**计算机系统能力培养实践课程系列**

**《操作系统》**

**课程设计指导手册**

**——基于Minisys平台**

**版本 12.0**

**杨全胜 谢家骏 徐逸凡 余航 编写**

**j0234657**

**东南大学计算机科学与工程学院、软件学院**

**东南大学计算机系统能力培养课题组**

2022年10月

# 目 录

[目 录 I](#_Toc115734825)

[前言 1](#_Toc115734826)

[第一章 开发之前要掌握的知识 2](#_Toc115734827)

[1.1 MIPSfpga概述 2](#_Toc115734828)

[1.1.1 MIPSfpga核心 2](#_Toc115734829)

[1.1.2 MIPSfpga系统 3](#_Toc115734830)

[1.1.3 MIPSfpga寄存器概述 4](#_Toc115734831)

[1.2 Minisys实验平台简介 8](#_Toc115734832)

[1.2.1 Minisys计算资源 9](#_Toc115734833)

[1.2.2 Minisys存储资源 10](#_Toc115734834)

[1.2.3 Minisys I/O外设资源 11](#_Toc115734835)

[1.2.4 Bus Blaster 11](#_Toc115734836)

[1.3.3中断以及异常 11](#_Toc115734837)

[第二章 MIPSfpga-SoC的移植 13](#_Toc115734838)

[2.1工具链的搭建 13](#_Toc115734839)

[2.1.1 OpenOCD的安装 13](#_Toc115734840)

[2.1.2 MIPS工具链的安装 13](#_Toc115734841)

[2.1.3 PuTTY以及驱动程序的安装 14](#_Toc115734842)

[2.2 MIPSfpga-SoC工程的创建 14](#_Toc115734843)

[2.3 MIPSfpga-SoC的搭建 16](#_Toc115734844)

[2.3.1MIPSfpga Core的导入 16](#_Toc115734845)

[2.3.2总线协议的转换 20](#_Toc115734846)

[2.3.3添加AXI总线GPIO设备 22](#_Toc115734847)

[2.3.4存储系统的构建 24](#_Toc115734848)

[2.3.5其他外设的添加 38](#_Toc115734849)

[2.3.6总线地址的分配 43](#_Toc115734850)

[2.4硬件工程的编译与验证 44](#_Toc115734851)

[2.5 操作系统工程的体系结构 46](#_Toc115734852)

[第三章 从BIOS到Bootload 49](#_Toc115734853)

[3.1工具链的使用 49](#_Toc115734854)

[3.1.1 MIPS交叉编译器的使用 49](#_Toc115734855)

[3.1.2 GDB调试 49](#_Toc115734856)

[3.2驱动程序的设计与实现 51](#_Toc115734857)

[3.3 Bootloader的实现 54](#_Toc115734858)

[3.3.1初始化TLB 55](#_Toc115734859)

[3.3.2 Load elf文件 56](#_Toc115734860)

[第四章 异常处理的设计与实现 59](#_Toc115734861)

[4.1 上下文的保存与恢复 59](#_Toc115734862)

[4.2 异常类型判断 60](#_Toc115734863)

[4.3 中断向量表 60](#_Toc115734864)

[4.4 时钟中断（handle\_int） 61](#_Toc115734865)

[4.5 系统调用（handle\_sys） 62](#_Toc115734866)

[4.6 访问地址越界中断（handle\_addr） 63](#_Toc115734867)

[4.7 TLB异常处理 64](#_Toc115734868)

[第五章 内存管理的设计与实现 65](#_Toc115734869)

[5.1 理解虚拟地址 65](#_Toc115734870)

[5.2 页式内存体系 66](#_Toc115734871)

[5.3 TLB作为页表的缓存 72](#_Toc115734872)

[第六章 进程管理的设计与实现 76](#_Toc115734873)

[6.1 进程控制块（PCB）与内存初始化 76](#_Toc115734874)

[6.2 进程创建 77](#_Toc115734875)

[6.3 进程调度 79](#_Toc115734876)

[6.4 进程释放 81](#_Toc115734877)

[6.5 线程创建 83](#_Toc115734878)

[6.6 设备管理 84](#_Toc115734879)

[第七章 关于拓展任务 89](#_Toc115734880)

[7.1 文件系统的抽象 89](#_Toc115734881)

[7.2 VGA显示屏 90](#_Toc115734882)

[7.3 页置换 90](#_Toc115734883)

[7.4 加载器的动态链接 90](#_Toc115734884)

[附录 如何运行整个工程？ 91](#_Toc115734885)

# 前言

操作系统 (Operating system)，作为计算机系统的基本组成部分，自产生之初就对计算机、通信技术领域产生了重大的影响。从最初的手工系统，到后来的多道程序以及分时系统，再到目前主流的PC端Windows、macOS X、Ubuntu等操作系统，以及移动端iOS、Android操作系统。可以说操作系统的出现使得计算机的发展进入了新的时代，也使得个人计算机进入了千家万户。而操作系统的日新月异，离不开计算机科学的教育。本教程正是为了计算机科学教学提出了一种教学用微内核设计方案，该微内核直接运行在真实的硬件平台上，而不是在虚拟机上运行，学生可以更直观的看到整个系统的运行过程。相比于在虚拟机环境运行的操作系统的学习，对学生的计算机系统综合设计能力的提高更为明显。本教程的方案可供高校的本科阶段操作系统课程以及相关课程设计参考。

# 第一章 开发之前要掌握的知识

## MIPSfpga概述

MIPSfpga是一款基于MIPS32 m14k处理器实现的商用软核处理器。该软核由

Imagination Technologies公司开发，并且向学术界免费提供许可。这使得它可以被用于教学的目的。MIPSfpga目前已经推出2.0版本，MIPSfpga 2.0提供了三个套件：MIPSfpga GSG（Getting Started Guide 入门指引手册）、MIPSfpga Labs以及MIPSfpga SoC（System-on-Chip 片上系统）。MIPSfpga GSG套件提供了MIPSfpga的Verilog以及VHDL两个版本的非模糊寄存器传输级（RTL）源代码，给出了MIPSfpga的详细文档，并且介绍了MIPSfpga系统使用、开发方法。MIPSfpga GSG还提供了开发MIPSfpga所需要的一系列工具链，包括交叉编译器以及OpenOCD等。并且MIPSfpga还支持指令集扩展，支持用户根据自己的需求增加MIPSfpga的指令集，来运行自定义的指令。

MIPSfpga Labs套件提供了基于MIPSfpga的一系列实验。这些实验解释了如何在

FPGA上创建microAptiv核心，以及如何在其上运行和调试程序，如何扩展MIPSfpga以添加新的外设并通过中断或直接内存访问（DMA）与它们通信，如何使用CorExtend特性和修改core的硬件本身，以及如何使用性能计数器来试验不同的内存层次结构、缓存大小、关联性和内容管理策略。

MIPSfpga SoC套件着重介绍了如何在MIPSfpga上搭建部署SOC，并且移植开源操作系统Linux内核。

### 1.1.1 MIPSfpga核心

MIPSfpga中使用的MIPS软核是通用微芯片PIC32MZ微控制器中使用的microAptiv UP core的一个版本。该软核由一组Verilog HDL文件组成，这些文件在5级流水线中实现MIPS32r3指令集体系结构（ISA）。如图1-1所示，正式发行版的MIPSfpga核心包括具有TLB的MMU、指令和数据cache，以及多个接口（例如EJTAG）。总线接口单元支持高级微控制器总线体系结构（AMBA）、高级高性能总线（AHB）精简协议。

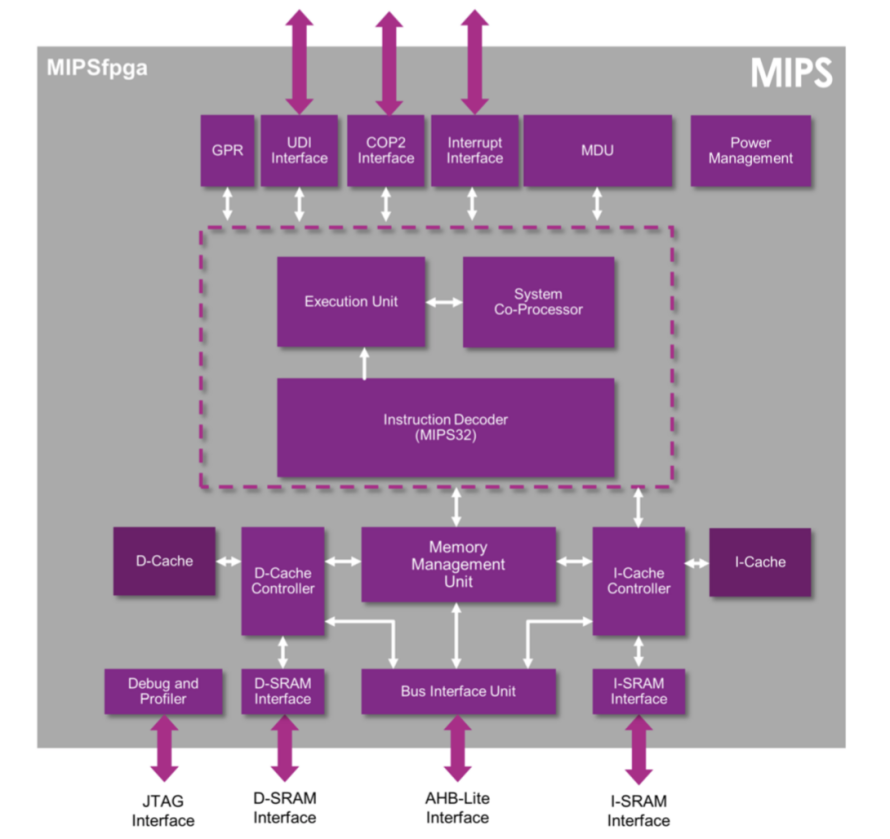


图1-1 MIPSfpga核心结构框图

### 1.1.2 MIPSfpga系统

如图1-2所示，MIPSfpga系统包括MIPSfpga核心以及基于AHB-Lite总线接口的外设。

MIPSfpga系统核心的运行至少需要两个信号，分别是系统输入时钟信号：SI\_ClkIn，系统输入复位信号（低电平触发）：SI\_Reset\_N。该核心还提供了EJTAG接口以便于开发者进行程序下载、调试，可以大大缩短程序开发的周期。

此外，MIPSfpga系统提供AHB-Lite总线接口，以便开发者进行外设的扩展。开发者可以据此添加RAM控制器以便于访问DDR和BRAM等片上存储，也可以添加自定义GPIO来访问片上外设，例如LED、拨码开关等。

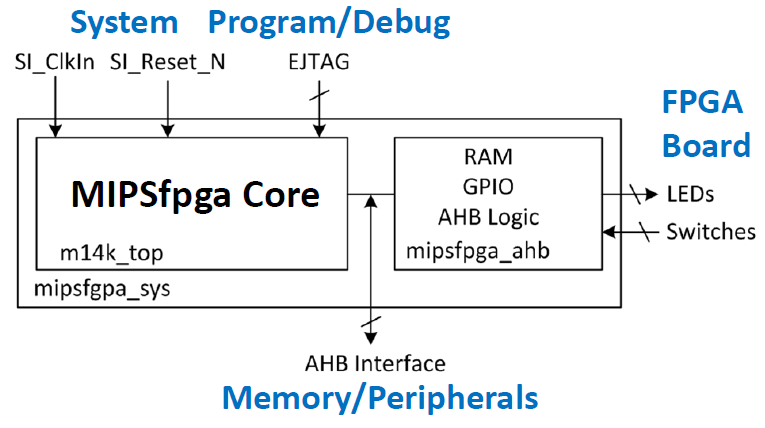


图1-2 MIPSfpga系统

### 1.1.3 MIPSfpga寄存器概述

操作系统，是最接近硬件层的软件。因此操作系统的许多操作依赖于对寄存器的操作，这也是本教程的操作系统采用C语言+汇编语言开发的原因。因此本教程选择性地介绍MIPSfpga中，对于操作系统内核开发十分重要的若干个寄存器。详细的MIPSfpga寄存器介绍请参考官方文档《MicroAptiv UP Software User's Manual MD00942.pdf》。

（1）通用寄存器

MIPS32体系结构共有32个通用寄存器，这些寄存器的长度都是32bit。各个寄存器的编号、助记符以及用途，如表1.1所述。

表1.1 MIPS32通用寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| 寄存器编号 | | 助记符 | 描述 |
| 0  1  2-3  4-7  8-15  24,25  16-23  26,27  28  29  30  31 | zero  at  v0, v1  a0-a3  t0-t7  t8-t9  s0-s7  k0, k1  gp  sp  s8/fp  ra | | 常量0  汇编暂存寄存器，为汇编保留  子程序返回值寄存器  子程序的前几个参数  暂存，子程序使用时不需要恢复  暂存，子程序使用时不需要恢复  子程序必须存储且恢复这些寄存器的值  为异常处理程序保留  全局指针  堆栈指针  子程序用它作为帧指针  子程序返回地址 |

值得注意的是k0，k1两个寄存器的使用。这两个寄存器为操作系统陷入保留，在异常处理程序中会使用到这两个寄存器，理论上不用做其他汇编程序。

sp寄存器被用作堆栈指针。MIPS32体系结构没有相应的堆栈指令，入栈出栈操作都需要显示的对sp寄存器进行修改。通常只在子程序开始和结束时修改sp寄存器的值。MIPS32采取向下生长的栈，入栈时减，出栈时加。

ra寄存器保存子程序的返回地址，在子程序中需要先将ra寄存器入栈，在子程序结束时恢复ra寄存器的值。

（2） CP0寄存器

CP0（Co-Processor 0，协处理器0）是MIPS32体系结构中非常重要的一个部件，CP0是MIPS所有协处理器中**唯一一个**必须实现的。它起到控制CPU的作用，CP0对CPU的这个控制作用体现在CP0的寄存器上面。本教程给出几个关键CP0寄存器的概述。

1. BadVAddr

BadVAddr寄存器各比特的含义如图1-3所示。该寄存器自动保存最近一次地址相关异常的虚拟地址。在发生TLB相关异常时，需要查寻该寄存器，来获取导致TLB异常的虚拟地址，来进行TLB重填操作。

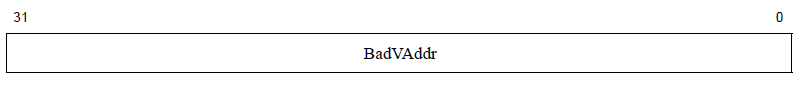


图1-3 BadVAddr寄存器

1. EntryLo0、EntryLo1

EntryLo0、EntryLo1两个寄存器也是与TLB有关的寄存器，EntryLo0存放偶数页的入口物理地址，EntryLo1存放奇数页的入口物理地址。这两个寄存器各比特的含义如图1-4所示。

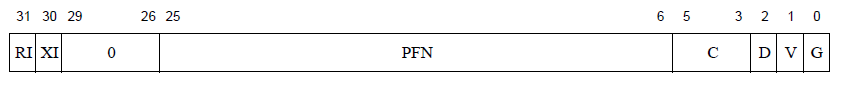


图1-4 EntryLo0和EntryLo1寄存器

RI，XI：读禁止和执行禁止，如果该位被设置位1，那么即使V = 1，对于该页的读取也会导致TLB Invalid异常。

29:26 ：强制写入0

PFN：Page Frame Number，即物理页帧号。对应物理地址的31:12位。

C：描述页面的缓存一致性属性，对于microAptiv UP核来说，2为不可缓存，3为可缓存。

D：脏位，或者叫做写使能位。D = 0时，不允许写入该页，否则会引起TLB修改异常。

V：有效位，表明当前TLB映射是否有效。V = 1，表示允许访问该页，V = 0，对该页面的访问会导致TLB无效异常。

G：全局位，如果G = 1，那么TLB表项匹配时会忽略ASID域。

1. EntryHi

EntryHi是与EntryLo0、EntryLo1功能类似的寄存器，也是用于对TLB读、写操作的寄存器。每当TLB发生异常时，需要将发生异常的虚拟地址的31:13位写入VPN2，该寄存器的各比特含义如图1-5所示。

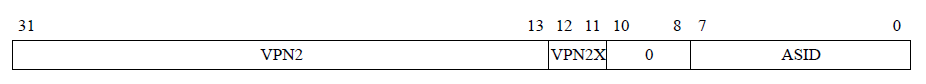


图1-5 EntryHi寄存器

VPN2：发生异常的虚拟地址的31:13bit

VPN2X：本教程中为常量0

ASID：表示地址空间标志，在对TLB进行读操作时，由硬件写入；写操作时，由操作系统软件写入。用于地址空间的隔离。

1. Status

Status是一个用于获取CPU状态的寄存器，其存储的内容可以表示CPU的中断使能、操作状态等信息。该寄存器的各比特含义如图1-6所示。

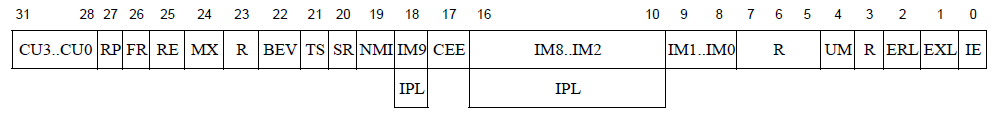


图1-6 Status寄存器

下面给出本教程用到的若干bit的定义：

BEV：是否启动Boot异常向量，用于定位异常向量的入口地址，为1时将异常向量入口定位在kseg1，为0时定位在通常的异常向量入口地址。

TS：TLB是否被关闭，当发生TLB严重错误时，该位被置为1。除非硬件复位，否则无法清除。

NMI：表示是否是不可屏蔽中断，软件只能对其写入0。

IM：是否进行中断屏蔽，0为中断屏蔽，1为不屏蔽。该段与Cause寄存器的IP段配合使用，来进行中断服务。

UM：是否为用户模式。0表示内核模式，1表示用户模式。

ERL：是否处于错误级，该位如果被置1，那么表示处理器处于错误级。eret指令使用ErrorEPC寄存器来返回而不是EPC寄存器。并且kuseg段会被当成一个不经过TLB映射，且不缓存的段。

