITING (等待状态)** 该状态由 READ_DATA5 (即最后一个数据字节接收完毕后) 进入。在 WAITING 状态，模块会禁止 SPI 时钟 (sck_en 置低)，拉高 SPI 片选信号 (cs_n_d[0] 置高以释放 Flash 芯片)，并清零 sdo_count。完成这些收尾工作后，状态机转换到 ENDING 状态。

• **ENDING (结束状态)** 此状态是 SPI 操作（读或写）完成后的状态，可由 INST_OUT、ADDR3_OUT (在无数据操作时)、WRITE_DATA (所有字节写完后) 或 WAITING (所有字节读完并处理后) 进入。

在 ENDING 状态下，若 init_count 计数器为零且 Wishbone 应答信号 wb_ack_o 尚未置位，则将 wb_ack_o 置为高电平，向 Wishbone 主设备发出操作完成的应答。状态机从 ENDING 转换到 IDLE 的具体逻辑为：

若当前状态是 ENDING、Wishbone 周期仍然有效 (wb_cyc_i & wb_stb_i 为高)、wb_ack_o 未置位且 init_count > 0，则状态机在此同一时钟周期内（逻辑上）会被直接设置回 IDLE 状态，同时 init_count 递减；

或者，在 wb_ack_o 置位后，一旦 wb_cyc_i & wb_stb_i 变为无效，状态机也会转换到 IDLE；若 Wishbone 周期直接结束，也会回到 IDLE。

(7) 增加 SDRAM 控制器

本模块未采用《自己动手写 CPU》中的代码，本次实验使用的是 Nexys4DDR 板载的 DDR2 模块，使用 DDR2 模拟一块一次最小读写宽度 8bits，最大读写宽度 32bits 的内存。

NEYXS4DDR 里的 DDR2SDRAM 型号为 MT47H64M16HR-25:H，由于 DDR 协议比较复杂，Xilinx 提供了一个简化控制 DDR 的内存控制器 IP 核，Memory Interface Generator(MIG)，此次实验中就使用这个 IP 核来进行操作。

• 添加步骤：

首先在 IP Catalog 中搜索 MIG。

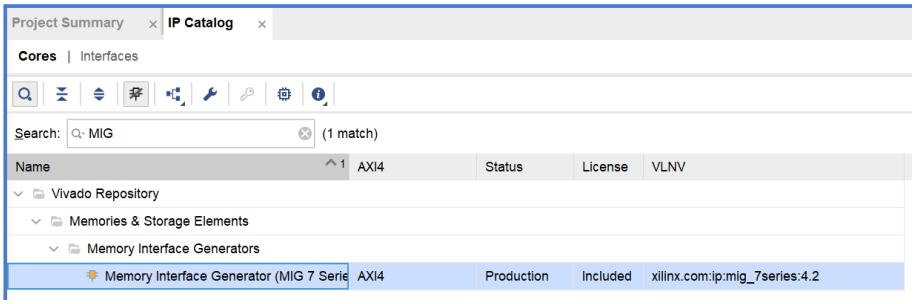


图 17：添加 IP 核步骤 1

双击 MIG，进入配置，在 Memory Seletion 中选择 DDR2 SDRAM，以下步骤中未作提示说明的部分，一律是默认配置。

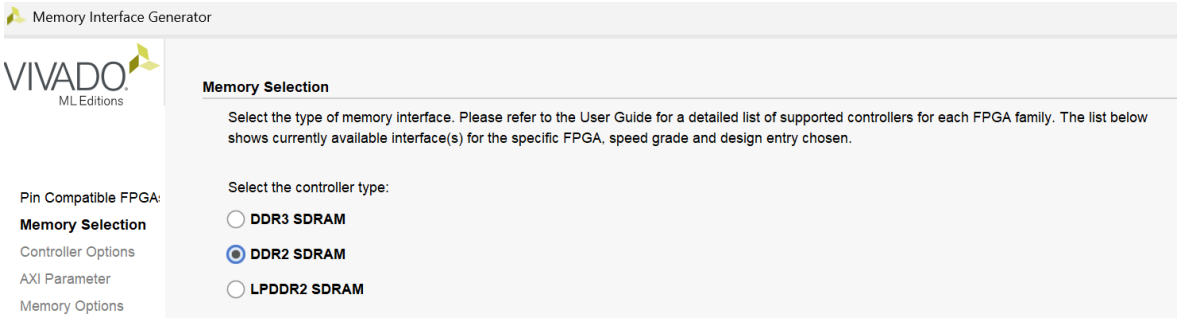


图 18：添加 IP 核步骤 2

在 Controller Options 中 Clock Period 改为 3333ps，Memory Part 选择 MT47H64M16HR-25E，Data Width 改为 16。

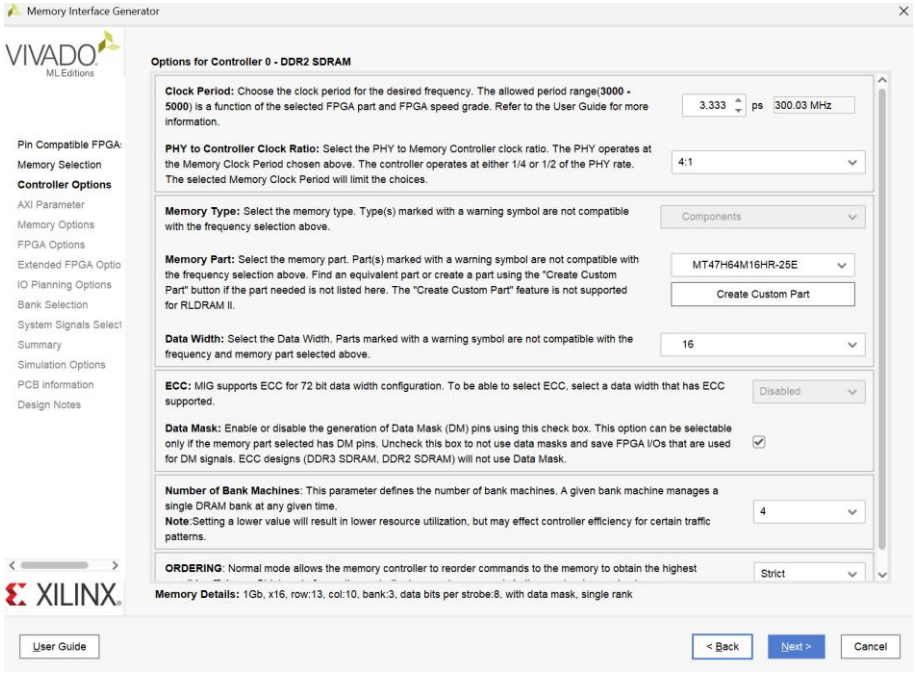


图 19：添加 IP 核步骤 3

在 Memory Option 部分，将 Input Clock Period 改为 5000ps，RTT 改为 75ohms。

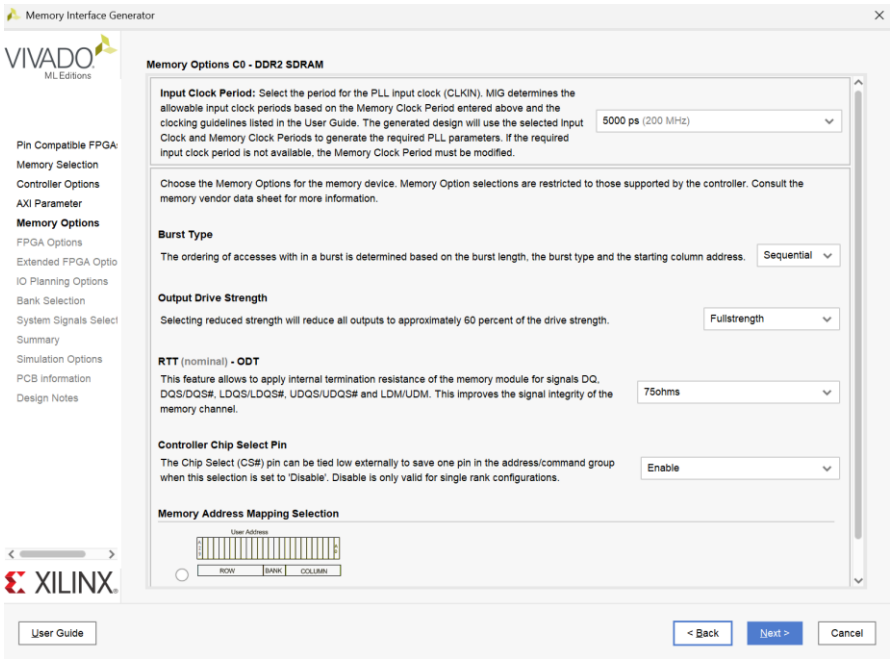


图 20：添加 IP 核步骤 4

在 FPGA Options 中，将 System Clock 改为 No Buffer，Reference Clock 改为 Use System Clock，将 Internal Vref 勾选，将 XADC Instantiation 改为 Disabled。

