

同濟大學

同济大学计算机科学与技术系

计算机系统结构课程设计报告



项目名称 μC/OS-II 操作系统移植

专 业 计算机科学与技术

授课老师 郭玉臣

姓 名

学 号

日 期 2024 年 5 月 20 日

1 实验目的

本次实验主要需要为 NEXYS DDR4 开发板移植一个操作系统，方便之后进行程序开发。移植的操作系统为 $\mu\text{C}/\text{OS-II}$ 操作系统，具体的移植过程参照雷思磊的《自己动手写 CPU》一书完成。

首先，移植的操作系统必须为嵌入式实时操作系统，这类操作系统往往执行带有特定要求、预先定义的任务，所以可以减小尺寸、降低成本；同时，当外界事件或数据产生时，能够接受并以足够快的速度予以处理，处理的结果能在规定时间内来控制生产过程，做出快速响应，并控制所有实时任务协调一致，因此及时响应和高可靠性是这类操作系统的主要特点。

$\mu\text{C}/\text{OS-II}$ 在诸多领域得到应用，包括手机、路由器、集线器、不间断电源、飞行器等，并且得到了美国航空管理局的认证，这充分证明了 $\mu\text{C}/\text{OS-II}$ 的高度可靠。它采用 ANSI 的 C 语言编写，包含一小部分汇编代码，使之可以在不同架构的微处理器上使用。

本次实验将在上一次完成的 89 条 CPU 基础之上继续添加相关的功能部件，增加 Wishbone 总线、GPIO、UART 控制器、Flash 控制器、SDRAM 控制器。并在 Ubuntu 上建立交叉编译环境对 $\mu\text{C}/\text{OS-II}$ 系统进行改写、编译，最终成功将 $\mu\text{C}/\text{OS-II}$ 系统移植到 NEXYS 4 DDR 开发板上并验证。

2 实验内容

按照《自己动手写 CPU》思路，增加 Wishbone 总线，GPIO、UART、Flash 控制器、SDRAM

控制器，同时在 Ubuntu 下交叉编译生成移植系统的二进制文件，并进行波特率等的调整，最终将 $\mu\text{C}/\text{OS-II}$ 系统一直移植到开发板上。大致流程如下：

1. 实现总线

- 增加 Wishbone 总线、GPIO、UART 控制器、Flash 控制器、实现一个小的 SOPC
- 参考《自己动手写 CPU》的基础篇

2. 系统移植 $\mu\text{C}/\text{OS-II}$

- 利用 Ubuntu 建立交叉编译环境
- 对 $\mu\text{C}/\text{OS-II}$ 系统进行改写、编译
- 参考《动手写 CPU》第十五章

3. 下板验证检查

使用串口助手观察加载结果进行检查

3 实验过程

3.1 实验环境

3.1.1 软件环境

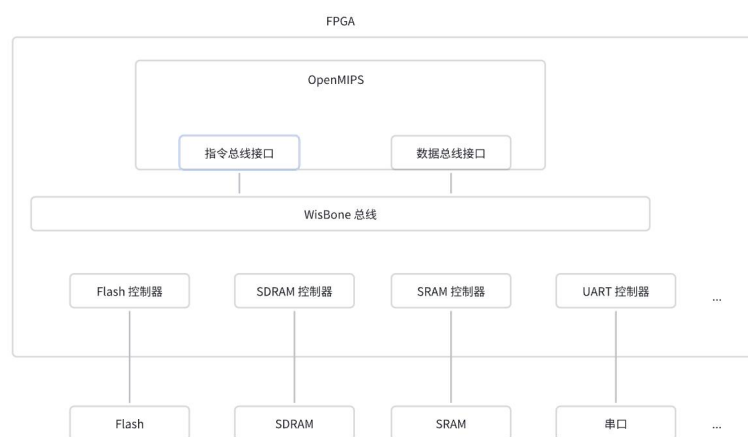
1. 开发环境：Vivado v2019.1(64bit)
2. 仿真环境：Vivado v2019.1(64bit)
3. 测试环境：MARS4.5
4. 文档管理：office

3.1.2 硬件环境

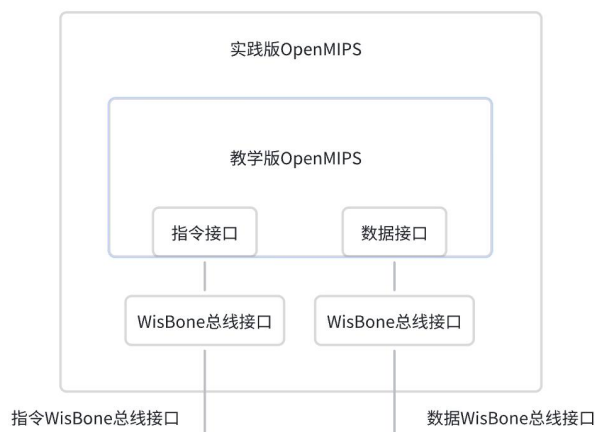
1. 计算机：XiaoXin Pro 16IHU 2021
2. 开发板：NEXYS 4DDR Atrix-7

3.2 实验原理

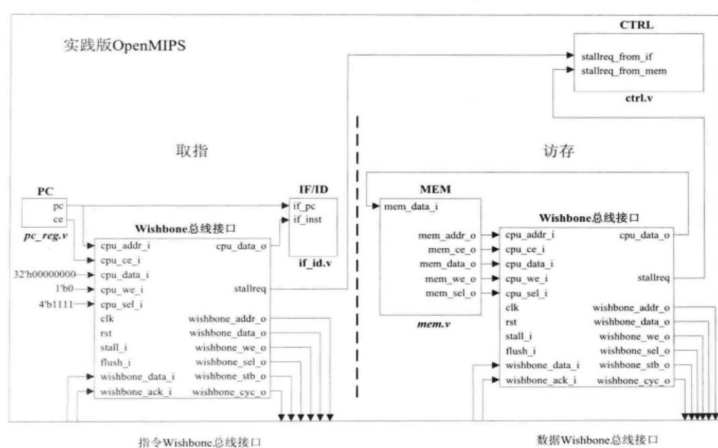
实践版 OpenMIPS 系统设计总框图如下：



本次实验将在之前 9 条实验版 OpenMIPS 基础上改进，用于系统移植，改进下图所示。



实践版 OpenMIPS 设计思路



实践版 OpenMIPS 架构

3.3 实验步骤

本次实验步骤在第一次实验:实现 89 条 CPU 的基础上进行模块添加。添加 Wishbone 总线、GPIO、UART 等模块后改进教学版 OpenMIPS, 构建实践版 OpenMIPS, 最后根据《自己动手写 CPU》进行操作系统的移植, 具体实验过程如下。

3.3.1 实现 89 条教学版 OpenMIPS

该过程已经由第一次实验完成, 在此不作详细描述。改造 CPU, 使得 CPU 能支持 89 条指令。经过仿真、综合、生成 bit 文件四个步骤, 下板验证并得到结果。需要新增的 35 条指令如下:

- 移动操作指令
movn、movz
- 算术操作指令

clo、madd、maddu、msub、msubu

- 转移指令

b、bal、bgezal、bgtz、blez、bltz、bltzal

- 加载存储指令

ll、lwl、lwr、sc、swl、swr

- 异常相关指令

tge、tgeu、tlr、tlru、tne、teqi、tgei、tgeiu、tlr、tlru、tnei

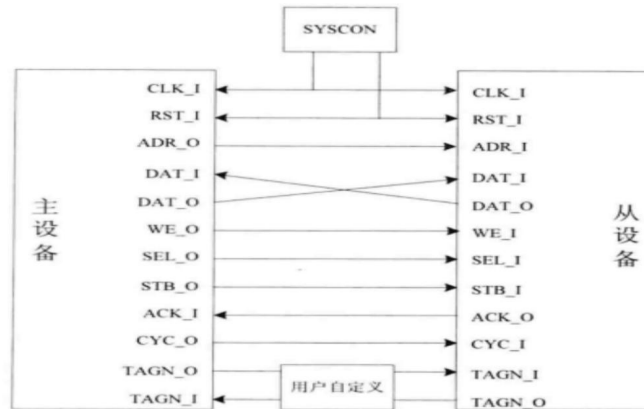
- 其他指令

nop、ssnop、sync、pref

89 条 CPU 实现实验中实现的 MIPS CPU 为五级流水线 CPU，分别有取指、译码、执行、访存、回写五个阶段。采用大端模式。采用哈佛结构，有分开的指令存储器和数据存储器。