EXL：是否处于异常级。

IE：是否中断使能，当IE = 1，EXL = 0，ERL = 0时，中断才会被启动。

1. Cause

Cause寄存器记录上一次发生异常的原因，该寄存器被用于定位错误或者异常的原因，其各比特含义如图1-7所示。

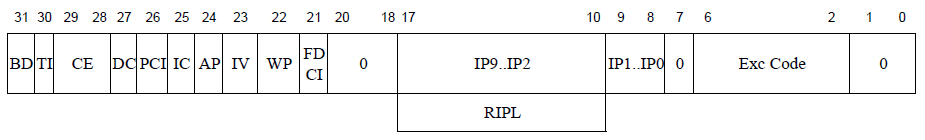


图1-7Cause寄存器

BD：表示最近发生的异常是否位于分支延迟槽，该位如果被置1，那么分支指令将无法进行跳转。在实验中发现，如果在异常处理中使用两个连续的分支或者跳转指令，该位会被置1，进而导致产生保留指令异常（10号异常）

IP：表示当前相应中断是否被挂起，其中IP1:0为软件中断，其他为硬件中断。

Exc Code：异常码（Exception Code），该字段的编码如表1.2所示

表1.2 异常编码

|  |  |  |  |
| --- | --- | --- | --- |
| Exc Code | | 简记 | 描述 |
| 0  1  2  3 | Int  Mod  TLBL  TLBS | | 中断  TLB修改异常  TLB加载或取指令异常  TLB存储异常 |

续表1.2 异常编码

|  |  |  |  |
| --- | --- | --- | --- |
| Exc Code | | 简记 | 描述 |
| 4  5  6  7  8  9  10  11  12  13  14-22  23  24  25-31 | AdEL  AdES  IBE  DBE  Sys  Bp  RI  CpU  Ov  Tr  —  WATCH  MCheck  — | | 加载或取指地址错误异常  存储地址错误异常  总线错误（取指）  总线错误（访问数据）  系统调用异常  断点异常  保留的指令异常  协处理器不可用异常  整数溢出异常  陷阱异常  保留  访问WatchHi/WatchLo地址  机器检查  保留 |

本教程中，操作系统设计需要重点关注的异常有0号异常（中断），1-3号异常（TLB异常），以及8号异常（系统调用异常）。在本教程的异常处理中，保留了其他异常的处理函数，仅实现了上述的异常类型的处理函数。

## 1.2 Minisys实验平台简介

Minisys实验开发板是由东南大学计算机科学与工程学院与依元素科技联合开发的，用于计算机系统综合能力培养的FPGA实验开发板。Minisys实验板是以Xilinx Artix-7TM系列FPGA（XC7A100T FGG484C-1）为主芯片的可用于“数字电路”、“计算机组成原理”、“计算机组成课程设计”、“微机原理与接口技术”、“计算机系统综合课程设计（SoC设计）”等多门课程实验的统一实验平台。

Minisys实验板拥有丰富的计算、存储资源以及I/O外设资源。以供操作系统内核的使用。Minisys实验板如图1-8所示，其资源对照表如表1.3所示。

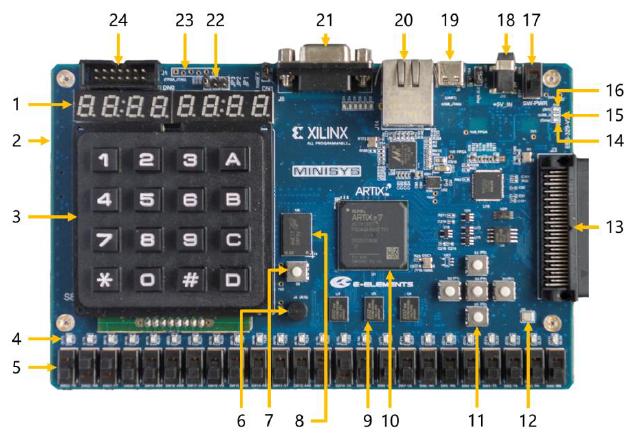


图1-8 Minisys实验板

表1.3 Minisys资源表

|  |  |  |  |
| --- | --- | --- | --- |
| 标注 | 描述 | 标注 | 描述 |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 8个七段数码管  Micro SD卡槽（实验板背面）  4\*4小键盘  LEDs（红、黄、绿各8个）  拨码开关  蜂鸣器  FPGA复位键  DDR3 SDRAM  SRAM  XC7A100T主芯片  五个按键开关  麦克风 | 13  14  15  16  17  18  19  20  21  22  23  24 | 接口板连接器  FPGA烧写完成指示灯  USB\_JTAG指示灯  电源指示灯  电源开关  电源连接口  Type-C接口（编程、串口）  以太网接口  VGA接口  编程跳线  用户扩展IO  JTAG接口 |

### 1.2.1 Minisys计算资源

Minisys的主芯片XC7A100T上有101440个逻辑单元，15850个Slice，每一个Slice中带有4个6输入的查找表（LUT）和8个触发器，片内近12.5%的查找表可以配置为64-bit分布式RAM或者32位的SRL（或两个16位SRL16），使得综合工具能够充分利用这些逻辑和存储资源。

拥有240个DSP48E1数字信号处理单元，每个DSP48E1中包含一个预加器，一个25×18乘法器，一个加法器以及一个累加器。

拥有6个时钟管理模块（CMT），每个包含1个混合模式时钟管理器（MMCM）及一个锁相环（PLL）。MMCM和PLL的中心都有一个可以根据输入电压而调速的晶振，由此能够生成频率范围很宽的时钟信号。同时，这两个部件又都能作为输入时钟信号的抖动滤波器。

XC7A100T的内部时钟最高可达450MHz，Minisys实验板采用100MHz主频。

### 1.2.2 Minisys存储资源

（1）Block RAM

Minisys的Block RAM（或简称BRAM）集成在主芯片XC7A100T的内部，片内集成135个36Kbit的Block RAM，并且每一个可以当作两个独立的18Kbit的Block RAM使用。这些Block RAM资源可以利用Vivado的IP集成器很方便地配置成单端口、双端口等多种类型RAM。

（2）DDR3 SDRAM

Minisys实验板上，将一个容量为256M×16bit的DDR3 SDRAM(芯片型号为MT41J256M16-FBGA96)连接到主芯片上。当主芯片访问SDRAM时，需要传送15位的行地址、3位块地址和10位列地址，其中行地址和列地址分时共用一组地址线。

（3）SRAM

SRAM模块由三块IS61WV51216BLL-10MI芯片并联组成，每块芯片的容量为512K×16bit，并联后的SRAM模块的容量为512K×48bit，通过19根地址线和48根数据线与主芯片连接。

（4）Flash Memory

非易失串行Flash 的容量是128Mbits，使用的是专用的Quad SPI 总线。FPGA 的配置文件可以写入Quad SPI Flash（型号N25Q032A13ESE40F），当短接了JP3后，板子在上电时，FPGA 自动从SPI Flash 中读取配置文件。当编程跳线连接JP3的位置时，可以将编程文件下载到Flash 中。

### 1.2.3 Minisys I/O外设资源

Minisys拥有丰富的I/O外设资源。包括24个拨码开关SW23~0，在实验中可以作为系统的输入。还包括24个LED灯，其中RLD7~0为红色，YLD7~0为黄色，GLD7~0为绿色，可以作为系统的输出。还包括5个方向按键、4\*4按键、一个蜂鸣器以及八个七段数码管。

Minisys丰富的计算以及存储资源使得开发者利用其开发、运行一个自定义的SoC变得非常方便且高效。其丰富的I/O外设资源也使得开发者可以使自己的SoC更有用户交互性。总的来说，选择Minisys作为本教程的操作系统微内核运行的硬件平台是合适的。

### 1.2.4 Bus Blaster

Bus Blaster如图1-9所示。它是一个可以用于FPGA、ARM、flash、CPLD等设备的高速JTAG调试器。在实验中，可以通过Bus Blaster配合OpenOCD进行程序装载、调试、运行等操作。这大大缩短了程序开发的周期。

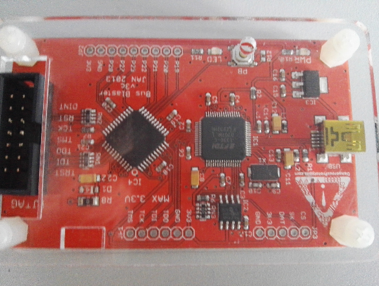


图1-9 Bus Blaster下载调试器

### 1.3.3中断以及异常

正如操作系统课程中讲的“操作系统是中断驱动的”。任何操作系统都是离不开中断的。可以说操作系统本身就是一个大型中断和异常处理程序。

MIPS中，中断、陷阱、系统调用和任何可以中断程序正常执行流的情况都称异常。也就是说，中断是异常的一种。各个异常的异常码可以在1.1.3节中的Cause寄存器的介绍中找到，可以看到中断是0号异常。

关于更多microAptiv UP核异常的内容，请读者参考官方文档《MicroAptiv UP Software User's Manual MD00942.pdf》的“Exceptions and Interrupts in the microAptiv™ UP Core”章节。该章节详细的描述了MIPS microAptiv UP核的异常产生条件、异常优先级、中断、异常向量入口、通用异常处理等内容。请读者在开发OS之前，掌握其中的内容，尤其是TLB相关异常的处理、系统调用。

# 第二章 MIPSfpga-SoC的移植

## 2.1工具链的搭建

### 2.1.1 OpenOCD的安装

首先将MIPS\_FPGA压缩包解压至路径D:\

接着通过D:\MIPS\_FPGA\路径找到OpenOCD-0.9.3-Installer.exe，双击打开，安装OpenOCD，当出现如图2-1所示界面时，去掉第二个选项的勾，然后继续安装。

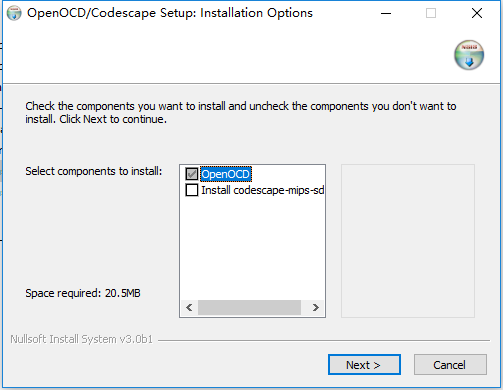


图2-1

### 2.1.2 MIPS工具链的安装

通过D:\MIPS\_FPGA\路径找到mipssdk.v2.0.0k.windows.x64.offline.exe，双击打开，默认安装（安装路径保持默认，并为C盘保证足够的空间）,当遇到如图2-2所示界面时，把空白处都打勾，点击next继续安装。

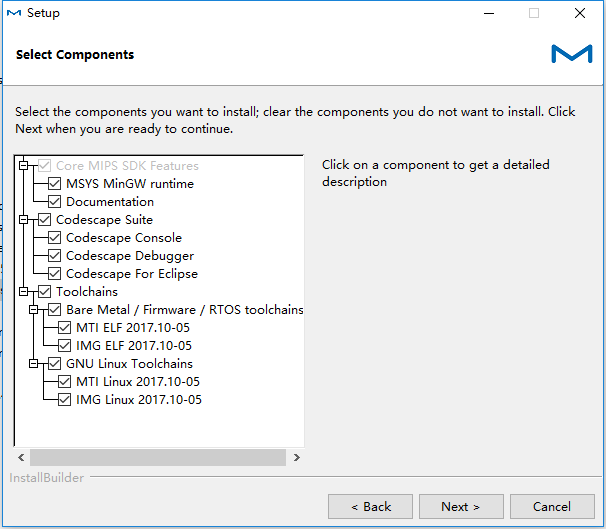


图2-2

等待出现图**2-3**所示界面时，点击**next**。安装完成。



图2-3

### 2.1.3 PuTTY以及驱动程序的安装

通过D:\MIPS\_FPGA找到putty-64bit-0.70-installer，双击打开，一直点击next,直到安装完成。如图2-4所示。点击Finish。PuTTY安装完成。

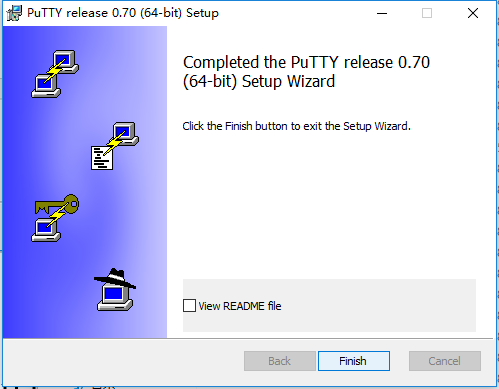


图2-4

最后通过C:\Program Files\Imagination Technologies\OpenOCD找到zadig\_2.1.1.exe，双击打开，把缺少的驱动程序选择安装。

## 2.2 MIPSfpga-SoC工程的创建

首先打开vivado2017.4，点击create project，然后填写工程名称并选择路径。如图2-5所示。

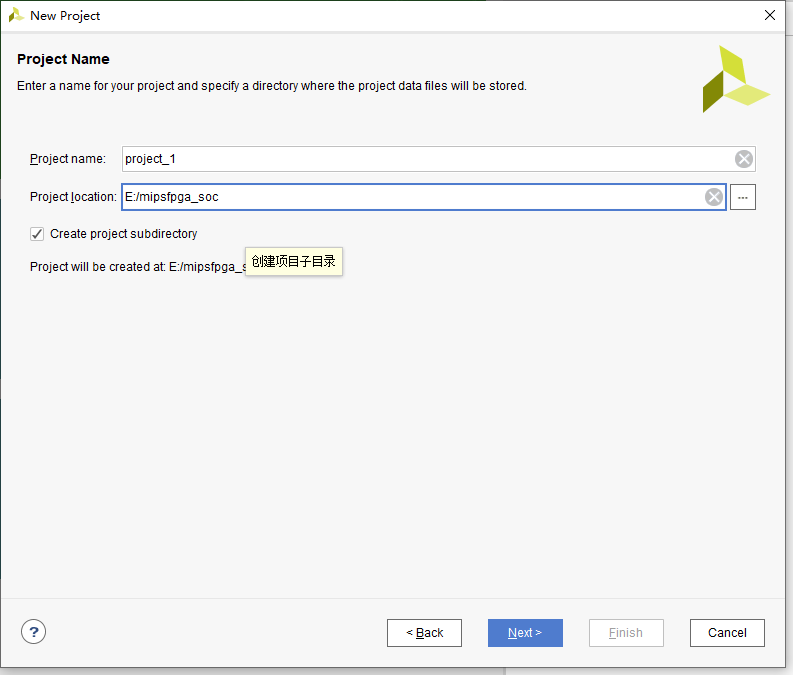


图2-5

点击Next，选择RTL project。

点击Next，这里暂时不导入任何源文件。

点击Next，这里暂时不添加约束文件。

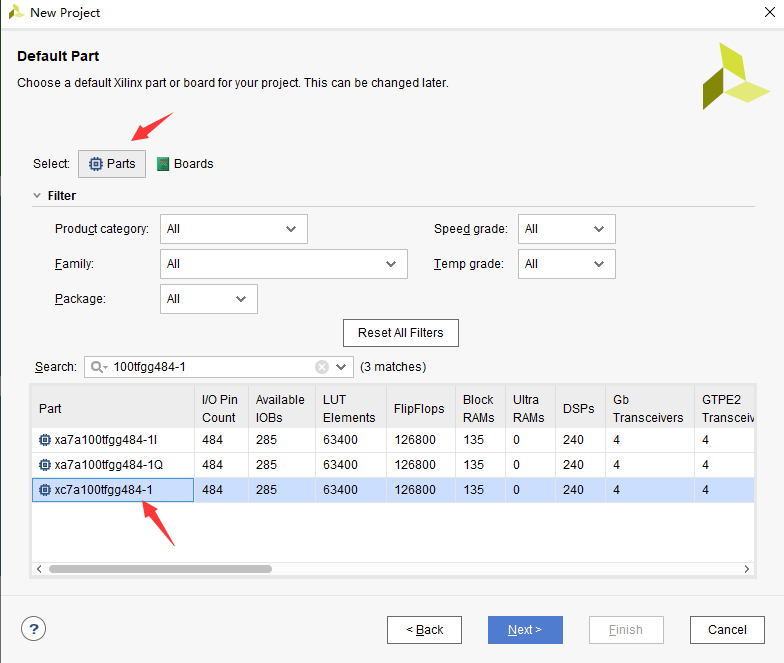


图2-6

点击Next，这一步选择FPGA芯片类型为**xc7a100tfgg484-1.**

点击Next，然后点击Finish完成工程的创建。在创建完成后，在左侧PROJECT MANAGER栏中点击Create Block Design，如图2-7所示，然后点击OK完成Block Design的创建。

