同济大学

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称	实验 2 操作系统移植实验
实验成员	王钧涛 2050254
日　　期	二零二三年　六月　四日

1、实验目的

贯通计算机系大学课程，将编译原理、操作系统等底层软件课程与硬件课程相结合，形成完整的硬件-软件结合知识体系。

2、实验内容

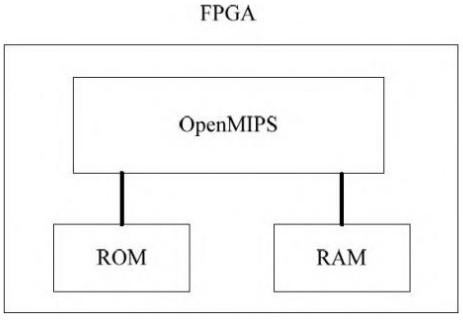
在上次实现的 89 条 MIPS CPU 的基础上增加以下内容：

- 实现总线
  - 增加 Wishbone 总线
  - 增加 GPIO
  - 增加 UART
  - 增加 Flash 控制器
  - 增加 SDRAM 控制器
  - 实现完整 SOPC。
- 系统移植 μC/OS-II
  - 利用 Ubuntu 上建立交叉编译环境
  - 对 μC/OS-II 系统进行改写、编译
- 检查方式：串口助手观察加载过程。

3、实验步骤

一、添加总线设计

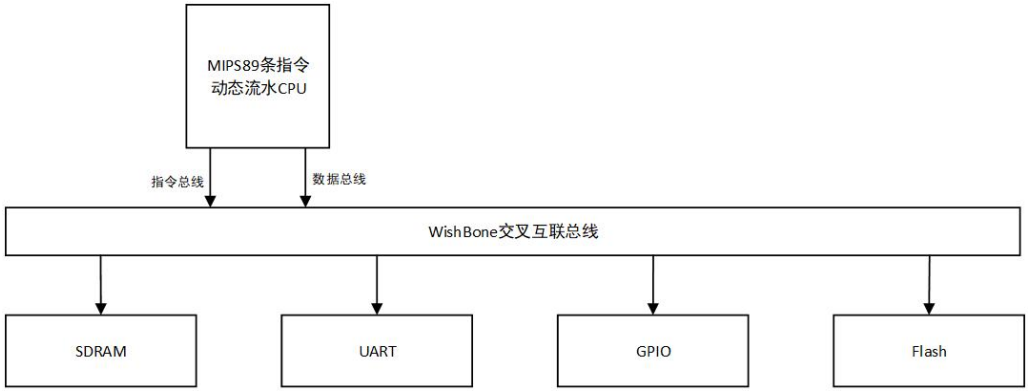
原理：上次实验设计的 CPU 采用哈佛结构，在一个时钟周期内可以取到指令，完成存储、加载数据。此时，指令存储器 ROM 和数据存储器 RAM 都位于 FPGA 内部。



而在本次实验中，程序的体积可能非常大，指令存储器无法集成在 FPGA 内部，因此，需要使用 FPGA 芯片外部的 Flash 作为指令存储器，使用 SDRAM 作为数据存储器。因此，需要在原 CPU 的基础上，加入 Flash 控制器和 SDRAM 控制器。

此外，我们还需要使用串口进行调试，因此需要为 CPU 添加 UART 控制器和 GPIO 控制器。从而就形成了两台主设备需要控制四台从设备总线的情况。在这种情况下，为了实现更好的拓展性，应当为其添加通用总线接口 WishBone，方便地接入新设备。CPU 通过总线接口挂在总线上，各种外部设备的控制器也挂在总线上。由于 OpenMIPS 采用的是哈佛结构，所以有两个总线接口，分别是指令总线接口和数据总线接口。Flash 控制器、SDRAM 控制器、SRAM 控制器、UART 控制器都具有同样的总线接口，都可以挂在总线上，并放在 FPGA 内部，如下图所示：

如下图所示：



接下来将依次介绍本次试验添加的各个模块。

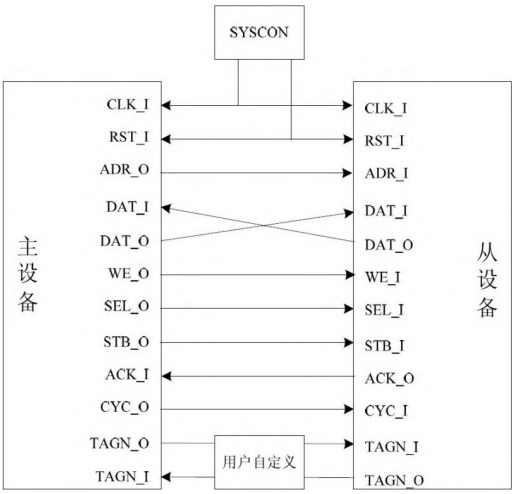
3.1 WishBone 交叉互联总线模块

Wishbone 是一种通用链接总线，有多种连接方式:点对点、数据流、共享总线、交叉互联等。在点对点连接方式中，有一个主设备，一个从设备，输出信号的名称使用“O”结束，输入信号的名称使用“I”结束。所有的信号都是高电平有效。

具体信号含义如下：

接口名称	接口含义
CLK_I / RST_I	时钟信号和复位信号

DAT_O / DAT_I	数据总线，数据可以由主设备传送给从设备，也可以由从设备传递给主设备
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
SEL_O / SEL_I	数据总线选择信号，用于表示当前操作中，数据总线上哪些 bit 是有效的，以总线粒度为单位
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正在占有总线，但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	主从设备操作结束信号，表示成功
ERR_O / ERR_I	主从设备操作结束信号，表示错误
RTY_O / RTY_I	主从设备操作结束信号，表示表示重试
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。



在本次实验中，因为有两台主机和至少四台从机，并且指令存储器和数据存储器需要同时进行访问，因此不太适合使用共享总线的方式（会造成潜在的冲突，从而降低程序的执行效率）。这里更倾向于使用交叉互联的方式进行实现。

接口定义代码如下：

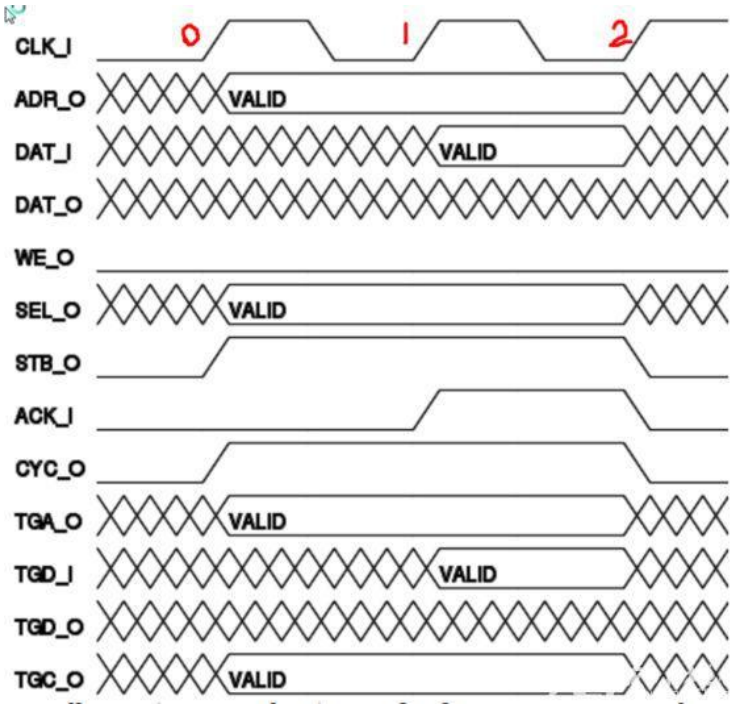
```
module wb_conmax_top(
    clk_i, rst_i,

    // 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 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 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
       );
```

WishBone 读写操作流程

查阅 WishBone 总线手册，对读写的操作有一条约束和一条允许：

1. 规则 3.75：
2. 所有支持单次读或者单次写周期的，主机和从机接口都应满足下文给出的相关时序要求。
3.
4. 允许 3.50：
5. Wishbone 主设备或者从设备也可以不支持单次读/写操作，甚至没有地址和数据总线。