3.3.2 添加 Wishbone 总线

PC 机一般都提供 PCI 插槽，各种板卡（包括显卡、语音卡、网卡甚至是用户自制的板卡）。只要们组 PCI 接口标准，就可以直接插在 PC 机的 PCI 插槽使用，十分方便。同样的道理，目前有很多 IP 核的研发者或公司，为了方便不同研发者或公司的 IP 核能够直接连接，就要求这些 IP 核遵守共同的接口标准。在片上系统（SoC）中，处理器核与其他 IP 核通过共享总线互通互联，这些 IP 核必须遵守相同的总线规范。总线规范定义了 IP 核之间的通用接口。目前常见的片上总线规范有 ARM 公司的 AMBA、IBM 公司的 CoreConnect、Altera 公司的 Avalon，以及这里所说的 Wishbone。Wishbone 总线规范是 Silicore 公司最先提出的，由于其开放性，现在已有不少用户群，特别是一些免费的 IP 核，大多数都采用 Wishbone 规范。Wishbone 除了开放、免费，还有简单、灵活、轻量、支持用户自定义标签的特点。目前已有 B4 版本的规范，OR1200 中遵循的是 Wishbone B2 与 B3 版本的规范，用户可以自行配置是采用 B2 还是 B3 规范。Wishbone 有多种互联方式：点对点、数据流、共享总线、交叉互联等。OR1200 内部使用的都是点对点连接方式。在点对点连接方式中，有一个主设备，一个从设备，连接关系如图所示，图中输出信号使用“_O”结束，输入信号使用“_I”结束，同时所有的信号都是高电平有效。



图中主从设备接口的含义如下

1. CLK_I/RST_I: 分别是时钟信号、复位信号，由外部输入。

2. DAT_O/DAT_I: 主设备和从设备的之间的数据信号，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。一对主设备和从设备之间最多存在两条数据总线，一条用于主设备向从设备传输数据，另一条用于从设备向主设备传输数据。

3. ADR_O/ADR_I: 地址信号，主设备输出地址到从设备。

4. WE_O/WE_I: 写使能信号，主设备输出到从设备，代表当前周期中进行的操作是写操作还是读操作，1 代表写，0 代表读。

5. SEL_O/SEL_I: 数据总线选择信号，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。比如一个具有 32 位宽、粒度为 1 个字节的数据总线的选择信号应定义为 SEL_O(3:0)/SEL_I(3:0)，SEL(4'b1001)代表当前操作中数据总线的最高和最低字节有效。

6. 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（是否正在进行总线操作取决于选通信号 STB_O/STB_I 是否有效）。只有该信号有效，Wishbone 主设备和从设备之间的其它信号才有意义。CYC_O/CYC_I 信号在一次总线操作过程中必须持续有效，比如一次块读操作可能需要多个时钟周期，那么 CYC_O/CYC_I 信号必须在多个时钟周期中持续有效。

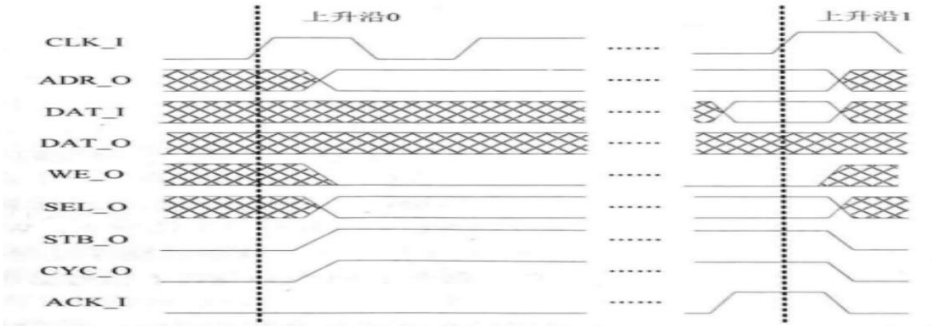
7. STB_O/STB_I: 选通信号。选通有效代表主设备发起一次总线操作。只有选通信号有效时（此时 CYC_O/CYC_I 也必须为高），ADR_O/ADR_I、DAT_O/DAT_I、SEL_O/SEL_I 才有意义。

8. ACK_O/ACK_I: 实际还可以有 ERR_O/ERR_I、RTY_O/RTY_I，都表示主从设备间的操作结束方式信号。ACK 表示成功，ERR 表示错误，RTY 表示重试。操作总是在某一总线周期内完成的，因此操作结束方式也称为总线周期结束方式。成功是操作的正常结束方式，错误表示操作失败，造成失败的原因可能是地址或者数据校验错误，写操作或者读操作不支持等。重试表示从设备当前忙，不能及时处理该操作，可以稍后重新发起。接收到操作失败或者重试后，主设备如何响应取决于主设备的设计者。

9. TAGN_O/TAGN_I: 标签信号，用户可以利用标签信号传递自定义的信息。一个总线周期由多个不同的时钟周期构成，完成单次读/写操作、块读/写操作、读改写操作，总线周

期也相应分为单次读/写周期、块读/写周期、读改写周期。

一般情况下，一次操作由主设备和从设备控制信号的一次握手，以及同时进行的地址和数据总线的一次传输构成。当主设备将 `CYC_O` 置高，一个总线周期开始，此后当 `STB_O` 为高时，一次总线操作开始。`CYC_O` 和 `STB_O` 可以同时从低电平变为高电平，表示发起总线周期的同时开始一次总线操作，因此在只有一个主设备时可以将两者合并为一个信号。在 OR1200 中就直接将 `CYC_O`、`STB_O` 合并成 `CYCSTB_O`。主从设备之间信号虽然很多，但单次读写操作实际上十分简单。单次读操作的 Wishbone 总线信号如下图所示，此处是从主设备的角度观察信号变化。

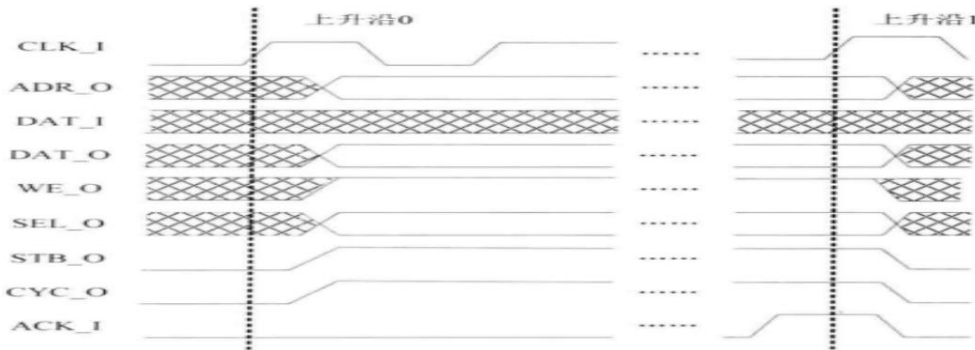


在时钟上升沿 0，主设备将地址信号 `ADR_O`、适当的 `SEL_O` 放到总线上，将 `WE_O` 置低表示读操作，将 `CYC_O`、`STB_O` 置高表示一次总线操作开始。

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信号 `DAT_I`，同时将主设备的 `ACK_I` 置高作为对主设备 `STB_O` 的响应。从设备可以在 `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，将 `DAT_I` 采样，并将 `STB_O` 和 `CYC_O` 置低表示操作完成。从设备发现 `STB_O` 置低后，将主设备的输入信号 `ACK_I` 也置低。单次读操作就完成了。