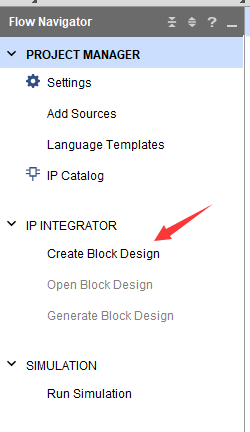


图2-7

## 2.3 MIPSfpga-SoC的搭建

### 2.3.1MIPSfpga Core的导入

首先点击PROJECT MANAGER的Settings进入设置界面。如图2-8所示，展开IP的选项，选择Repository，然后点击加号“+”，添加准备好的MIPSfpga Core IP核（mips\_packaged，该IP核在MIPS\_FPGA目录下可以找到）的路径。点击确定完成添加。如图2-9所示，本教程将该IP的目录复制到工程文件的子目录下，便于移植。

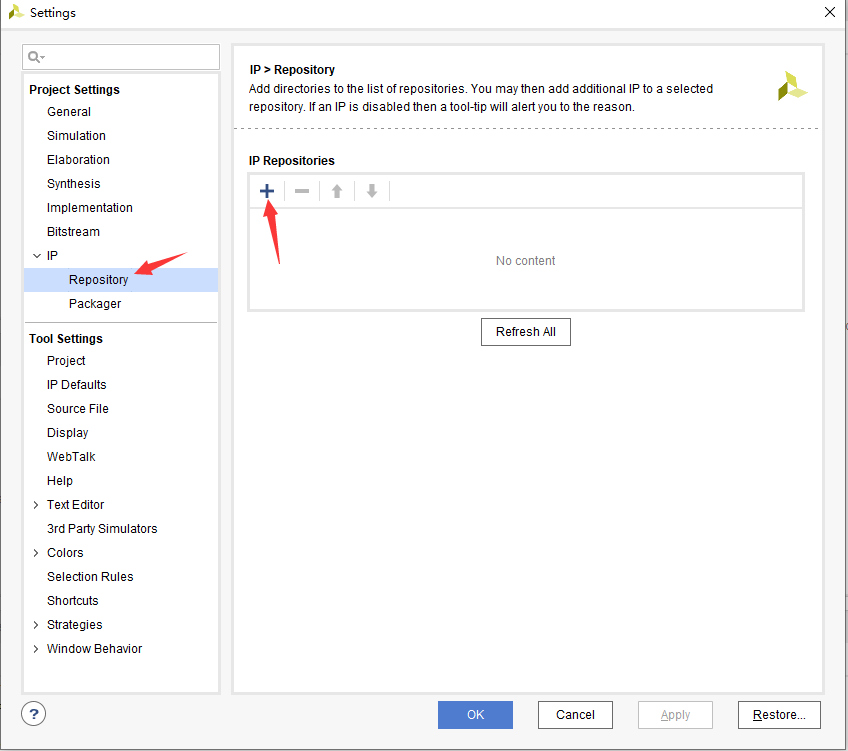


图2-8

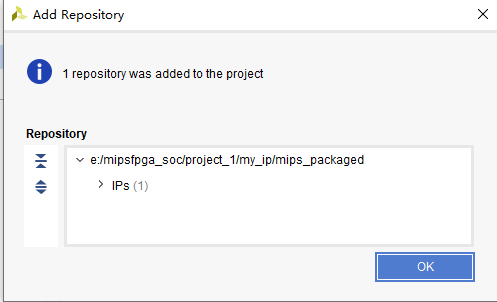


图2-9

接着，回到我们创建的Block Design中，此时它还是空白的。我们选择Add IP，搜素MIPS即可找到刚刚添加的MIPSfpga Core的IP核。点击完成添加，如图2-10所示。

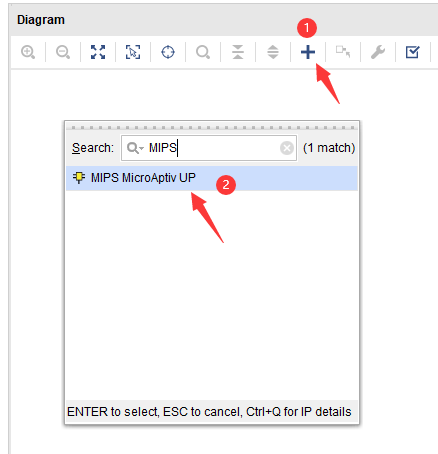


图2-10

这是我们可以在Block Design中看到我们刚才添加的MIPSfpga Core，也就是CPU部分。如图2-11所示。可以看到它上面有许多信号，遵循左进右出原则，这些信号/接口的功能描述如表2.1所示。

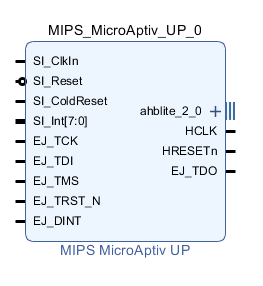


图2-11

表2.1 CPU信号/接口描述

|  |  |  |
| --- | --- | --- |
| 信号/接口名 | 功能描述 | |
| SI\_ClkIn  SI\_Reset  SI\_ColdReset  SI\_Int  EJ\_\*  ahblite\_2\_0  HCLK  HRESETn | | CPU时钟输入  CPU复位  CPU冷复位  8bit中断输入  EJTAG调试相关信号  AHB-Lite总线接口  AHB-Lite总线同步时钟  AHB-Lite总线复位 | |

下面我们将对CPU上的信号进行连接。首先按照如图2-12的方式创建输入输出port进行连接。左侧三个IP核分别为Utility Vector Logic（非门），Utility Buffer（BUFG，CSIZE:1），Constant（位宽8bit，低电平）。这里完成了EJTAG的连接以及CPU复位信号的连接。

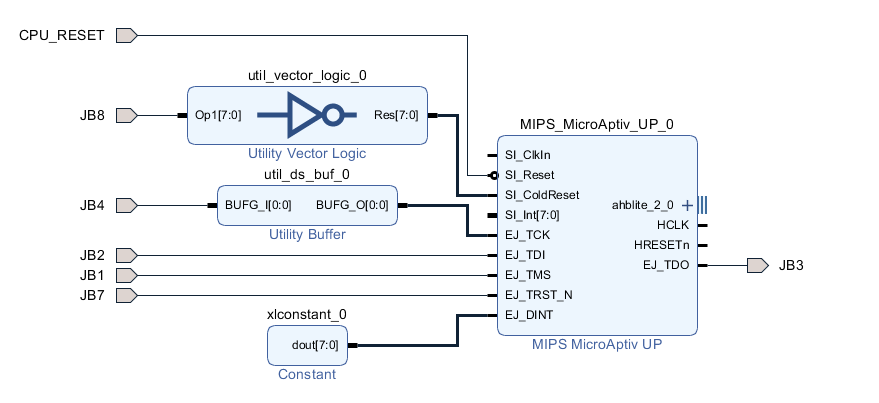


图2-12

在链接CPU时钟前，首先已知Minisys开发板的晶振时钟频率为100MHz。CPU不需要这么高的时钟频率，因此需要创建时钟向导，来创建CPU的输入时钟。

Add IP，找到Clock Wizard，在primitive中选择PLL，如图2-13所示。接着进入Output Option中，进行如图2-14所示的设置。两个时钟输出端口，50MHz的时钟直接连接置CPU的时钟输入，200MHz的后面的教程中会用到。

至此CPU的输入信号，除了中断信号外全部连接完成，下面进行AHB-Lite总线接口的连接。

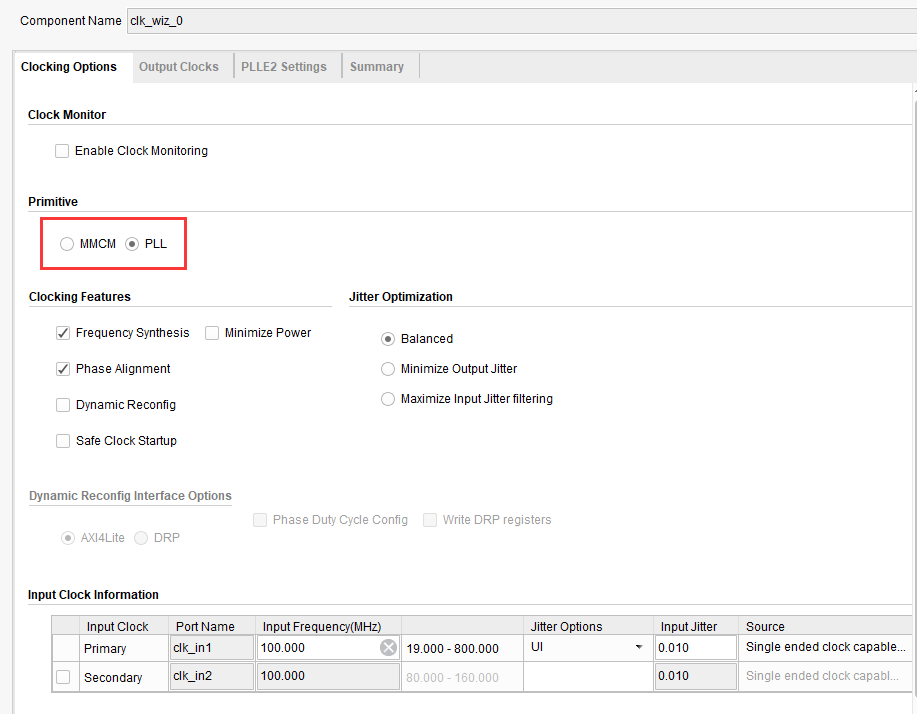


图2-13

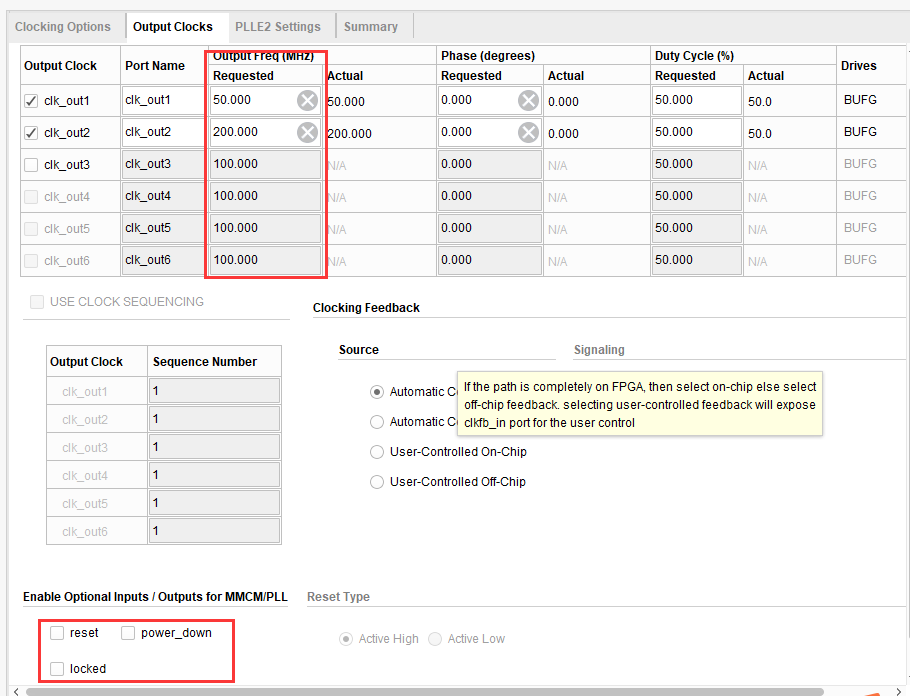


图2-14

### 2.3.2总线协议的转换

由于Xilinx提供的大部分的总线设备控制器使用的总线标准是AXI或AXI-Lite，为了方便硬件设计，这里需要将microAptiv UP核的AHB-Lite总线进行转换，转换成常用的AXI总线标准。Vivado提供了相应的IP核来完成这个转换，找到并添加AHB-Lite to AXI Bridge这个IP核。

将CPU上的AHB-Lite接口展开，找到名为HREADY的信号。再将AHB-Lite to AXI Bridge的AHB-Lite接口展开，找到s\_ahb\_hready\_in和s\_ahb\_hready\_out两个信号，将它们连接在一起，再将两个AHB-Lite接口相连，并且连接时钟和复位信号。这样就完成了总线的桥接，连接好的情况如图2-15所示。

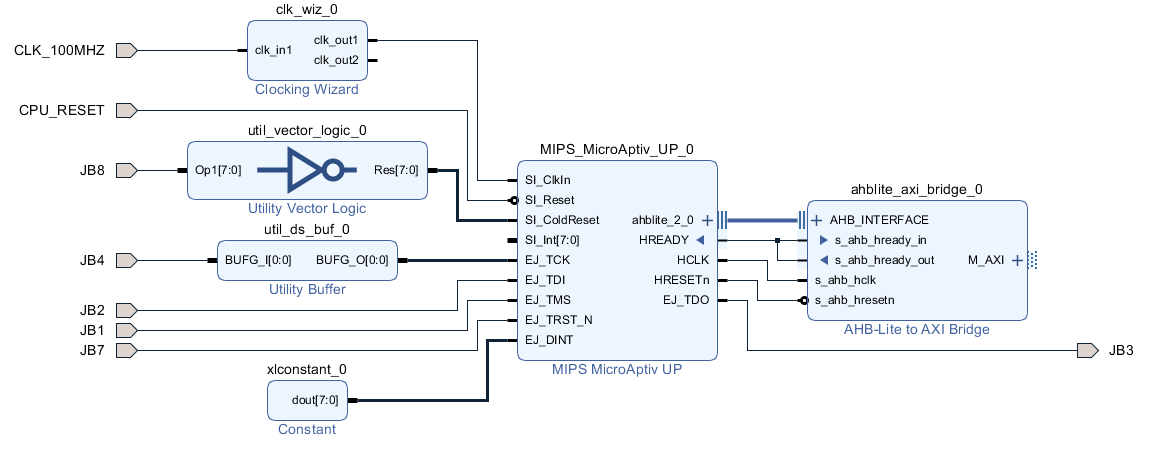


图2-15 CPU与总线桥的连接

此时我们已经将MIPS MicroAptiv核上的AHB总线协议转换成了我们常用的AXI总线协议。下一步是添加AXI总线互联IP核，来实现AXI设备的扩展。

Add IP，找到AXI Interconnect这个IP核并添加，双击它进行自定义，在这里可以自定义Master Interface的数量，可以根据实际需求进行增减。本例程中设置了11个Master Interface（也就是11个AXI Slave设备）。

连接好的情况如图2-16所示，各个AXI从设备的Clk和Reset信号在添加设备的时候进行连接。

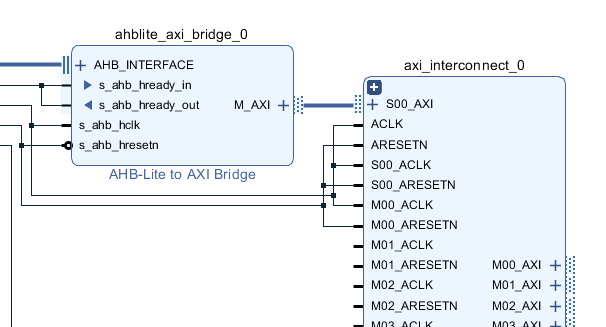


图2-16

### 2.3.3添加AXI总线GPIO设备

LED、蜂鸣器、七段数码管、拨码开关，这些设备的控制都是基于GPIO进行的。以LED为例，添加名为AXI GPIO的IP核，双击进行自定义。按照图2-17的方式，选择All Output模式，并选择位宽为24bit（Minisys实验板共有24个LED）。将该IP核的AXI接口与AXI Interconnect的M00相连，并连接总线时钟和复位信号（这些设备的时钟和复位信号全部连接在一起，即全部连接到CPU的HClk、HRESETn），并且创建LED输出端口，位宽24。连接好的情况如图2-18所示。

