

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 |

# 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

# 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

# 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：

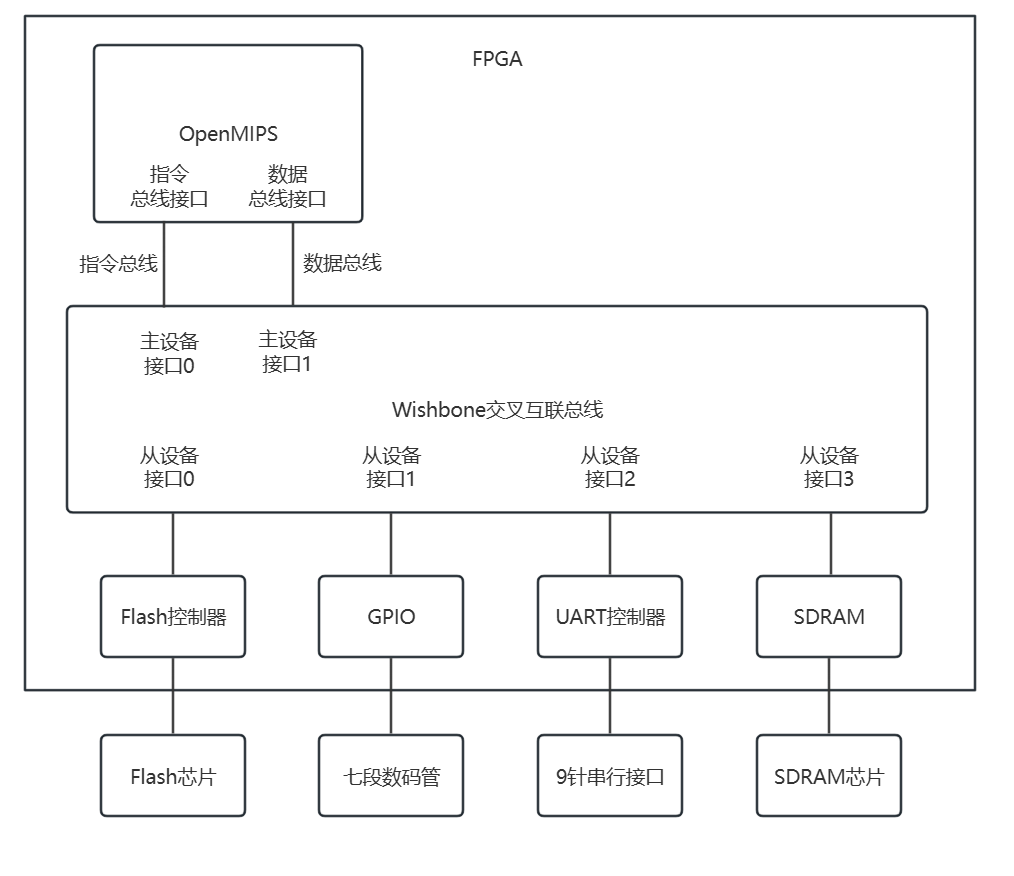
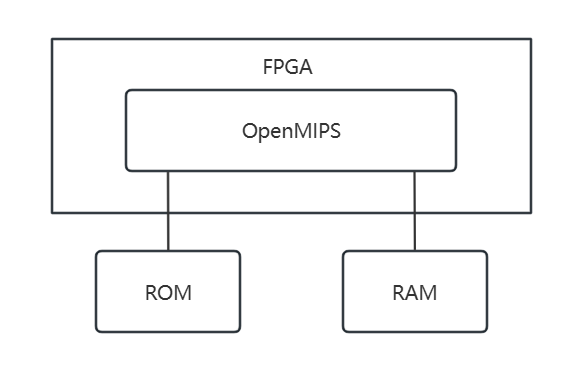