单次写操作的 Wishbone 总线信号如下图所示，此处还是从主设备的角度观察信号变化。



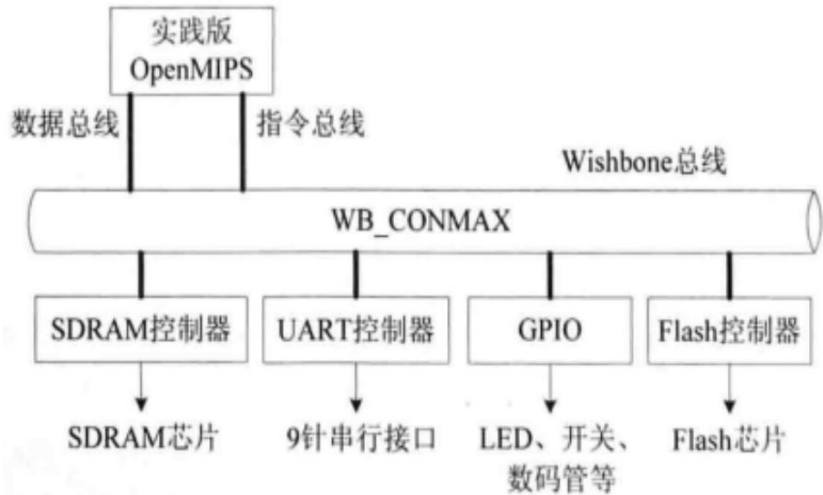
在时钟上升沿 0，主设备将地址信号 `ADR_O`、数据信号 `DAT_O` 放到总线上，将 `WE_O` 置高表示写操作，将适当的 `SEL_O` 放到总线上指示从设备 `DAT_O` 中哪些字节是有效的，将 `CYC_O`、`STB_O` 置高表示一次总线操作开始。

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，将锁存 `DAT_O` 的数据，同时将主设备的 `ACK_I` 置高作为对主设备 `STB_O` 的响应。从设备可以在 `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，将 STB_O 和 CYC_O 置低表示操作完成。从设备发现 STB_O 置低后，将主设备的输入信号 ACK_I 也置低。单次写操作就完成了。

在此次实验中，采用交叉互联方式建立 Wishbone 总线，其结构如下图所示。在 Wishbone 总线上挂接了五个模块：OpenMIPS 处理器、GPIO、UART 控制器、Flash 控制器、SDRAM 控制器。其中 Wishbone 总线使用的是 OpenCores 站点提供的开源项目 WB_CONMAX，这是一个 Wishbone 总线互联矩阵，采用的是交叉互联方式，允许多对主从设备同时进行通信。

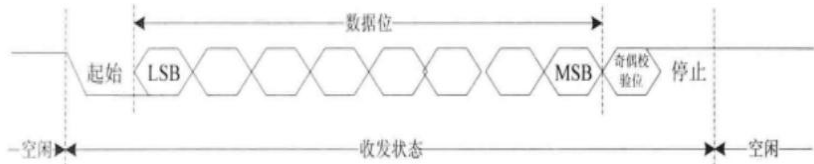


3.3.3 添加 GPIO 模块

GPIO 是以位为单位进行数字输入输出的 IO 接口，作为单纯的通用输入/输出 IO，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，例如 LED、开关、七段数码管等。在实践版 OpenMIPS 设计过程中，采用 OpenCores 站点提供的开源项目 GPIO IP Core，添加整个系统中。

3.3.4 添加 UART 控制器

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。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。传输格式如下图所示。

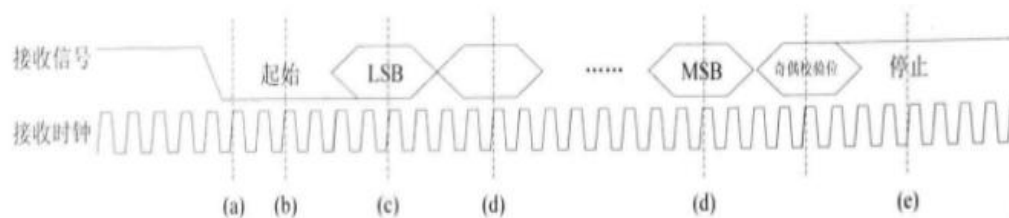


依次传输起始位、数据位、奇偶校验位、停止位，分别说明如下：

- 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。

UART 的通信速率用波特率（**baud rate**）来表示。波特率指的是信号被调制以后的变化率，即单位时间内载波变化的次数。用于波特率计算的信号除了数据位，还包括起始位、奇偶校验位、停止位，因此，波特率与单纯的数据传输速率是不同的。UART 常用的波特率有 9600 baud、19200 baud、38400 baud 等。

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。为了便于说明，下图以接收时钟是波特率的 4 倍为例，给出了数据接收过程。



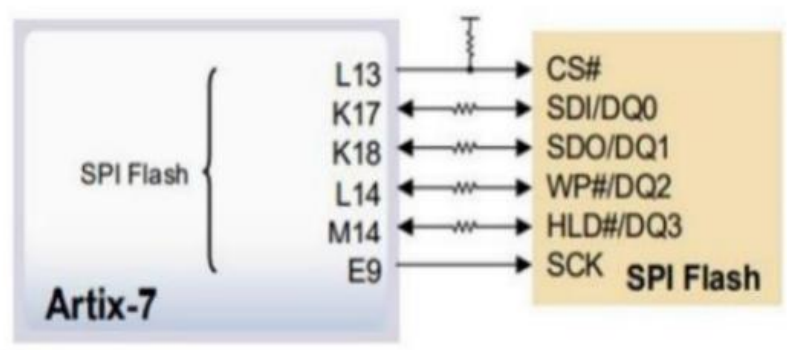
1. 当接收信号由高电平变为低电平时，表示检测到起始位。
2. 检测到起始位后，在接下来的第 2 个时钟周期检查接收信号，如果保持为低电平，说明确实是起始位，开始接收数据。否则认为起始位检测错误，将其忽略。
3. 确定是起始位后，等待 4 个时钟周期检查接收信号，得到的值就是接收到的第一个 bit，也就是 LSB。
4. 之后每隔 4 个时钟周期检查接收信号，依次得到传送过来的数据位、奇偶校验位。从图中可以发现，每次采样都是在接收数据的中部，这样采样得到的数据更加准确。
5. 数据接收完成后，接收停止位。在实践版 OpenMIPS 设计过程中，采用的 UART 控制器是 OpenCores 站点提供的开源项目 UART16550 IP Core，可以在 OpenCores 站点下载源代码。添加了 UART 控制器之后，就可以通过串口与计算机进行通信了。

3.3.5 添加 Flash 控制器

在操作系统移植中，我们需要一个非易失性存储来存放 BootLoader 程序和操作系统的二进制文件，Flash 就充当这样的角色。与《自己动手写 CPU》书上实现的 Flash 不同，此次实验中使用的是 Nexys4 DDR 板载的 SPI FLASH 模块。

在实践版 OpenMIPS 的设计过程中，实现了一个一次读取 4Bytes 的 flash 控制器，用来模拟“硬盘”，用于读取只读数据，包括 BootLoader 程序和操作系统的二进制文件。

Nexys4 DDR 使用的 SPI Flash 芯片的具体型号是 S25FL128S 芯片。其线路图如下图所示。



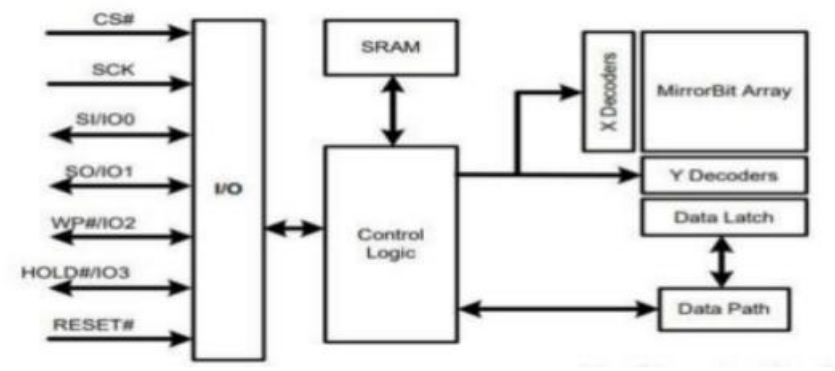
各接口的具体功能和说明如下图所示。

PAD NO.	PAD NAME	I/O	FUNCTION
1	/CS	I	Chip Select Input
2	DO (IO1)	I/O	Data Output (Data Input Output 1)* ¹
3	/WP (IO2)	I/O	Write Protect Input (Data Input Output 2)* ²
4	GND		Ground
5	DI (IO0)	I/O	Data Input (Data Input Output 0)* ¹
6	CLK	I	Serial Clock Input
7	/HOLD (IO3)	I/O	Hold Input (Data Input Output 3)* ²
8	VCC		Power Supply