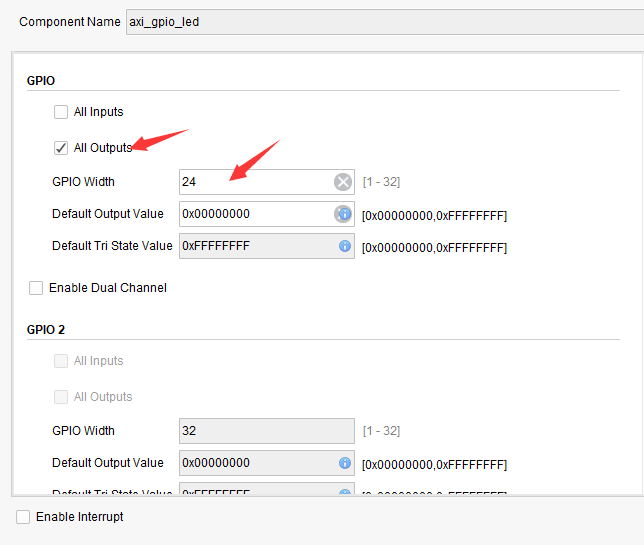


图2-17

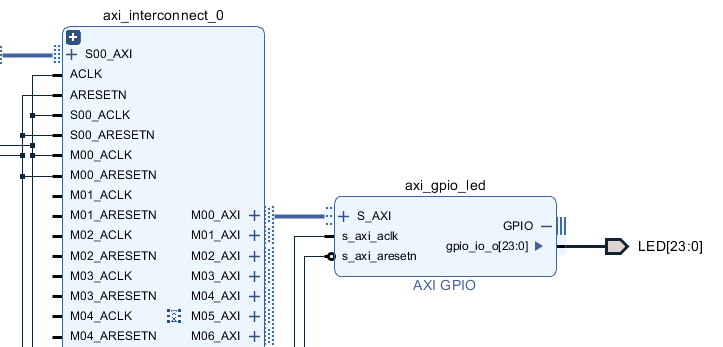


图2-18

类似的，添加蜂鸣器、拨码开关、七段数码管的GPIO。值得注意的是，拨码开关应选择All Input模式。对于蜂鸣器，除了添加GPIO作为控制寄存器外，还应添加控制蜂鸣器的输出波形的控制器，读者可以选择使用本教程提供的IP核（路径MIPS\_FPGA/my\_ip/buzzer），也可以自行设计。七段数码管的GPIO可以分为使能和数据两个，并且也需要添加相应的控制器来点亮数码管，这个数码管实质上就是一个译码器，读者可以根据数字电路的知识自行设计，也可以使用本教程提供的IP核来进行控制（路径MIPS\_FPGA/my\_ip/seven\_seg）。全部添加好后，此时的Block Design如图2-19所示。

添加完外设之后，下一步是创建MIPSfpga-SoC的存储系统。

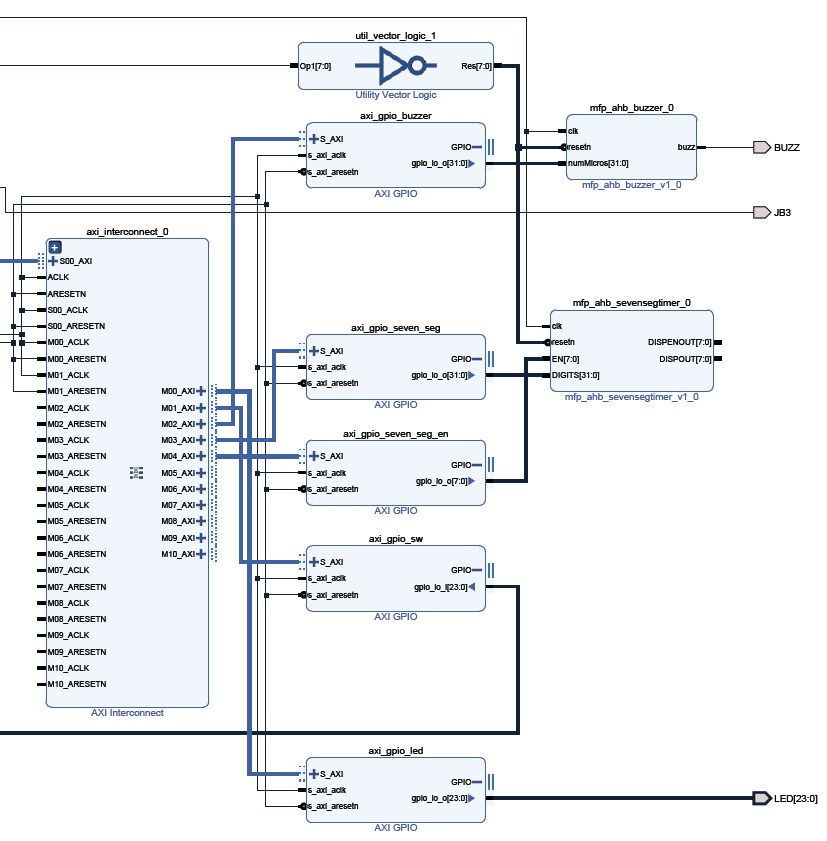


图2-19

### 2.3.4存储系统的构建

MIPSfpga-SoC的存储系统主要由两个部分构成，一部分是Block RAM，该部分容量小速度快，不足以支撑OS运行；另外一部分是由DDR3 SDRAM构成，这一部分容量大，但速度相对较慢，可以支撑OS运行。根据这个特点，我们应当选择DDR3作为系统的主存储器，而将Block RAM映射为存放Bootloader的存储器。

（1）添加Block RAM控制器

Add IP，找到AXI BRAM Controller这个IP核，点击添加。然后双击自定义这个IP，按照图2-20的选择修改端口数为1。点击确定。

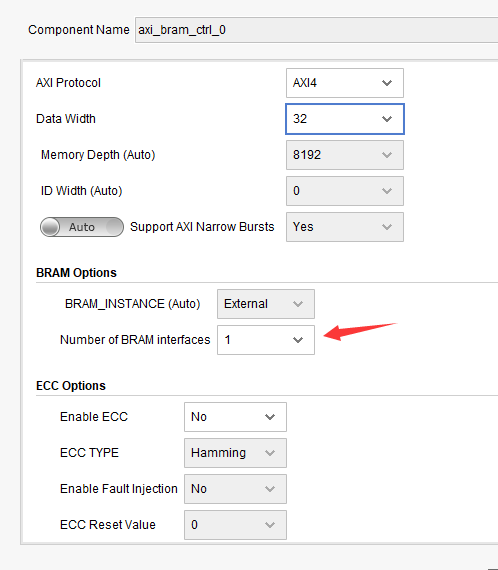


图2-20

接着添加第二个IP，找到Block Memory Generator这个IP，双击并添加。然后自定义这个IP，在basic选择Mode为Stand Alone，其他保持默认，如图2-21所示。

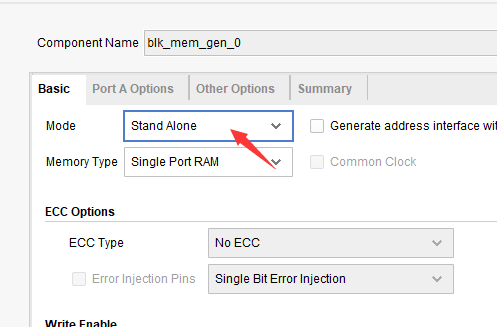


图2-21

接着进入Port A Option，设置数据深度为131072（128k），剩余选项的设置如图2-22所示。在Other Option中，可以设置初始化.coe文件，我们可以将编译好的代码生成.coe文件，将其预读取到BRAM中去。读者可以使用教程中提供好的.coe文件（MIPS\_FPGA/ram\_init/ram\_init.coe，使用此文件时地址分配要和本教程保持一致），也可以自行编译生成.coe文件。

然后连接它们和AXI Interconnect，如图2-23所示。

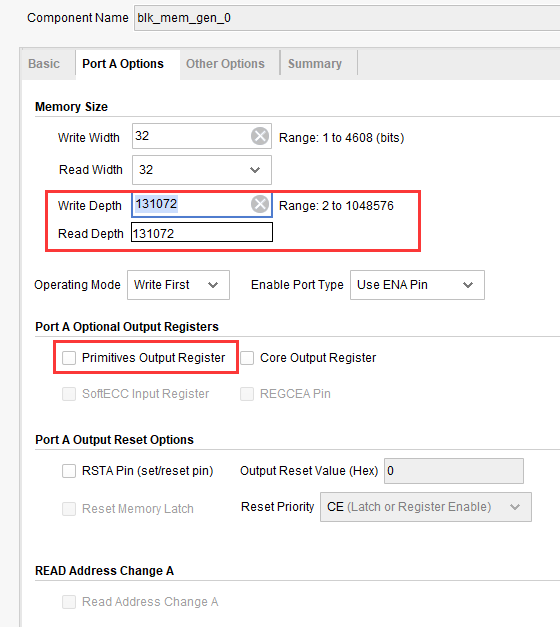


图2-22

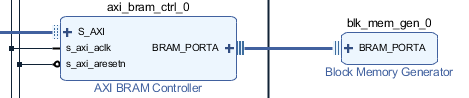


图2-23

在本教程中，共添加了两套上述的BRAM控制器和Block Memory Generator，一个用于Bootloader，映射到物理地址0x1fc0\_0000（请读者思考这是为什么）；另一个是Linux启动所需要的一个8kB内存空间，映射到物理地址0x1000\_0000，如果不需要移植Linux，也可以不添加这组IP。

（2）添加DDR3 SDRAM控制器

DDR3 SDRAM控制器采用Xilinx的Memory Interface Generator（MIG）这个IP核。首先找到这个IP核并添加，接着按照如下步骤自定义该IP。