读操作较为简单，因为不牵涉到数据相关问题，其时序图如下：



总线协议的执行流程如下：

时钟上升沿 0：

- 主机将有效地址输出到 ADR_O()和 TGA_O()；
- 主机将 WE_O 复位，表示进入读周期；
- 主机输出 SEL_O()（bank select）表明其操作的数据地址；
- 主机将 CYC_O 和 TCG_O 置位，以表明读周期开始；
- 主机将 STB_O 信号置位，以表明操作开始（start of phase）。

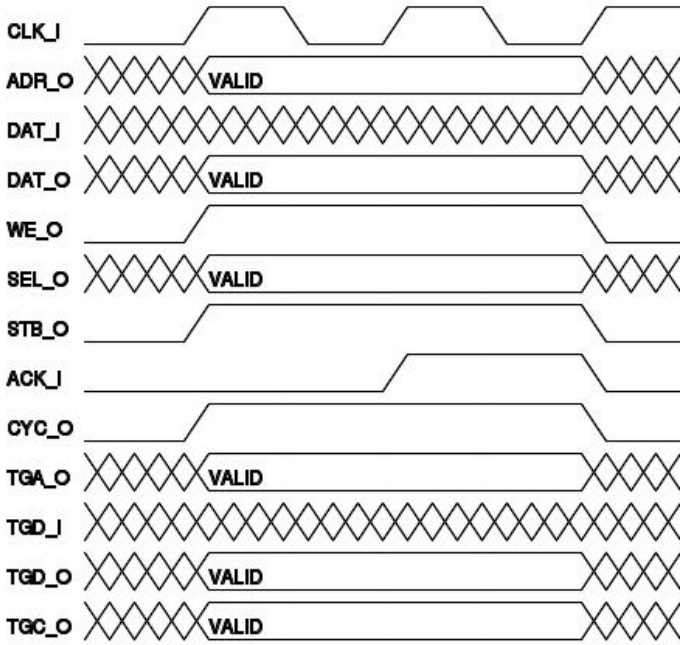
时钟上升沿 1：

- 从机解码输入（检测 STB_O，以验证数据是否有效），并将 ACK_I 置位，以做出响应；
- 从机将有效数据放入 DAT_O()和 TGD_I()；（如之前博文所述，默认信号名为主机信号名!）；
- 主机监控 ACK_I 信号，并准备将 DAT_O()和 TGD_I()信号上的数据进行锁存；

时钟上升沿 2：

- 主机锁存 DAT_O()和 TCG_O()上的数据；
- 主机将 STB_O 和 CYC_O 复位，以表明周期的结束；
- 从机将 ACK_I 信号复位，以响应 STB_O 信号的复位。

写操作则较为复杂，因为要牵涉到**同步异步**读写问题，这也是之后调试 SD 卡一直失败，浪费大量时间的问题所在，其时序如下：



最后在实现时，统一采用同步读写方式。

其具体读写流程如下：

在时钟上升沿 0：

- Master 在[ADR_O()]和[TGA_O()]发出有效的地址
- Master 在[DAT_O()]和[TGD_O()]发出数据
- Master 发出[WE_O]，表明是一个写周期
- Master 发出有效数据选择信号[SEL_O()]表明哪些数据是有效的
- Master 发出[CYC_O]和[TGC_O()]表明总线周期的开始
- Master 发出[STB_O]表明操作的开始

在时钟上升沿 1：

• Slave 检测到主设备发起的操作，准备发出[ACK_I]

• Slave 准备锁存[DAT_O]和[TGD_O()]

• Slave 发出[ACK_I]应答[STB_O]，表明数据有效，可以读取数据了

• Master 发现[ACK_I]，准备结束总线周期

• Slave 可以在发出[ACK_I]前插入等待周期(-WSS-)，以控制传速度。可以插入任意多个等待周期。（**最初设计 SD 卡的时候没有考虑到这点，因为调试的时候会将 CPU 频率调的很低，而由于 SD 卡读取速度是要慢于正常 CPU 频率的，就导致 Slave 发出 Wait 信号的时候，CPU 仍然当其为正常代码进行读写，就出现了时序问题。后来发现这个问题之后，为 waiting 周期加上了指令流暂停指令 stall，但是又发现最初设计 CPU 的时候因为没有考虑这个问题，stall 指令的设计是不完善的，没有考虑数据相关时候的写回问题，造成了错误的数据运行覆盖。由于每次运行都需要下板观察代码执行结果，烧写 SD 卡的调试太麻烦了，浪费了大量时间在调试上）**

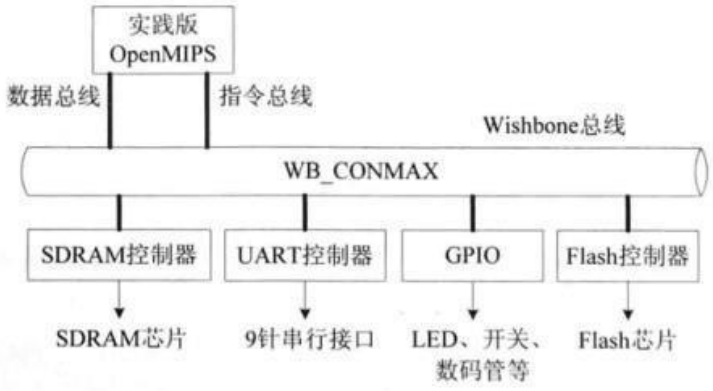
在时钟上升沿 2：

• Slave 锁存[DAT_I]和[TGD_I()]

• Master 拉低[STB_O]和[CYC_O]，表明总线周期的结束

• Slave 发现 Master 拉低[STB_O]，也将[ACK_I]拉低

改造后，整体的 SPOC 架构变为如下图所示：



先介绍如何我的代码的实验 1 MIPS 89 条 CPU 是如何接上 WB 总线的：

首先从 CPU 的接口入手，修改接口代码如下：

```
module cpu_top(
    input CLK100MHZ,
    input rst,

    input [31:0] im_idata,
    output [31:0] im_odata,
    output [31:0] im_addr,
    output [3:0] im_sel,
    output im_we,
     output reg im_cyc,
     output reg im_stb,
     input im_ack,
     input im_err,
     input im_rty,

     input [31:0] dm_idata,
     output [31:0] dm_odata,
     output [31:0] dm_addr,
     output [3:0] dm_sel,
     output reg dm_we,
     output reg dm_cyc,
     output reg dm_stb,
     input dm_ack,
     input dm_err,
     input dm_rty,

     input [5:0] int_i,
     output timer_int,

     output [31:0] debug,
     output stall,
     output [3:0] st
);
```

加粗的部分是新添加的信号控制部分，addr、data 等数据接口是不用修改的。

在修改完代码之后，需要实际将这些信号进行输出或转换为 CPU 的某些控制信号，这需要在 CPU 的顶层模块中接入信号状态机如下：

指令存储交互：

```
iIDLE:begin
    im_cyc<=1'b0;
    im_stb<=1'b0;
    iCPU_stall<=1'b1;
    state<=iREAD;
```

```
end
iREAD:begin
    im_cyc<=1'b1;
    im_stb<=1'b1;
     if(im_ack)
     begin
         iCPU_im_dout<=im_idata;
         state<=dIDLE;
     end
     else
     begin
         state<=iREAD;
     end
end
```

数据存储交互：

```
dIDLE:begin
    im_cyc<=1'b0;
    im_stb<=1'b0;
    dm_cyc<=1'b0;
    dm_stb<=1'b0;
    if(oCPU_dm_ena&&~oCPU_dm_we)
    begin
        dm_we<=1'b0;
        state<=dREAD;
     end
     else if(oCPU_dm_ena&&oCPU_dm_we)
     begin
         dm_we<=1'b1;
         state<=WRITE;
     end
     else
     begin
         dm_we<=1'b0;
         state<=OK1;
     end
end
dREAD:begin
    dm_cyc<=1'b1;
    dm_stb<=1'b1;
    if(dm_ack)
    begin
         iCPU_dm_dout<=dm_idata;
         state<=OK1;
     end
     else
     begin
```

```
32.         state<=dREAD;
33.     end
34. end
35. WRITE:begin
36.     dm_cyc<=1'b1;
37.     dm_stb<=1'b1;
38.     if(dm_ack)
39.     begin
40.         state<=OK1;
41.     end
42.     else
43.     begin
44.         state<=WRITE;
45.     end
46. end
```