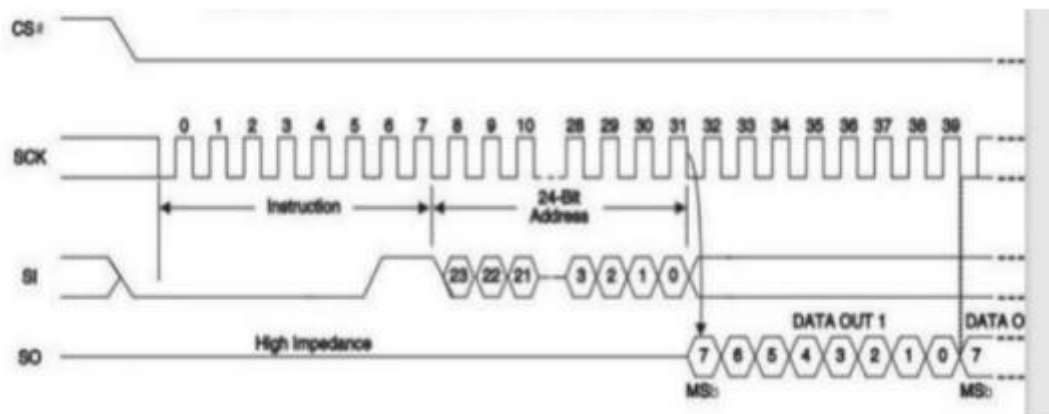
*1: IO0 and IO1 are used for Standard and Dual SPI instructions
*2: IO0 – IO3 are used for Quad SPI instructions

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 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，"_n" 的意思是低电平有效，故当 cs_n 信号被拉低的时候，这个芯片才开始工作。si 是输入到主设备的信号线，flash 芯片读取数据的时候需要靠它。so 是输入到 flash 的信号线，flash 芯片写入数据的时候需要靠它。flash 芯片是一种有一堆可读可写的颗粒（EEPROM）和一个控制器组成的，这个控制器封装了许多指令来操作那些颗粒。于是要想使用 flash 芯片，最主要的是对 flash 控制器发出正确的指令。

Flash 芯片的结构如图所示。



由于仅仅需要读取数据，此次实验中主要用到了一条指令：READ。根据 S25FL128S 的文档，可以看到读取指令的时序如下图所示。



从图上看，要想读取数据，需要做的是：

1. 先将 `cs_n` 拉低。
2. 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，于是此次实验中用 50MHz 会比较好分频。
3. 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）。
4. 最后 flash 芯片就会通过 `si` 信号线一位一位的输出数据。

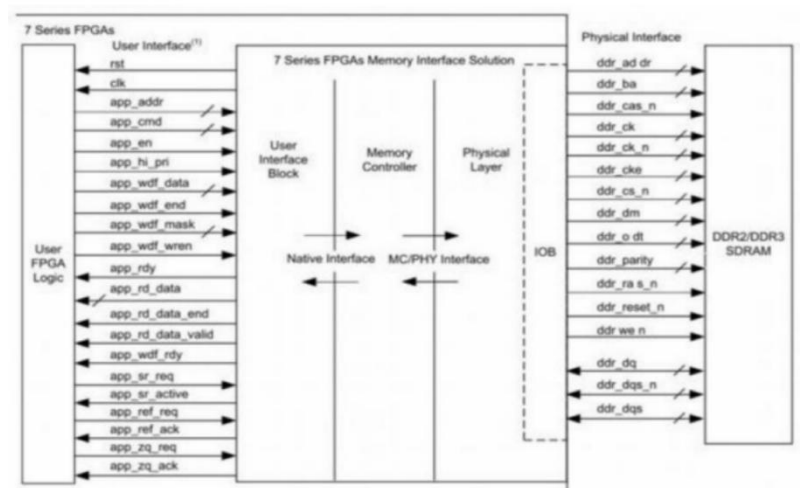
3.3.6 添加 SDRAM 控制器

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。与《自己动手写 CPU》书上实现的 SDRAM 不同，此次实验中使用的是 Nexys4 DDR 板载的 DDR 2 模块。此次实验中，使用 DDR 2 模拟一块一次最小读写宽度 8bits，最大读写宽度 32bits 的内存。

NEYXS 4 DDR 里的 DDR2 SDRAM 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。DDR RAM 时序一共有以下几个信号：

- `app_addr [26:0] in` 数据地址
- `app_cmd [2:0] in` 操作指令，3'b001 为读，3'b000 为写
- `app_en in` 指令使能
- `app_rdy out` 指令接受，SDRAM 接受到信号之后就会将其拉高

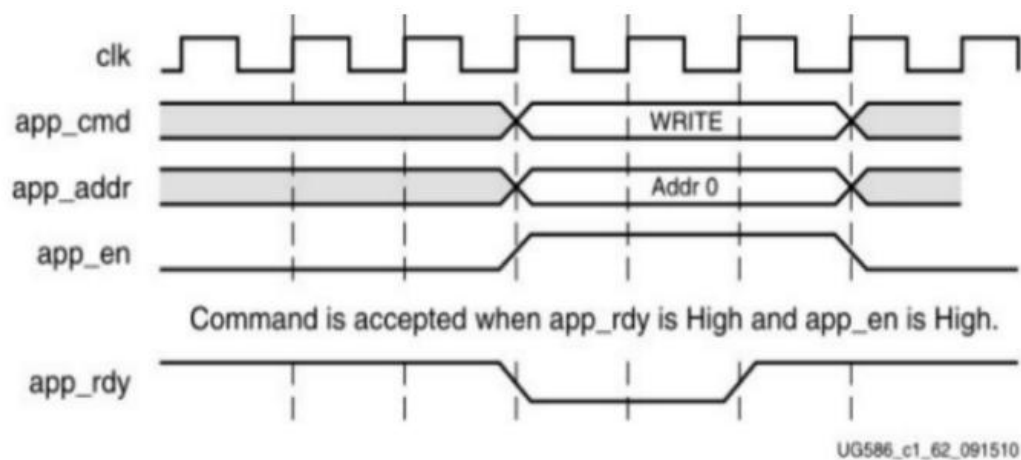
- `app_wdf_data [127:0]` in 写数据，通过 `app_wdf_mask` 处理后写入 SDRAM
- `app_wdf_mask [15:0]` in 写数据中一个 byte 对应这里一个 bit，将某一位设高就意味着忽略该位对应的数据 byte 后写入 SDRAM
- `app_wdf_end` in 在发送最后一个数据的时候将其拉高，以通知 SDRAM 结束写入操作
- `app_wdf_wren` in 写使能，在把前面的 `app_wdf_data` 和 `app_wdf_mask` 都写好了之后，把这个信号拉高，来通知 SDRAM 可以开始写入
- `app_wdf_rdy` out 写数据指令接收，SDRAM 接收到 `app_wdf_data` 和 `app_wdf_mask` 之后就会将其拉高
- `app_rd_data [127:0]` out 读取的数据
- `app_rd_data_valid` out 读取数据有效，当 SDRAM 将数据读取到 `app_rd_data` 时会将这个信号拉高
- `app_sr_req` in 保留引脚，应该拉低
- `app_ref_req` in 拉高这个信号发送一个 refresh 指令给 DRAM，一次只能拉高一个时钟周期
- `app_zq_req` in 拉高这个信号发送一个 ZQ 校准指令，校准 ODT 电阻值，也是一次只能拉高一个时钟周期
- `app_sr_active` out 保留引脚
- `app_ref_ack` out SDRAM 响应 refresh 信号时拉高
- `app_zq_ack` out SDRAM 响应 ZQ 校准信号时拉高
- `ui_clk` out IP 核给用户用的时钟，对于这个 IP 核操作的时序电路必须用这个时钟信号
- `ui_clk_sync_rst` out 给用户用的复位信号，同上，对于这个 IP 核操作的时序电路必须用这个时钟信号



DDR 是 Nexys4 板上的 Nexys4 板上的一个资源，容量为 128M。DDR 17 的操作非常复杂，需要借助一个 IP 核 Memory Interface Generator，简称 MIG（当然用了 MIG 还是非常复杂，需要多级封装）。MIG 能够封装 DDR 的物理层信号，用户不需要对物理信号有所了解，只需要关注应用信号。MIG 需要按照资料中指导的操作生成。但是由于 DDR 相较于 cpu 属于慢速的设备，所以要协调工作的话还要靠下面的 MIG 控制器。IP 核生成后，需要将 DDR 的物理接口添加到顶层模块，但无需再在 XDC 文件中配置端口，因为生成过程中已经配置好了。