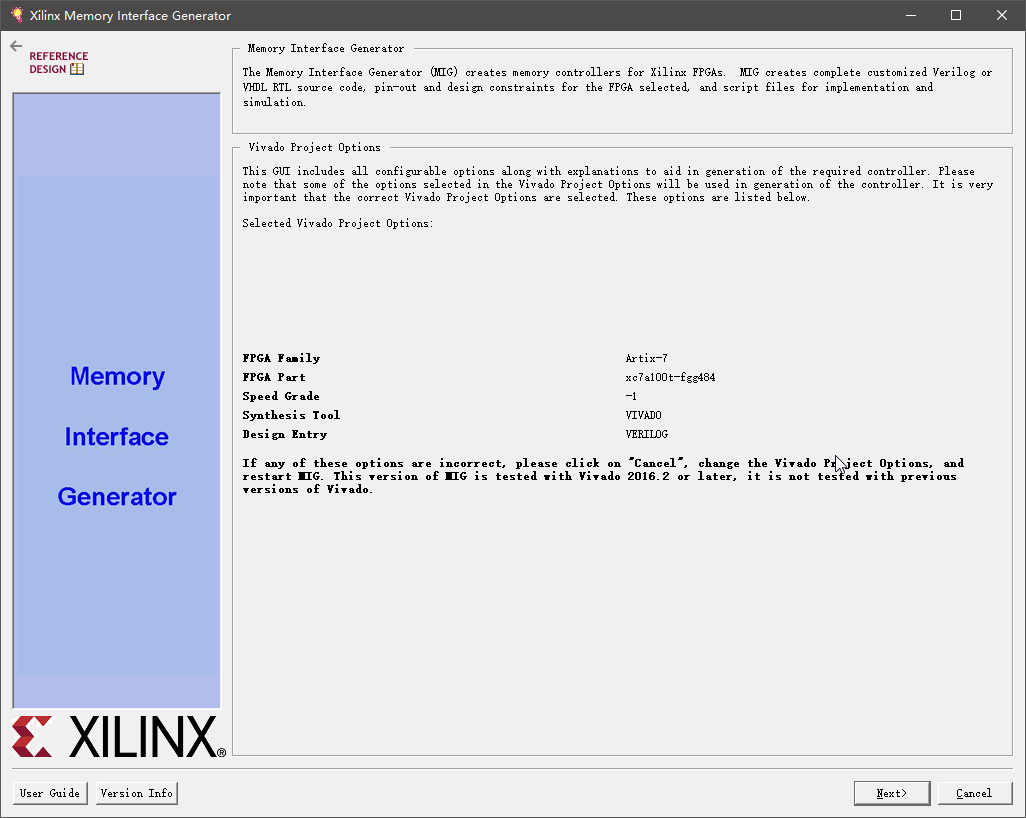


图2-24

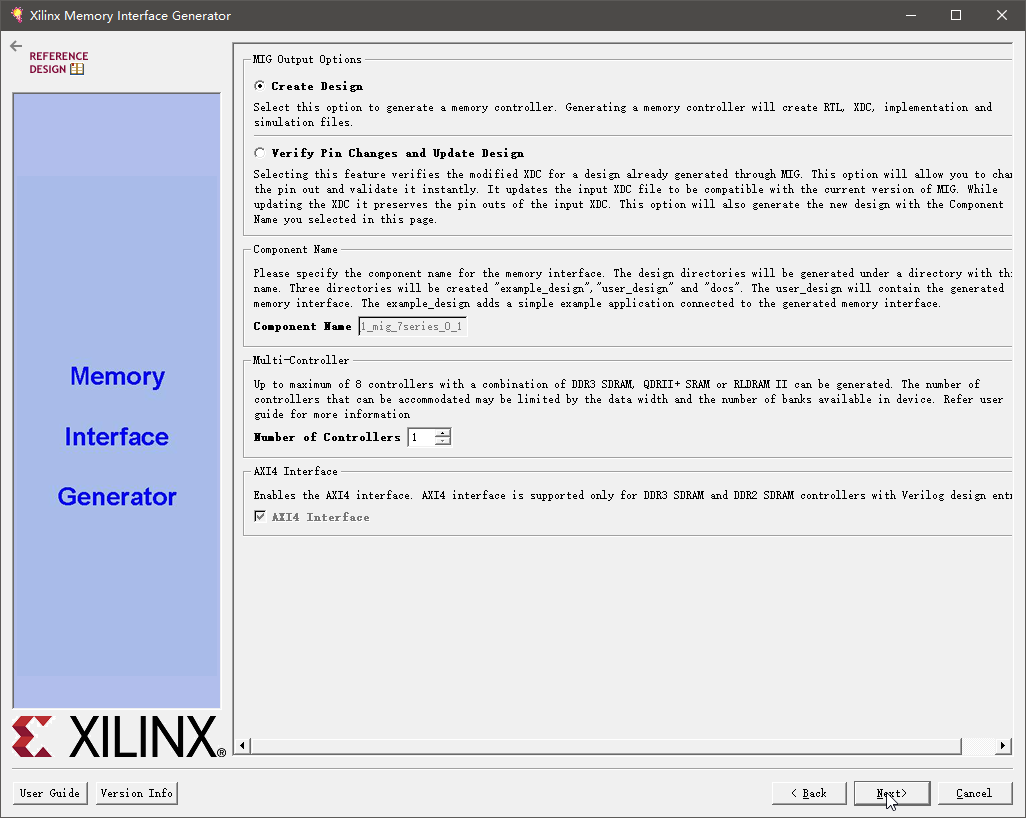


图2-25

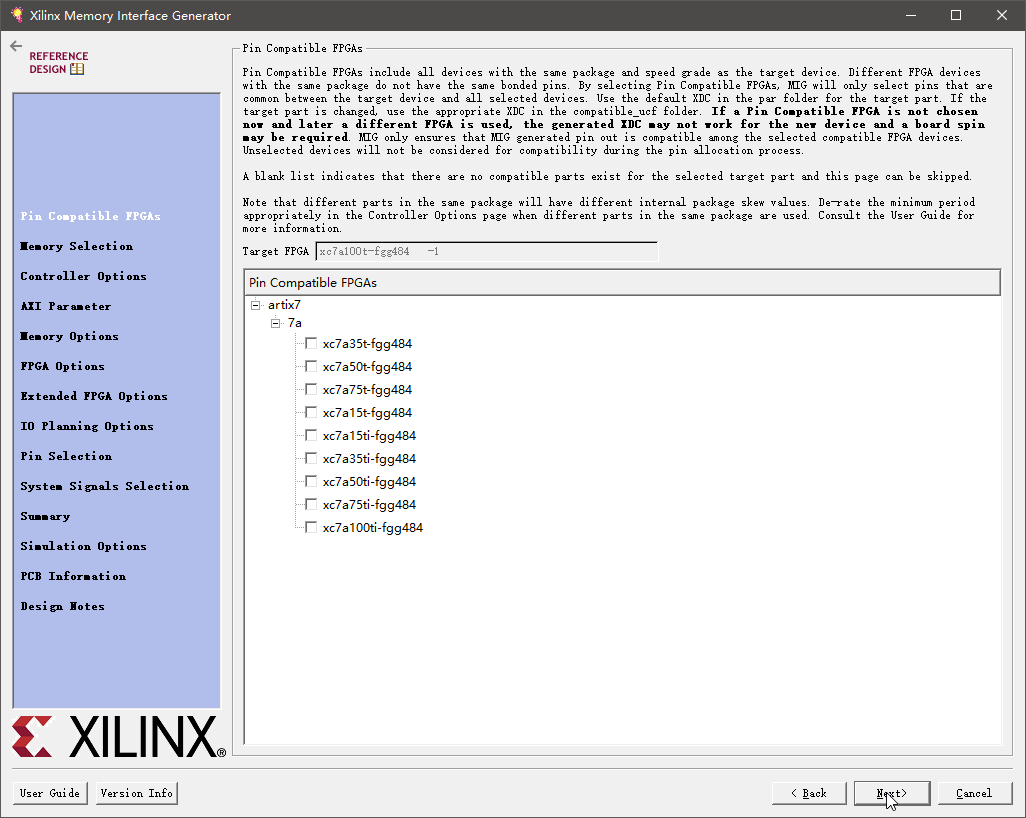


图2-26

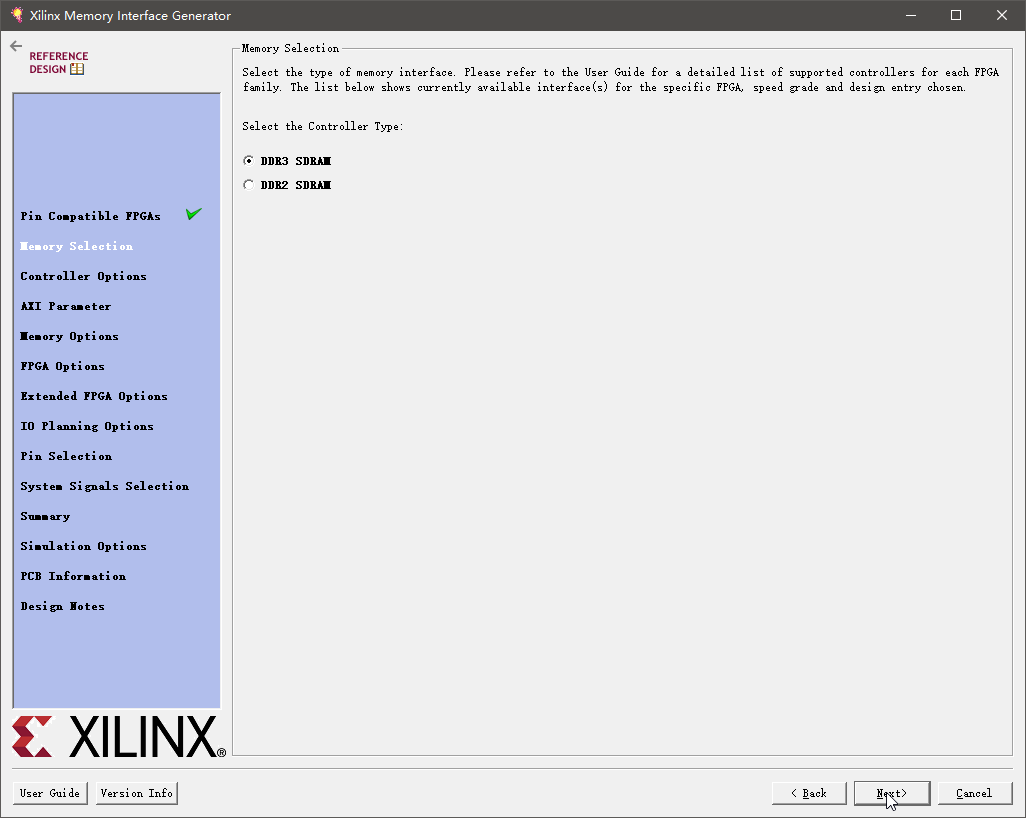


图2-27

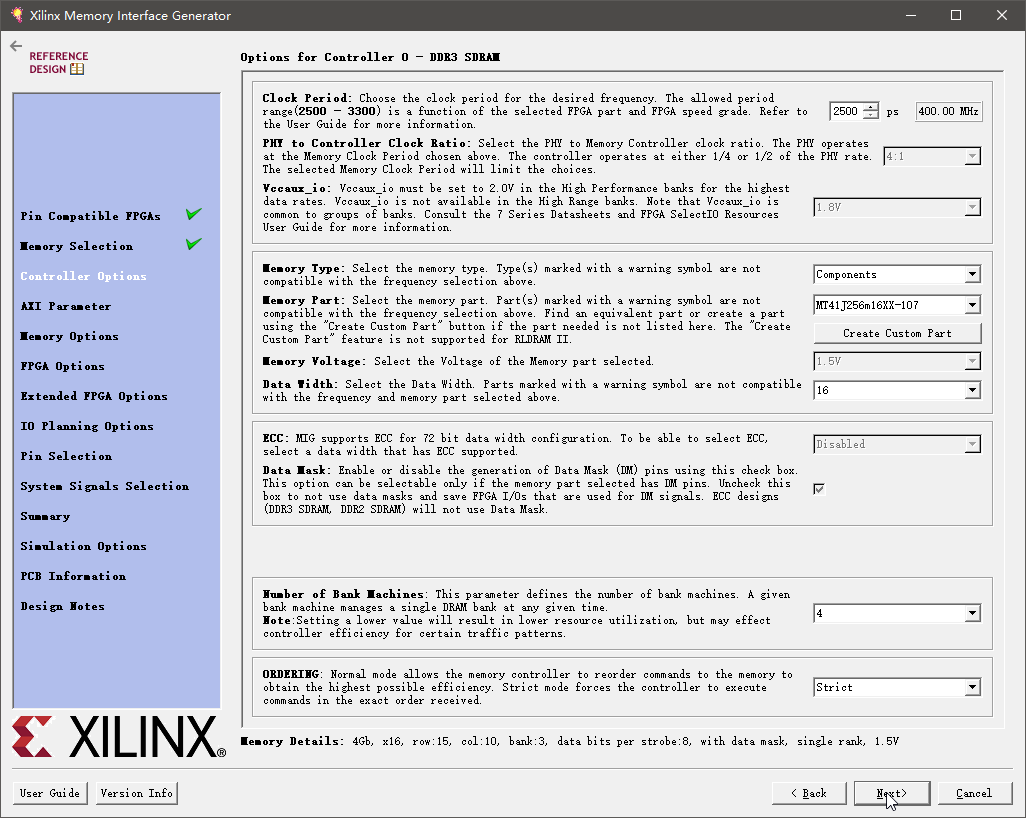


图2-28

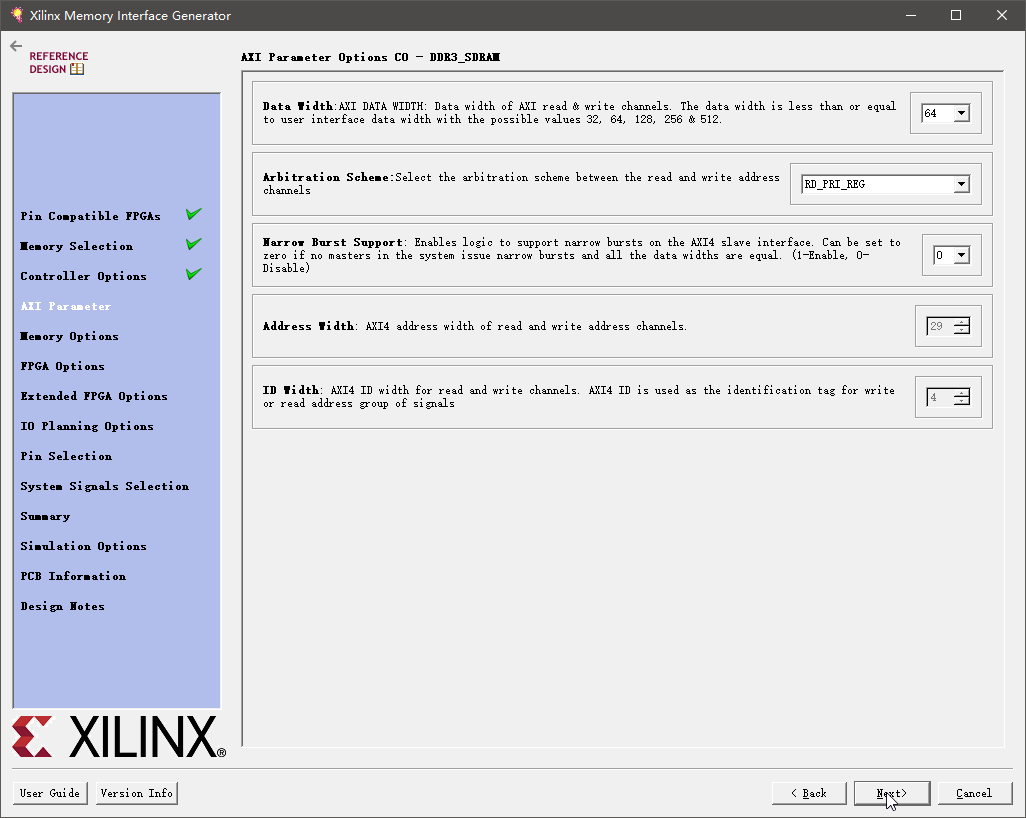


图2-29

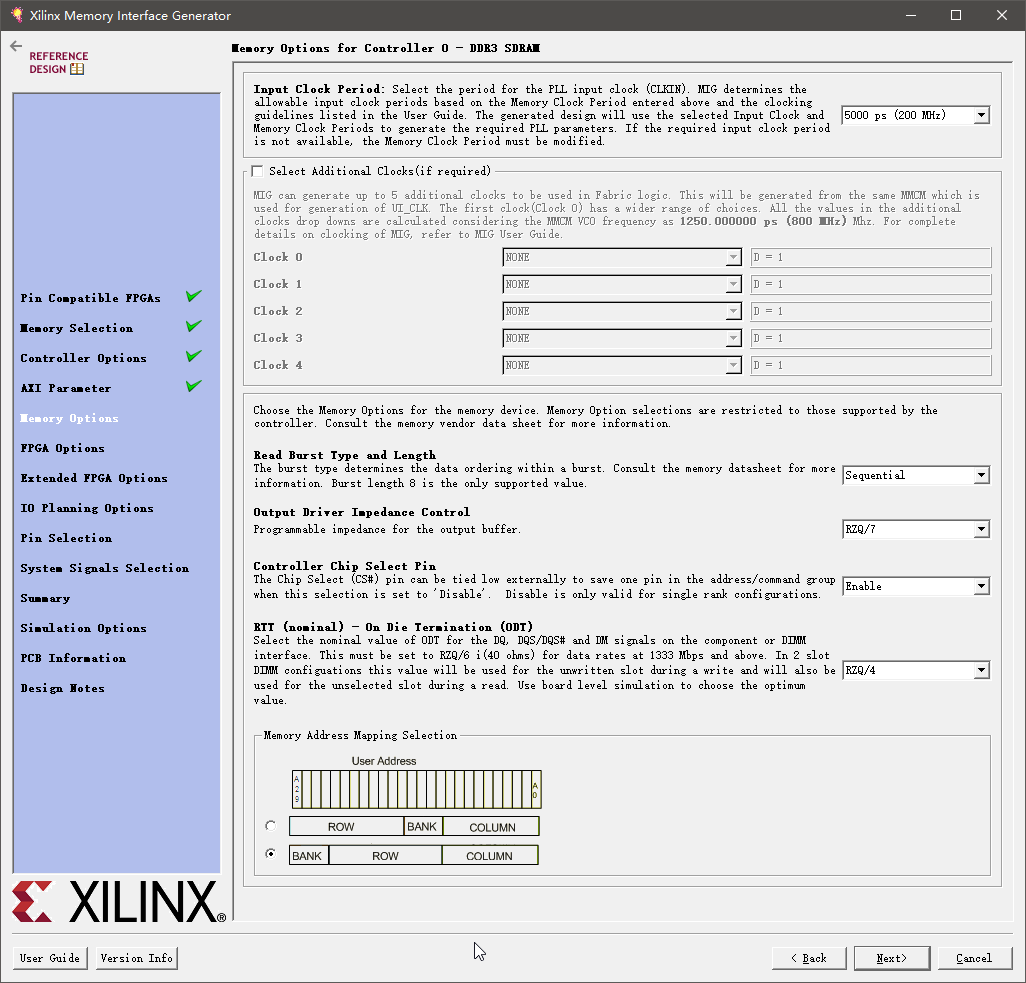


图2-30

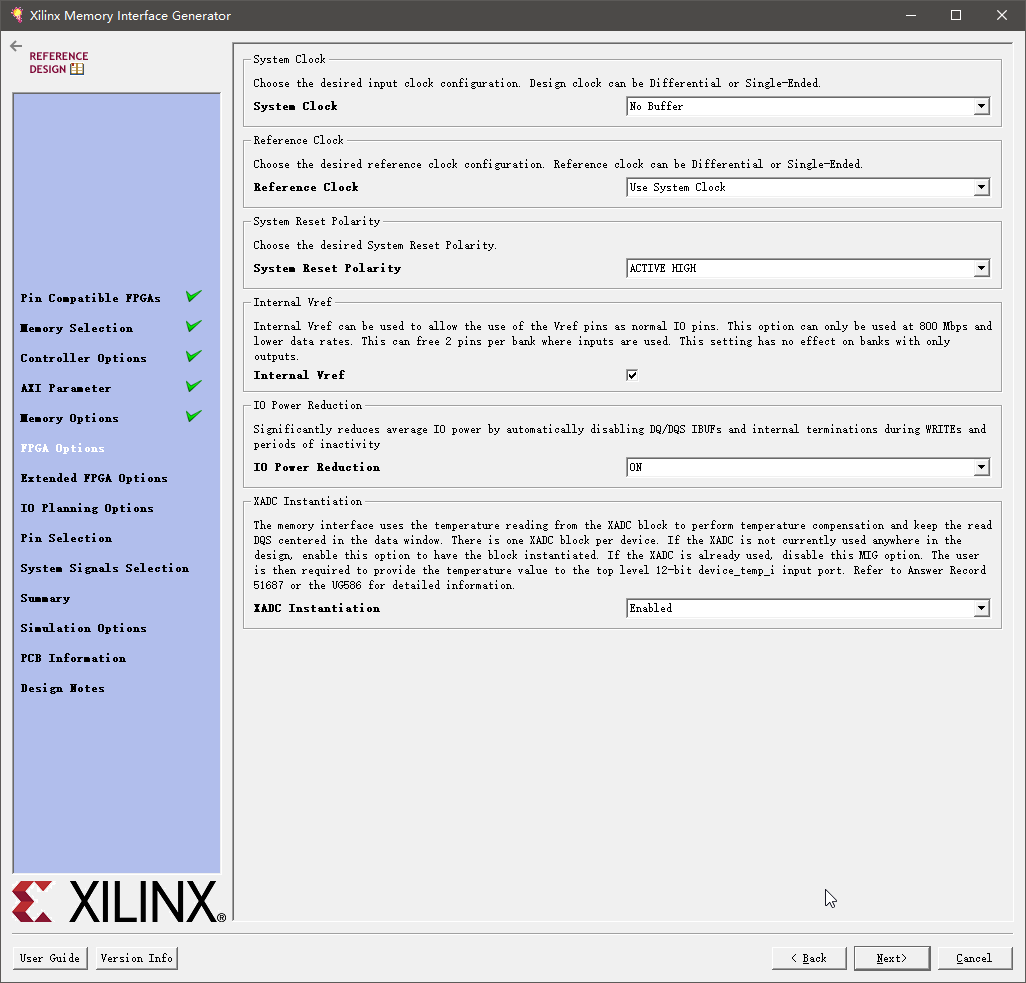


图2-31

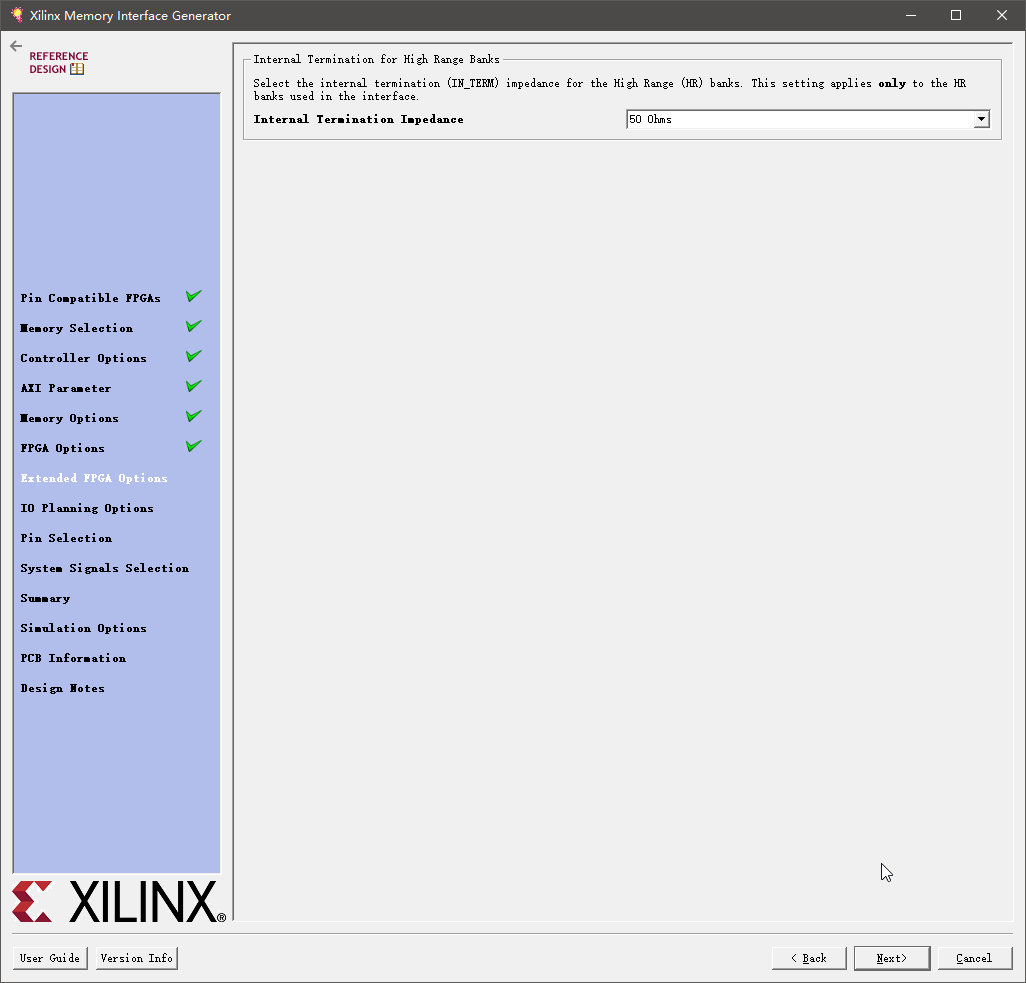


图2-32

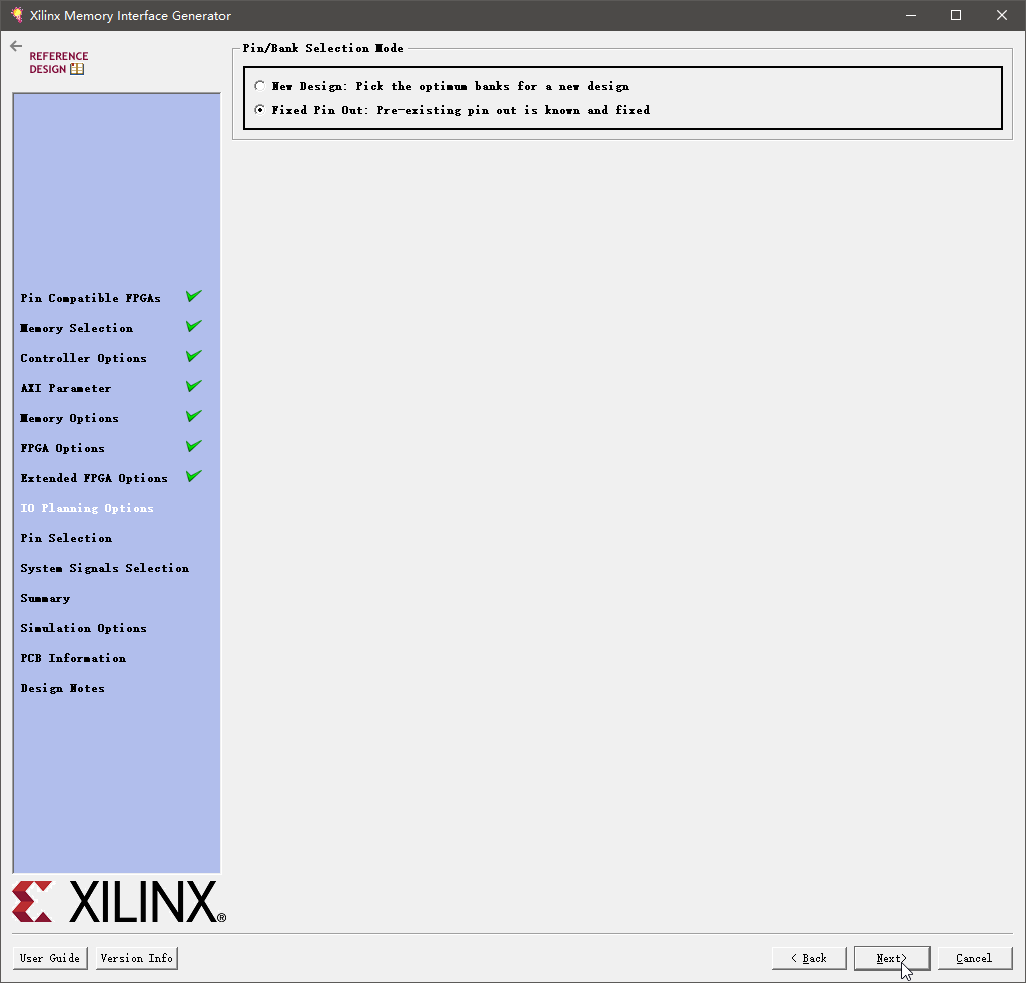


图2-33