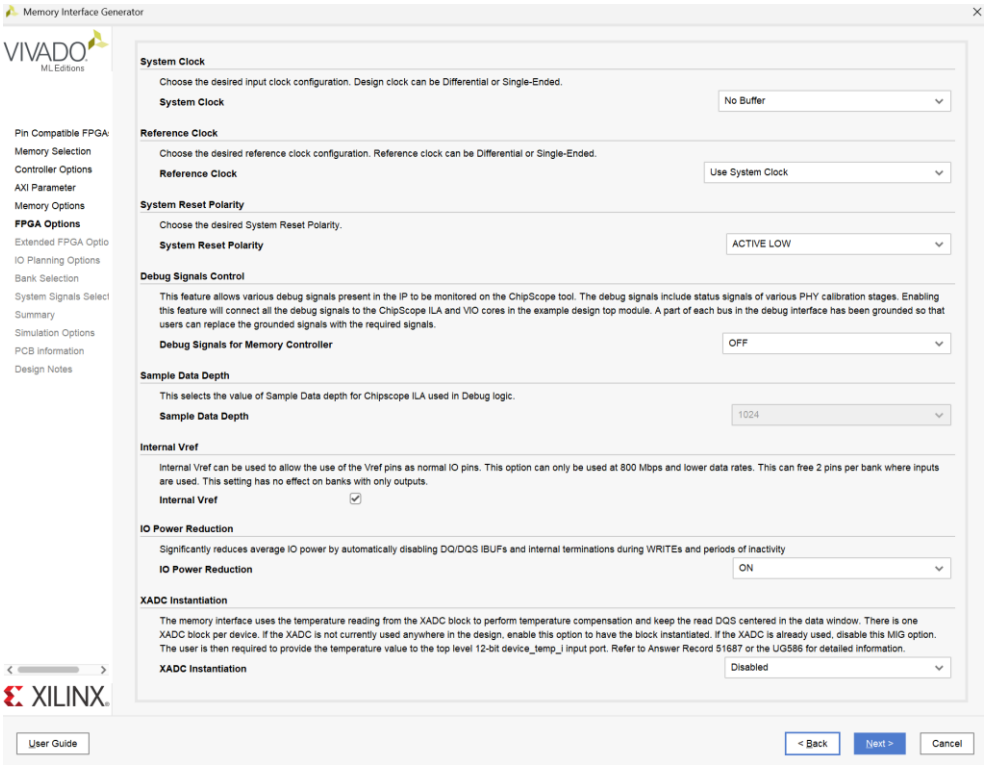


图 21：添加 IP 核步骤 5

在 IO Planning Options 中选择 Fixed Pin Out。

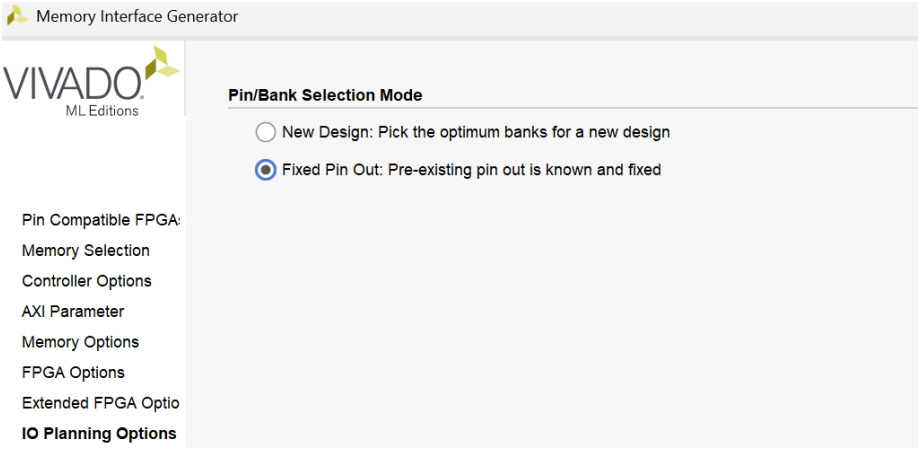
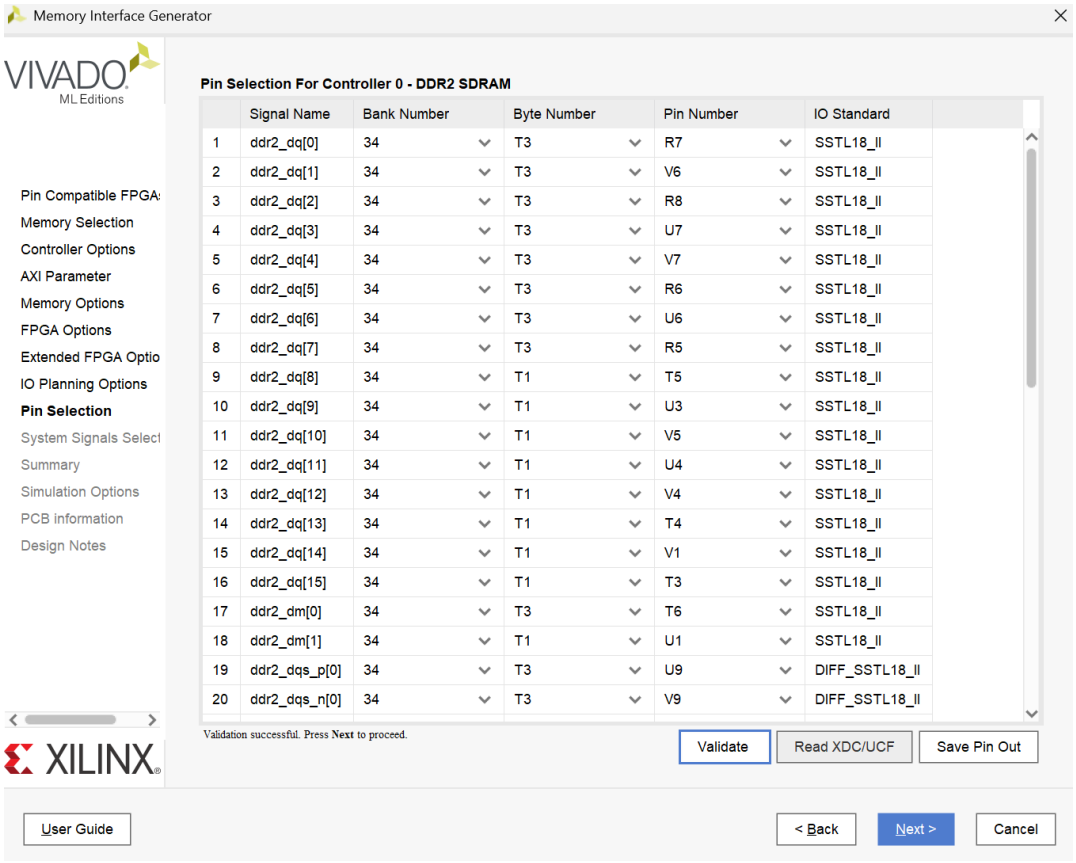


图 22：添加 IP 核步骤 6

在 Pin Seletion 中，点击 Read XDC/UCF，使用 Nexys4DDR_DMA_controller/PROJECT/Nexys4DDRmemorypinout.ucf at master · HackLinux/Nexys4DDR_DMA_controller 中下载的文件，导入后点击 validate。后续一路 Next 即可。



	Signal Name	Bank Number	Byte Number	Pin Number	IO Standard
1	ddr2_dq[0]	34	T3	R7	SSTL18_II
2	ddr2_dq[1]	34	T3	V6	SSTL18_II
3	ddr2_dq[2]	34	T3	R8	SSTL18_II
4	ddr2_dq[3]	34	T3	U7	SSTL18_II
5	ddr2_dq[4]	34	T3	V7	SSTL18_II
6	ddr2_dq[5]	34	T3	R6	SSTL18_II
7	ddr2_dq[6]	34	T3	U6	SSTL18_II
8	ddr2_dq[7]	34	T3	R5	SSTL18_II
9	ddr2_dq[8]	34	T1	T5	SSTL18_II
10	ddr2_dq[9]	34	T1	U3	SSTL18_II
11	ddr2_dq[10]	34	T1	V5	SSTL18_II
12	ddr2_dq[11]	34	T1	U4	SSTL18_II
13	ddr2_dq[12]	34	T1	V4	SSTL18_II
14	ddr2_dq[13]	34	T1	T4	SSTL18_II
15	ddr2_dq[14]	34	T1	V1	SSTL18_II
16	ddr2_dq[15]	34	T1	T3	SSTL18_II
17	ddr2_dm[0]	34	T3	T6	SSTL18_II
18	ddr2_dm[1]	34	T1	U1	SSTL18_II
19	ddr2_dqs_p[0]	34	T3	U9	DIFF_SSTL18_II
20	ddr2_dqs_n[0]	34	T3	V9	DIFF_SSTL18_II