图1：实验1与实验2的系统总结构框架图

## (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。

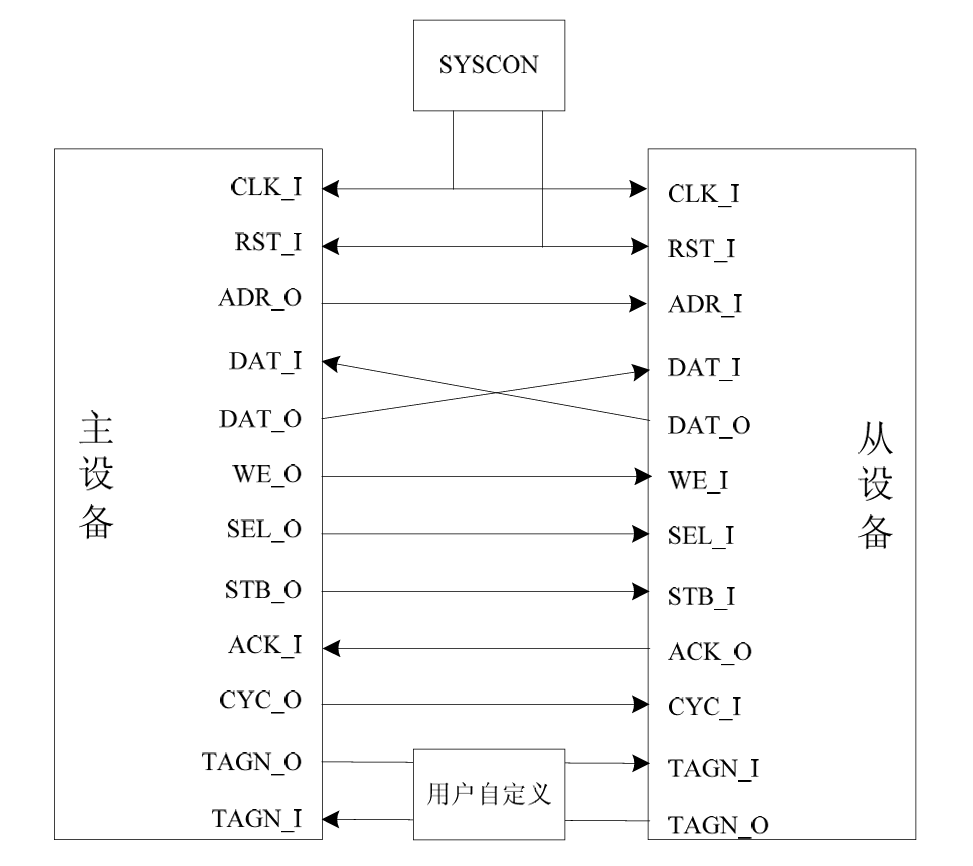


图2：Wishbone 总线规范点对点连接方式

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**

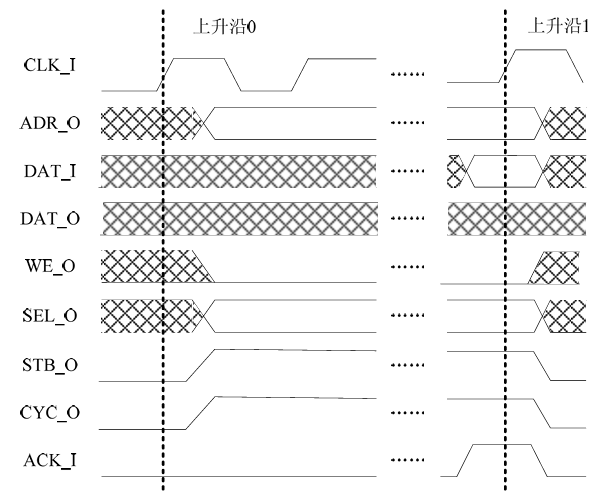
****

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。

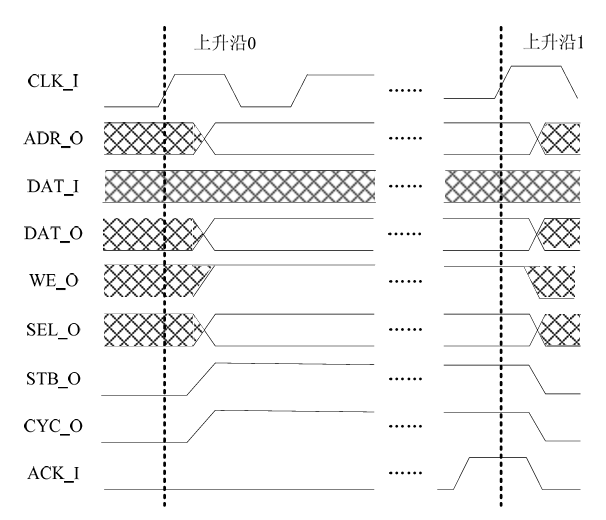
****

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。

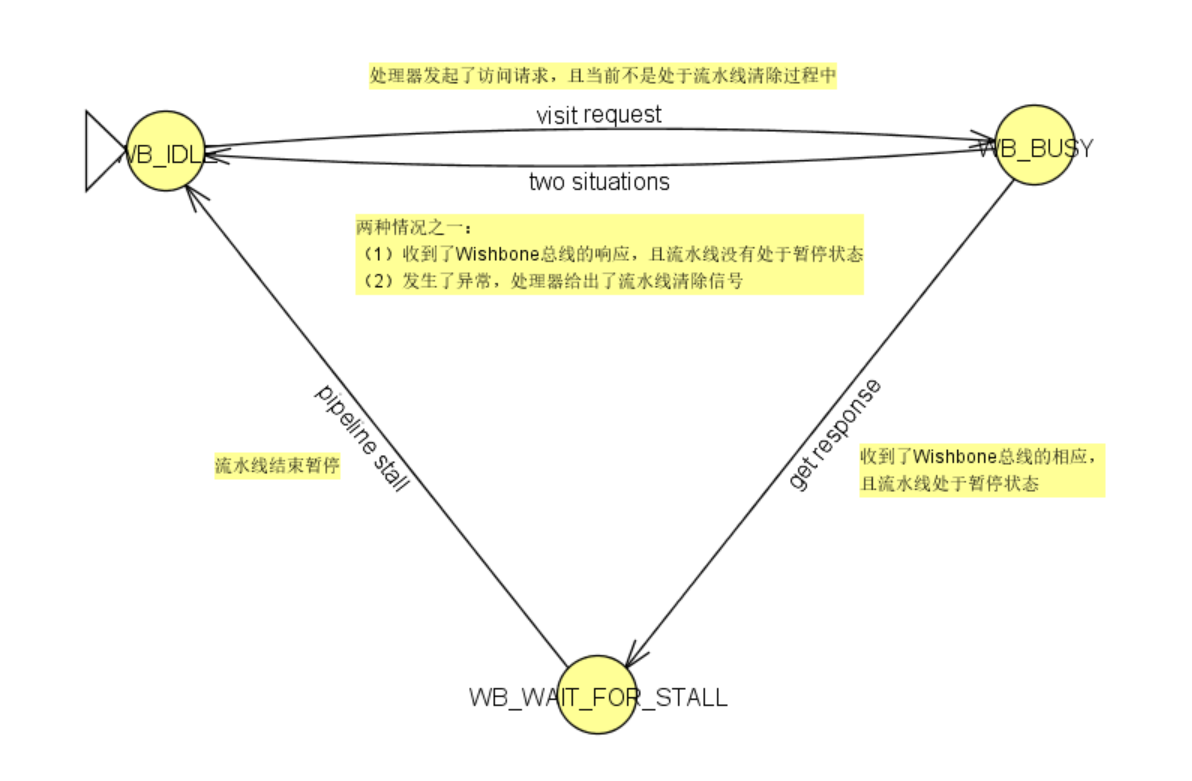


图5：Wishbone 总线接口模块的状态机

**·模块设计**

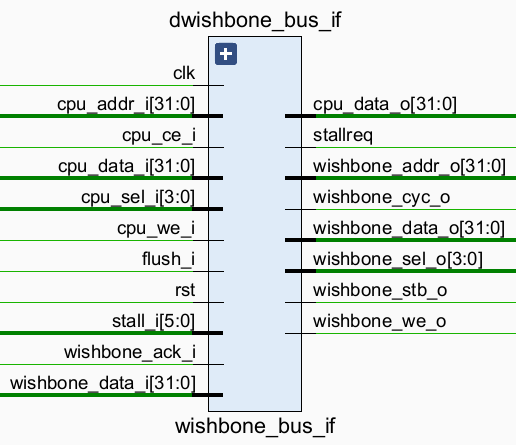
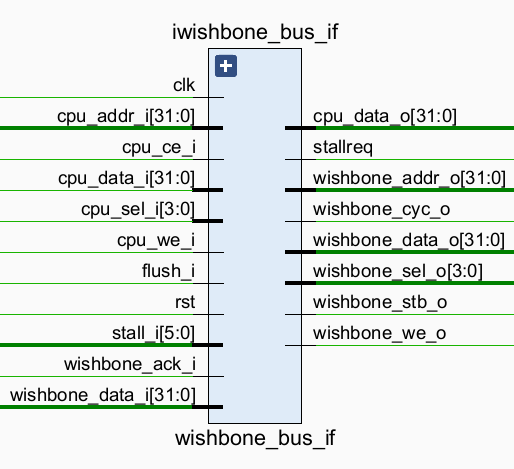