在这一步你应该看到的是空白的。这个时候找到这份文件（MIPS\_FPGA/ constraints/ Minisys \_DDR\_Pin\_Map.ucf），用来约束Minisys上的DDR3 SDRAM，安排引脚。

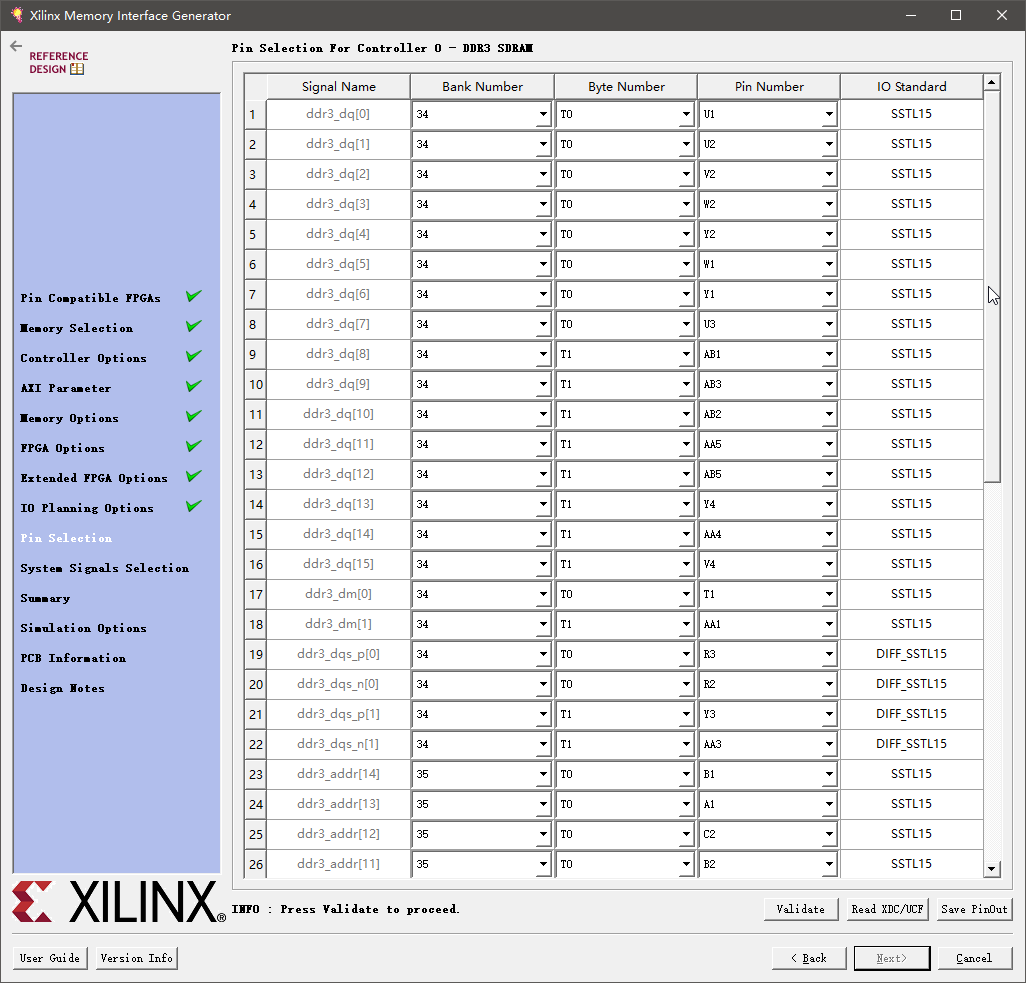


图2-33

根据下列步骤导入ucf。

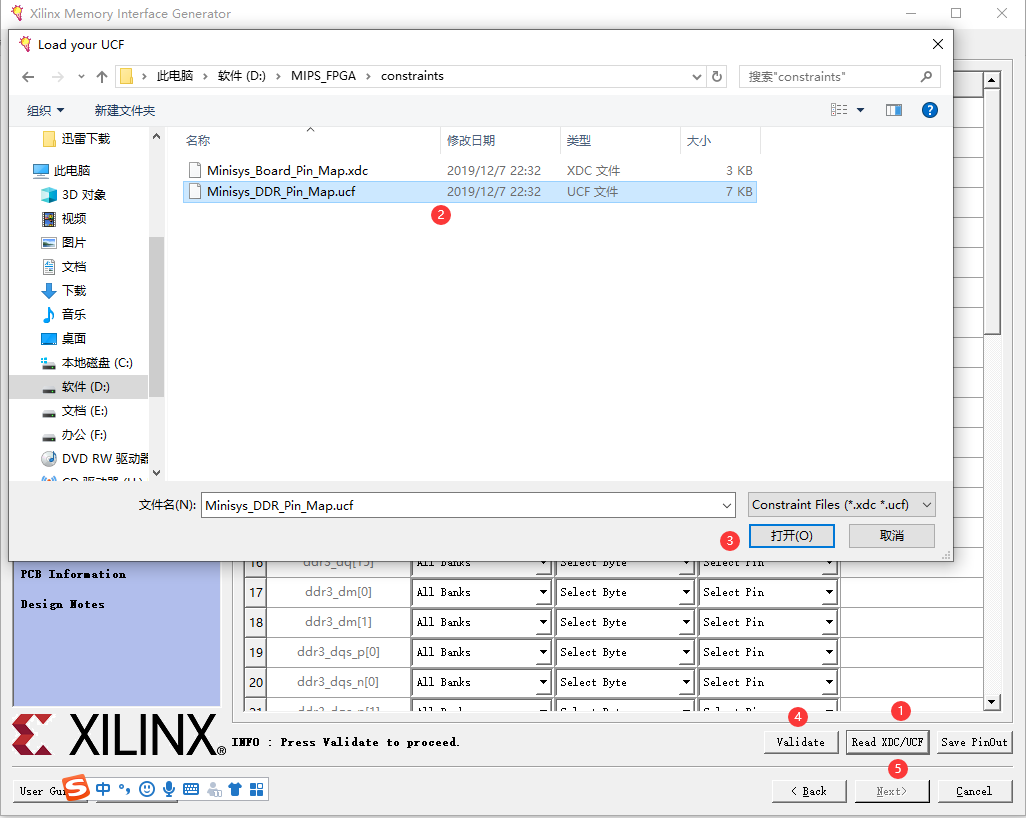


图2-34

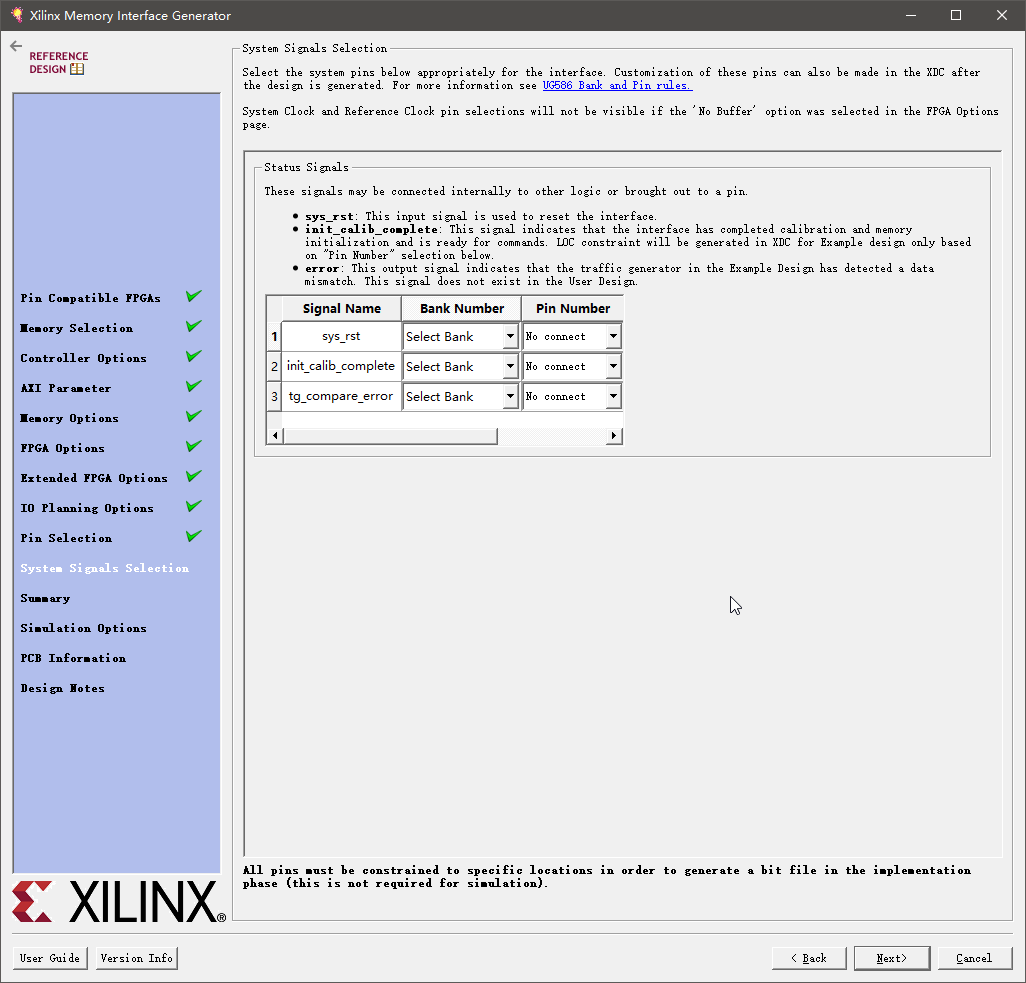


图2-35

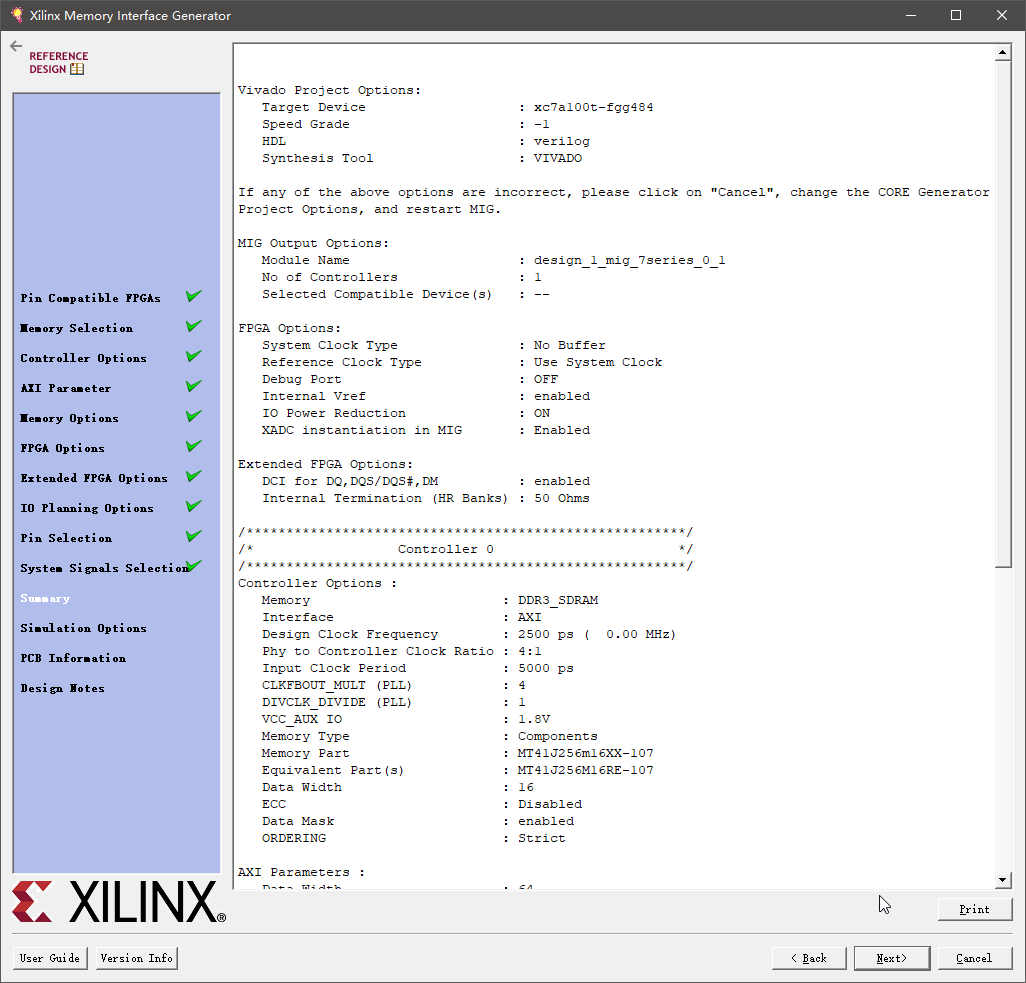


图2-36

之后没有需要的配置项，直接下一步到结束即可。

MIG引脚的连接：

* 添加完成后的MIG IP核如图2-37所示，其中S\_AXI与AXI Interconnect的一个Matser接口相连（本教程是M07\_AXI）。
* sys\_rst直接连在CPU\_RESET上即可。
* sys\_clk\_i与之前创建的时钟向导的clk\_out2（200MHz，MIG需要较高频率的参考时钟）。
* aresetn与其他AXI设备的复位信号直接相连即可。
* 在DDR3接口处创建输出端口，并重名为DDR3\_SDRAM，如图2-38所示。