读取状态机的原理解释如下：

iIDLE（空闲状态）：

将 im_cyc 和 im_stb 都设置为逻辑 0。

将 iCPU_stall 设置为逻辑 1。

切换到 iREAD 状态。

iREAD（读取状态）：

将 im_cyc 和 im_stb 都设置为逻辑 1。

如果 im_ack 为真，则执行以下操作：

将 im_idata 赋值给 iCPU_im_dout。

切换到 dIDLE 状态。

否则，切换回 iREAD 状态。

写入状态机的原理解释如下：

dIDLE（空闲状态）：

将 im_cyc、im_stb、dm_cyc 和 dm_stb 都设置为逻辑 0。

如果 oCPU_dm_ena 为真且 oCPU_dm_we 为假，则执行以下操作：

将 dm_we 设置为逻辑 0。

切换到 dREAD 状态。

否则，如果 oCPU_dm_ena 为真且 oCPU_dm_we 也为真，则执行以下操作：

将 dm_we 设置为逻辑 1。

切换到 WRITE 状态。

否则，执行以下操作：

将 dm_we 设置为逻辑 0。

切换到 OK1 状态。

dREAD（读取状态）：

将 dm_cyc 和 dm_stb 都设置为逻辑 1。

如果 dm_ack 为真，则执行以下操作：

将 dm_idata 赋值给 iCPU_dm_dout。

切换到 OK1 状态。

否则，切换回 dREAD 状态。

WRITE（写入状态）：

将 dm_cyc 和 dm_stb 都设置为逻辑 1。

如果 dm_ack 为真，则切换到 OK1 状态。

否则，切换回 WRITE 状态。

此外，修复了上一个实验设计的 CPU 在 STALL 的情况下不能正确暂停的问题：

将读取到的暂停状态传入 CPU：

```
assign stall=iCPU_stall;
assign st=state;
```

并在 if 段到 ir 段的过程、mem 段存储数据的过程中添加流水线暂停参数 stall：

```
module PipeIR(
    input clk,
    input rst,
    input stall,
    input [31:0] pc4,
```

```
    input [31:0] inst,
    input blockade,
    output reg [31:0] D_pc4,
    output reg [31:0] D_inst,
     output reg D_blockade
);
    always@(posedge clk or posedge rst)
        ...
        else if(~stall)
        begin
            D_pc4<=pc4;
            D_inst<=inst;
            D_blockade<=blockade;
        end
    end
endmodule
```

在 IR 段进行阻断后，数据仍然能够正常取指，但是不会进行解析，后面尚未执行完的代码会正常进行执行。

WishBone 的代码我直接使用的发布版的代码，基本没做额外的修改，只需要保证在读取冲突异常的时候：

1. 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 `WBIDLE`

此时清除流水线的指令需要 CPU 通过 SPOC 接受。

WishBone 总线互联矩阵分配的地址空间如下，考虑到之后实现的拓展功能，为 VGA 和 PS2 预留了位置：

接口	连接	寻址空间
主设备接口 0	CPU 指令总线	/
主设备接口 1	CPU 数据总线	/
从设备接口 0	SDRAM DDR2 内存控制器	0x00000000-0x0FFFFFFF
从设备接口 1	UART 控制器	0x10000000-0x1FFFFFFF
从设备接口 2	GPIO 控制器	0x20000000-0x2FFFFFFF
从设备接口 3	SD 卡控制器	0x30000000-0x3FFFFFF
从设备接口 4	VGA 控制器（预留，目前没	0x40000000-0x4FFFFFF

	有实现）	
从设备接口 5	PS2 键盘鼠标控制器（预留，目前没有实现）	0x50000000-0x5FFFFFF

3.2 GPIO 模块设计

GPIO (General Purpose Input Output）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。

该模块较为简单，定义如下：

```
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，主要用于数据和使能传输的寄存器如下：

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

本次试验将七段数码管交由 GPIO 控制。

3.3 UART 模块设计

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter，通常称为 UART）是一种异步收发传输器，是电脑硬体的一部分，将数据通过串列通讯进行传输。它在发送端执行并行到串行数据转换，在接收端执行串行到并行数据转换。它是通用的，因为传输速度、数据速度等参数是可配置的。UART 通常用在与其他通讯接口（如 EIA RS-232）的连接上。

UART 中的字母“A”代表异步，即没有时钟信号来同步或验证从发送器发送并由接收器接收的数据（异步串行通信）。这与同步串行通信相反，同步串行通信使用发送器和接收器之间共享的时钟信号来“同步”它们之间的数据。在 UART 中，发送器和接收器必须事先就时序参数达成一致。此外，UART 在每个数据字的开头和结尾使用特殊位来同步发送器和接收器。

在数字逻辑课程中已经利用过 UART 进行过数据收发，其传输如下：

发送：依次传输起始位、数据位、奇偶校验位、停止

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。

在接受时，只需要反向操作即可。

模块定义如下：