图6：Wishbone 总线接口模块设计图

**·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

    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**

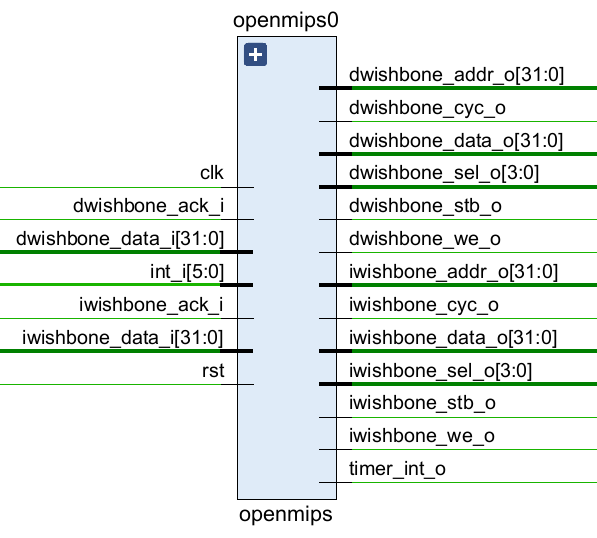


图7：修改后openmips模块设计图

**· 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**

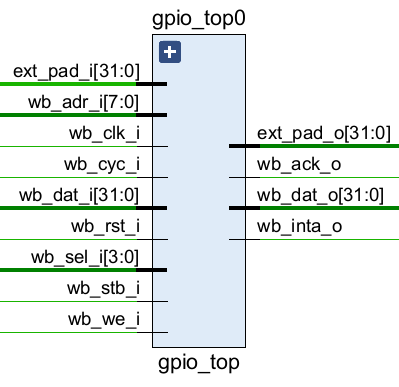


图8：GPIO模块设计图

**·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**

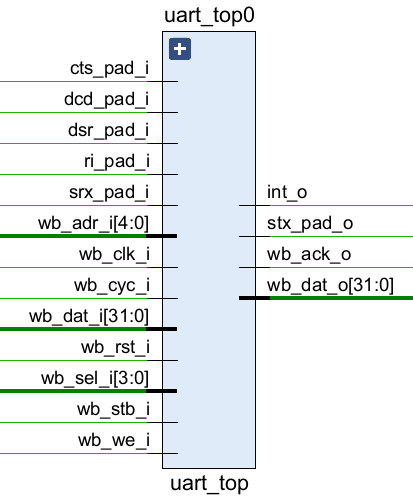


图9：UART模块设计图

**·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**

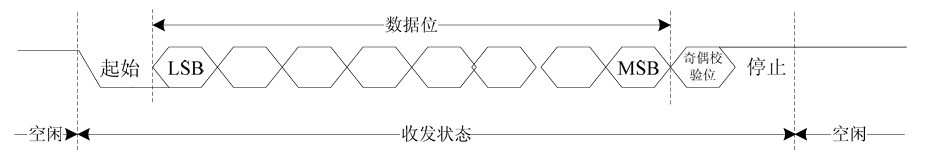


图10：UART 的数据传输过程

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**

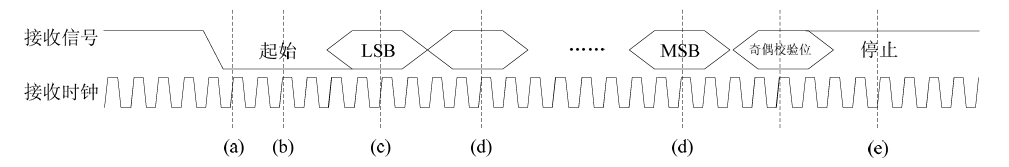


图11：UART 的数据接收过程

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。

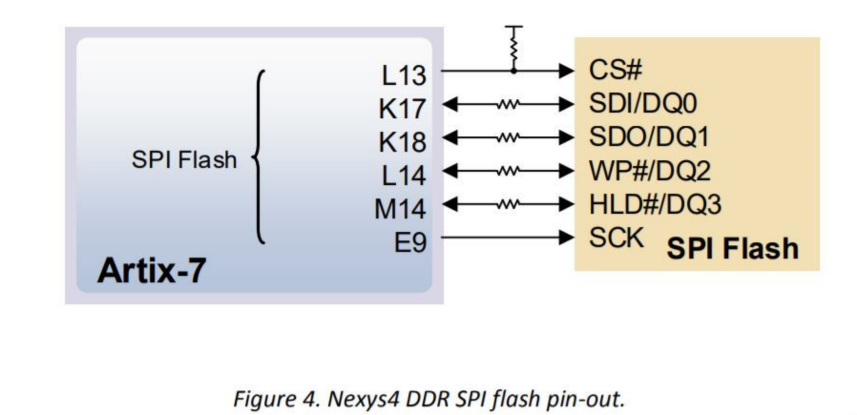


图12：S25FL128S芯片对应接口图

**·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**

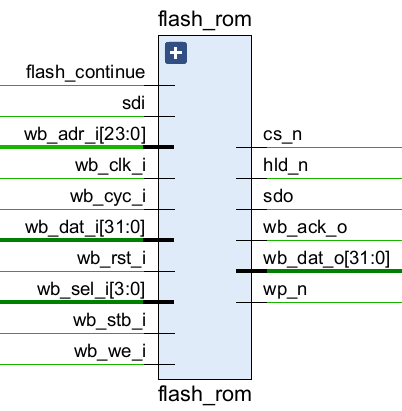


图13：Flash模块设计图