图 23：添加 IP 核步骤 7

此外，我们还需一个转换接口，充当静态 RAM (SRAM) 接口到 DDR2 SDRAM 接口的转换器。简单来说，它允许一个系统像与简单的 SRAM 交互那样，去访问和控制实际连接的 DDR2 内存。我们需要使用 Nexys4 官方的 Ram2Ddr 的 VHDL 库文件。在这里下载 https://reference.digilentinc.com/_media/nexys4-ddr/nexys4ddrmemorypinout.zip，并把 Inst_DDR 后的名称改成刚刚生成 IP 核的名称。

```
------------------------------------------------------------------------------
-- DDR controller instance
------------------------------------------------------------------------------
   Inst_DDR: mig_7series_0
   port map (
      ddr2_dq              => ddr2_dq,
      ddr2_dqs_p           => ddr2_dqs_p,
      ddr2_dqs_n           => ddr2_dqs_n,
      ddr2_addr            => ddr2_addr,
      ddr2_ba              => ddr2_ba,
      ddr2_ras_n           => ddr2_ras_n,
      ddr2_cas_n           => ddr2_cas_n,
      ddr2_we_n            => ddr2_we_n,
      ddr2_ck_p            => ddr2_ck_p,
      ddr2_ck_n            => ddr2_ck_n,
      ddr2_cke             => ddr2_cke,
      ddr2_cs_n            => ddr2_cs_n,
      ddr2_dm              => ddr2_dm,
      ddr2_odt             => ddr2_odt,
      -- Inputs
      sys_clk_i            => clk_200MHz_i,
      sys_rst              => rstn,
      -- user interface signals
      app_addr             => mem_addr,
      app_cmd              => mem_cmd,
      app_en               => mem_en,
      app_wdf_data         => mem_wdf_data,
      app_wdf_end          => mem_wdf_end,
      app_wdf_mask         => mem_wdf_mask,
      app_wdf_wren         => mem_wdf_wren,
      app_rd_data          => mem_rd_data,
      app_rd_data_end      => mem_rd_data_end,
      app_rd_data_valid    => mem_rd_data_valid,
      app_rdy              => mem_rdy,
      app_wdf_rdy          => mem_wdf_rdy,
      app_sr_req           => '0',
      app_sr_active        => open,
```

```
    app_ref_req          => '0',
    app_ref_ack          => open,
    app_zq_req           => '0',
    app_zq_ack           => open,
    ui_clk               => mem_ui_clk,
    ui_clk_sync_rst      => mem_ui_rst,
    device_temp_i        => device_temp_i,
    init_calib_complete  => calib_complete);
```

在使用该 IP 核后，只需要考虑再在 IP 核操作行为上加以封装状态机，使得该 IP 核能与 WB 总线进行交互即可。

·功能：SDRAM（Synchronous Dynamic Random Access Memory）是同步动态随机访问存储器， 同步是指 Memory 工作需要同步时钟，内部命令的发送与数据的传输都以它为基准；动态是 指存储阵列需要不断地刷新以保证数据不丢失；随机访问是指数据不是线性依次读写，而是可以自由指定地址进行读/写。

·模块设计

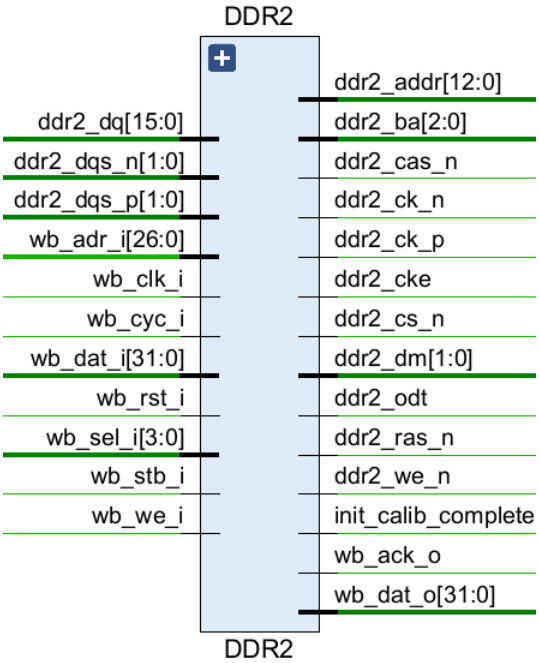


图 24：DDR2 模块设计图

·接口定义

```
module DDR2(
    input wire wb_clk_i,            // Wishbone 时钟
    input wire wb_rst_i,            // Wishbone 复位
    input wire wb_cyc_i,            // Wishbone 总线周期有效
```


装 订 线

```
    input wire wb_stb_i,            // Wishbone 选通信号
    input wire wb_we_i,             // Wishbone 写使能
    input wire [3:0] wb_sel_i,      // Wishbone 字节选择
    input wire [26:0] wb_adr_i,     // Wishbone 地址
    input wire [31:0] wb_dat_i,     // Wishbone 写数据
    output reg [31:0] wb_dat_o,     // Wishbone 读数据
    output reg        wb_ack_o,     // Wishbone 应答
    output reg init_calib_complete, // DDR2 初始化完成标志

    // DDR2 SDRAM 物理接口信号
    output   [12:0] ddr2_addr,
    output   [2:0] ddr2_ba,
    output   ddr2_ras_n,
    output   ddr2_cas_n,
    output   ddr2_we_n,
    output   ddr2_ck_p,
    output   ddr2_ck_n,
    output   ddr2_cke,
    output   ddr2_cs_n,
    output   [1:0] ddr2_dm,
    output   ddr2_odt,
    inout    [15:0] ddr2_dq,
    inout    [1:0] ddr2_dqs_p,
    inout    [1:0] ddr2_dqs_n
);
```

·状态机设计

为 DDR2 模块作出状态机图：

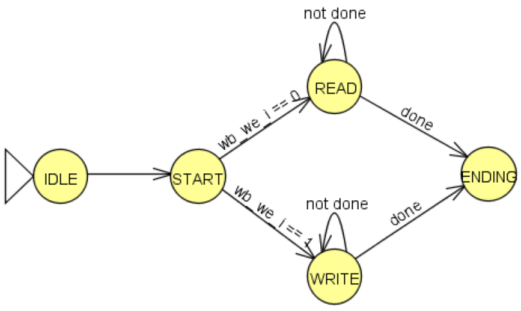


图 25：DDR2 模块状态机图

·IDLE (空闲状态) 该状态是系统的初始或待命状态，通常在系统复位、Wishbone 事务完成并由主设备撤销总线周期后进入。在 IDLE 状态下，与 Ram2Ddr 模块交互的内存控制信号（mem_cen、mem_oen、mem_wen）均被设置为非活动状态，内部的 wait_count 计数器被清

零。当检测到有效的 Wishbone 周期（wb_cyc_i & wb_stb_i 为高）时，状态机无条件地将下一状态设置为 START，准备解析并执行新的总线请求。

• **START (开始状态)** 此状态由 IDLE 状态在接收到有效 Wishbone 周期时转换而来。在进入 START 状态时，它会首先锁存从 Wishbone 总线传入的地址 wb_adr_i 到内部的 mem_a 寄存器中。

紧接着，它会根据 Wishbone 写使能信号 wb_we_i 的值来决定后续操作：

如果 wb_we_i 为高电平，表示这是一个写请求，状态机将转换到 WRITE 状态；

如果 wb_we_i 为低电平，则表示是一个读请求，状态机将转换到 READ 状态。

• **WRITE (写状态)** 当 START 状态判断当前 Wishbone 请求为写操作时，便会进入此 WRITE 状态。在此状态期间，内存接口的相关控制信号被设置为执行写操作的模式（即 mem_cen 和 mem_wen 有效，而 mem_oen 无效），同时将锁存的 Wishbone 地址 mem_a 和当前的 Wishbone 写数据 wb_dat_i (通过 mem_dq_i) 提供给 Ram2Ddr 模块。内部的 wait_count 计数器会在此状态下持续递增。只要 wait_count 的值小于预设的等待阈值 16'd80，状态机就会保持在 WRITE 状态以允许足够的时间进行写操作；一旦 wait_count 达到或超过 16'd80，表示写操作的必要时序已满足，此时内存接口控制信号会恢复到非活动状态，然后状态机转换到 ENDING 状态。

• **READ (读状态)** 当 START 状态判断当前 Wishbone 请求为读操作时，便会进入此 READ 状态。在此状态期间，内存接口的相关控制信号被设置为执行读操作的模式（即 mem_cen 和 mem_oen 有效，而 mem_wen 无效），同时将锁存的 Wishbone 地址 mem_a 提供给 Ram2Ddr 模块。内部的 wait_count 计数器会在此状态下持续递增。只要 wait_count 的值小于预设的等待阈值 16'd80，状态机就会保持在 READ 状态以等待数据从内存中读取；一旦 wait_count 达到或超过 16'd80，表示读操作的必要时序已满足，此时内存接口控制信号会恢复到非活动状态，从 Ram2Ddr 模块通过 mem_dq_o 返回的读数据会被驱动到 wb_dat_o 输出端口，随后状态机转换到 ENDING 状态。