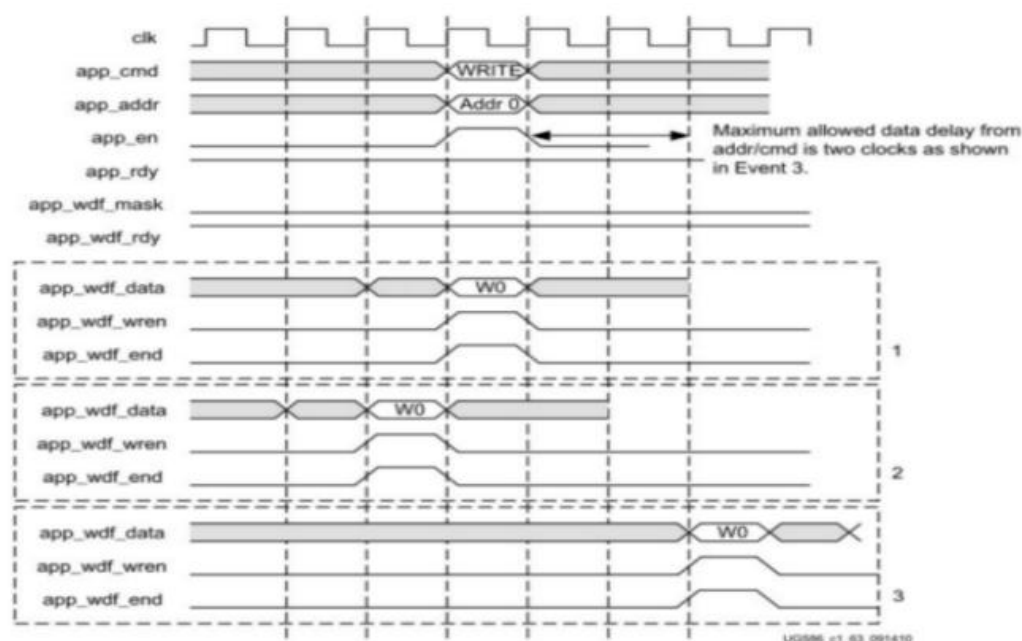
FPGA 如果需要对 DDR 进行读写，则需要一个 DDR 的控制器。DDR 控制器的时序主要有三个。

1. 控制信号



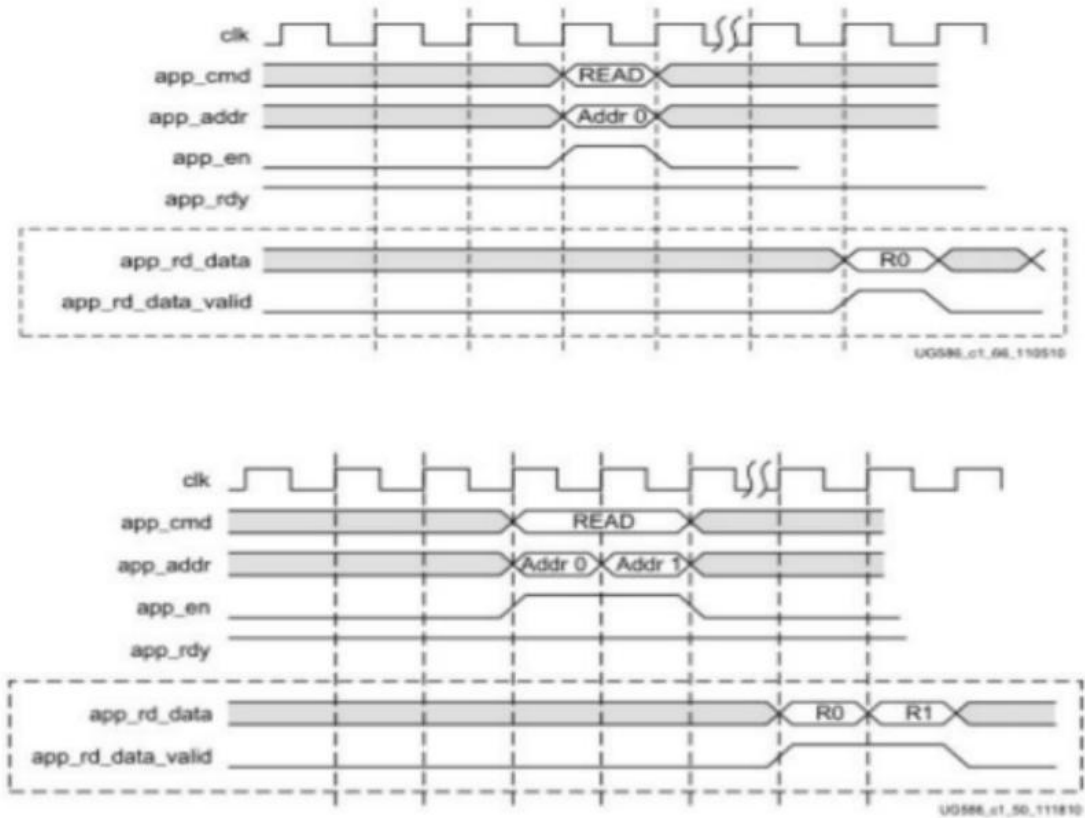
由上图可以看出，只有当 app_rdy 信号有效时，程序所发出的读写命令才会被控制器接收。这点必须注意。

2. 写操作的时序



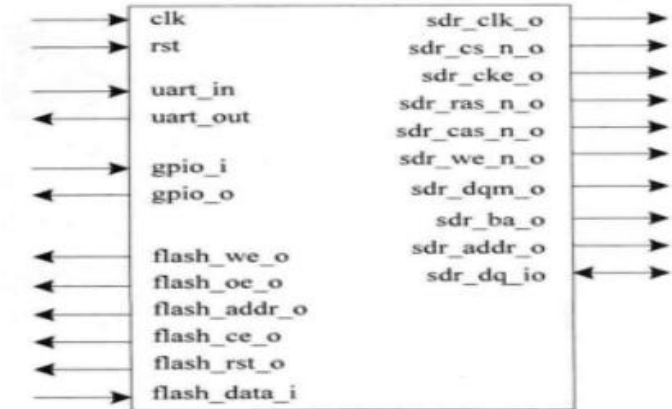
由上图可知，在向 DDR 写数据时，需要提供写命令 `app_cmd`、地址 `app_addr`、数据 `app_wdf_data` 等信号，且写入的数据最多可以比 `app_cmd` 提前一个时钟周期有效，最迟可以比 `app_cmd` 晚两个时钟周期有效。在写数据的时候必须检测 `app_rdy` 和 `app_wdf_rdy` 信号是否同时有效，否则写入命令无法成功写入到 DDR 控制器的命令 FIFO 中，从而导致写操作失败。

3. 读操作的时序



读操作的时序比较简单，只需要注意 `app_rdy` 是否有效即可，其余不再赘述。

至此，实现了基于实践版 OpenMIPS 的小型 SOPC，OpenMIPS 处理器、Wishbone 总线互联矩阵、GPIO 模块、UART 控制器、Flash 控制器、SDRAM 控制器都准备好了，现在只需将这些模块连接起来即可，如下图可知小型 SOPC 的接口描述情况。



序 号	接 口 名	宽 度 (bit)	输 入/输 出	作 用
1	clk	1	输入	时钟
2	rst	1	输入	复位信号
3	uart_in	1	输入	串口输入信号
4	uart_out	1	输出	串口输出信号
5	gpio_i	16	输入	GPIO 输入信号
6	gpio_o	32	输出	GPIO 输出信号
7	flash_oe_o	1	输出	Flash 输出使能信号，低电平有效
8	flash_addr_o	22	输出	Flash 地址信号
9	flash_ce_o	32	输出	Flash 片选信号，低电平有效
10	flash_rst_o	1	输出	Flash 复位信号，低电平有效
11	flash_we_o	1	输出	Flash 写使能信号，低电平有效
12	flash_data_i	8	输入	从 Flash 读出的数据