**·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

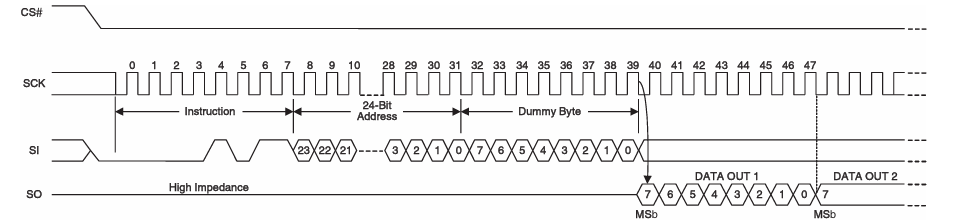


图14：Flash快速读取时序图

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。

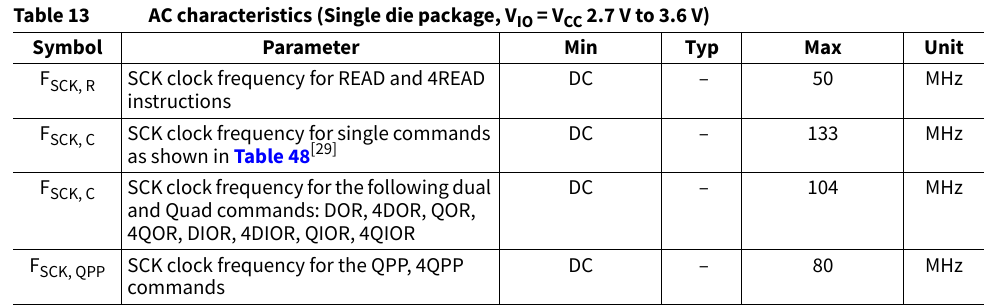


图15：S25FL128S官方文档对SCK频率的限制

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：

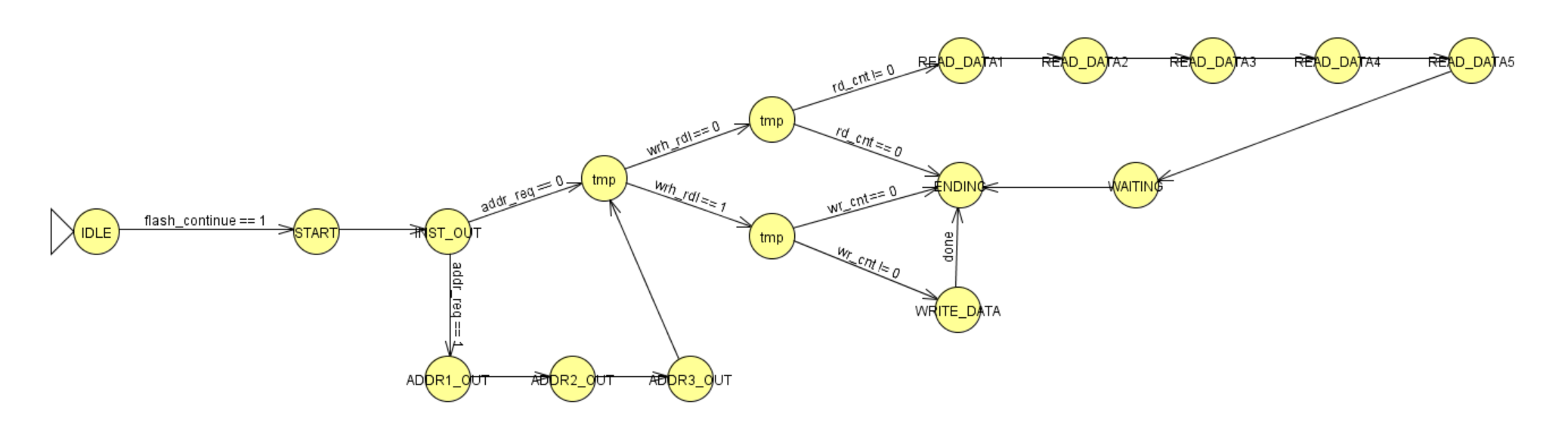


图16：Flash模块的状态机

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。

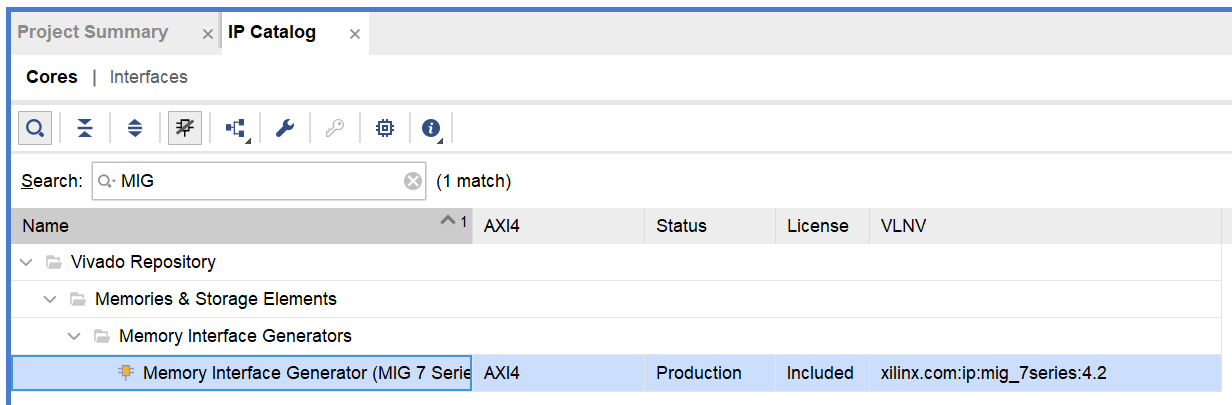


图17：添加IP核步骤1

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。

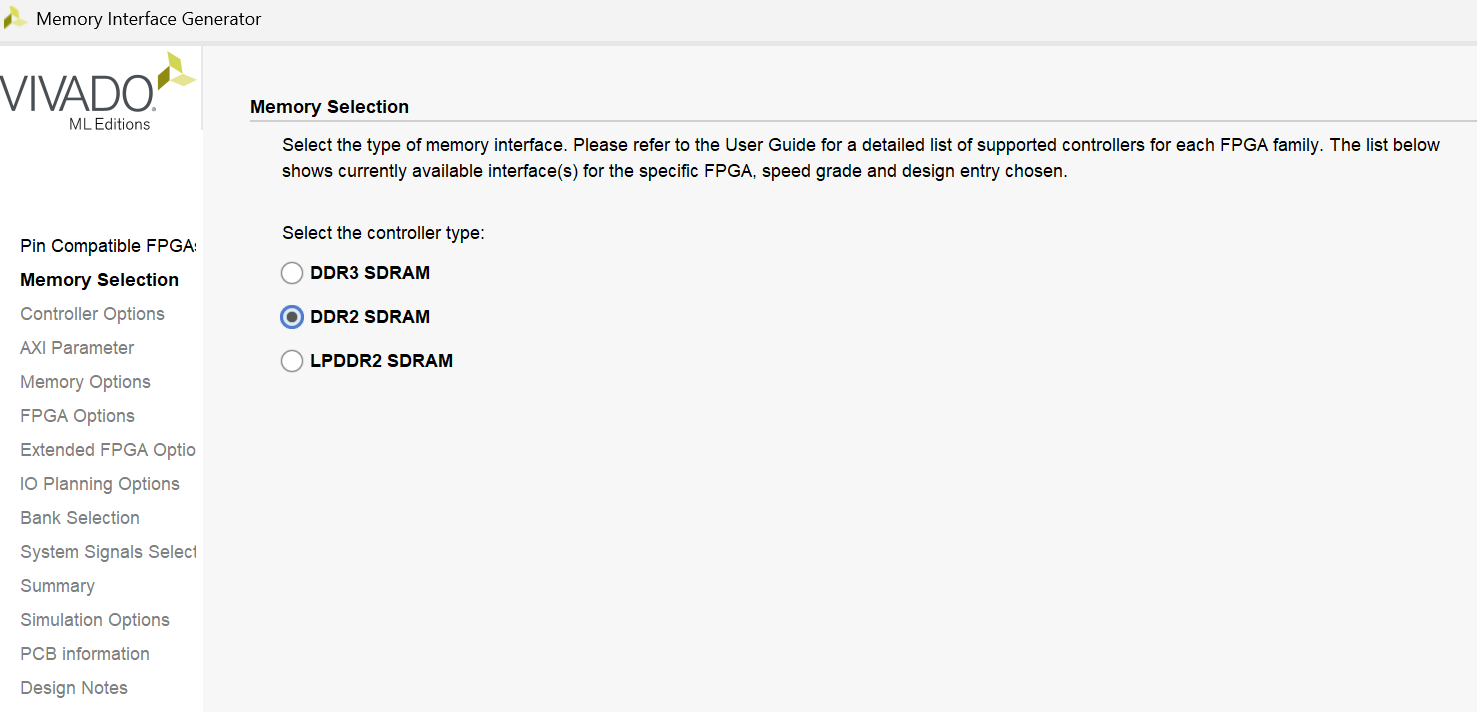


图18：添加IP核步骤2

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

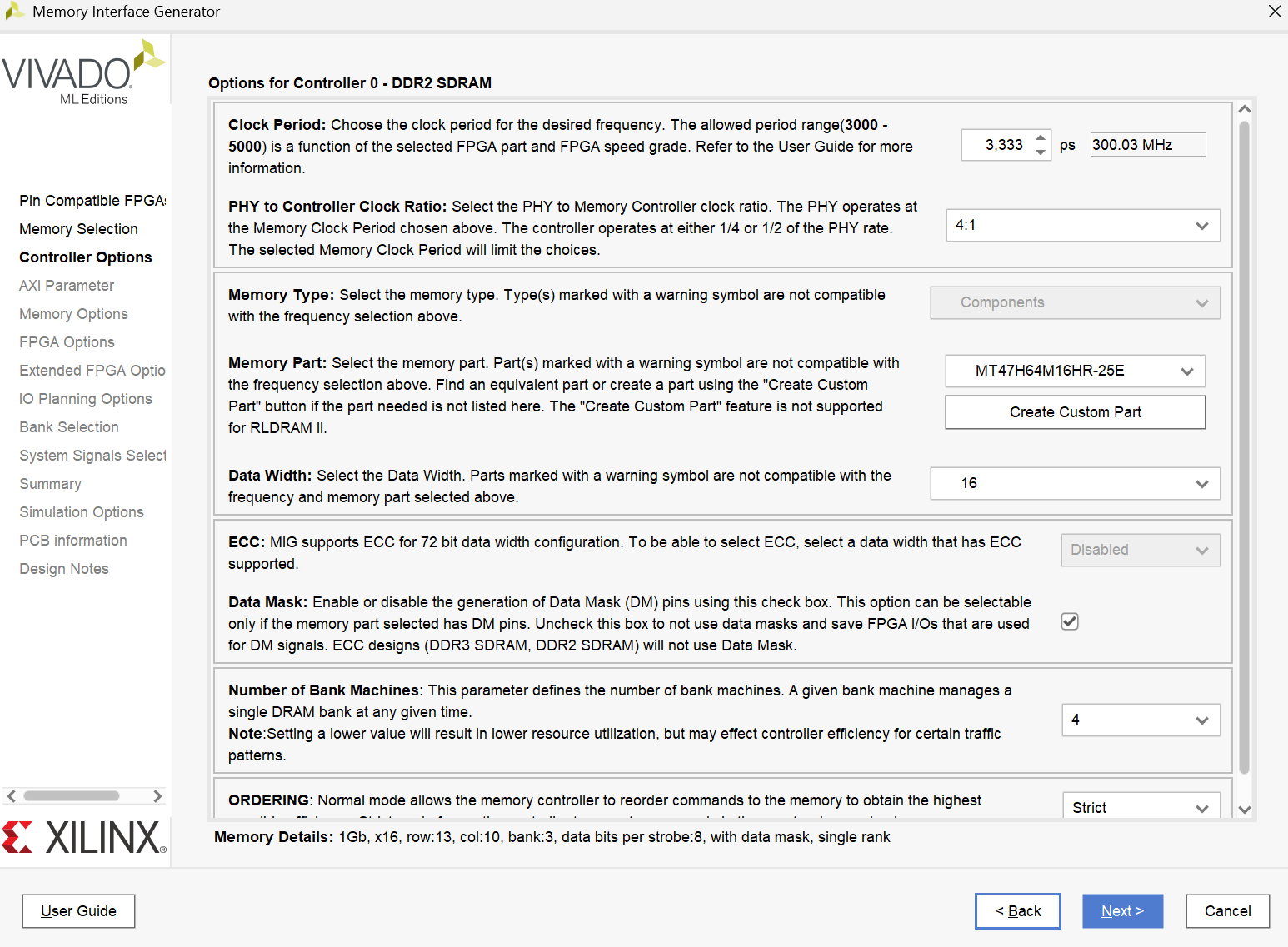


图19：添加IP核步骤3

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

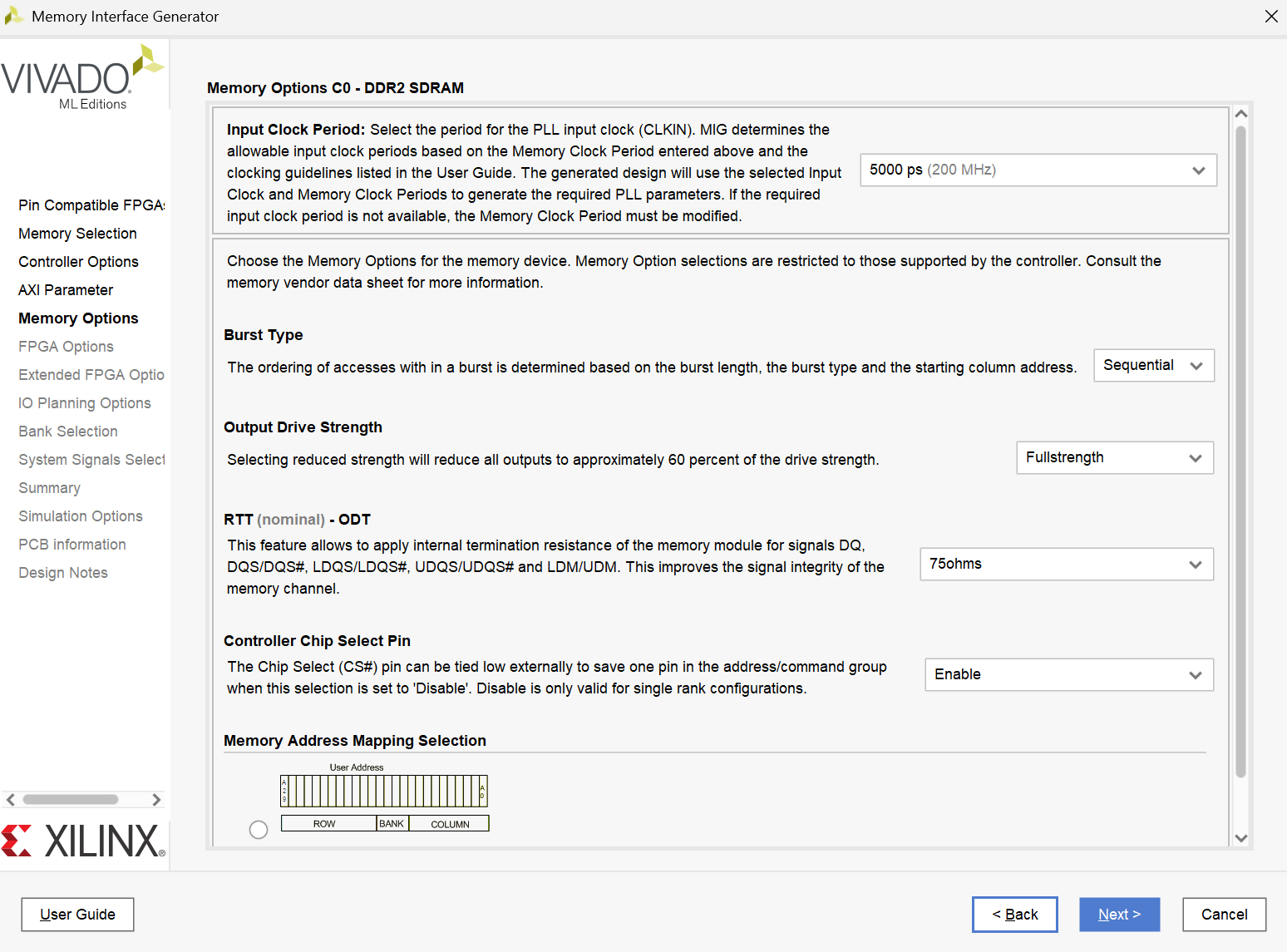
****

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。

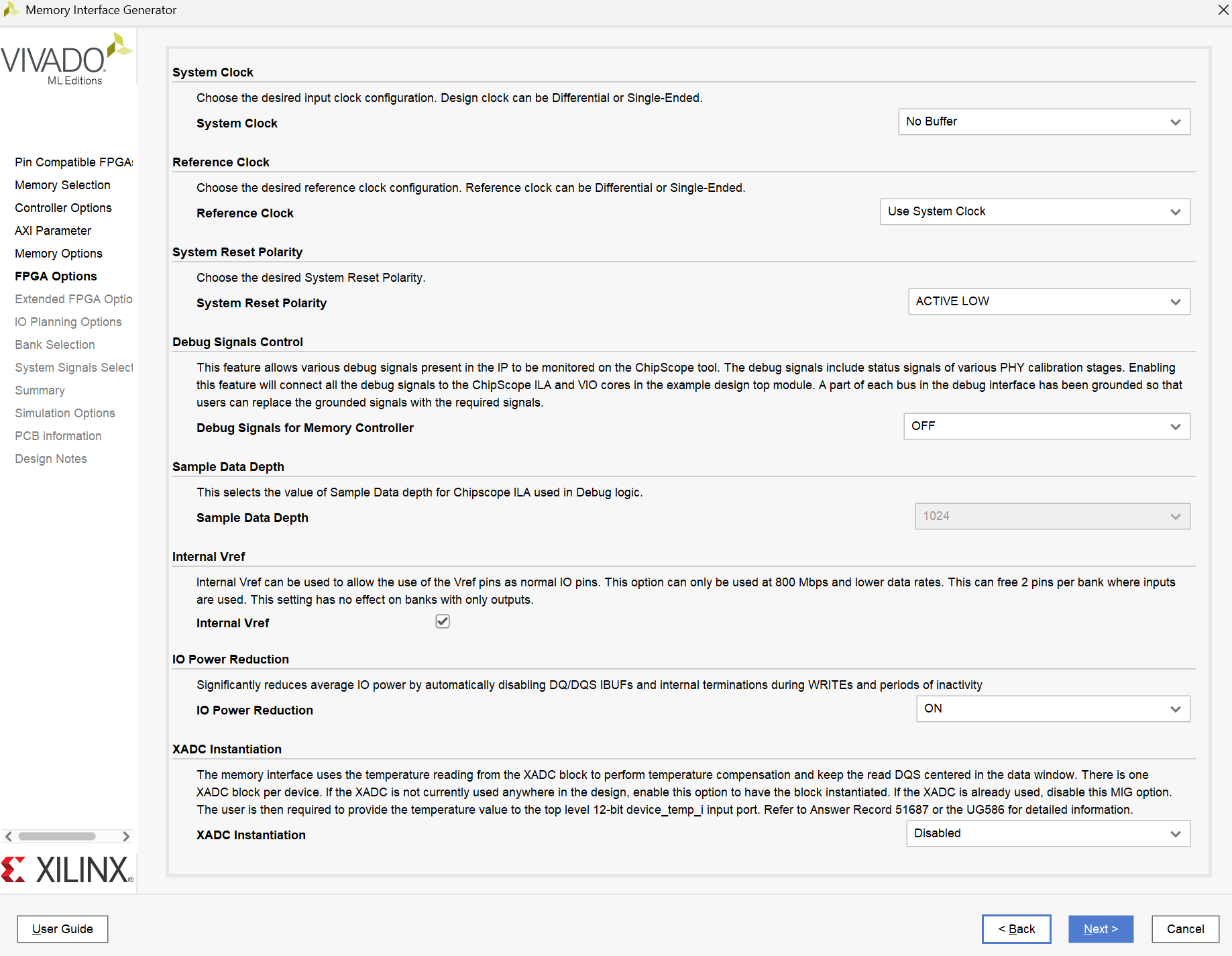


图21：添加IP核步骤5

在IO Planning Options中选择Fixed Pin Out。

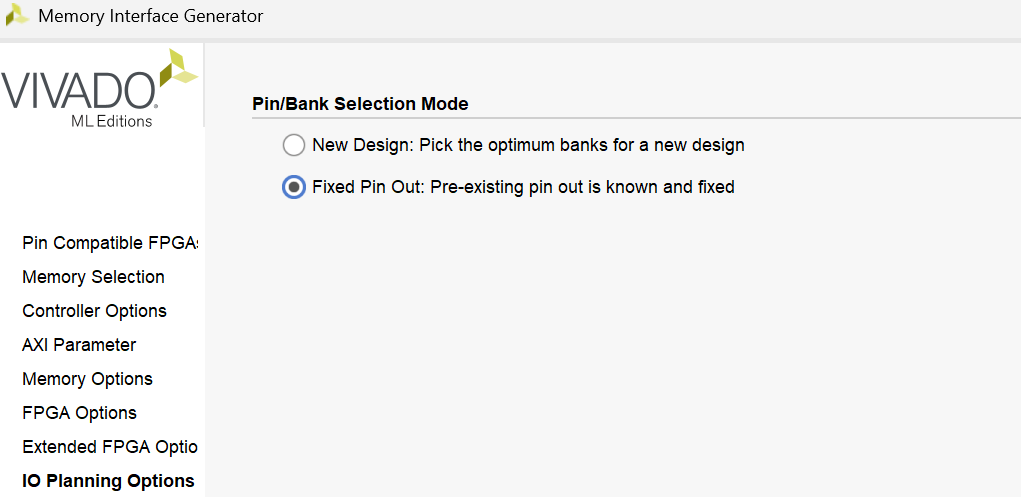