```
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
     );
```

这里采用的代码与数字逻辑课程采用的代码相同，只是在其上为适配 WB 总线添加了状态控制机（UART 自带的状态已经能够覆盖 WB 总线的状态，只需在其上做逻辑计算进行转换即可，不另行设计状态机模块）。

3.4 FLASH 模块设计

本次试验没有采用 FLASH 模块，而是采用 SD 卡模块进行替代。尽管如此，其工作原理却是一致的，FLASH 通过 SPI 协议进行交互，而 SD 卡模块也可以不适用高速的协议，而是使用 SPI 协议。采用 SD 卡的好处是，我们无需在每次进行烧录操作，只需要在系统代码更新的时候将 SD 卡插入机器即可，大大方便开发过程。

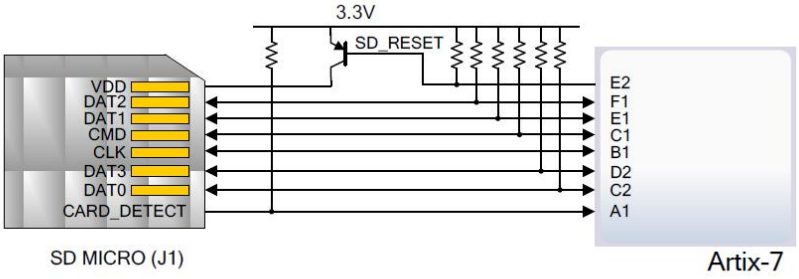


Figure 21. Artix-7 microSD card connector interface (PIC24 connections not shown).

这里只用到 DAT0，VDD，CD，CLK,CMD，DAT3 管脚，其中 DAT0 为单向输入管脚，CMD 为单项输出管脚。

引脚编号	引脚名称	功能（SDIO 模式）	功能（SPI 模式）
Pin 1	DAT2	数据线 2	保留
Pin 2	DAT3/CS	数据线 3	片选信号
Pin 3	CMD/MODI	命令线	主机输出，从机输入
Pin 4	VDD	电源	电源
Pin 5	CLK	时钟	时钟
Pin 6	VSS	电源地	电源地
Pin 7	DAT0/MISO	数据线 0	主机输入，从机输出
Pin 8	DAT1	数据线 1	保留

DAT1、DAT2 作为保留管脚，不使用。

与 flash 不同的是，SD 卡在正常读写操作之前，必须先对 SD 卡进行初始化，SD 卡的初始化过程就是向 SD 中写入命令。在对 SD 卡进行读写操作时同样需要先发送写命令和读命令，SD 卡的命令格式由 6 个字节组成（先发送高位再发送低位）

读取命令如下：

byte1：命令号，格式为 01xx_xxxx,

byte2~byte5:命令参数

byte6: 前 7 位为 CRC（循环冗余校验）校验位，最后一位为停止位 0. 在 SPI 模式下默认不开启 CRC 校验(校验位全部设为 1)，在 SDIO 模式下开启 CRC 校验.SD 卡上电默认是 SDIO 模式，在接收 SD 卡返回 CMD0 的响应命令时，拉低片选 CS，这样就可以进入 SPI 模式。

初始化过程如下：

1、先对从机 SD 卡发送至少 74 个以上的同步时钟，在上电同步期间，片选 CS 引脚和 MOSI 引脚必须为高电平（MOSI 引脚除发送命令或数据外，其余时刻都为高电平）

2，拉低片选 CS 引脚，发送命令 CMD0（0x40）复位 SD 卡，等待返回数据

3，在接收返回信息期间片选 CS 为低电平，判断数据为复位完成信号 0x01，SD 卡返回响应数据的 8 个时钟周期后再拉高片选 CS 信号，SD 卡进入 SPI 模式。如果返回其他值，重新上一步。

4，拉低片选 CS 引脚，发送命令 CMD8（0x48）查询 SD 卡的版本号，等待返回数据。

5，SD 卡返回响应数据后，先等待 8 个时钟周期再拉高片选 CS 信号，此时判断返回的响应数据，如果为 4'b0001,(即 2.7V~3.6V)，SD 卡位 2.0 版本，否则上一步。

6、拉低片选 CS 引脚，发送命令 CMD55（0x77）告诉 SD 卡下一次发送的命令是应用相关命令，等待返回数据。

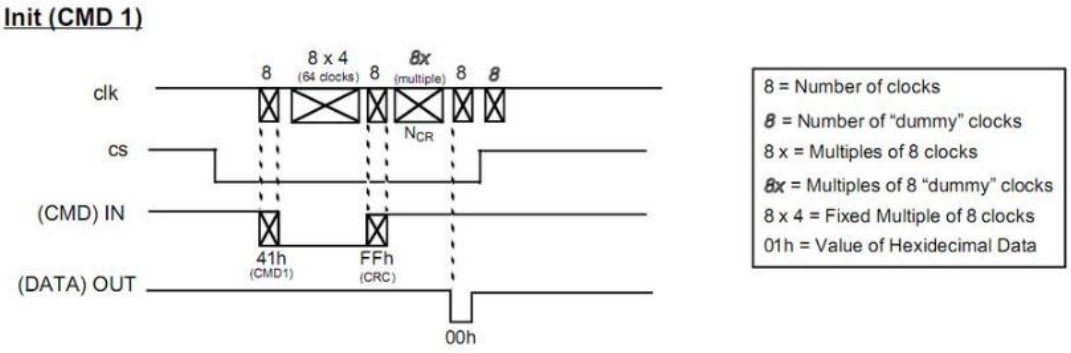
7、SD 卡返回响应数据后，先等待 8 个时钟周期再拉高片选 CS 信号，此时判断返回的响应数据。如果返

回的数据为空闲信号 0x01，开始进行下一步

8、拉低片选 CS 引脚，发送命令 ACMD41（0x69）查询 SD 卡是否初始化完成，等待返回数据。

9、SD 卡返回响应数据后，先等待 8 个时钟周期再拉高片选 CS 信号，此时判断返回的响应数据若为 0x00,则初始化完成。否则第 6 步。

SD 卡在初始化的时候，SPI_CLK 的时钟频率不能超过 400KHz，在初始化完成之后，再将 SPI_CLK 的时钟频率切换至 SD 卡的最大时钟频率。



由于本次移植的操作系统只需要对磁盘进行写入而不需要读出，因此没有设计写操作，读操作的流程如下：

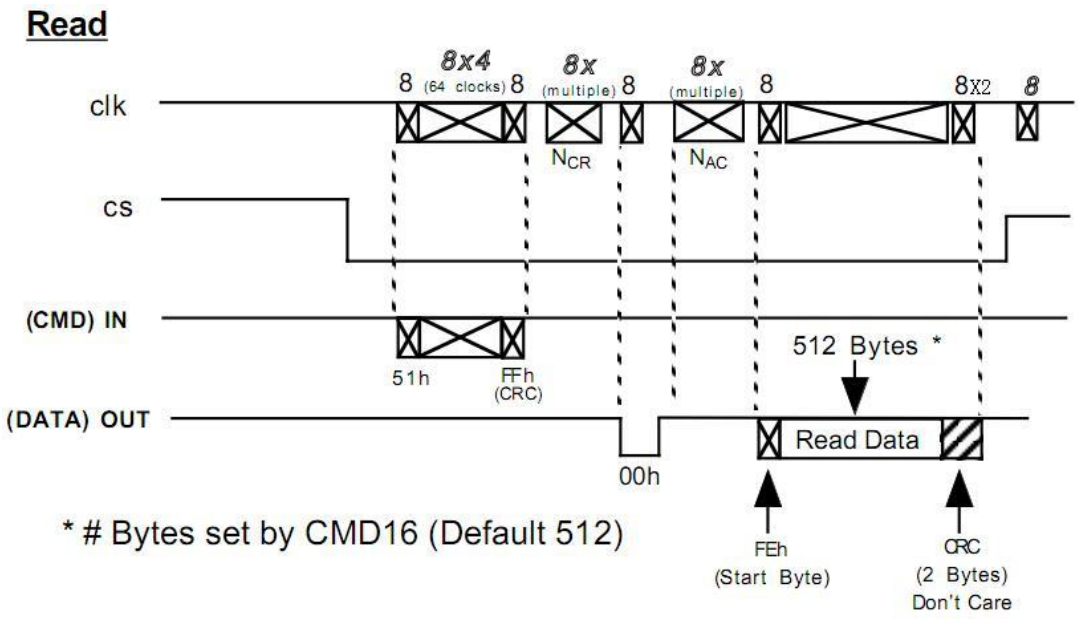
1、 拉低片选 CS 引脚，发送命令 CMD17（0x51）读取单个数据块，等待返回响应数据；

2、 SD 卡返回正确响应数据 0x00 后，准备开始解析 SD 卡返回的数据头 0xfe；

3、 解析到数据头 0xfe 后，接下来接收 SD 卡返回的 512 个字节的数据；

4、 数据解析完成后，接下来接收两个字节的 CRC 校验值。由于 SPI 模式下不对数据进行 CRC 校验，可直接忽略这两个字节；

6、 拉高片选 CS 引脚，等待 8 个时钟周期后允许进行其它操作。



具体模块定义如下：