• **ENDING (结束状态)** 该状态在 WRITE 或 READ 状态完成了相应的操作（即 wait_count 达到阈值）后进入。当状态机进入或保持在 ENDING 状态时，如果 Wishbone 应答信号 wb_ack_o 尚未被置位，它将被设置为高电平 1'b1，以此向 Wishbone 主设备指示当前的读/写操作已经结束。状态机将停留在 ENDING 状态并持续输出 wb_ack_o，直到 Wishbone 主设备响应该应答并撤销当前的 Wishbone 周期（即 wb_cyc_i 和 wb_stb_i 变为无效）。一旦总线周期结束，状态机将依据全局转换规则返回到 IDLE 状态，为下一次操作做准备。

装 订 线

(8) 编写顶层模块 openmips_min_sopc.v

•**功能：**将 OpenMIPS 处理器、Wishbone 总线互联 1 矩阵、GPIO 模块、UART 控制器、Flash 控制器、SDRAM 控制器连接起来。

• **模块设计**

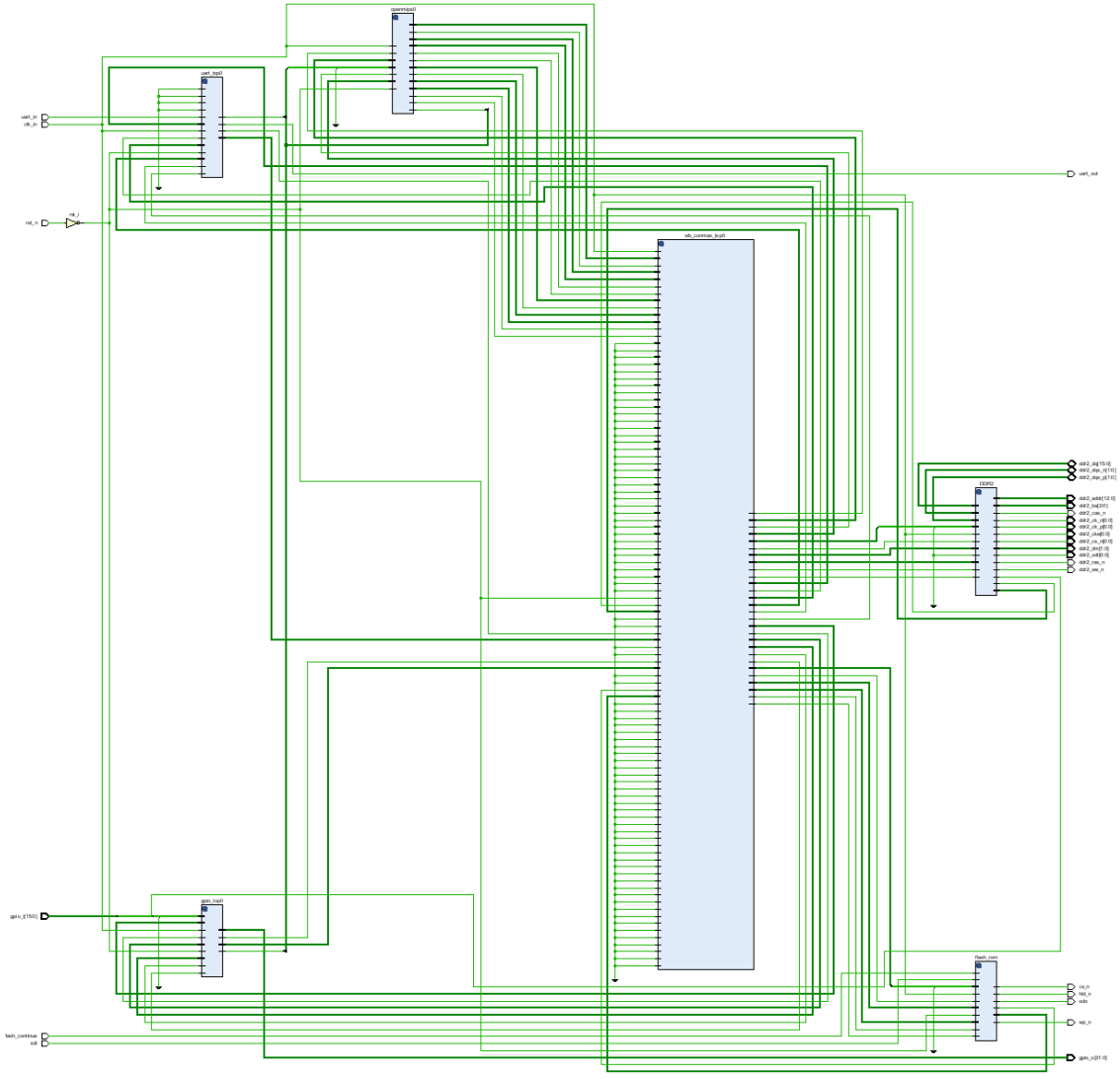


图 26：openmips_min_sopc 模块设计图

• **接口定义**

```
module openmips_min_sopc(

    input  wire         clk_in,
    input  wire         rst_n,
    input  wire         flash_continue,
```

```
    // 新增 UART 接口
    input  wire          uart_in,
    output wire          uart_out,

    // 新增 16 位输入接口
    input  wire[15:0]    gpio_i,

    // 新增 32 位输出接口
    output wire[31:0]    gpio_o, // 31:16-led, 15:8-o_seg, 7:0-o_sel

    // 新增与外部 Flash 相连的接口
    // Flash
    output  cs_n,
    input   sdi,
    output  sdo,
    output  wp_n,
    output  hld_n,

    // 新增与外部 SDRAM 相连的接口
    // DDR2
    inout [15:0]           ddr2_dq,
    inout [1:0]            ddr2_dqs_n,
    inout [1:0]            ddr2_dqs_p,
    output [12:0]          ddr2_addr,
    output [2:0]           ddr2_ba,
    output                 ddr2_ras_n,
    output                 ddr2_cas_n,
    output                 ddr2_we_n,
    output [0:0]           ddr2_ck_p,
    output [0:0]           ddr2_ck_n,
    output [0:0]           ddr2_cke,
    output [0:0]           ddr2_cs_n,
    output [1:0]           ddr2_dm,
    output [0:0]           ddr2_odt

);
```

(9) 建立交叉编译环境

1. Linux 环境的配置

本实验需要在 Linux 虚拟机环境下进行。可以采用 VMware Workstation 并在其中安装 Linux 系统，使用 Ubuntu 的光盘映像文件进行安装；同样可以在 Windows 系统中配置 WSL(Windows

Subsystem for Linux)，安装教程见 Windows Subsystem for Linux (WSL, Ubuntu) 最新安装教程（2024.11 更新）-CSDN 博客，在此不作过多赘述。

2. MIPS 编译环境的建立

首先需要安装 GNU 工具链。《自己动手写 CPU》一书的附件中提供了安装文件，mips-sde-elf- i686-pc-linux-gnu.tar.tar，将安装文件复制到 Ubuntu 的 /opt 目录下，打开 Ubuntu 的终端，使用如下命令解压缩：

```
cd /opt
tar vfxj mips-sde-elf-i686-pc-linux-gnu.tar.tar
```

然后在 Home 文件夹中找到隐藏文件.bashrc，可以在终端输入 code . 在 VSCode 中打开，便于查看隐藏文件，在此文件的最后加入环境变量的设置：

```
export PATH="$PATH:/opt/mips-4.3/bin"
```

重新启动 Ubuntu 系统，重启后，打开终端，在其中输入 mips-sde-elf-，然后按两次 Tab 键，会列出刚刚安装的针对 MIPS 平台的所有编译工具，如下图所示，即安装成功。

```
chenovo@LAPTOP-MGRUHPKO:/opt$ mips-sde-elf-
mips-sde-elf-addr2line  mips-sde-elf-cpp        mips-sde-elf-gdbtui     mips-sde-elf-ranlib
mips-sde-elf-ar         mips-sde-elf-g++        mips-sde-elf-gprof      mips-sde-elf-readelf
mips-sde-elf-as         mips-sde-elf-gcc        mips-sde-elf-ld         mips-sde-elf-run
mips-sde-elf-c++        mips-sde-elf-gcc-4.3.2  mips-sde-elf-nm         mips-sde-elf-size
mips-sde-elf-c++filt    mips-sde-elf-gcov       mips-sde-elf-objcopy    mips-sde-elf-strings
mips-sde-elf-conv       mips-sde-elf-gdb        mips-sde-elf-objdump    mips-sde-elf-strip
```