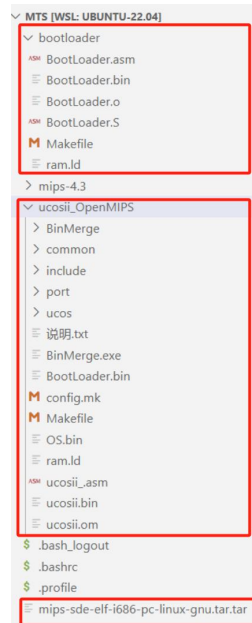
序 号	接 口 名	宽 度 (bit)	输 入/输 出	作 用
13	sdr_clk_o	1	输出	SDRAM 时钟信号
14	sdr_cs_n_o	1	输出	SDRAM 片选信号，低电平有效
15	sdr_cke_o	1	输出	SDRAM 时钟使能信号
16	sdr_ras_n_o	1	输出	SDRAM 行地址选通信号，低电平有效
17	sdr_cas_n_o	1	输出	SDRAM 列地址选通信号，低电平有效
18	sdr_we_n_o	1	输出	SDRAM 写操作信号，低电平有效
19	sdr_dqm_o	2	输出	SDRAM 字节选择和输出使能，低电平有效
20	sdr_ba_o	2	输出	SDRAM 的 Bank 选择信号
21	sdr_addr_o	13	输出	SDRAM 地址总线
22	sdr_dq_io	16	双向	SDRAM 数据总线

3.3.7 利用 Ubuntu 建立交叉编译环境

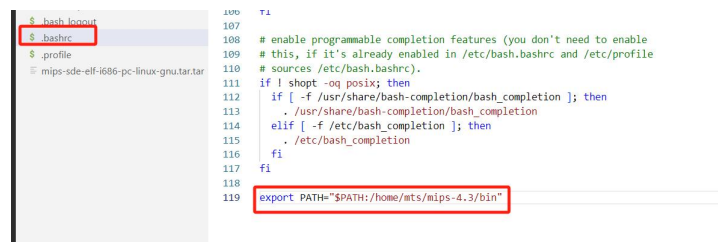
这里与《动手写 CPU》中方法不同，将在 Ubuntu 下配置支持 MIPS 编译的 GNU 对 μ C/OS-II 操作系统进行编译，生成对应的 OS.bin，支持实践版 OpenMIPS 执行的二进制文件。其中构建交叉编译环境的具体过程如下：

1. 将《自己动手写 CPU》tools 目录下提供的 GNU 工具链安装文件 mips-sde-elf-i686-pc-linux-gnu.tar.tar 复制到 Ubuntu 系统的/opt 目录下并解压；复制必要文件到 Ubuntu 系统中。

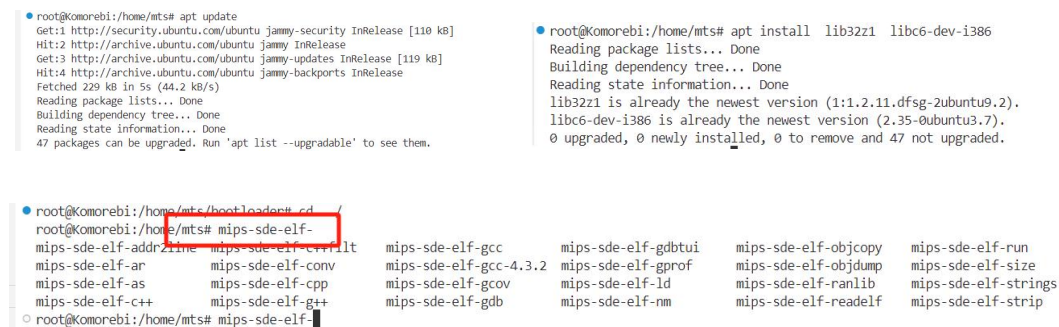
文件包括：； Bootloader 文件夹，ucosii_OpenMIPS 文件夹以及编译工具 mips-sde-elf-i686-pc-linux-gnu.tar.tar 压缩包。将压缩包解压后生成 mips-4.3 编译工具文件夹。



2. 修改环境变量，在.bashrc 中加入编译工具 PATH，确保系统能够访问到该编译工具。



3. 重启 Ubuntu 系统，完成换源操作，安装 lib32z1、libc6-dev-i386 库。输入 mips-sde-elf- 并双击 Tab，出现 mips 编译指令说明 GNU MIPS 工具链安装成功。



4. 完成以上步骤，Ubuntu 下的交叉编译环境配置成功。

3.3.8 对 μ C/OS-II 操作系统改写编译

1. 修改 Bootloader.S 并编译


```

mips-sde-elf-i686-pc-linux-gnu.tar $ .bashrc BootLoader.S X
bootloader > BootLoader.S
9      _start:
10
11      lui $1,0x1000
12      ori $1,$1,0x0003
13      ori $2,$0,0x80
14      sb $2,0x0($1)
15
16      lui $1,0x1000
17      ori $1,$1,0x0001
18      ori $2,$0,0x01
19      sb $2,0x0($1) # MSB of divisor latch
20
21      lui $1,0x1000
22      ori $1,$1,0x0000
23      ori $2,$0,0x45
24      sb $2,0x0($1) # LSB of divisor latch
25
26      lui $1,0x1000
27      ori $1,$1,0x0003
28      ori $2,$0,0x03
29      sb $2,0x0($1) # 8bit, no parity, 1 stop bit
30
31      lui $1,0x2000
32      ori $1,$1,0x0008
33      lui $2,0xffff
34      ori $2,$2,0xffff
35      sw $2,0x0($1)

```

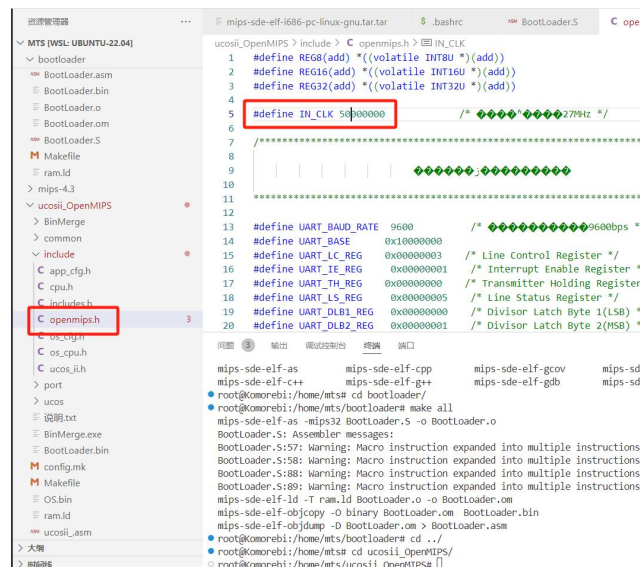
make all 对 BootLoader 进行编译，生成可执行对象文件 BootLoader.o，将其转为可执行的汇编代码 BootLoader.asm（在 make all 中已实现）

```

root@Komorebi:/home/mts/bootloader# make all
mips-sde-elf-as -mips32 BootLoader.S -o BootLoader.o
BootLoader.S: Assembler messages:
BootLoader.S:57: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:58: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:88: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:89: 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
root@Komorebi:/home/mts/bootloader#

```

2. 修改 openmips.h 中的时钟频率并编译



```

1  #define REG8(add) *((volatile INT8U *) (add))
2  #define REG16(add) *((volatile INT16U *) (add))
3  #define REG32(add) *((volatile INT32U *) (add))
4
5  #define IN_CLK 50000000 /* 50MHz */
6
7
8
9
10
11
12
13 #define UART_BAUD_RATE 9600 /* 9600bps */
14 #define UART_BASE 0x10000000
15 #define UART_LC_REG 0x00000003 /* Line Control Register */
16 #define UART_IE_REG 0x00000001 /* Interrupt Enable Register */
17 #define UART_TH_REG 0x00000000 /* Transmitter Holding Register */
18 #define UART_LS_REG 0x00000005 /* Line Status Register */
19 #define UART_DLB1_REG 0x00000000 /* Divisor Latch Byte 1 (LSB) */
20 #define UART_DLB2_REG 0x00000001 /* Divisor Latch Byte 2 (MSB) */

```

```

mips-sde-elf-as mips-sde-elf-cpp mips-sde-elf-gcov mips-sde
mips-sde-elf-c++ mips-sde-elf-g++ mips-sde-elf-gdb mips-sde
root@Komorebi:/home/mts/bootloader# make all
BootLoader.S: Assembler messages:
BootLoader.S:57: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:58: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:88: Warning: Macro instruction expanded into multiple instructions
BootLoader.S:89: 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
root@Komorebi:/home/mts/ucosii_OpenMIPS/
root@Komorebi:/home/mts/ucosii_OpenMIPS/