```
1. module sd_func_controller(
2.     input CLK100MHZ,
3.     input rst,
4.     input CLK25MHZ,
5.     input locked,
6.
7.     input SD_CD,
8.     output SD_RESET,
9.     output SD_SCK,
10.     output SD_CMD,
11.     inout [3:0] SD_DAT,
12.
13.     output [31:0] s_odata,
14.     input [31:0] s_idata,
15.     input [31:0] s_addr,
16.     input [3:0] s_sel,
17.     input s_we,
18.     input s_cyc,
19.     input s_stb,
20.     output reg s_ack,
21.     output reg s_err,
22.     output reg s_rty,
23.     output o_dirty,
24.     output [3:0] st,
25.     output [31:0] bt,
26.     output [4:0] signal_controller_status,
27.     output read
28.     );
```

本模块采用 https://github.com/mczerski/SD-card-controller 项目的 IP 核进行读写控制。

3.4 DDR2 SDRAM 模块设计

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。

DDR2 SDRAM（Double Data Rate 2 Synchronous Dynamic Random-Access Memory）是一种广泛应用于计算机系统和其他电子设备中的存储器类型，在计算机系统和其他电子设备中广泛用于主存储器、图形处理器、网络设备和嵌入式系统等方面。其高带宽、低延迟和较低的功耗特性使其成为现代计算和通信系统中的重要组成部分。

根据 nexys4 的开发手册，开发板上共设置了 1GiB 大小的 DDR SDRAM 存储空间，刚开始开发的时候我没有看清，还以为是设置了 1GB 大小，而 wishbone 控制总线最多只支持 256MB 的寻址空间，吓得我找了好久如何解决，最后才发现实际的 DDR 容量为 128MB，完全不用担心地址越界的问题。

More advanced users or those who wish to learn more about DDR SDRAM technology may want to use the Xilinx 7-series memory interface solutions core generated by the MIG (Memory Interface Generator) Wizard. Depending on the tool used (ISE, EDK or Vivado), the MIG Wizard can generate a native FIFO-style or an AXI4 interface to connect to user logic. This workflow allows the customization of several DDR parameters optimized for the particular application. Table 2 below lists the MIG Wizard settings optimized for the Nexys4 DDR.

Setting	Value
Memory type	DDR2 SDRAM
Max. clock period	3000ps (667Mbps data rate)
Recommended clock period (for easy clock generation)	3077ps (650Mbps data rate)
Memory part	MT47H64M16HR-25E
Data width	16
Data mask	Enabled
Chip Select pin	Enabled
Rtt (nominal) – On-die termination	50ohms
Internal Vref	Enabled
Internal termination impedance	50ohms

Table 2. DDR2 settings for the Nexys4 DDR.

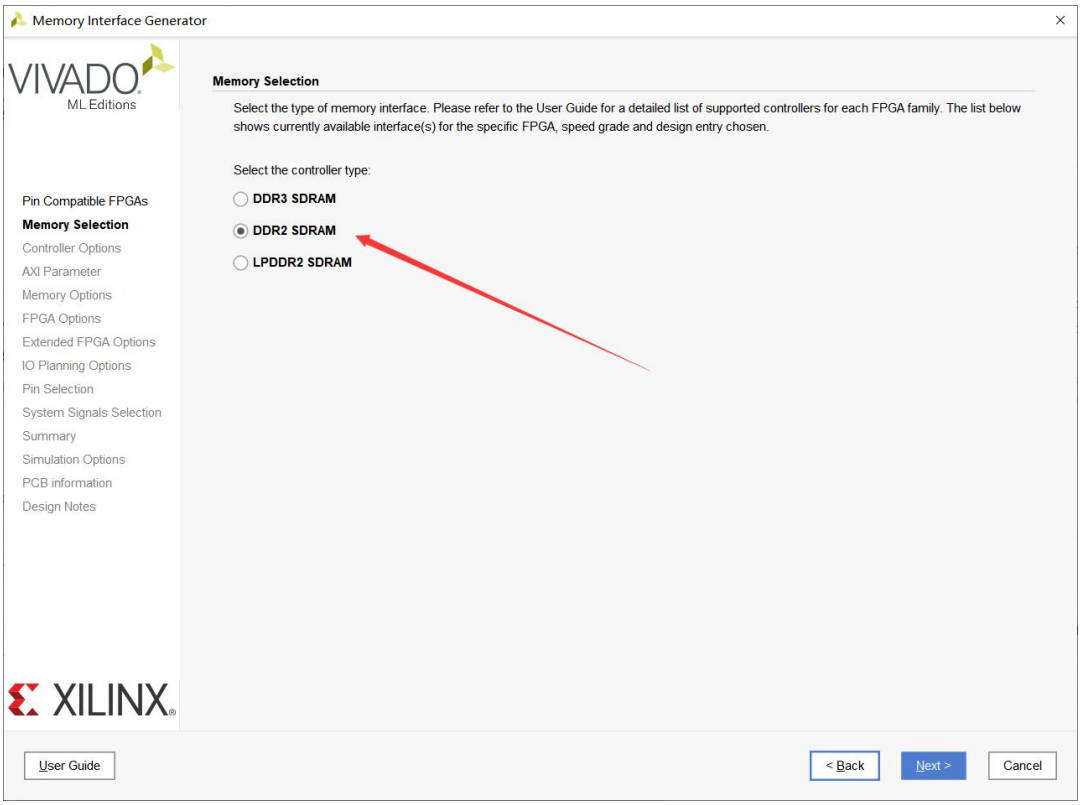
其写入过程相当复杂，我看手册真的是一头雾水，这里借助 IP 核 Memory

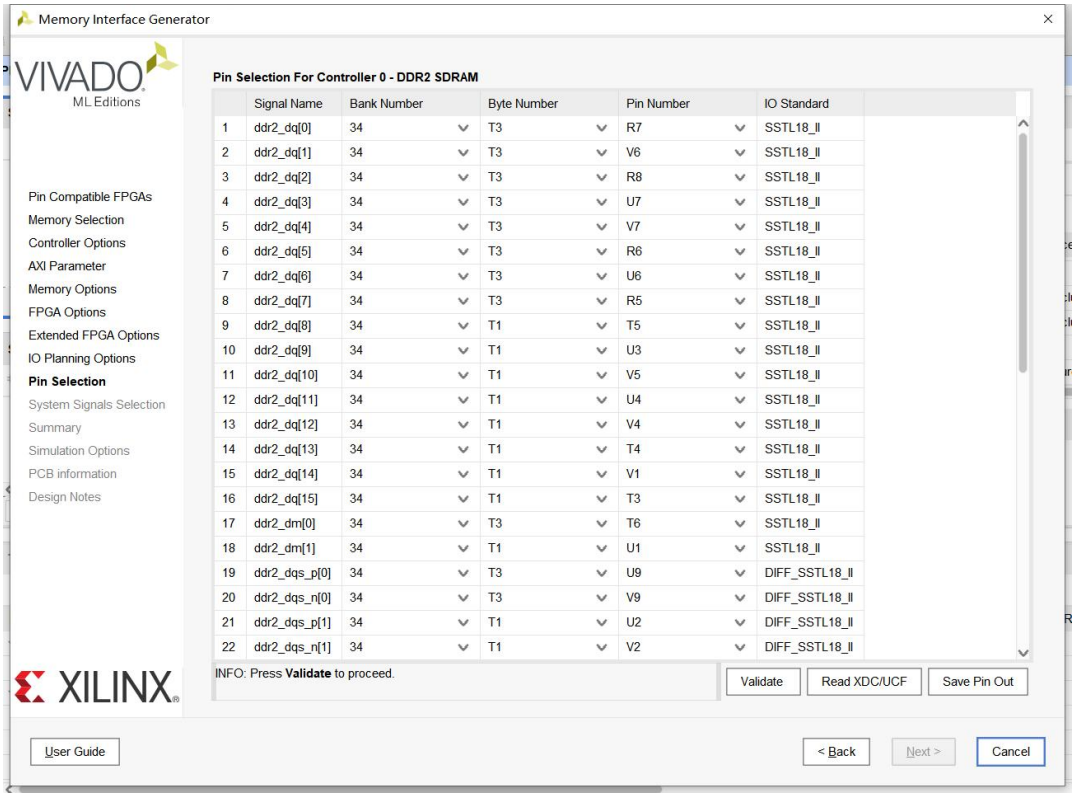
Interface Generator 封装 DDR 的物理层信号，然后在其上为 WB 总线作状态机的适配。 该 IP 核具有以下特点。

- ✧ 支持 SDRAM 的数据总线宽度可以为 8、16、32。
- ✧ 列地址宽度可配置。
- ✧ 支持 4 个 Bank 的 SDRAM。
- ✧ CAS 延迟可配置。
- ✧ 支持数据掩码，从而实现“部分写”操作(PartialWrite Operation)。
- ✧ 自动控制刷新。
- ✧ 支持所有标准的 SDRAM 功能。
- ✧ 支持 WishboneB 版本。

装 订 线

主要需要选择的配置，其他直接使用默认配置：

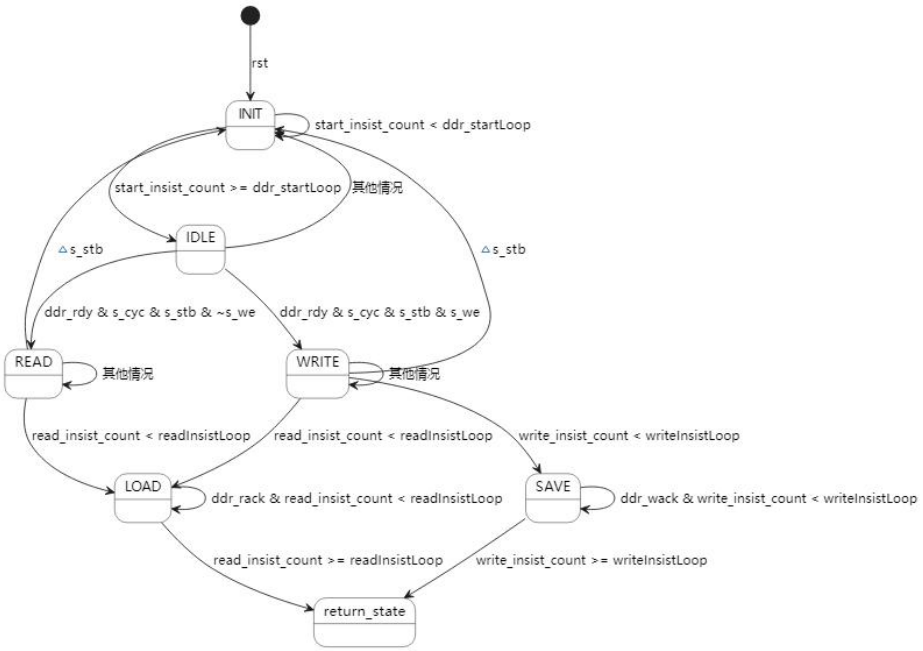




	Signal Name	Bank Number	Byte Number	Pin Number	IO Standard
1	ddr2_dq[0]	34	T3	R7	SSTL18_II
2	ddr2_dq[1]	34	T3	V6	SSTL18_II
3	ddr2_dq[2]	34	T3	R8	SSTL18_II
4	ddr2_dq[3]	34	T3	U7	SSTL18_II
5	ddr2_dq[4]	34	T3	V7	SSTL18_II
6	ddr2_dq[5]	34	T3	R6	SSTL18_II
7	ddr2_dq[6]	34	T3	U6	SSTL18_II
8	ddr2_dq[7]	34	T3	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
21	ddr2_dqs_p[1]	34	T1	U2	DIFF_SSTL18_II
22	ddr2_dqs_n[1]	34	T1	V2	DIFF_SSTL18_II

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。

所采用的状态机如下：



状态机在时钟上升沿（posedge CLK100MHZ）或复位（posedge rst）时工作。

在复位状态下，状态机将各个信号和计数器重置，并将状态设置为 INIT。

在非复位状态下，状态机根据当前状态执行相应的操作并根据条件进行状态转移。

INIT 状态：等待 DDR2 SDRAM 就绪，如果超过了指定的起始等待时间（start_insist_count >= ddr_startLoop），则转移到 IDLE 状态；否则继续保持在 INIT 状态。

IDLE 状态：根据输入信号和操作类型判断是读取（READ）还是写入（WRITE），并根据相应的条件进行状态转移。

READ 状态：在一定次数内重复读取数据，如果达到指定次数（read_insist_count >= readInsistLoop），则转移到 LOAD 状态；如果输入信号 s_stb 为低电平，则转移到 INIT 状态；否则继续保持在 READ 状态。

WRITE 状态：在一定次数内重复写入数据，如果达到指定次数（write_insist_count >= writeInsistLoop），则转移到 SAVE 状态；如果输入信号 s_stb 为低电平，则转移到 INIT 状态；否则继续保持在 WRITE 状态。

LOAD 状态：等待 DDR2 SDRAM 读取操作完成，如果未达到指定次数（read_insist_count < readInsistLoop），则继续重复读取并保持在 LOAD 状态；否则转移到之前记录的返回状态（return_state）。

SAVE 状态：等待 DDR2 SDRAM 写入操作完成，如果未达到指定次数（write_insist_count < writeInsistLoop），则继续重复写入并保持在 SAVE 状态；否则转移到之前记录的返回状态（return_state）

模块定义如下：