图 27：GNU 工具链安装成功示意图

(10) 编写 BootLoader

1. 修改 BootLoader.S

本文件在《自己动手写 CPU》中的 BootLoader.S 文件上修改。

• **功能：**BootLoader 可以用于模拟操作系统加载的过程。BootLoader 存放在 Flash 从 0x0 处开始的空间，操作系统存放在 Flash 从 0x304 处开始的空间，另外，在 Flash 的 0x300 处存放的是操作系统的长度信息。OpenMIPS 启动后，会首先执行 BootLoader。BootLoader 读取存放在 Flash 的 0x300 处的长度信息 length，根据该信息，将 Flash 从 0x304 处开始的 length 个字，依次复制到 SDRAM 从 0x0 处开始的空间，也就是将操作系统读取到 SDRAM。读取结束后，跳转到 SDRAM 的 0x0 地址，将控制权交给操作系统。

本处只实现了 UART 的回显，当 PC 通过 UART 发送数据给小型 SOPC 时，会引发 UART 控制器的中断，操作系统中的中断处理程序会读取传递过来的数据，然后回送给 PC，从而实现 UART

的回显。

- **模块设计**：该 BootLoader 含有九个模块，分别是：
  - UART 控制器初始化：设置分频系数的高低字节，并设置数据格式为 8 位数据位、没有奇偶校验位、1 位停止位。
  - GPIO 模块初始化：使能所有 GPIO 输出端口并禁止 GPIO 输入中断。
  - 等待 SDRAM 初始化完毕：获取 GPIO 的输入并判断第 16 位是否为 1。
  - 显示启动开始字符串。
  - 获取 OS 的长度信息：获取 Flash 的 0x300 处存放的 OS 长度信息并保存到寄存器 $1。
  - 将 OS 复制到 SDRAM。
  - 显示启动结束字符串。
  - 定义串口输出函数。
  - 一些预定义信息。
- **修改部分**

主要是将分频系数修改为适合当前实现的 CPU 的主频。Nexys4 DDR 对应 E3 管脚的时钟频率为 100MHz，根据分频系数计算公式：

分频系数 = 系统时钟 / （16 倍的波特率）

这里可以得到分频系数为 $\frac{100000000}{16*9600}=651$（已取整），转换为十六进得到 028BH。

```
lui $1,0x1000
ori $1,$1,0x0001
ori $2,$0,0x02
sb  $2,0x0($1)     # 向地址 0x10000001 写入 0x02，此时对应的是分频系数的高字节

lui $1,0x1000
ori $1,$1,0x0000
ori $2,$0,0x8B
sb  $2,0x0($1)     # 向地址 0x10000000 写入 0x8B，此时对应的是分频系数的低字节
```

2. 得到 BootLoader.bin 二进制文件

首先需要 Makefile 文件，Makefile 代码来自《自己动手写 CPU》的附件。

得到可以使用的指令存储器 ROM 初始化文件一共需要 3 步：编译、链接、得到 bin 文件，如下：

```
编译： mips-sde-elf-as -mips32 BootLoader.S -o BootLoader.o
链接： mips-sde-elf-ld -T ram.ld BootLoader.o -o BootLoader.om
```

得到 bin 文件：`mips-sde-elf-objcopy -O binary BootLoader.om BootLoader.bin`

Makefile 文件即把上述几个步骤写在一起。

其次，需要 ram.ld 文件，这是一个链接描述脚本，描述了输入文件的各个 Section 如何映射到输出文件的各个 Section 中，并控制输出文件中 Section 和符号的内存布局。该文件也直接采用《自己动手写 CPU》附件。

最后将上述三个文件放置在同一个目录下，如下图所示。

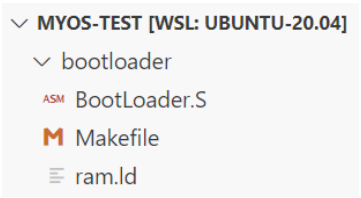


图 28：BootLoader 目录结构

在终端输入 `make all`，得到如下文件，保存好 BootLoader.bin。

```
chenovo@LAPTOP-MGRUHPKO:~/MyOS-test/bootloader$ make all
mips-sde-elf-as -mips32 BootLoader.S -o BootLoader.o
BootLoader.S: Assembler messages:
BootLoader.S:65: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:66: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:103: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:104: Warning: Macro instruction expanded into multiple instructions
mips-sde-elf-ld -T ram.ld BootLoader.o -o BootLoader.om
mips-sde-elf-objcopy -O binary BootLoader.om  BootLoader.bin
mips-sde-elf-objdump -D BootLoader.om > BootLoader.asm
```

图 29：BootLoader 的编译

(11) 为 OpenMIPS 处理器移植 μC/OS-II

1. 建立文件目录

• 首先在 Ubuntu 虚拟机中新建文件夹 MyOS-test，这里我将刚刚的 BootLoader 文件夹也放在该目录下，其实真正实验时秩序刚刚得到的 BootLoader.bin 文件。

• 在主目录下建立操作系统目录 ucosii_OpenMIPS。

• 在 ucosii_OpenMIPS 目录下新建文件夹 ucos，将 μC/OS-II 源代码的文件（除了 os_cfg_r.h、ucos_ii.h 两个头文件之外）复制到 ucos 文件夹下。

• 在 ucosii_OpenMIPS 目录下新建文件夹 port，将针对 MIPS M14K 的 μC/OS-II 移植代码中的 os_cpu_a.S、os_cpu_c.c 两个文件复制到该文件夹下。

• 在 ucosii_OpenMIPS 目录下新建文件夹 include，将 μC/OS-II 源代码中的 ucos_ii.h、os_cfg_r.h 两个头文件，以及针对 MIPS M14K 的 μC/OS-II 移植代码中的 cpu.h、os_cpu.h 两个头文件，一共四个头文件复制到该文件夹下，并将 os_cfg_r.h 重命名为 os_cfg.h。

• 针对 MIPS M14K 的 μC/OS-II 移植代码中的 os_cpu_c.c 文件需要引用 includes.h 文件，在 include 目录下新建文件 includes.h，内容如下：

```
#include <stdarg.h>
#include <stddef.h>
#include <limits.h>
#include "ucos_ii.h"
```

• 在 include 目录下新建文件 app_cfg.h，建立定时器任务的优先级，内容如下：

```
#ifndef _APP_CFG_H_
#define _APP_CFG_H_

#define  OS_TASK_TMR_PRIO (OS_LOWEST_PRIO - 2)

#endif
```

• 修改 includes 目录下的 cpu.h 文件，去掉其中对如下两个头文件的引用，因为移植过 程没有用到这两个头文件。

```
// #include  <cpu_def.h>
// #include  <cpu_cfg.h>                                    /* See Note 3.
```

• 在 ucosii_OpenMIPS 目录下新建 common 文件夹，其中用于存放测试程序。

最终得到的文件结构如下：

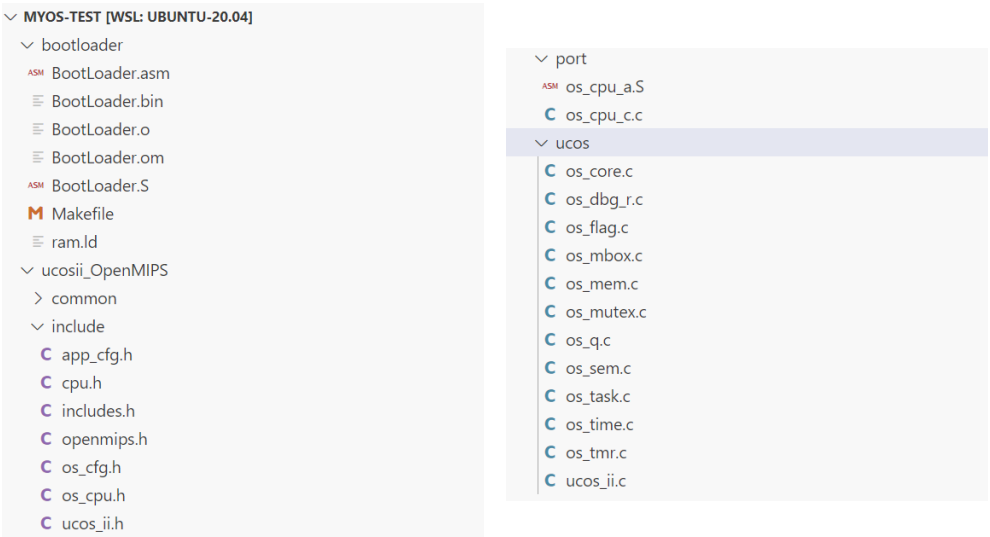


图 30：μC/OS-II 的文件结构

2. 修改 os_cpu_a.S 文件

该文件直接采用《自己动手写 CPU》附件，在此基础上修改。修改部分过多，在此不作展示，在本实验的代码中我写了详细的修改部分，原有代码只是注释掉，并未删除，便于比对。