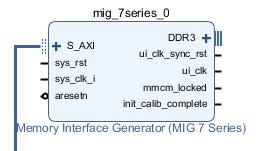


图2-37

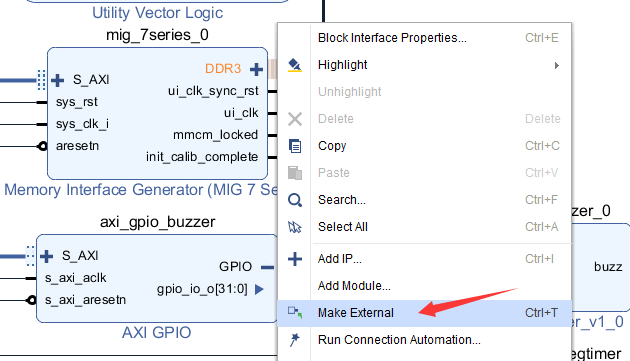


图2-38

* ui\_clk\_sync\_rst通过一个非门连接到AXI Interconnect的M07\_ARESET引脚。
* ui\_clk连接回AXI Interconnect的M07\_ACLK引脚。
* 剩下的两个引脚悬空即可。

至此，存储系统的构建完成。

### 2.3.5其他外设的添加

（1）UART

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter)，通常称作UART。它将要传输的资料在串行通信与并行通信之间加以转换。作为把并行输入信号转成串行输出信号的芯片，用于与上位机的通讯。本教程中使用Xilinx提供的UART16650 IP核进行串口通信的控制。

该IP的详细功能、使用方法以及样例代码可参考该官方文档：

<https://www.xilinx.com/support/documentation/ip_documentation/axi_uart16550/v2_0/pg143-axi-uart16550.pdf>

* Add IP，找到AXI UART16650，双击添加。
* 该IP不需要自定义，这里直接使用默认配置。
* 连接该IP核到AXI Interconnect，如图2-39所示，该IP的freeze引脚置为低电平（使用CONSTANT），将UART接口展开，导出sin和sout引脚。
* 本教程中UART未使用中断模式，因此中断信号悬空。

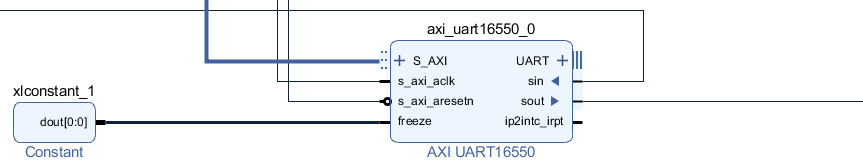


图2-39

（2）定时器

操作系统的运行离不开时钟中断，例如进程调度中的时间片轮转算法。定时器的作用就是产生时钟中断，本教程中使用Xilinx的AXI Timer IP核，该IP核可设置两个独立的定时计数器，读者可根据自己的需要进行配置。

该IP的详细功能、使用方法以及样例代码可参考该官方文档：

<https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf>

本教程的使用方法仅作为参考。

* Add IP，找到AXI Timer，双击添加。
* 该IP核可根据实际需要决定是否启用第二个计数器。以及计数器的最大值。如图2-40所示。
* 连接IP到AXI Interconnect，将该IP的interrupt信号引出，连接到一个8位的concat的某个引脚（本教程连接到0号引脚，这里最好参考一下MIPSfpga中断信号的引脚定义），这个拼接后的8bit信号，直接送到CPU的中断输入引脚。如图2-41所示。

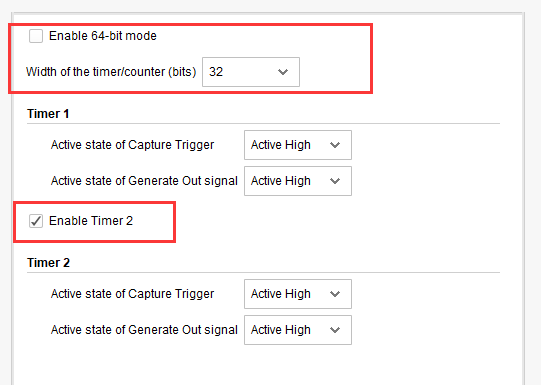


图2-40

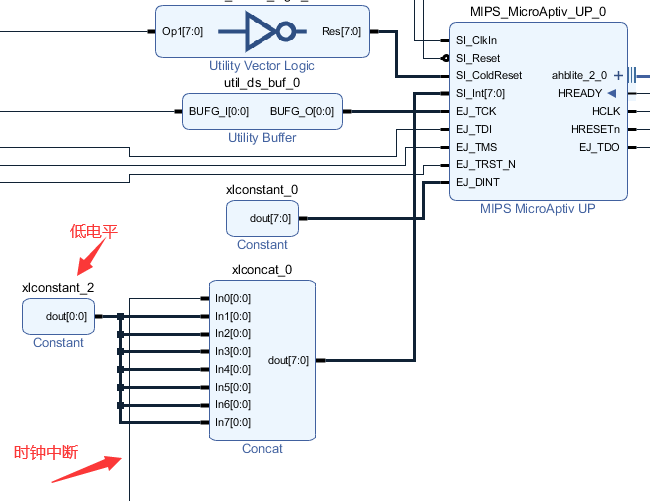


图2-41

（3）SD卡控制器

Minisys支持Micro SD卡的扩展，操作系统的镜像应放在SD卡中，由Bootloader加载到RAM中运行。SD卡的控制有SPI模式和SD模式，本教程选择了控制逻辑较为清晰的SPI模式进行SD的初始化以及控制。下面简单对SD卡的SPI模式进行简单的介绍。

首先介绍Micro SD卡的引脚定义，如图2-42所示。需要我们控制的有CS、MISO、MOSI、CLK。SPI模式通过对这四个信号的控制，我们可以实现对SD卡的初始化、读写操作。

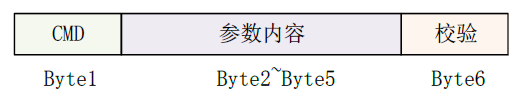


图2-42

SPI模式的命令类型与格式、返回数据格式、初始化时序和过程、读写时序和过程参见附录的内容。请读者阅读后再进行IP核的配置。

~~SPI模式是通过发送串行命令的方式，进行控制的。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，使其工作在预期的工作模式。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，因此SD卡的命令格式是学习SD卡的重要内容。SD卡的命令格式由6个字节组成，发送数据时高位在前，SD卡的写入命令格式如图2-43所示：~~