```
1. module ddr2_func_controller(
2.     input CLK100MHZ,
3.     input rst,
4.     input CLK200MHZ,
5.     input locked,
6.
7.     output ddr2_ck_p,
8.     output ddr2_ck_n,
9.     output ddr2_cke,
10.     output ddr2_cs_n,
11.     output ddr2_ras_n,
```

```
    output ddr2_cas_n,
    output ddr2_we_n,
    output [1:0] ddr2_dm,
    output [2:0] ddr2_ba,
    output [12:0] ddr2_addr,
    inout [15:0] ddr2_dq,
    inout [1:0] ddr2_dqs_p,
    inout [1:0] ddr2_dqs_n,
    output ddr2_odt,

    output reg [31:0] s_odata,
    input [31:0] s_idata,
    input [31:0] s_addr,
    input [3:0] s_sel,
    input s_we,
    input s_cyc,
    input s_stb,
    output reg s_ack,
    output reg s_err,
    output reg s_rty,

    output [7:0] st
);
```

装 订 线

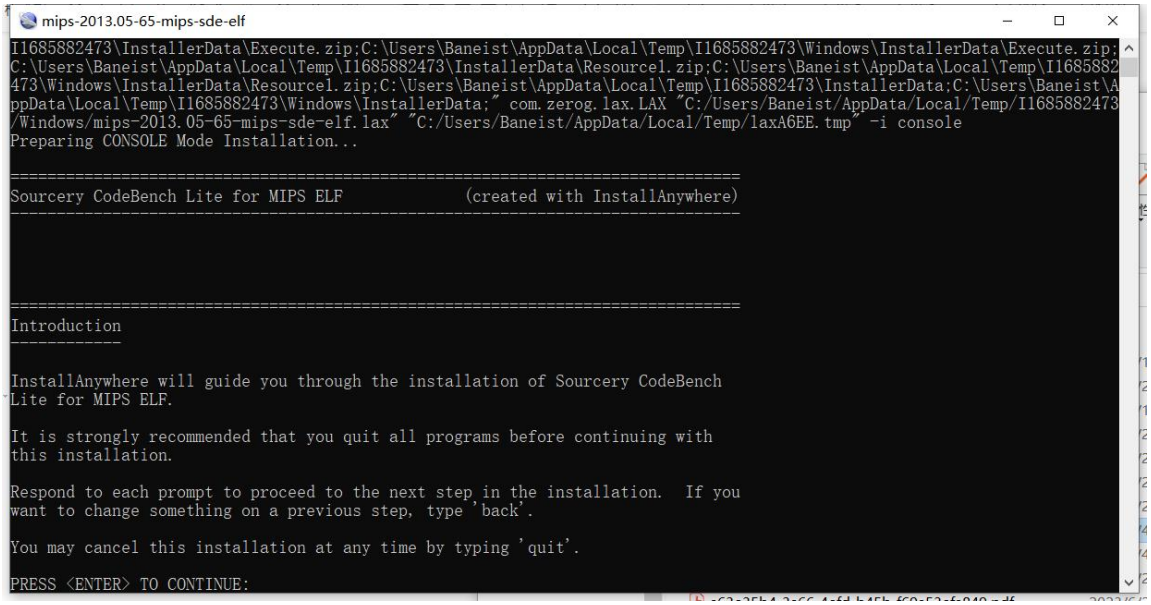
3.5 交叉编译环境建立

本次试验成功在 Windows 环境下建立起交叉编译环境，并且在 Ubuntu 上进行交叉开发。

1. 下载 mips-2013.05-65-mips-sde-elf.exe：注意，只有该版本可以成功编译，其他在 windows 平台下的版本都无法进行编译。

https://download.csdn.net/download/wfxzf/11082585

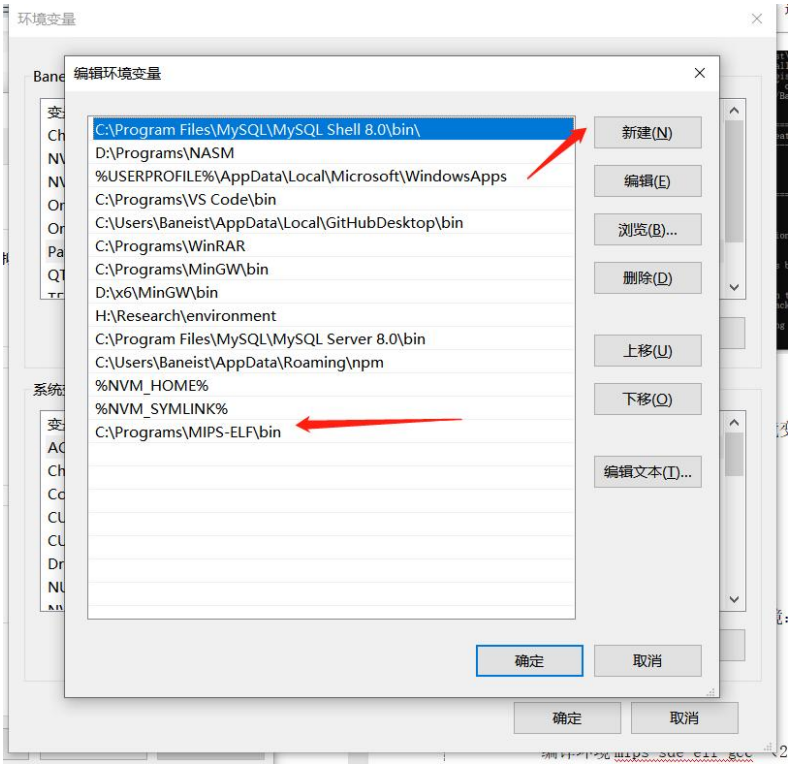
2. 运行后在命令行下执行 mips-2013.05-65-mips-sde-elf -i console，进入命令行安装界面。该程序其实自带有 GUI 安装界面，但是由于版本太过古老，其用的 GUI API 是十年前的 MVC，目前在 windows10 上已经无法运行，因此需要用命令行的方式进行安装。



一直按 ENTER 默认安装即可。

3. 将安装好的编译环境添加入环境变量：

点击新建 - 输入安装的目录的 bin 文件夹



4. 修改 makefile，使其适配 Windows 环境：

更改 shell：