3. 修改 os_cpu_c.c 文件

该文件采用《自己动手写 CPU》附件，在此基础上修改。将下方两段 VECTOR 的定义删除：

```
/*
*********************************************************************************
*                                          HARDWARE INTERRUPT VECTOR
*********************************************************************************
*/

extern char vec[], endvec[];  /* Create the hardware interrupt vector    */
asm (".set push\n"
     ".set nomicromips\n"
     ".align 2\n"
     "vec:\n"
     "\tla  $26,InterruptHandler\n"
     "\tjr  $26\n"
     "endvec:\n"
     ".set pop\n");

/*
*********************************************************************************
*                                              EXCEPTION VECTOR
*********************************************************************************
*/

extern char vec2[], endvec2[];  /* Create the exception vector        */
asm (".set push\n"
     ".set nomicromips\n"
     ".align 2\n"
     "vec2:\n"
     "\tla  $26,ExceptionHandler\n"
     "\tjr  $26\n"
     "endvec2:\n"
     ".set pop\n");
```

将所有 Hook 函数中的内容删除，只保留 `ptcb = ptcb` 的部分。

将 OSTaskStkInit 函数中的 `sr_val |= 0x0000C001;`改为 `sr_val |= 0x00000401。`

```
    /* Status 寄存器的值保存在变量 sr_val 中，设置其第 10 位为 1，设置其第 0 位也
```

```
        为 1，sr_val 将作为新任务的对应 Status 寄存器的值，此处的设置就是使得新任
        务在执行时允许时钟中断 */
    sr_val  |= 0x00000401; /* Initialize stack to allow for tick interrupt */
```

增加函数 BSP_Interrupt_Handler，进行具体的中断处理：

```
/*
*********************************************************************************
*                                                BSP_Interrupt_Handler
*
* Description: 中断发生时会调用本函数处理具体的中断事宜
*
* Arguments  : None
*
* Note(s)    : 1) Interrupts may or may not be ENABLED during this call.
*********************************************************************************
*/
void  BSP_Interrupt_Handler (void)
{
    INT32U   cause_val;
    INT32U   cause_reg;
    INT32U   cause_ip;

     /* 读取 Cause 寄存器，获得其中的 IP（Interrupt Pending）字段 */
    asm ("mfc0   %0,$13"   : "=r"(cause_val));
    cause_reg  = cause_val;                    /* 得到 Exc Code        */
    cause_ip = cause_reg & 0x0000FF00;

    if((cause_ip & 0x00000400) != 0 )
    {
        /* 如果 IP 字段表示是时钟中断，那么调用函数 TickISR，在该函数中将
          Compare 寄存器增加 0x50000，同时清除时钟中断声明   */
        TickISR(0x50000);
    }
}
```

增加 BSP_Exception_Handler 进行异常处理：

```
/*
*********************************************************************************
*                                                BSP_Exception_Handler
*
* Description: 调用 syscall 指令、Txx 指令、invalid instruciton 发生时会调用本
函数处理具体的异常事宜
*
* Arguments  : None
```

```
*
* Note(s)    : 1) Interrupts may or may not be ENABLED during this call.
*********************************************************************************
*/
void  BSP_Exception_Handler (void)
{
    INT32U   cause_val;
    INT32U   cause_exccode;
    INT32U   EPC;

    /* 读取 Cause 寄存器，获得其中的 ExcCode 字段，该字段存储的是异常原因 */
    asm volatile("mfc0   %0,$13"   : "=r"(cause_val));
    cause_exccode  = (cause_val & 0x0000007C);    /* 得到 Exc Code        */
    if(cause_exccode == 0x00000020 )    /* 判断是否是由于 syscall 指令引起 */
    {
        OSIntCtxSw();
    }
    else if(cause_exccode == 0x00000034)     /* 判断是否是由于 Txx 指令引起 */
    {
        OSIntCtxSw();
    }
    else if(cause_exccode == 0x00000030)          /* 判断是否是由于溢出引起 */
    {
        OSIntCtxSw();
    }
    else if(cause_exccode == 0x00000028)   /* 判断是否是由于 invalid
instruction 引起 */
    {
        OSIntCtxSw();
    }
}
```

4. 创建 openmips.h 文件

该文件采用《自己动手写 CPU》附件，在此基础上修改。修改系统时钟为 Nexys4 DDR 的系统时钟，频率为 100Hz。

```
/*****************************************************************
***********                  第二段：系统时钟              **********
*****************************************************************/
#define IN_CLK 100000000              /* 输入时钟是 100MHz */
```

5. 创建 openmips.c 文件

该文件采用《自己动手写 CPU》附件，未作修改，将该文件放入 common 文件夹中。

6. 交叉编译

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。

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

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：

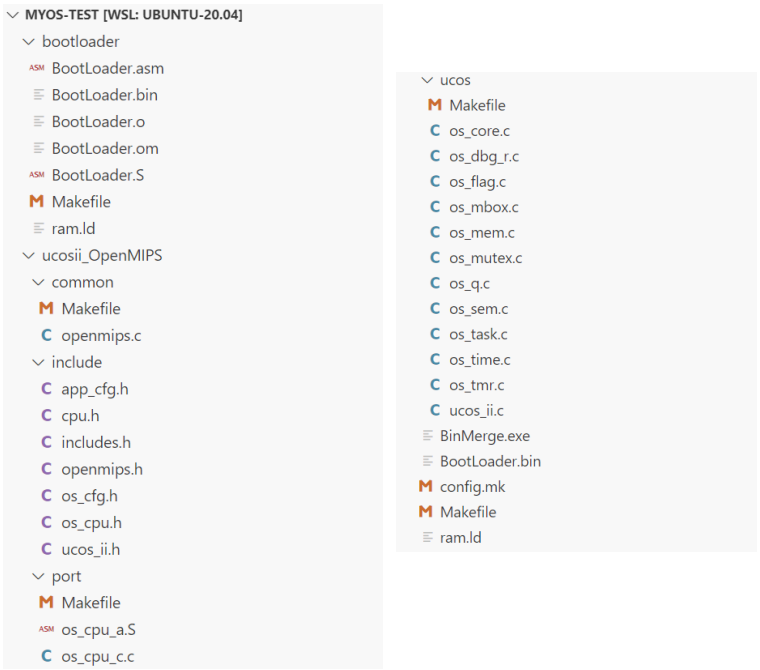


图 31：整个项目的文件架构

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 32：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) 生成 bit 流文件

与所有实验相似，首先进行 Synthesis、使用如下 xdc 文件进行管脚的配置，再进行 Implementation，最后生成 Bitstream 文件。

```
## Clock signal
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports clk_in]

## Switches
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports
flash_continue]
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]}]
```

```
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[16] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[17] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[18] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[19] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[20] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[21] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[22] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[23] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[24] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[25] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[26] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[27] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[28] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[29] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[30] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports
{ gpio_o[31] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

## SEG
set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[15]}]
set_property -dict {PACKAGE_PIN L18 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[14]}]
set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[13]}]
set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[12]}]
```

```
set_property -dict {PACKAGE_PIN K13 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[11]}]
set_property -dict {PACKAGE_PIN K16 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[10]}]
set_property -dict {PACKAGE_PIN R10 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[9]}]
set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[8]}]
set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[7]}]
set_property -dict {PACKAGE_PIN K2  IOSTANDARD LVCMOS33} [get_ports
{gpio_o[6]}]
set_property -dict {PACKAGE_PIN T14 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[5]}]
set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[4]}]
set_property -dict {PACKAGE_PIN J14 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[3]}]
set_property -dict {PACKAGE_PIN T9  IOSTANDARD LVCMOS33} [get_ports
{gpio_o[2]}]
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[1]}]
set_property -dict {PACKAGE_PIN J17 IOSTANDARD LVCMOS33} [get_ports
{gpio_o[0]}]

## Buttons
set_property -dict {PACKAGE_PIN C12 IOSTANDARD LVCMOS33} [get_ports rst_n]
set_property -dict {PACKAGE_PIN N17 IOSTANDARD LVCMOS33} [get_ports
{gpio_i[0]}]

## USB-RS232 Interface
set_property -dict {PACKAGE_PIN C4 IOSTANDARD LVCMOS33} [get_ports uart_in]
set_property -dict {PACKAGE_PIN D4 IOSTANDARD LVCMOS33} [get_ports uart_out]

## Quad SPI Flash
set_property -dict {PACKAGE_PIN K17 IOSTANDARD LVCMOS33} [get_ports sdo]
set_property -dict {PACKAGE_PIN K18 IOSTANDARD LVCMOS33} [get_ports sdi]
set_property -dict {PACKAGE_PIN L14 IOSTANDARD LVCMOS33} [get_ports wp_n]
set_property -dict {PACKAGE_PIN M14 IOSTANDARD LVCMOS33} [get_ports hld_n]
set_property -dict {PACKAGE_PIN L13 IOSTANDARD LVCMOS33} [get_ports cs_n]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets rst_n_IBUF]
```