图22：添加IP核步骤6

在Pin Seletion中，点击Read XDC/UCF，使用[Nexys4DDR\_DMA\_controller/PROJECT/Nexys4DDRmemorypinout.ucf at master · HackLinux/Nexys4DDR\_DMA\_controller](https://github.com/HackLinux/Nexys4DDR_DMA_controller/blob/master/PROJECT/Nexys4DDRmemorypinout.ucf)中下载的文件，导入后点击validate。后续一路Next即可。



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**

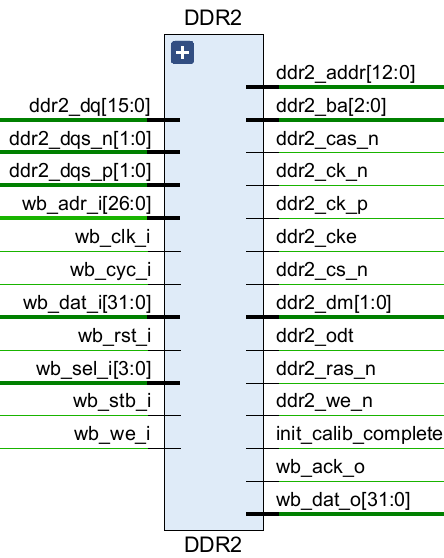


图24：DDR2模块设计图

·**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

    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：

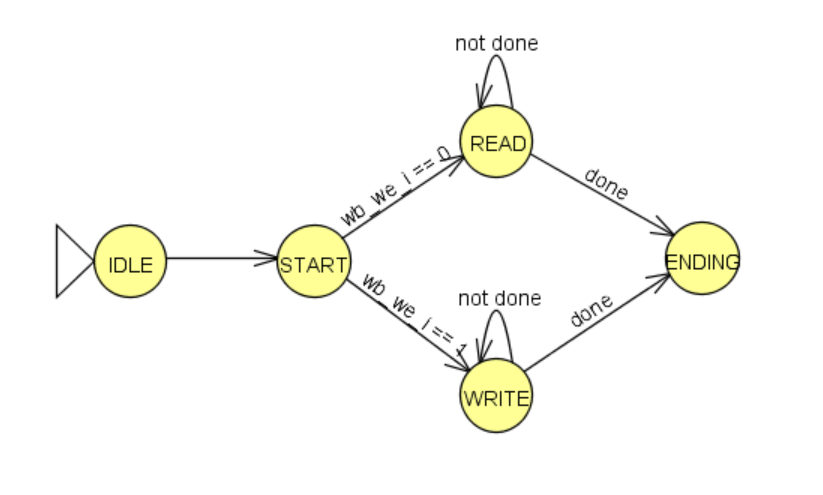


图25：DDR2模块状态机图

·**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。

## (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**

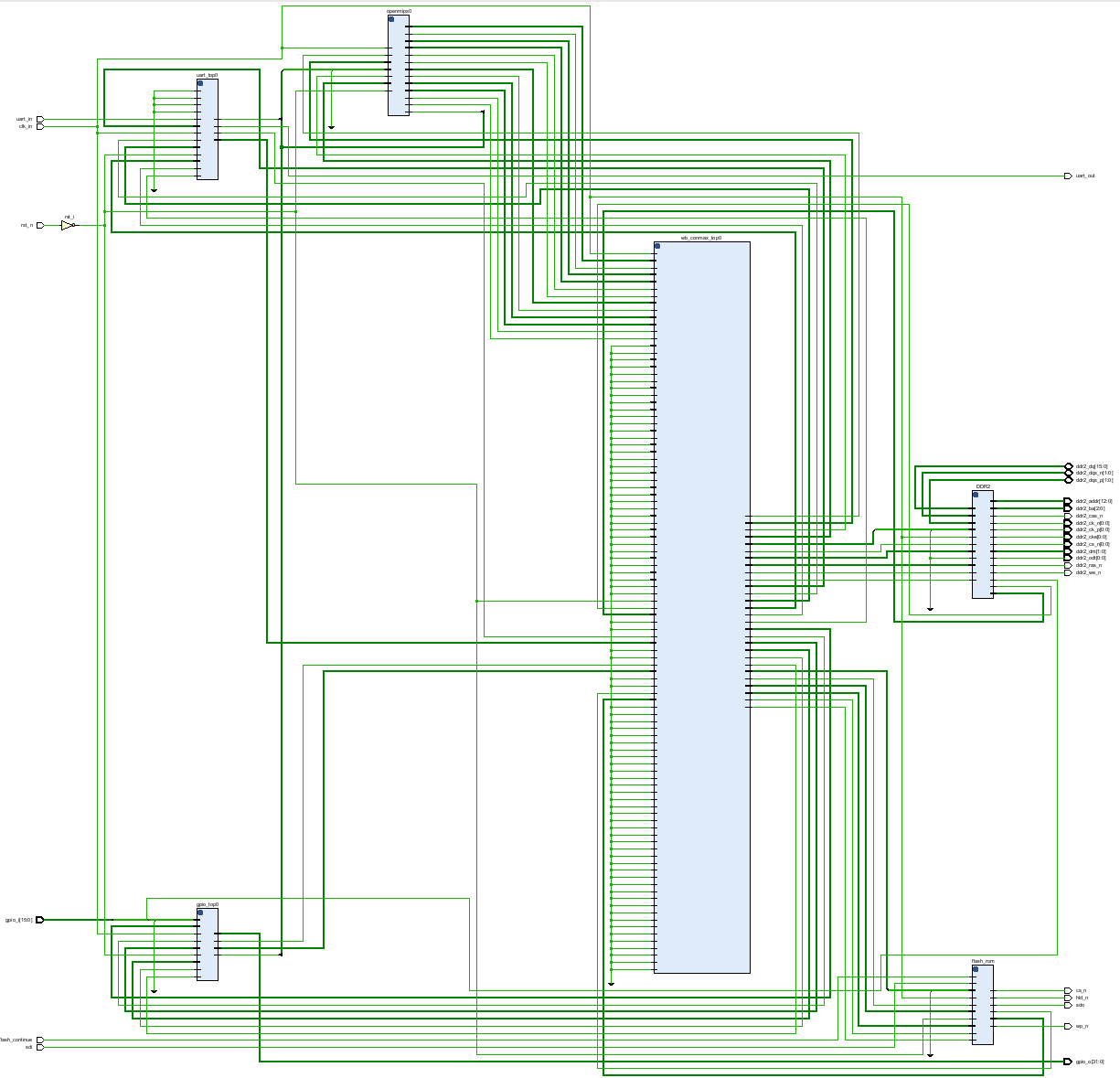


图26：openmips\_min\_sopc模块设计图

**·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](https://blog.csdn.net/wangtcCSDN/article/details/137950545)，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。