```

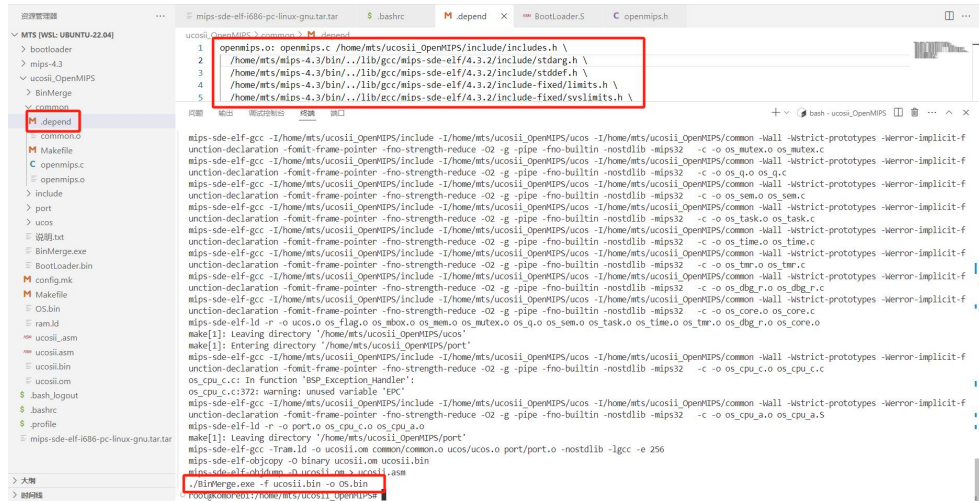
由于 Ubuntu 系统权限控制，此处需要更改 BinMerge.exe 的执行权限之后进行 make all:

```

root@Komorebi:/home/mts/ucosii_OpenMIPS# chmod +x BinMerge.exe
root@Komorebi:/home/mts/ucosii_OpenMIPS#

```

同时由于依赖文件中的路径没有被修改，所以需要将 ucosii_OpenMIPS/.depend 中的路径修改为 mips 编译器路径，之后执行 make all:



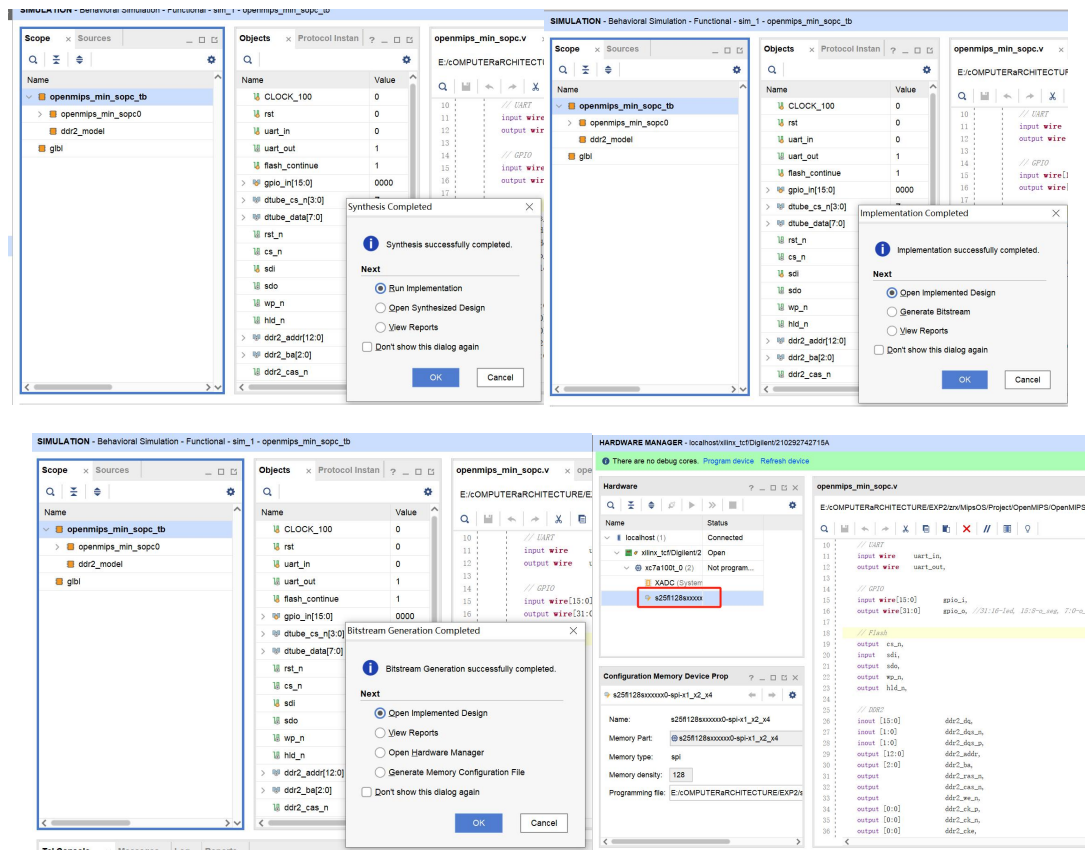
至此，得到二进制文件，将 OS.bin 写入板子上的 Flash，当 OpenMIPS 运行时，会首先运行 BootLoader，后者将 μ C/OC-的代码复制到 SDRAM 中，然后跳转到 SDRAM，把控制权交给 μ C/OS-II，于是 μ C/OS-II 就运行起来了。

4 实验验证

将文件 OS.bin 写入板子上的 SPIflash 中，打开 PC 上的串口程序，将参数设置为 9600bps、8 位数据位、没有奇偶校验位、1 位停止位。拨动开关复位 OpenMIPS，然后启动 OpenMIPS，串口程序将得到结果，汉字每隔 100ms 显示一个，另外 4 个 7 段数码管的显示大概每隔 100ms 变化一次。由此可知，BootLoader 加载 μ C/OS-II 成功， μ C/OS-II 工作正常、移植成功。具体结果展示如下。

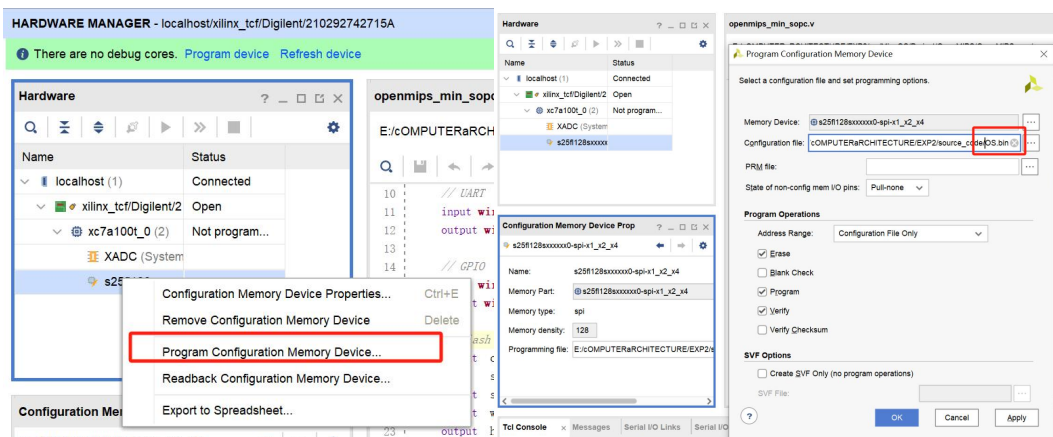
4.1 Bit 流文件生成

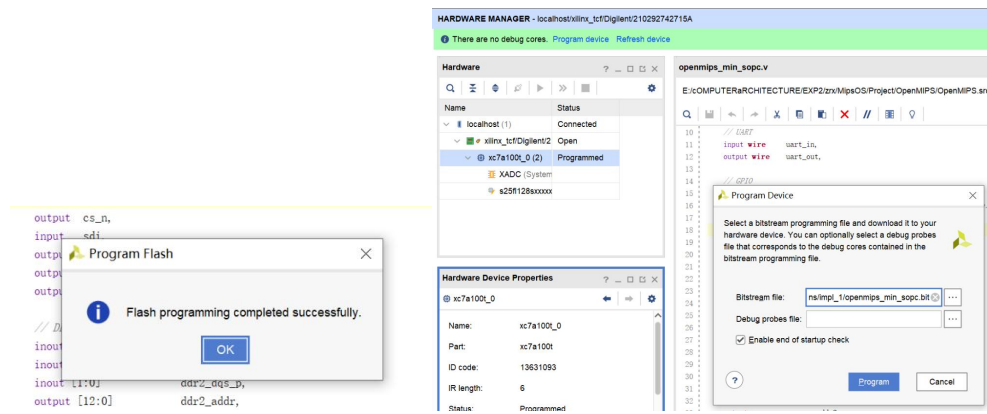
将项目 openmips_min_sopc 导入 Vivado，进行 Synthesis，Implementation 和 Bit 流文件生成得到 Bitstream 文件，打开 Hardware Manager，其中多出配置文件选项：