装 订 线

(2) 串口通讯

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。

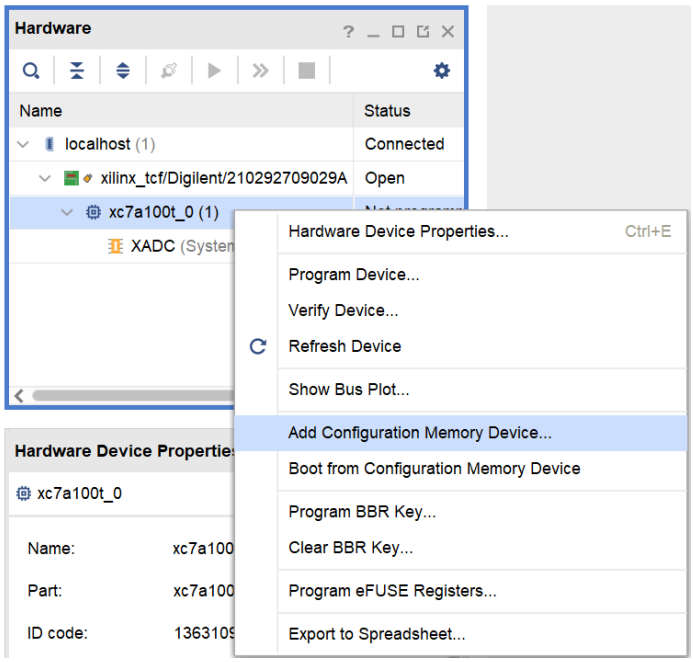


图 33：配置 SPI-Flash

选中 s25fl128sxxxxxx0-spi-x1_x2_x4。

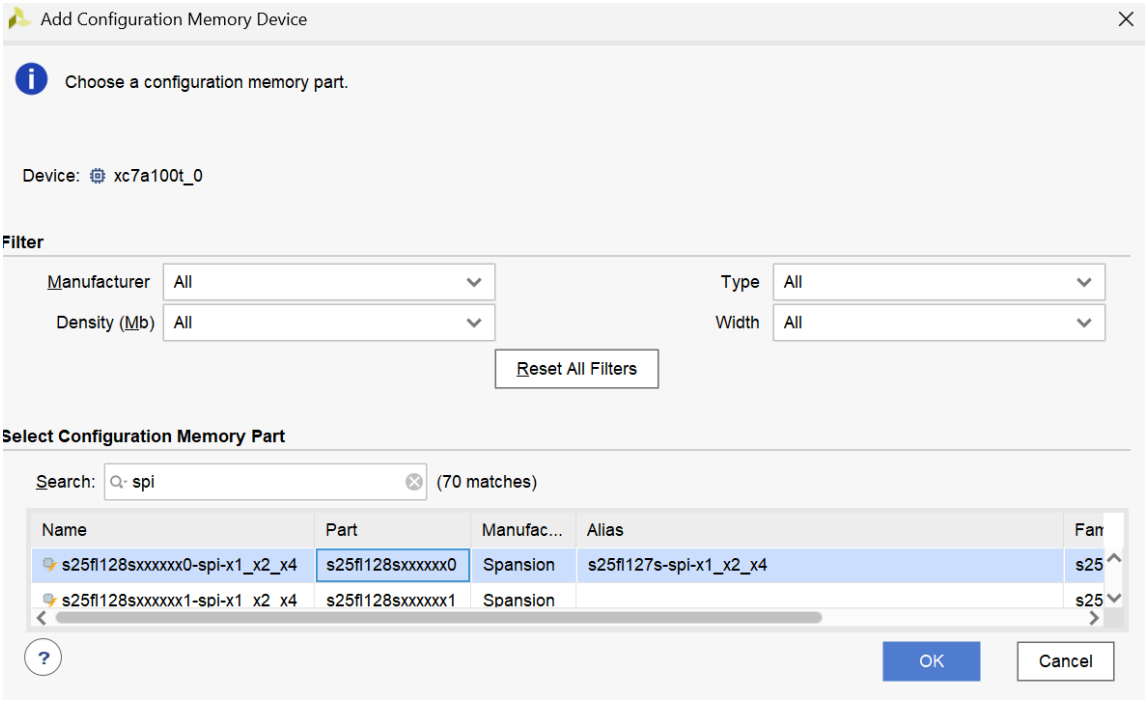


图 34：配置 SPI-Flash

装 订 线

选择编译得到的 bin 二进制文件下板。

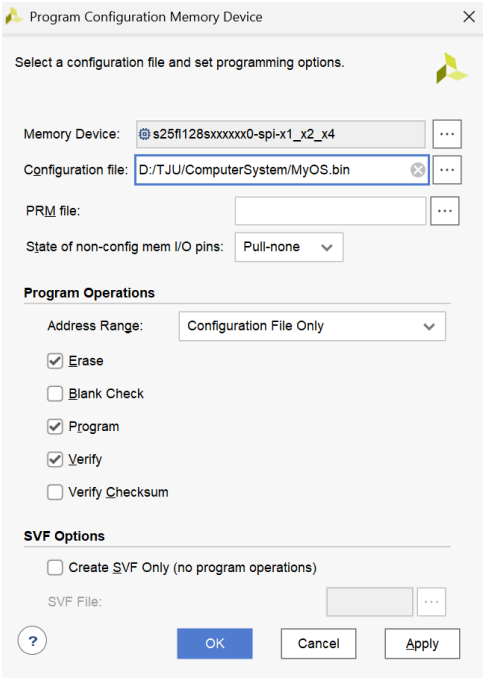


图 35：配置 SPI-Flash

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，并按下 C12 对应的 CPU RESET 按钮，即可看到实验结果。

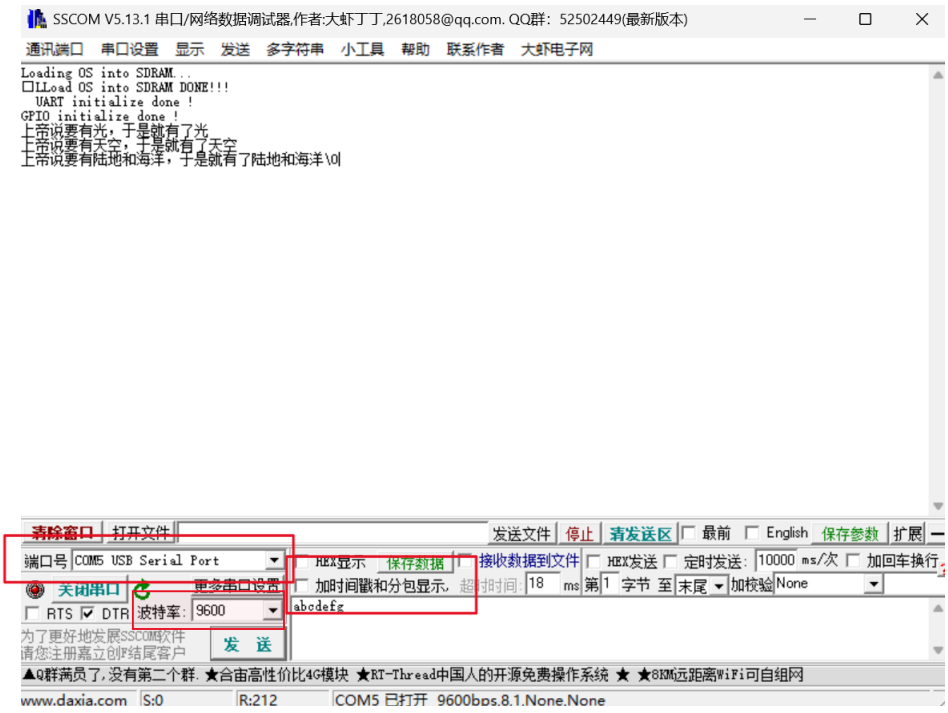


图 36：串口通讯结果

(3) 下板结果

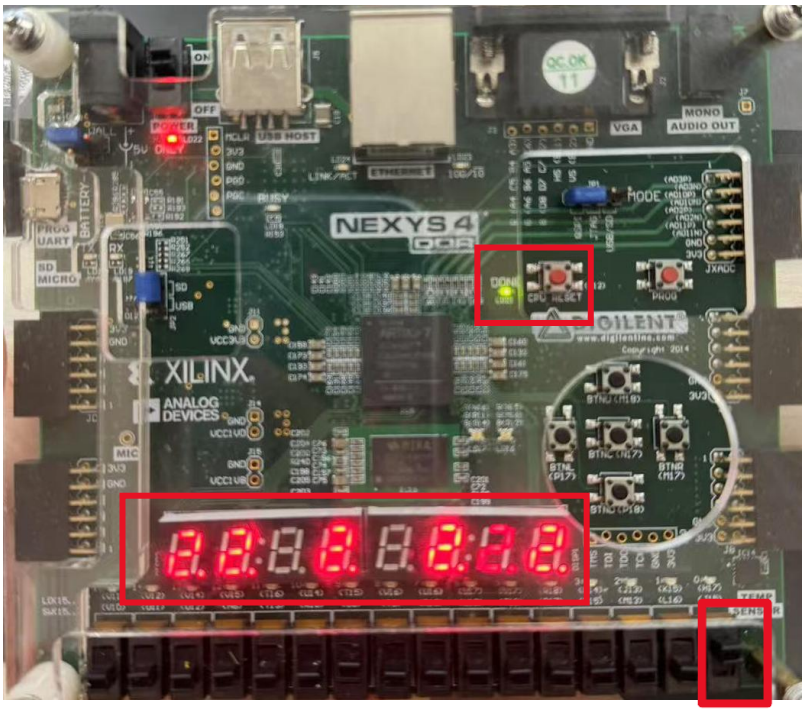


图 37：下板结果演示

可以看到数码管是由 GPIO 控制，不断闪烁，右侧开关上拨表示使能，按下上方红色按钮可以复位，验证结果正确，这证明了本次试验移植的操作系统可以在 100Mhz 下平稳运行，为之后的应用程序开发打下坚实的基础。