```
TOPDIR  := $(shell cd)
export  TOPDIR

include ./config.mk
```

更改 depend 和 subdirs：

```
depend dep:
    @for %%i in ($(SUBDIRS)) do ( $(MAKE) -C %%i .depend )

subdirs:
    @for %%i in ($(SUBDIRS)) do ( $(MAKE) -C %%i || exit /b )
```

5. 修改 binMerge.exe

使用 binMerge 时会报错：

因此只好用 python 自己实现一个：

```
import os
with open('BootLoader.bin', 'rb') as f1:
    BootLoader = f1.read()
while len(BootLoader) < 512 + 256:
    BootLoader += bytearray([0xff])
with open('ucosii.bin', 'rb') as f2:
    ucosii = f2.read()
BootLoader += bytearray([0x00, 0x00, len(ucosii) // 256, len(ucosii) % 256])
ret = BootLoader + ucosii
while len(ret) % 512 != 0:
    ret += bytearray([0x00])
with open('2050254-OS.bin', 'wb') as f:
    f.write(ret)
```

继续修改 makefile 代码如下：

```
OS.bin: ucosii.bin
    python ./merge.py
```

执行 make clean & make：

```
make[1]: Entering directory `D:/Repos/computer-system/testos/common'
make[1]: `.depend' is up to date.
make[1]: Leaving directory `D:/Repos/computer-system/testos/common'
make[1]: Entering directory `D:/Repos/computer-system/testos/ucos'
make[1]: `.depend' is up to date.
make[1]: Leaving directory `D:/Repos/computer-system/testos/ucos'
make[1]: Entering directory `D:/Repos/computer-system/testos/port'
make[1]: `.depend' is up to date.
make[1]: Leaving directory `D:/Repos/computer-system/testos/port'
make[1]: Entering directory `D:/Repos/computer-system/testos/common'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `D:/Repos/computer-system/testos/common'
make[1]: Entering directory `D:/Repos/computer-system/testos/ucos'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `D:/Repos/computer-system/testos/ucos'
make[1]: Entering directory `D:/Repos/computer-system/testos/port'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `D:/Repos/computer-system/testos/port'
mips-sde-elf-gcc -Tram.ld -o ucosii.om common/common.o ucos/ucos.o port/port.o -nostdlib -lgcc -e 256
mips-sde-elf-objcopy -O binary ucosii.om ucosii.bin
mips-sde-elf-objdump -D ucosii.om > ucosii.asm
python ./merge.py

D:\Repos\computer-system\testos>
```

正确编译出 OS 系统。

名称	修改日期	类型	大小
ucos	2023/6/4 13:49	文件夹	
2050254-OS.bin	2023/6/4 20:49	BIN 文件	31 KB
BinMerge.exe	2023/5/29 14:44	应用程序	12 KB
BootLoader.bin	2023/5/29 14:44	BIN 文件	1 KB
config.mk	2023/6/4 13:40	Makefile 源文件	2 KB
Makefile	2023/6/4 14:04	文件	2 KB
merge.py	2023/6/4 20:48	Python 源文件	1 KB
ram.ld	2023/5/29 14:44	LD 文件	1 KB
report.txt	2023/6/4 12:57	文本文档	1 KB
ucosii.asm	2023/6/4 20:49	ASM 文件	1,014 KB
ucosii.bin	2023/6/4 20:49	BIN 文件	30 KB
ucosii.om	2023/6/4 20:49	OM 文件	192 KB
utils.py	2023/6/4 13:14	Python 源文件	2 KB
说明.txt	2023/5/29 14:44	文本文档	1 KB

3.6 修改 OS 系统

首先修改 BootLoader 的 UART 频率，将频率设置到我们开发板的 100Mhz，其次在 OS 系统的 openmips.h 文件下修改 IN_CLK 为 100Mhz：

```
μCOSii操作系统工程 > include > C openmips.h > IN_CLK
U
  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 100000000            /* 输入时钟是100MHz */
  6
  7  /*****************************************************************
  8
  9                          串口相关参数、函数
 10
 11  *****************************************************************/
 12
 13  #define UART_BAUD_RATE  9600          /* 串口速率是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) */
 21
 22  /* Line Status Register的标志位 */
 23  #define UART_LS_TEMT    0x40    /* Transmitter empty */
 24  #define UART_LS_THRE    0x20    /* Transmit-hold-register empty */
 25
 26  /* Line Control Register的标志位 */
 27  #define UART_LC_NO_PARITY   0x00    /* Parity Disable */
 28  #define UART_LC_ONE_STOP    0x00    /* Stop bits: 0x00表示one stop bit */
 29  #define UART_LC_WLEN8  0x03 /* Wordlength: 8 bits */
 30
 31  extern void uart_init(void);
 32  extern void uart_putc(char);
 33  extern void uart_print_str(char*);
 34  extern void uart_print_int(unsigned int);
 35
 36  /*****************************************************************
```

再在 openmips.c 中添加上我自己的信息：