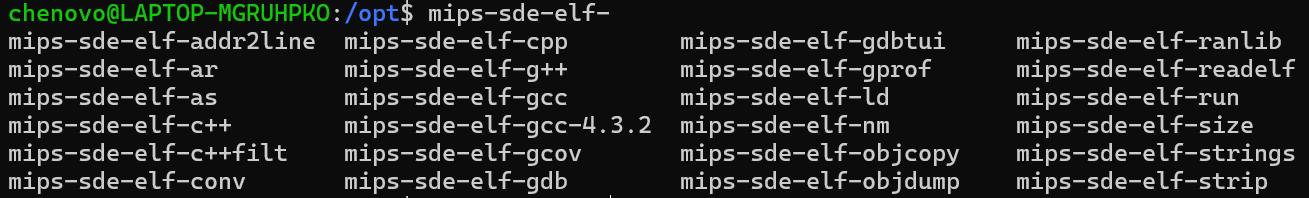


图27：GNU工具链安装成功示意图

## (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 （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。

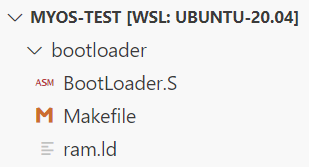


图28：BootLoader目录结构

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

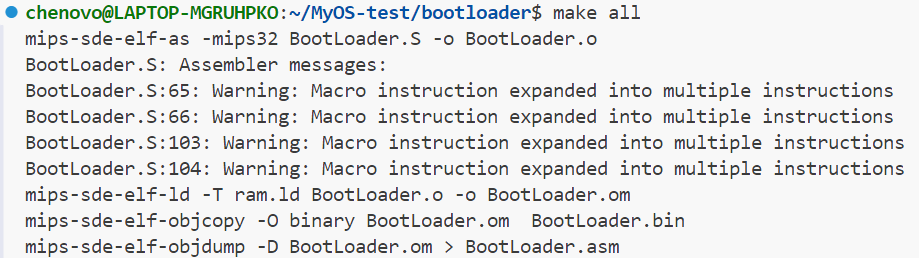


图29：BootLoader的编译

## (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：

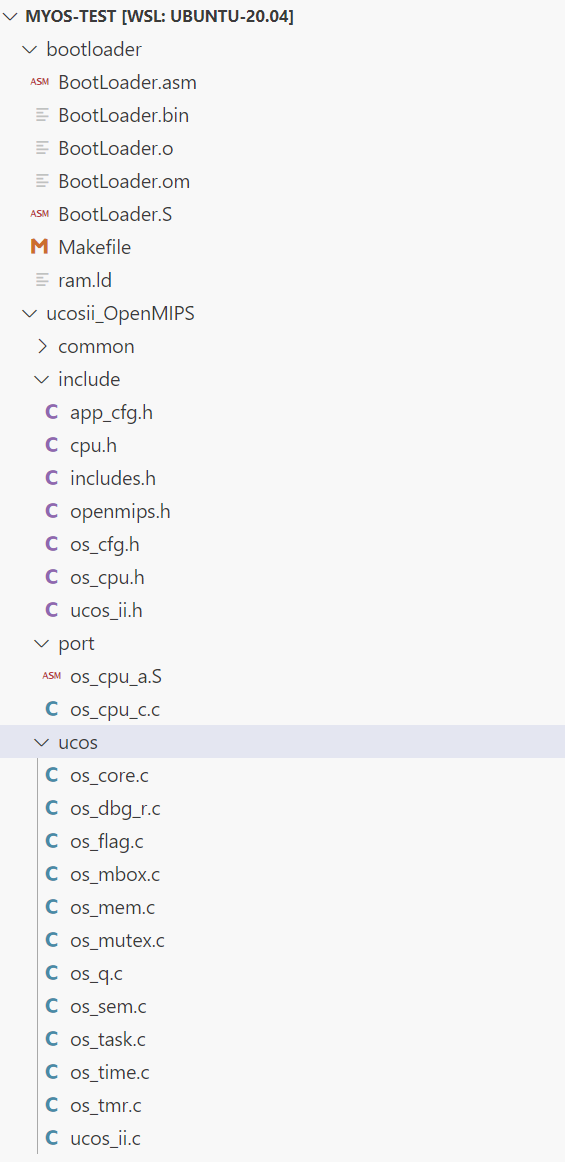
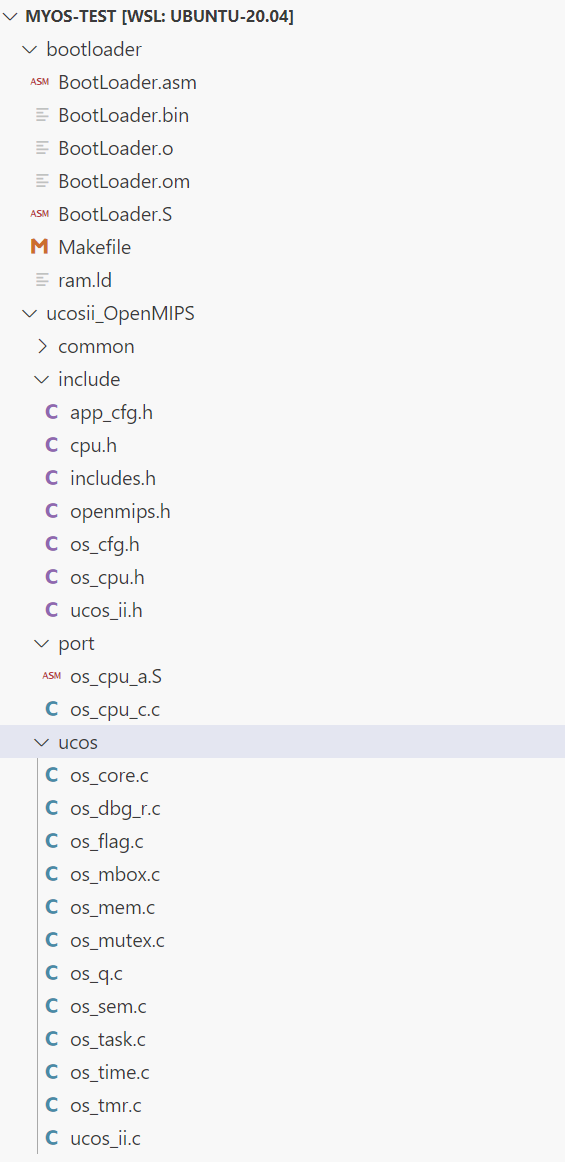