5、实验总结

(1) 遇到的问题

1. Implementation 阶段报错 [Place 30-574]

具体信息如下：

[Place 30-574] Poor placement for routing between an IO pin and BUFG. If this sub optimal condition is acceptable for this design, you may use the CLOCK_DEDICATED_ROUTE constraint in the .xdc file to demote this message to a WARNING. However, the use of this override is highly discouraged. These examples can be used directly in the .xdc file to override this clock rule.

< set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets rst_n_IBUF] >

报错原因为，编译器在综合时会自动的为工程中的时钟信号生成一个全局时钟 BUF，然后如果管脚分配将这个 BUFG 连接到普通管脚上，就会报以上错误。即使坚持使用 IO 管脚做为全局

时钟管脚，这个错误也并不是不能消除的，可以使用 CLOCK_DEDICATED_ROUTE 约束来将这种错误降级为 WARNING。

经查阅资料，可以根据给出的一条语句 set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets rst_n_IBUF]，在 xdc 文件中插入，问题得以解决。注意，左右两侧的尖括号是不要添加的！

2. 串口通讯信息乱码，不能正常显示

编译得到二进制文件后下板打开串口后显示乱码。

图 38：乱码问题复现

经过与《自己动手写 CPU》代码的仔细比对，发现两个三个问题：

• 由于想要添加注释，我并没有在书给出的附件上直接修改，而是在电子书上直接复制，导致换行符后仍有无关内容，导致不可预见的错误。

```
/* 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)
```

• 系统时钟忘记设置，导致时序错误，应改为 100MHz。

• Status 寄存器写入错误，为粗心比对不仔细导致，忘记修改文件。

3. 串口通讯时，操作系统可以正常启动，但只能收到“上”一个字

就此问题，我与其他同学进行了充分讨论！原因有些复杂。

有同学同样出现此问题，Vivado 版本为 2020.2，后来更换为 2019.1，问题未得到解决，最终更改至 2019.2 版本，问题得以解决。

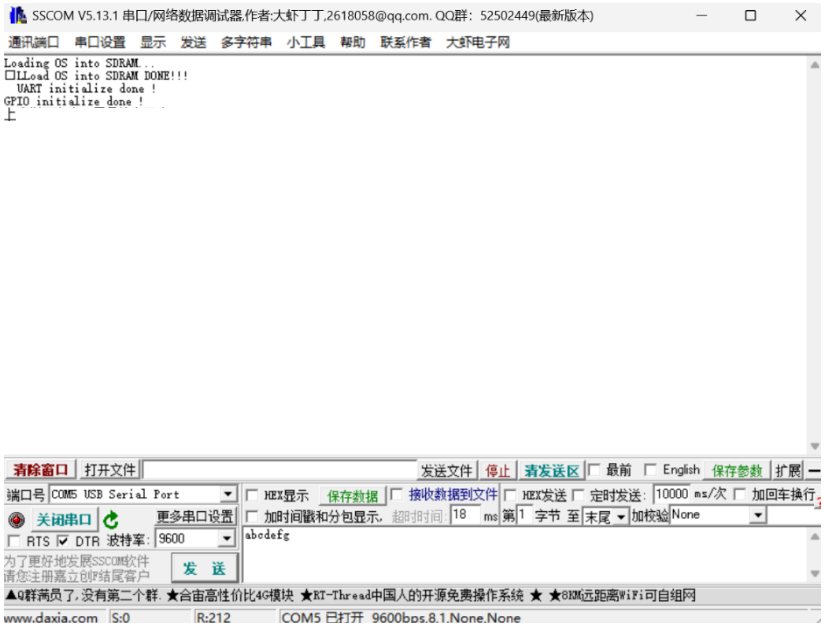


图 39：缺少字符问题复现

我在反复调试后，发现我的问题在于 OpenMIPS CPU。若我更换为雷思磊老师给我们的附件代码中的 CPU，则可以显示完整信息，但是换成自己上个实验修改过的 MIPS89 就是不行，但上次实验明明已经通过。再次整理信息发现，可能是新增的 break 指令存在问题，通过单个文件替换的排除法，得到了问题的根源所在：

• ex_mem.v 模块，由于实验 1 测试用例较少，可能并未测出真正的 CPU 漏洞。这个文件我后来排查时发现是某个变量未赋值。但修改过后仍然出现相同的卡死错误。

• ctrl.v 模块，由于实验 1 是针对 MARS 的，故代码段的起始地址是 0x00400000，已经与实验 2 截然不同。且实验 2 开辟了相应放置异常处理程序的入口，故应修改异常入口的地址，第高三位的 4 应该为 0。修改后仍然出现相同的卡死错误。

• id.v、cp0_reg.v、mem.v 模块，实验 1 要求的 89 条指令中包括一条 break 指令，但书中并没有实现，当时是我自行实现的，功能正常。故我猜测是此处新增的 break 断点指令，导致系统某些位置发生冲突。故我将上述模块中包含 break 指令的取指、译码等部分注释掉，即去掉 break 指令，最终系统可以实现正常的串口通信。

(2) 心得体会

操作系统我们天天都在使用，早已习以为常！看似较为简单的操作系统移植，到头来还是花了不少功夫。这次实验已经不仅仅是计算机系统结构本身的知识，更加横跨了编译原理、操作系统等多个学科，也是本次实验，让我真正感受到本科这么多年以来学习到的知识串起来的感觉，所有在课堂上枯燥乏味的理论，都藏在这一行行简明而又有意思的代码中了。

总感觉寥寥几个模块搭起来的 CPU 有些摇摇欲坠，但真正当它能够撑起一个操作系统的时候，我在屏幕前看呆了，被这些精密的设计所折服。

无论是几个控制器和总线的添加，还是 Ubuntu 下 OS.bin 的生成，对我来说都是陌生且有难度的。尤其是 Flash 控制器和 SDRAM 控制器，这两个芯片因开发板而异，这将是对我们资料检索能力、阅读能力和学习能力的考验。对此，我付出了大量时间在寻找官方手册、查阅芯片时序等工作上。同时，这些外设的使用，也和我们平时在计组或者数字逻辑中的小实验不同，需要配置额外的 IP 核，大量参数的配置更是需要我们谨小慎微的细致。

尽管雷思磊老师已经给出了完整的修改过的代码，但是，我认为仅仅复制粘贴这是远远不够的。秉持着严谨的态度，我还是跟着书上的步骤走了一遍，添加详细的注释，尽量让自己能够理解每一步那么做的原因吧。这也导致了后续由于不够细致，操作系统串口调试的结果一直都是乱码，增加了测试查错的时间。

我发现这次实验和其他专业学习的嵌入式课程有异曲同工之妙（就是），也算是能够体会到他们做作业调试时的感受了！硬件的错误和软件的错误确实大相径庭，哈哈，但调试起来都是一样的折磨体验！希望下次实验的小程序开发能够一切顺利~

装 订 线

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.

[4] IP 核调用教程：FPGA 基础入门【9】开发板外部存储器 DDR2 访问_fpga 的 pin 口 nc 后缀什么意思-CSDN 博客

[5] Flash 整体设计参考：SPI FLASH 读取模块[verilog]

[6] Ram2Ddr.vhd 库文件说明：SRAM to DDR Component - Digilent Reference

[7] SPI-Flash 教程：FPGA 基础入门【8】开发板外部存储器 SPI flash 访问_chipscope capture mode-CSDN 博客

[8] S25FL128S 芯片说明文档：S25FL128S, S25FL256S, 128 Mb (16 MB)/256 Mb (32 MB) FL-S Flash SPI Multi-I/O, 3.0V

[9] DDR 接口使用手册：7 Series FPGAs Memory Interface Solutions v1.9 and v1.9a, User Guide (AXI) • 查看器 • AMD 技术信息门户网站