```
124  }
125
126  void  TaskStart (void *pdata)
127  {
128      INT32U count = 0;
129      pdata = pdata;            /* Prevent compiler warning                */
130      OSInitTick();             /* don't put this function in main()       */
131      uart_print_str(
132         "2050254 Wangjuntao Operation Systen Exam 2 Copyright 2023.";
133      );
134      for (;;) {
135         if(count <= 102)
136          {
137           uart_putc(Info[count]);
138           uart_putc(Info[count+1]);
139          }
140          gpio_out(count);
141          count=count+2;
142          OSTimeDly(10);  /* Wait 100ms                  */
143      }
144
145  }
146
```

4、实验结果

先介绍本次试验采用的硬软件环境：

开发环境 Vivado 2022.2

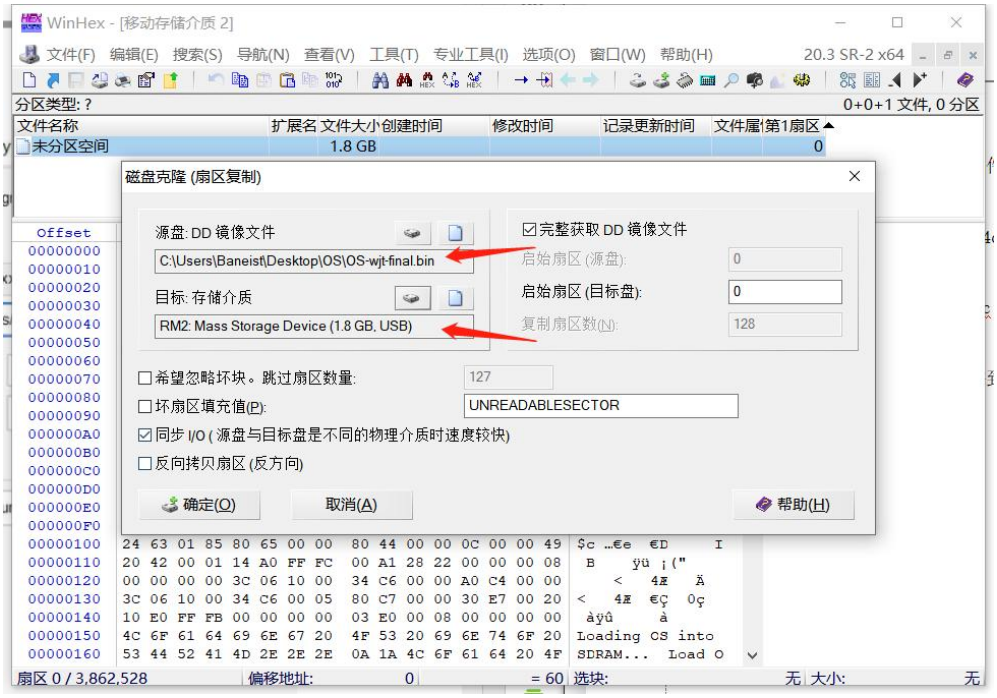
仿真环境 ModelSim PE 10.4c

测试环境 sscom5

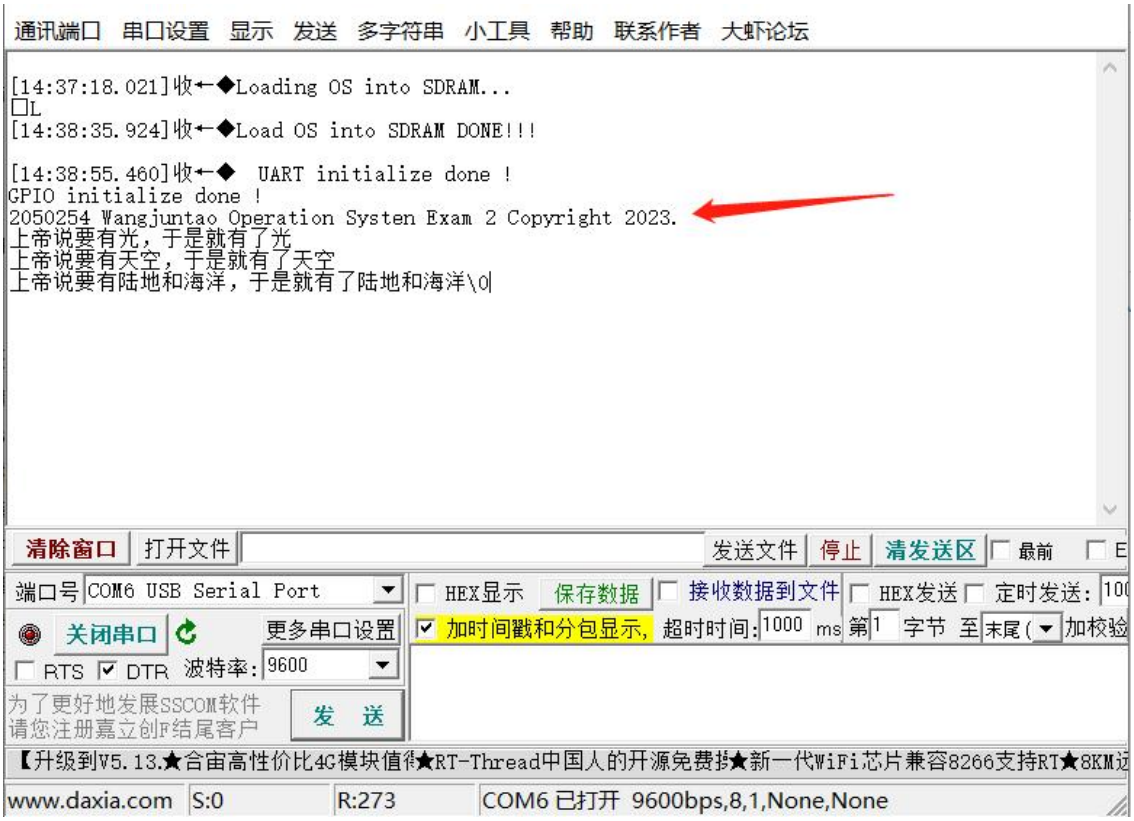
编译环境 mips-sde-elf-gcc （2013 年 5 月 Windows 发布版）。

使用 Winhex 将 OS 系统状态到 SD 卡中：

选择源为镜像文件 OS-wjt-final.bin，目标选择存储介质，也即 SD 卡



生成比特流文件到 SD 卡中，打开串口调试器，可以看系统输出正确。Bootloader 的加载正确，UART 和 GPIO 都初始化完毕，并且我的输出语句和作者给的中文输出语句均被成功执行：



发送完毕后操作系统进入等待状态，所有进程均结束。

这证明了本次试验的成功。

5、实验总结

本次试验是硬件课程有史以来以来难度最大的一次实验，我在前面几个 CPU 的设计实验中做的一帆风顺，也就以为本次实验难度和之前都差不太多，然而现实给了我巨大的打击。现在回顾前面实现的 CPU，我脑海里冒出的第一个词就是“TOY”。没有严格时序的收敛，没有进行高强度、全面的抗鲁棒性测试、没有进行流水线不同情况下的暂停启动测试、没有进行突发的指令/数据存储器故障测试等等等等，很多看似繁杂并因而省略的步骤背后就决定了这个 CPU 是否“可堪大任”。实验 1 编写的 89 条 MIPS CPU，虽然能够通过助教给的测试程序的所有测试点，成功完成验收，但是这个 CPU 其实有很多很多的弱点，就比如前面提到的 stall 指令连续触发的故障，以及高频下偶发的时序问题。在本次实验中，为了克服这些弱点，浪费了几周的大量时间在故障定位和故障原因分析上，最终也不能完全解决，只能使用折中妥协的降频

方案，让延迟升高到能接受这些时序问题为止。即使到快写完报告的现在，我依然不知道这些高频运行下偶发的时序问题是如何产生的，只能勉强给出一个大致的定位。不能以 100Mhz 的速度全功率运作就像是买了块 i9 到手一看却是一块 i3，也算是某种同类的遗憾吧（笑）。

相较而言几个协议控制器反而没有这么复杂了，主要是这些代码既有丰富的材料可以学习，又有成熟的稳定源码可以使用和查阅，并且 SD 卡、UART、GPIO 这些协议其实在数字逻辑中我就已经使用到并写出了很稳定的版本，这次只是加以封装使其适合 WB 总线罢了。主要的问题在于 DDR2 SDRAM 的控制器，他的运行频率太高了（300Mhz），并且其异步逻辑行为的波动很大，无法像 SD 卡这样写死一个固定的 timeout 数值，就只能在封装状态机的基础上再套一个读写控制小状态机完成动态的波动延迟下的读写逻辑，其开发就相对比较麻烦,最终勉强在 50Mhz 下做到了时序收敛（也不是很稳定），不过随着 CPU 的降频，这些更高频的时序问题也就无关紧要了，有点木桶效应的感觉。

操作系统的交叉编译之所以迁移到 Windows 上也是出于偷懒的缘故吧，不想再配一个 ubuntu 环境了，但是现在看看其实配一个环境所耗费的时间其实要比在 windows 上配环境少很多吧，Windows 上的交叉编译有很多很多的问题，大小端、库函数版本错误、32 位 64 位问题，架构不支持等等，最后在查阅了多方资料下才勉强找到了一个能完美解决问题的交叉编译器版本，稍微新一点或者稍微老一点都不可以。之后还修改 MAKEFILE 使其能在 Windows 上运行，还要修改 binMerge，真的非常浪费时间，但是配出来的时候真的成就感满满啊！！

相信老师也注意到了，前面在设计 wishbone 交叉互联总线的时候已经分配了 vga 和 ps2 的设备空间，之后的实验 3 也希望能够进一步实现这两个设备驱动器，将其接到 wishbone 总线上，实现一个真正意义上的，有现代 GUI 界面的微型电脑！