图30：µC/OS-II的文件结构

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：

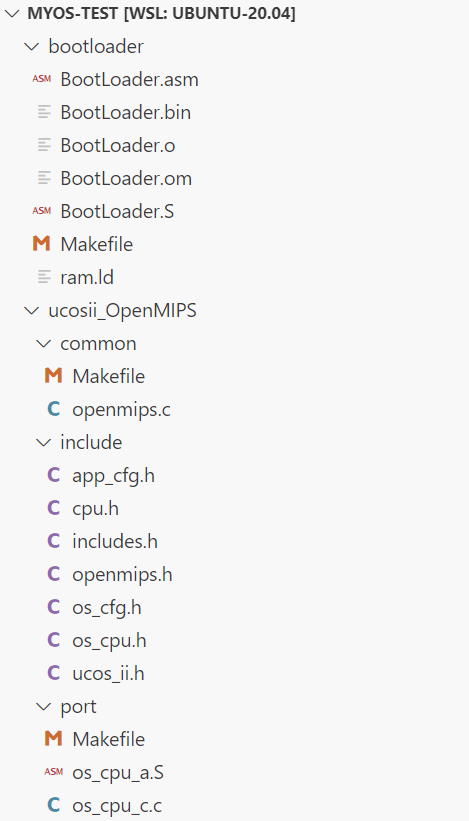


图31：整个项目的文件架构

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。

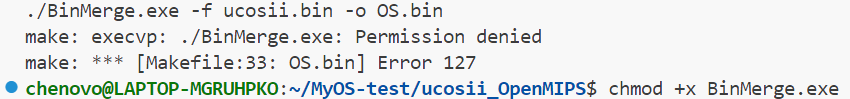


图32：更改BinMerge.exe的权限

最终再次执行make all指令，得到OS.bin文件。

# 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]

## (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。

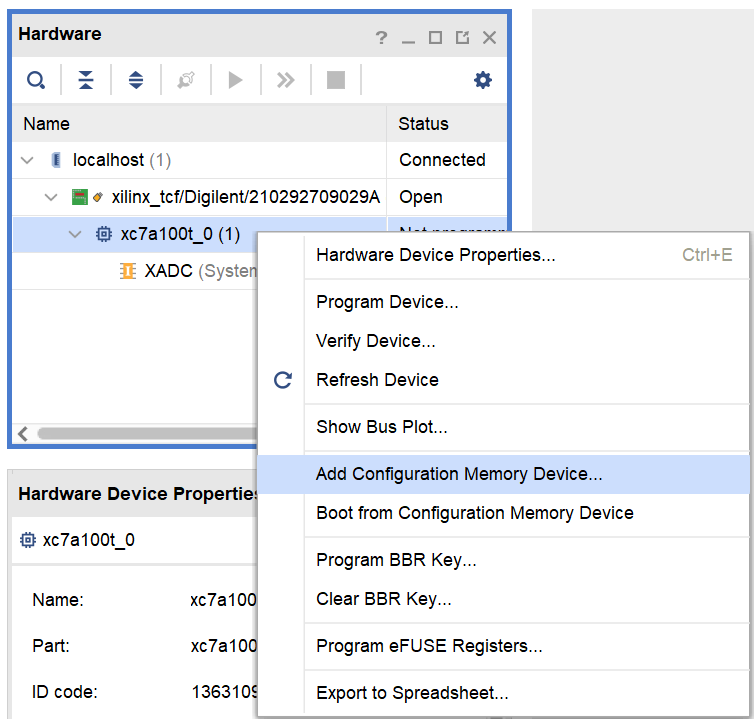


图33：配置SPI-Flash

选中s25fl128sxxxxxx0-spi-x1\_x2\_x4。

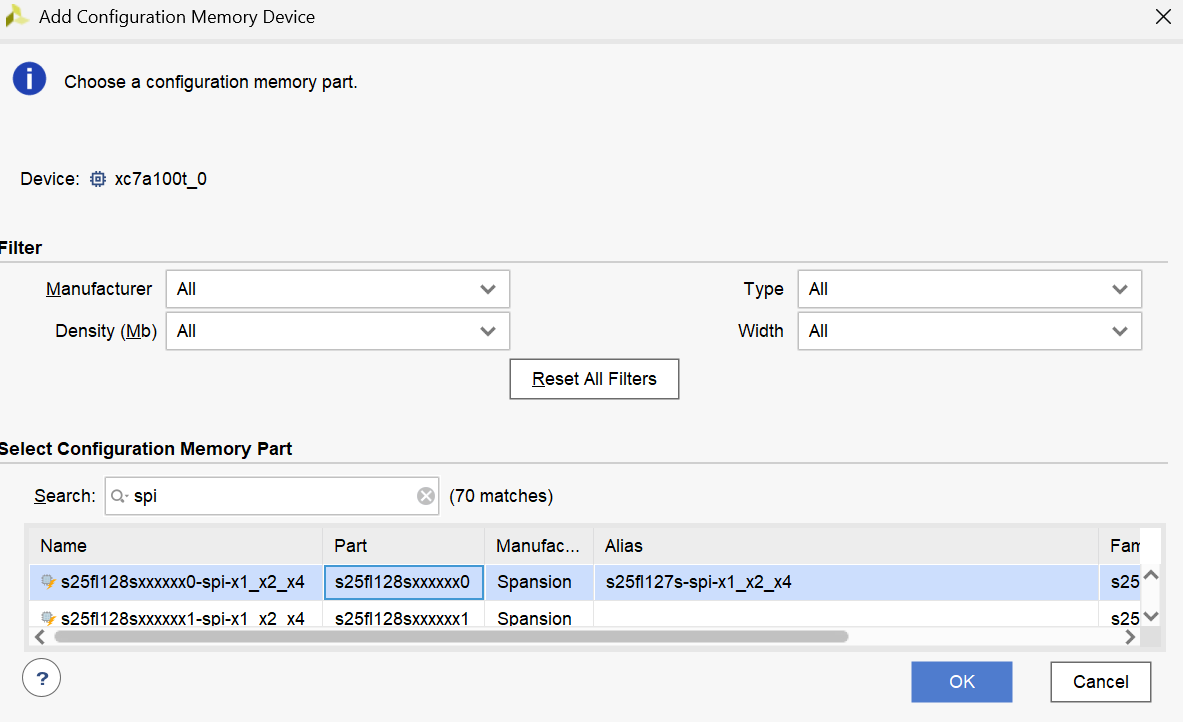


图34：配置SPI-Flash

选择编译得到的bin二进制文件下板。

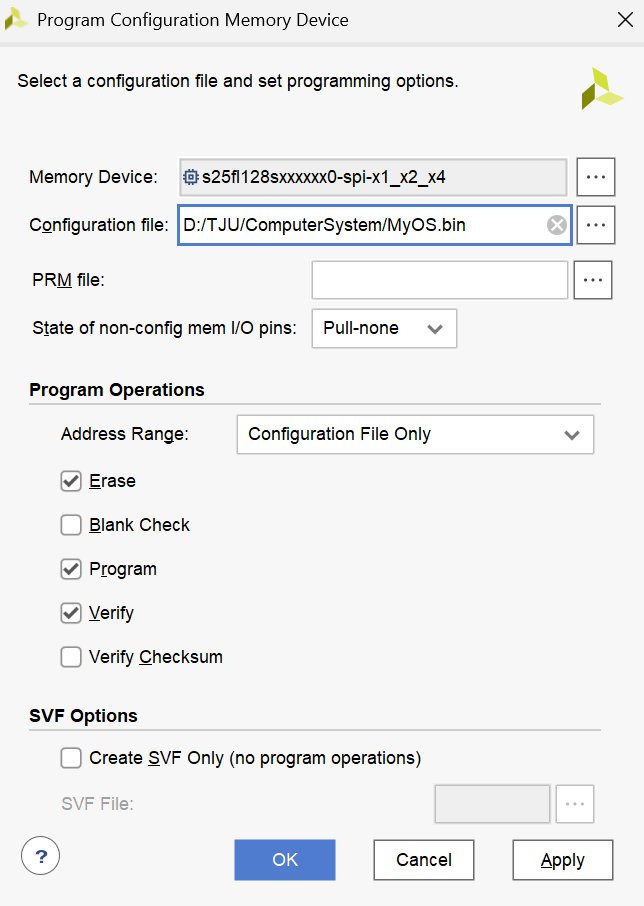


图35：配置SPI-Flash

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。

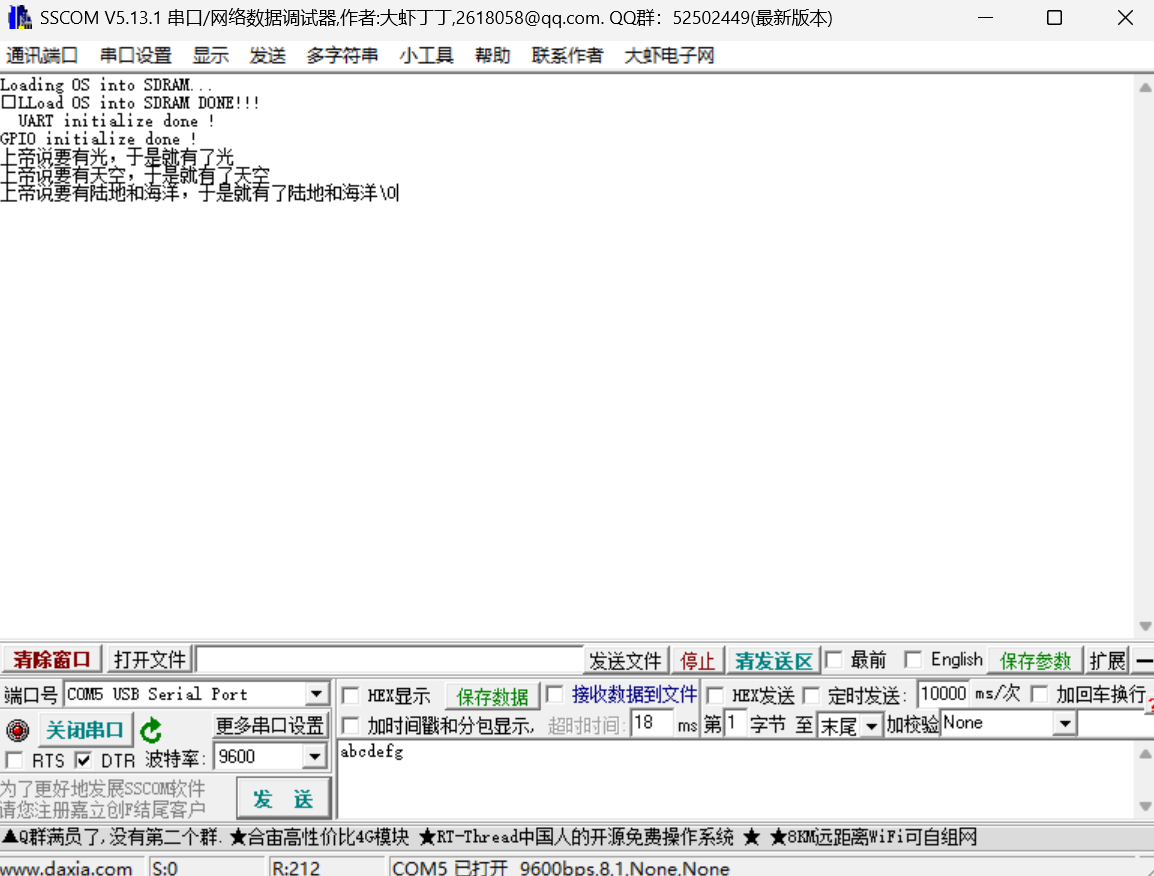




图36：串口通讯结果

## (3) 下板结果

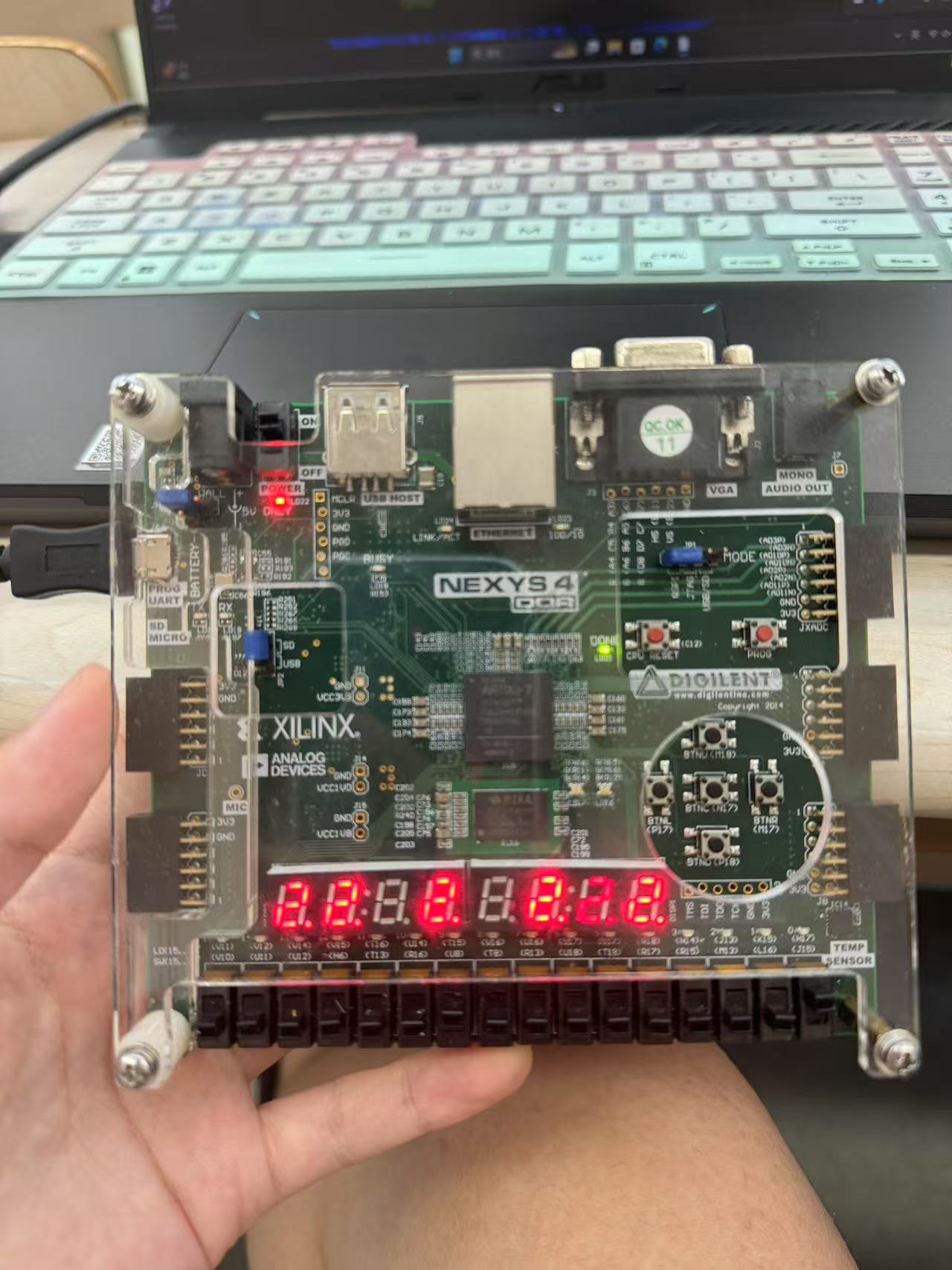




图37：下板结果演示

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。

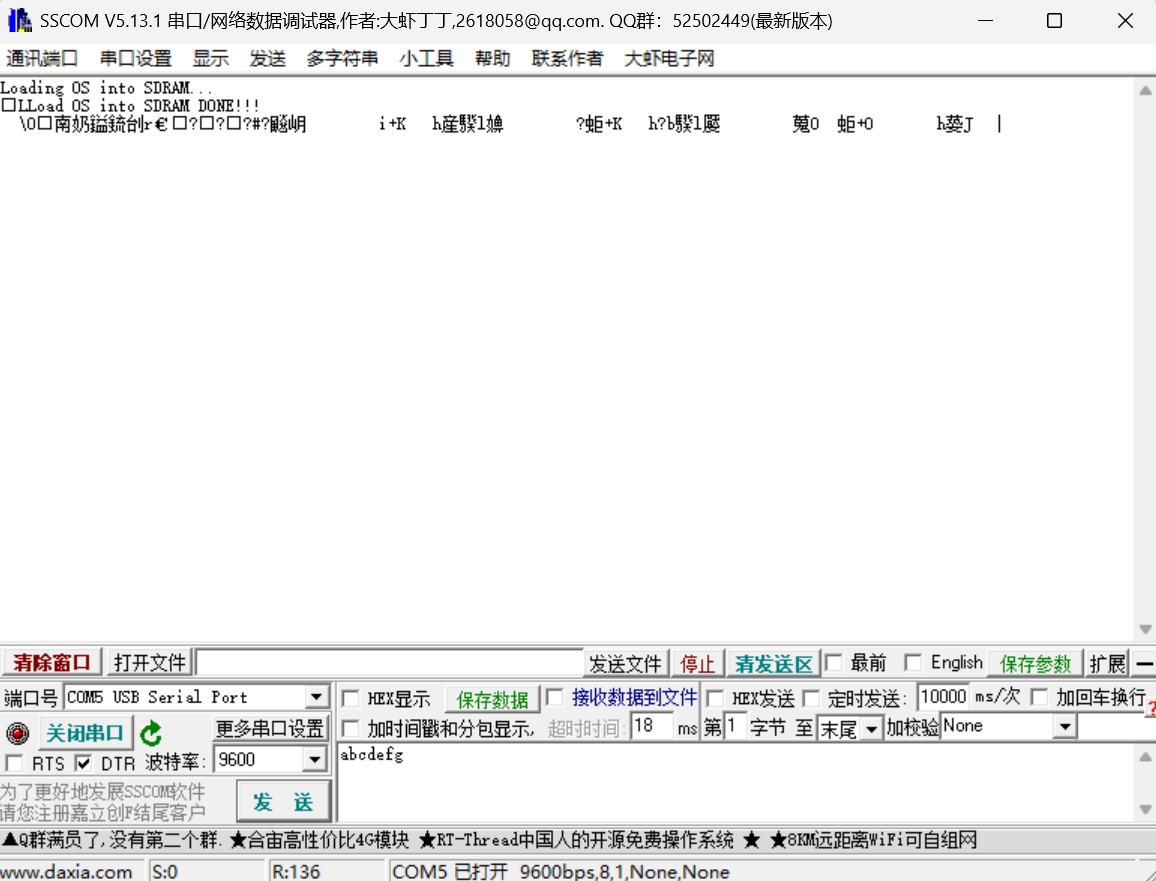


图38：乱码问题复现

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。

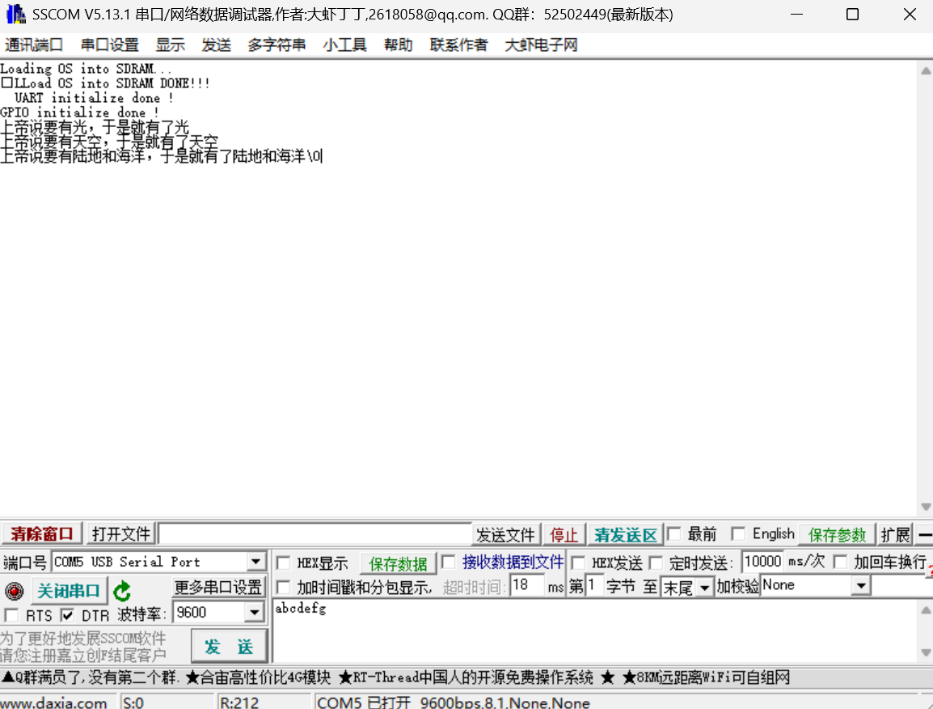
3. 串口通讯时，操作系统可以正常启动，但只能收到“上”一个字

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~

# 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](https://blog.csdn.net/qimoDIY/article/details/88986354)

[5] Flash整体设计参考：[SPI FLASH 读取模块[verilog]](https://yufeiran.com/spi-flash-du-qu-mo-kuai/)

[6] Ram2Ddr.vhd库文件说明：[SRAM to DDR Component - Digilent Reference](https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-4-ddr-sram-to-ddr-component/start)

[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](https://blog.csdn.net/qimoDIY/article/details/86770745)

[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](https://www.infineon.com/dgdl/Infineon-S25FL128S_S25FL256S_128_Mb_(16_MB)_256_Mb_(32_MB)_3.0V_SPI_Flash_Memory-DataSheet-v18_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0ecfb6a64a17&utm_source=cypress&utm_medium=referral&utm_campaign=202110_globe_en_all_integration-files&redirId=file_4_1_282)

[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](https://docs.amd.com/v/u/1.9-English/ug586_7Series_MIS)