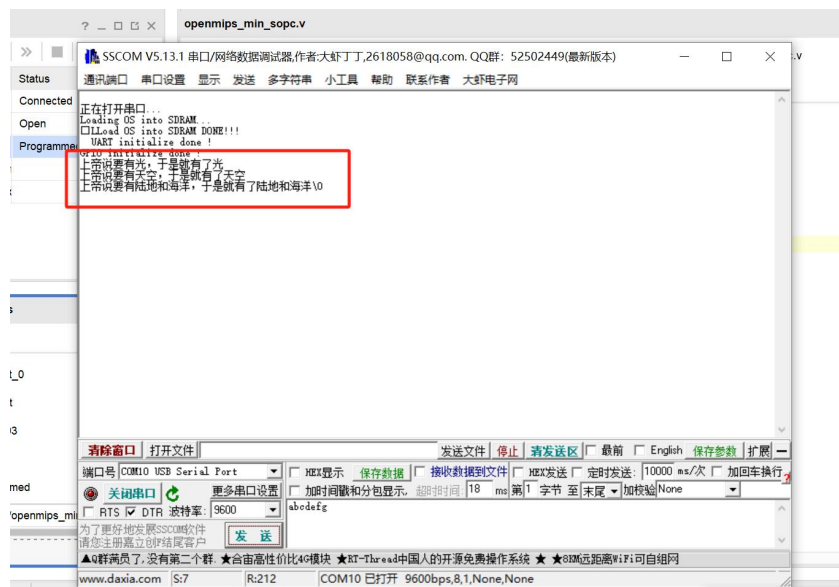
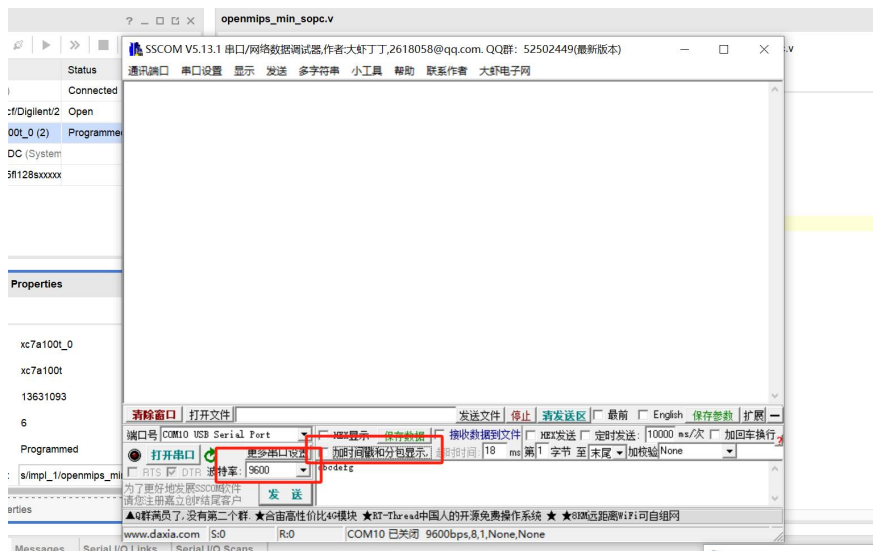
4.2 串口通讯

- 配置 SPIflash 中的数据为 CμC/OS-II 操作系统的二进制文件 OS.bin，将 bit 文件下板：



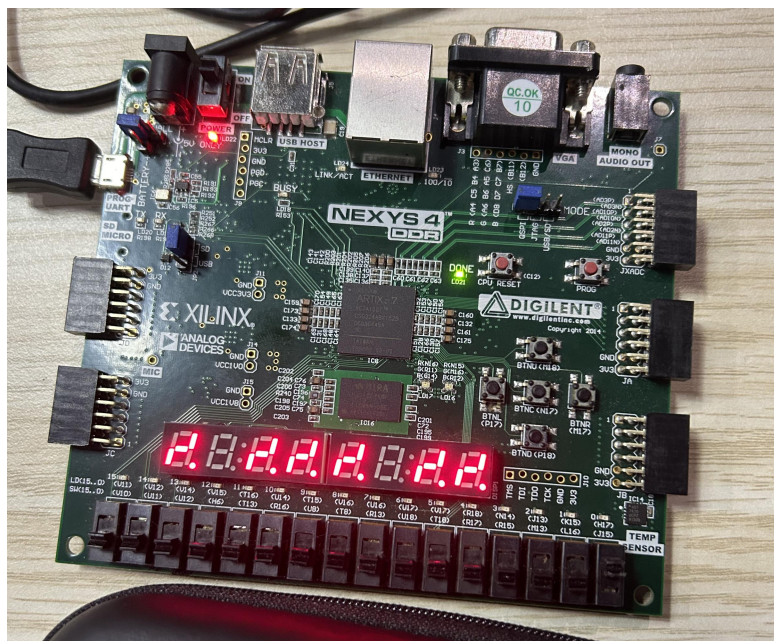


• 打开串口工具，设置波特率为 9600，取消加时间戳和分包显示。串口通信成功



4.3 数码管显示

- 观察开发板，点击开发板 reset，观察 PC 中串口监视工具 SSCOM 是否接收到信息



5 实验总结

5.1 分析

本实验参考雷思磊《自己动手写 CPU》后半部分进行。实验中 $\mu\text{C}/\text{OS-II}$ 操作系统，BootLoader 加载程序，编译配置文件及编译工具均为该书资源文件，本实验主要工作为对部分编译文件修改以及对部分路径文件、环境文件进行配置，从而更好适配 NEXYS DDR Atrix-7 开发板。

在修改 bootloader 源代码的过程中，需要对汇编代码中的标签进行重命名。原先代码中多个标签被简单地标记为值 1，这样的命名方式可能导致代码的可读性较差，且容易出错。为了提高代码的可维护性和可读性，决定将这些标签更改为更具描述性的命名方式，如 L1、L2、L3 等。

同时本次实验中涉及到 bootloader 的串口配置，特别是 UART 控制器的相关设置。考虑到 UART 控制器的时钟频率设定为 100MHz，串口波特率需要设定为 9600，最后需要在 bootloader 的汇编程序中，精确配置 UART 分频系数寄存器为 28BH。

对于操作系统源代码的修改，主要集中在 openmips.h 文件中，调整系统的时钟频率和串口配置。由于 UART 控制器的时钟频率此次设定为 500MHz，而波特率仍旧为 9600，因此需要保持分频系数为 28BH。所以必须在 openmips.h 文件中更新系统时钟频率为

50000000, 保证基于波特率预设值, 应用程序能够正确计算出所需的分频系数。

5.2 总结

本次实验主要分为两部分, 分别在 Ubuntu 系统上进行 OS.bin 文件的生成和在 Vivado 将结果进行下板。

在 OS.bin 文件生成过程中, 编译环境的配置消耗了一定时间。主要原因是我的服务器架构较为特殊, 在安装编译环境时需要使用 yum 或者 RedHat 寻找替代包进行安装。最后更换服务器, 仍使用 apt 进行安装解决问题。同时还需要注意环境变量的配置问题。在多用户系统中或者带有权限的系统中, 环境变量配置 .bashrc 文件的作用域不同, 某些情况下需要直接在控制台进行 export 更新环境变量。在执行 BinMerge.exe 时需要 chmod 更改执行权限防止 Make all 在最后一步无法执行。

Vivado 进行 Bitstream 生成时, 需要尤其注意 Vivado 的版本问题, 我在 2019.2 版本 Vivado 进行实验, 实验过程较为顺利, 可以正常进行下板验证。而 2016 版本中由于版本冲突, 需要手动配置 MIG, 配置过程中常常由于未知原因闪退或报错, 消耗时间较多。其中一种解决方法是将 2019 版本的 ip 核迁移到 2016 版本中进行实验。

本实验完成了简单操作系统的移植, 实现了基于 Wisbone 总线的串口通信, 锻炼了 Linux 系统配置和 Vivado 配置的能力, 收获颇多。