* ~~Byte1：命令字的第一个字节为命令号（如CMD0、CMD1等），格式为“0 1 x x x x x x”。命令号的最高位始终为0，是命令号的起始位；次高位始终为1，是命令号的发送位；低6位为具体的命令号（如CMD55，8’d55 = 8’b0011\_0111，命令号为 0 1 1 1 0 1 1 1 = 0x77）。~~
* ~~Byte2~Byte5：命令参数，有些命令参数是保留位，没有定义参数的内容，保留位应设置为0。~~
* ~~Byte6：前7位为CRC（循环冗余校验）校验位，最后一位为停止位0。SD卡在SPI模式下默认不开启CRC校验，在SDIO模式下开启CRC校验。也就是说在SPI模式下，CRC校验位必须要发，但是SD卡会在读到CRC校验位时自动忽略它，所以校验位全部设置为1即可。需要注意的是，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，进入SPI模式。所以在发送CMD0命令的时候，SD卡处于SDIO模式，需要开启CRC校验。另外CMD8的CRC校验是始终启用的，也需要启用CRC校验。除了这两个命令，其它命令的CRC可以不用做校验。~~

~~~~

~~图2-43~~

~~SD卡的命令分为标准命令（如CMD0）和应用相关命令（如ACMD41）。ACMD命令是特殊命令，发送方法同标准命令一样，但是在发送应用相关命令之前，必须先发送CMD55命令，告诉SD卡接下来的命令是应用相关命令，而非标准命令。发送完命令后，SD卡会返回响应命令的信息，不同的CMD命令会有不同类型的返回值，常用的返回值有R1类型、R3类型和R7类型（R7类型是CMD8命令专用）。SD卡的常用命令说明如图~~

Xilinx提供了AXI Quad SPI IP核来实现对SPI设备的控制。该IP的详细功能、使用方法以及样例代码可参考该官方文档：

<https://www.xilinx.com/support/documentation/ip_documentation/axi_quad_spi/v3_2/pg153-axi-quad-spi.pdf>

下面介绍该IP的配置方法。

* Add IP，找到AXI Quad SPI这个IP核并添加。
* 双击该IP进行自定义，按照图2-43进行配置。
* 连接该IP置AXI Interconnect IP核。
* 创建输入端口spi\_miso，输出端口spi\_sclk、spi\_cs、spi\_mosi。展开SPI接口，进行如图2-44所示的连接。
* ext\_spi\_clk为SPI模式的参考时钟，本教程使用50MHz，直接连接到时钟向导的clk\_out1即可。
* 本教程未使用中断模式，中断引脚悬空。

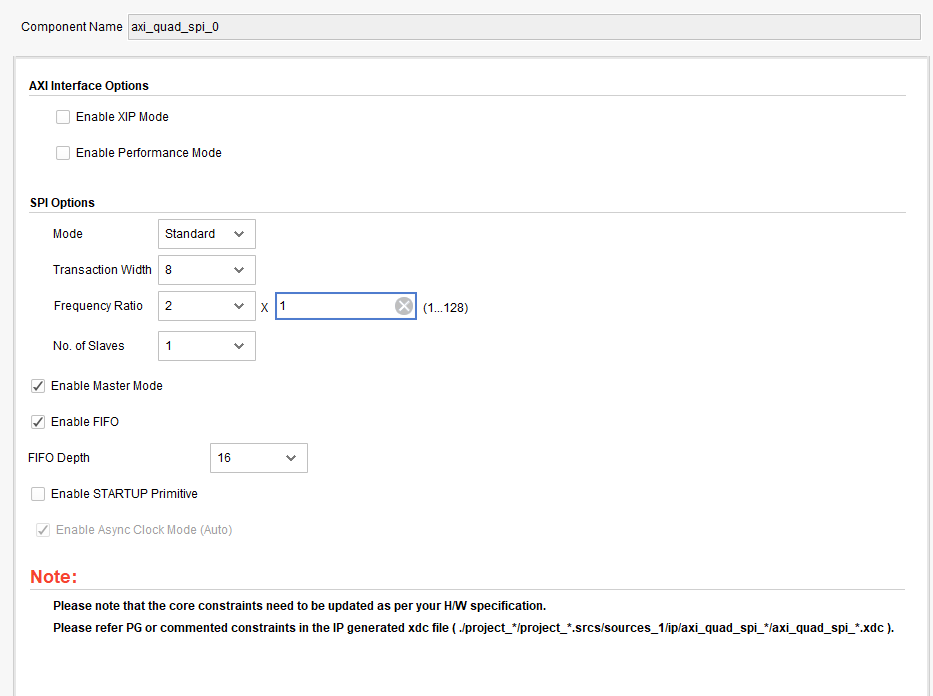


图2-43

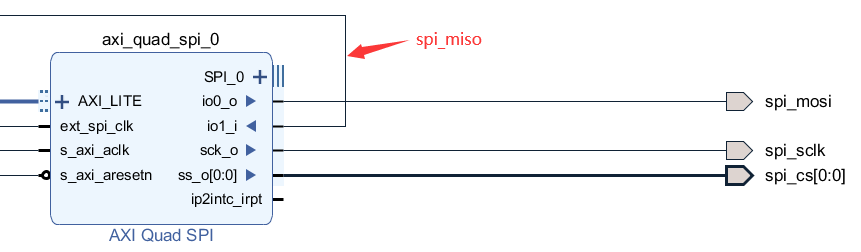


图2-44

### 2.3.6总线地址的分配

在添加完AXI设备之后，需要将这些设备映射到对应的**物理**地址上。回顾一下我们在1.3.1节中介绍的MIPS32 4G虚拟地址的划分规则，以及MMU映射策略。请读者思考一下，BRAM、DDR3、其他外设分别应该映射到4G的虚拟地址空间的哪一段？对应的物理地址是什么？

首先BRAM中的一个是用于存放Bootloader的，MIPS32的复位向量位于虚拟地址0xBFC0\_0000，对应物理地址为0x1FC0\_0000。另一个是Linux启动需要的，映射到虚拟地址0x9000\_0000，对应物理地址为0x1000\_0000。DDR3作为系统的主存储器，操作系统的内核放在其中，内核应起始于虚拟地址0x8000\_0000，对应物理地址为0x0000\_0000。kseg1（0xA000\_0000~0xBFFF\_FFFF）是内核段地址，由于该区域也不经过TLB映射，也不经过cache缓存，因此适合把外设（即外设的控制寄存器）映射到这个段地址中去。因此我们的外设，应该映射到这一段虚拟地址中，也就是物理地址0x0000\_0000 ~ 0x1FFF\_FFFF（注意物理地址应该互相隔离）。

本教程给出的地址分配如图2-45所示，读者可以参考进行分配。

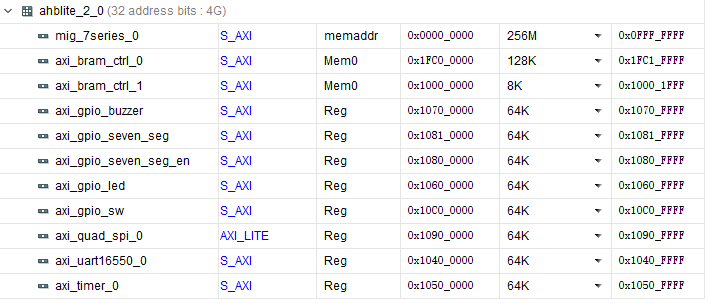


图2-45

## 2.4硬件工程的编译与验证

* 右键点击Block Design文件，选择Create HDL Wrapper，如图2-46所示，点击确定。
* 等待vivado自动生成HDL代码，就可以进行综合了。首先ctrl+s保存，然后点击左侧的Run Synthesis，等待完成。
* 综合完成后，点击Run Implementation，进行实现。

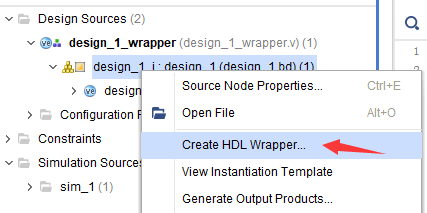


图2-46

* 实现完成后，现在导入约束文件。点击Add Source，选择Add or create constraints，如图2-47所示。然后点击下一步，创建约束文件。将教程中提供的约束文件内容复制进来（MIPS\_FPGA/ constraints/ Minisys\_Board\_Pin\_Map.ucf），然后保存。读者应注意自己的信号名是否与给的例程一致，若不一致应修改约束文件中的信号名。

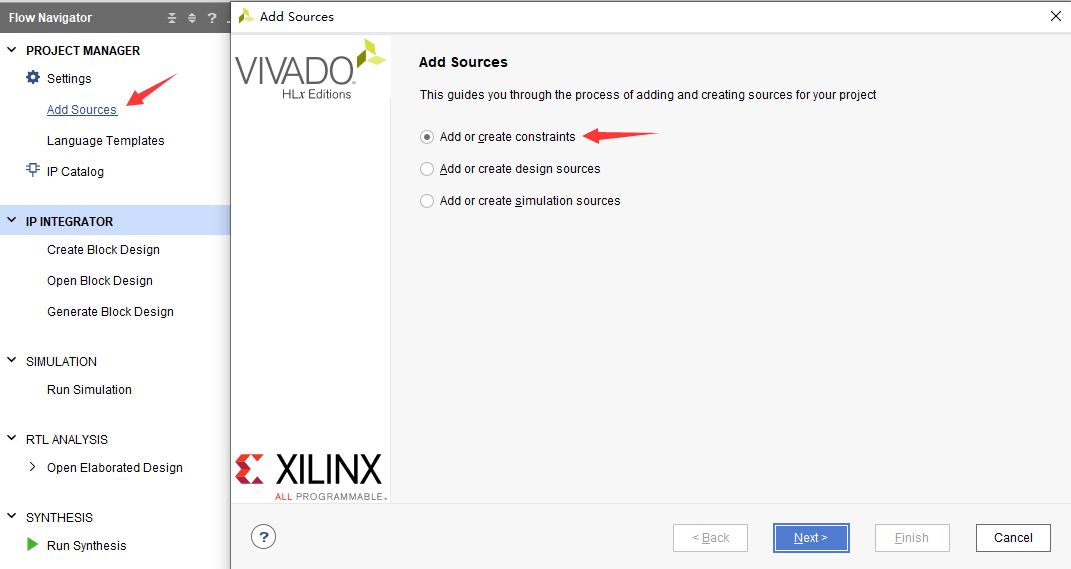


图2-47

* 在添加了约束文件后，就可以生成比特流了。点击Generate Bitstream，等待完成即可。
* 等待比特流生成完毕，点击Open Hardware Manager，选择Auto connect后，就可以进行下板测试了。选择生成好的bit流，点击program device。等待进度条完成后，首先将Minisys实验板的串口用Type-C下载线与PC的USB接口相连，并且打开实验板的电源开关。然后打开PC的设备管理器，找到端口(COM和LPT)，找到Minisys所在的端口，如图2-48所示，可以看出Minisys所在的端口号为COM4。紧接着打开putty，如图2-49所示，选择Serial，输入端口号为COM4，并且设置波特率为115200，然后点击Open即可开始与系统进行通信。
* 打开PuTTY后，按下Minisys开发板的复位键，此时串口循环打印MIPS FPGA，LED灯循环点亮即证明MIPSfpga-SoC移植成功（需要BRAM中加载教程提供的.coe文件）。



图2-48 Minisys的端口号

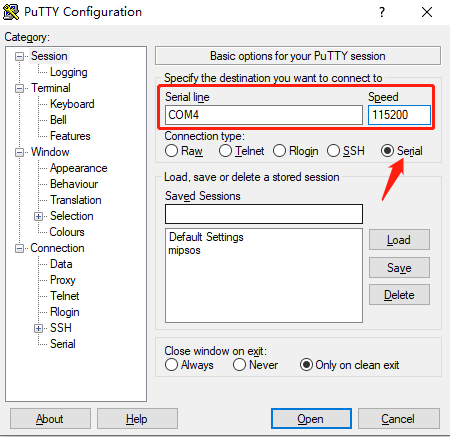


图2-49 Putty设置界面

## 2.5 操作系统工程的体系结构

本小节主要给同学们梳理一下整个操作系统如何“从零到一”的整体架构，是具有全局指导意义的一小节，**十分重要**！ 在图2-50中，

我们可以看到最底层是FPGA的硬件资源，在这一层中我们首先给同学们提供了-一些

图2-50 OS工程体系结构

IP核，因为这几部分的硬件时序比较复杂，就不要大家手写了。但是为了使大家熟悉Verilog语言，还是有一些简单的硬件控制器需要大家亲自写，比如非常经典的4\*4键盘、数码管控制器、VGA显示屏控制器（拓展内容）等。

在硬件资源上面一层，就是硬件抽象层，或者是类似Linux使用的板级支持包（BSP），这一部分是使用C语言编写的驱动程序，所有文件不论是写好的还是需要大家补充完整的，都在./drivers目录下面，用来处理基本输入输出系统（BIOS）。

从第三层的bootload开始，按照“微内核-模块化”的设计思路都可以视作操作系统内核代码，之后的每一个章节就是对应的每个子模块的讲解，在这里就不展开了。

值得注意的是，这次的操作系统实验并不是给大家一个白板，而是给出了一定量的代码框架让大家填充，发给大家的软件代码包文件夹中各子文件的描述如下：

1) boot:bootloader 代码

2) drivers:设备驱动

3) env:进程管理相关

4) fs:文件系统相关

5) inc:include 头文件

6) init: main()函数和初始化函数

7) lib:一些外部库

8) mm:内存管理相关

9) makefile:最顶层 makefile

10) MicroAptiv UP Software User's Manual MD00942 read.pdf: CPU 的说明书

11) run.bat: 启动 openocd-gdb 的脚本

12) scse0\_3.lds: 链接脚本

13) ushell: 用户命令行相关代码

读者根据以下每个章节中的引导，将对应的代码文件补充完整即可实现一个功能完整的Aurora操作系统。对于每个文件夹下面每个子文件的进一步介绍，我们将放在对应的章节里面来阐述。

最后请读者们注意：对于最为完整的寄存器使用准则，请参阅附件中的《MicroAptiv UP Software User's Manual.pdf》（下文简称Manual），因为这是针对MIPS cpu 核所写的硬件架构方面的规范，而本教程中所列举出的寄存器仅仅是可能要用到的内容。

# 第三章 从BIOS到Bootload

上一章节结束了关于SOC所有硬件方面的操作，本章节开始则正式进入OS**软件**实验部分，所谓软件实验，即所有内容都将以高级语言(C语言)或者汇编语言的形式来展现。

在本章中，首先给大家介绍了一套工具链的使用，他们的作用就是把我们写的一段段的程序文件整合成一个完整的操作系统内核，并且最终以二进制位流的形式传送到FPGA板子上。其次，介绍了BIOS的软件代码编写，主要包括的就是基本的输入输出设备的初始化操作和寄存器用法。最后，我们介绍整个操作系统最初的起点：自举程序(bootload)的实现，之所以采用这样的先后逻辑是因为想让同学们感受打通外围部件互联（PCI）的过程，外部设备严格的来说属于微机接口技术，但是操作系统对于外设的控制也是需要关注的。

## 3.1工具链的使用

### 3.1.1 MIPS交叉编译器的使用

（1） Makefile的设计

Makefile是自动化编译工具，如果读者对Makefile不熟悉，请先自行学习关于Makefile的相关知识。这不是本教程的重点，但对于OS开发十分重要，对程序员来说也是一个基本技能。读者可以自行设计编写Makefile，也可以参考代码仓库中的Makefile的写法来完成。

（2）链接脚本的设计

链接脚本（ld脚本）是将目标代码链接到指定的虚拟地址的工具，需要配合Makefile一起使用。同样的，读者可以自行设计编写链接脚本，也可以参考代码仓库中的链接脚本的写法来完成。

### 3.1.2 GDB调试

GDB（GNU symbolic debugger）简单地说就是一个调试工具。它是一个受通用公共许可证即GPL保护的自由软件。

像所有的调试器一样，GDB可以让你调试一个程序，包括让程序在你希望的地方停下，此时你可以查看变量、寄存器、内存及堆栈。更进一步你可以修改变量及内存值。GDB是一个功能很强大的调试器，它可以调试多种语言。在此我们仅涉及 C语言的调试，而不包括其它语言。还有一点要说明的是，GDB是一个调试器，而不像 VC 是一个集成环境。

下面是我们搭建的Windows环境下，启动GDB并运行、调试程序的过程。

* win+R打开cmd控制台黑窗，输入以下命令，回车

cd C:\Program Files\Imagination Technologies\OpenOCD

openocd-nexys4.bat

* 此时应该弹出图3-1所示的窗口。
* 这时进入MIPS\_FPGA目录，找到Starter\_Tutorial并进入，在图3-2的位置输入cmd并回车，启动一个新的控制台。
* 在新的控制台输入以下命令并回车：

“C:\Program Files\Imagination Technologies\Toolchains\mips-mti-elf\2017.10-05\bin\mips-mti-elf-gdb.exe” -x startup.txt

* 这时应出现图3-3所示的界面，按下Ctrl + C，此时出现“(gdb)”字样，就可以输入GDB命令了。
* 我们提前把要load的编译好的elf文件复制到Starter\_Tutorial目录下，输入GDB命令：load xxx（xxx为elf文件的文件名）并回车，此时elf镜像会被加载到内存中。如图3-4所示。
* 如果需要调试，则需要输入file xxx（xxx为elf文件的文件名）来加载symbols，加载完成后可以输入b命令来设置断点。
* 这时输入continue即可运行加载到内存中的代码到下一个断点处了。
* 常用的GDB命令参考网上的教程，本教程中常用的命令有info reg、b、continue、s、n等等。

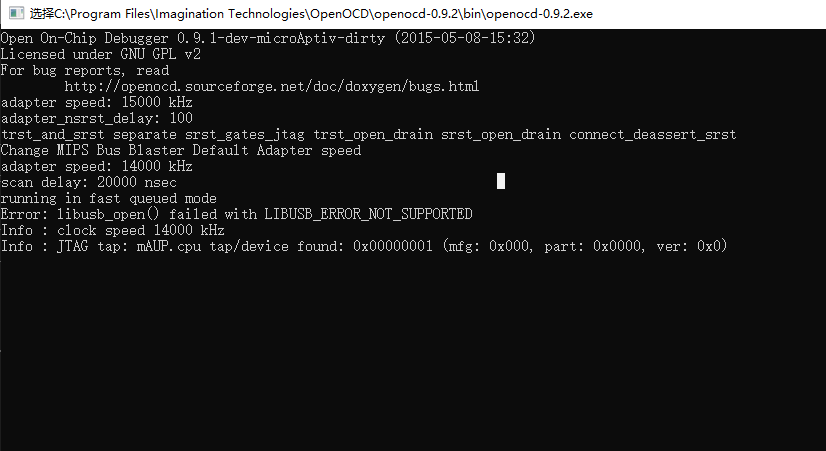


图3-1



图3-2

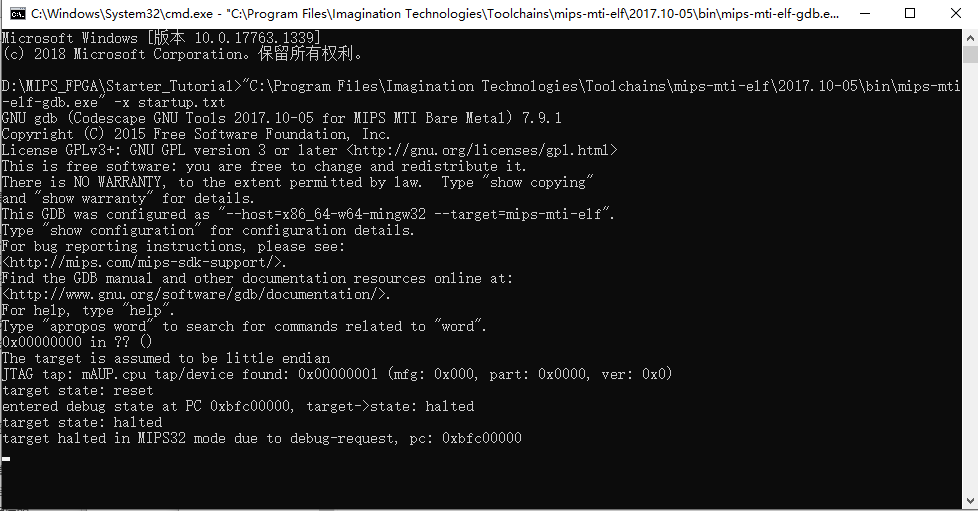


图3-3

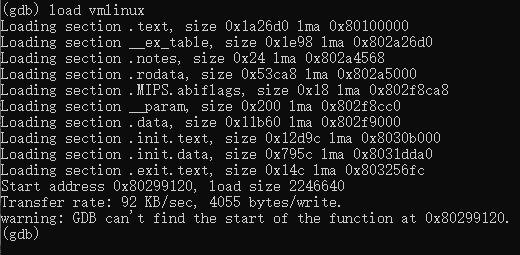


图3-4

## 3.2驱动程序的设计与实现

根据本教程的硬件系统设计，操作系统的设备驱动程序至少要支持外围部件：LED、拨码开关、蜂鸣器、串口、定时器、七段数码管以及SD卡的驱动。本模块的软件代码实现在./drivers 文件夹下面，子文件的名字就对应它所控制的外设的名字，非常容易理解就没有必要一一介绍了。下面介绍以下几个外设相关的控制逻辑：

（1）GPIO设备驱动

硬件中通过GPIO控制的外设有LED、拨码开关、蜂鸣器和七段数码管， LED与拨码开关没有控制逻辑，蜂鸣器与七段数码管得控制逻辑已经在硬件IP核中实现，因此这些外设可以直接通过对GPIO的读写来实现控制与访问。其读写控制函数如表3.1所示。

表3.1 GPIO外设读写控制函数

|  |  |
| --- | --- |
| 函数名 | 功能描述 |
| set\_buzzers  set\_leds  get\_switchs  set\_seven\_seg\_value  get\_seven\_seg\_value  set\_seven\_seg\_en  get\_seven\_seg\_en | 设置蜂鸣器频率  设置24个LED的状态  获取24个拨码开关的状态  设置七段数码管的显示  读取七段数码管的显示  设置七段数码管的使能  获取七段数码管的使能 |

（2）定时器驱动

定时器在Bootloader中是用不到的，但是对于OS的开发来说十分重要。进程调度的时间片算法依赖于设置外部的时钟中断。写在这里供读者参考。

定时器Timer的控制逻辑较为复杂，因为其端口寄存器较多，控制逻辑也更为复杂。Timer IP核提供了两个定时器Timer0以及Timer1，本教程只用到了Timer0。Timer0拥有端口三个寄存器，如表3.2所示。

表3.2 Timer0端口寄存器

|  |  |
| --- | --- |
| 寄存器名 | 功能描述 |
| TCSR0  TLR0  TCR0 | Timer0控制和状态寄存器  Timer0加载寄存器  Timer0计数寄存器 |

其中TCSR0的格式如图3-5所示，

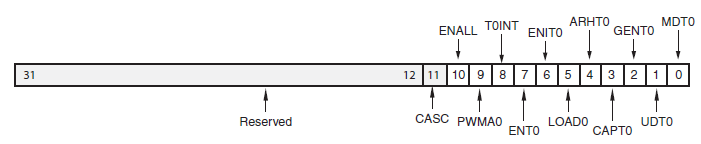


图3-5 TCSR0的格式

本教程的定时器初始化和设置需要用到的字段有ENT0：第7位，置为1表示使能Timer0。ENIT0：第6位，置1为表示使能Timer0的中断信号。ARHT0：第4位，置为1表示Timer0每次计数完成后会自动加载计数值。计数值位于TLR0中。

驱动程序在初始化Timer0时，首先清空TCSR0寄存器，接着设置上述字段为1，目的是使能Timer0、中断和自动加载计数值。然后设置用户输入的计数值到TLR0中，即可完成对Timer的驱动。每当Timer0计数到TLR0中的数时，就会产生中断信号。

（3）串口驱动

Xilinx的UART 16650 IP核中共有13个端口寄存器。本教程对串口的驱动用到了如表3.3所示的五个寄存器。

表3.3 UART16650端口寄存器

|  |  |
| --- | --- |
| 寄存器名 | 功能描述 |
| RBR  THR  LCR  DLL  DLM  IER | 数据接收缓冲寄存器  发送保持寄存器  线路控制寄存器  除数锁存（最低有效字节）寄存器  除数锁存（最高有效字节）寄存器  中断使能寄存器 |

UART的驱动逻辑是，首先对LCR、DLL以及DLM三个寄存器进行初始化，初始化的代码如图3-6所示。查询Xilinx的手册，DLL的作用是设置波特率的除数最低有效字节，其值的计算方法如公式3.1所示：

DLL = Faxi / (16 \* baudrate) （3.1）

Faxi表示AXI总线的时钟频率，baudrate指的是要设置的波特率。本教程的波特率为115200，AXI总线时钟频率为50MHz。因此初始化设置DLL的值为：50 \* 106 / 16 / 115200 = 27

这个值没有超过8bit，因此DLM需要设置为0。

本教程的串口不使用中断方式，因此初始化时设置IER为0。

初始化完成后，就可以通过RBR和THR进行串口数据的收发了。至此，串口的驱动设计就完成了。

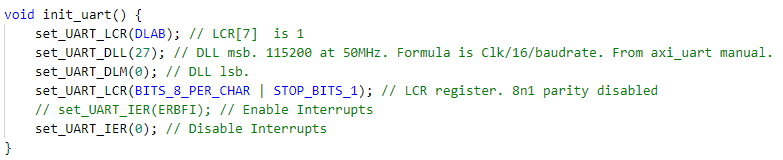


图3-6 串口驱动初始化

**请读者根据给出的串口驱动程序，自行设计一个简单的printf函数，该函数可以实现通过串口打印一个字符串和一个参数到PC的PuTTY中显示出来。**

（4）SD卡驱动（已经帮大家做好了）

SD卡的驱动程序较为复杂，大致可以分为：

* 对AXI\_QUAD\_SPI IP核的驱动
* 对SD卡SPI模式的初始化以及读写的驱动
* 对SD卡文件系统FAT32的驱动

其中比较难的部分是第二部分，即三个函数：disk\_initialize, disk\_read, disk\_write.请读者仔细阅读附录中对SD卡SPI模式的介绍后，根据已有的代码和提供的函数模板进行相关代码的编写。

## 3.3 Bootloader的实现

Bootloader有两大功能：boot和load。其中，boot的主要功能是将软硬件进行初始化，例如设置通用寄存器值、初始化堆栈、初始化TLB、初始化cache等；load的功能就是要将操作系统的内核代码从SD卡加载到内存。

操作系统自举相关代码在 ./boot 文件夹下，首先分别介绍一下子文件：boot.h是一些体系结构相关的宏定义，不需要大家写。Start.s 是中断相关的一些初始化，这个会在下一章异常处理提及。四个init\_ 打头的文件分别初始化了对应的内容，boot.s 是cpu总初始化，整合了子模块的初始化。

读者在Bootloader实验中，需要实现的功能有两个：

* 完成初始化TLB代码
* 完成load\_elf代码

值得注意的是在 TLB 初始化时需要给 EntryHi 寄存器不同的值，不可给 0，因为 EntryHi 代表了虚拟地址，写重复值会触发 TLB项的重复的24号Exception，具体可以看内存管理和异常处理部分。另外此实验操作系统在 Bootloader 结束，进入内核 main 函数之前，将 CP0\_Status 寄存器设为了 0x00007c01。CP0\_Status 寄存器 直接影响了处理器处于用户态还是内核态，以及 CP0 寄存器相关的指令是否可以被使用，具体每一位的含义可以看 Manual p153，下面列举几条关键信息：

1) CPU 处于内核态的条件为：UM=0 或 EXL=1 或 ERL=1；

2) 用户态条件为：UM=1 且 EXL=0 且 ERL=0；

3) 用户态访问 0X80000000 以上地址时会触发地址越界的 Exception。

4) Status 寄存器 CU0 位的值为 1 或者 CPU 处于内核态时，访问 CP0 的指令（比 如 mfc0、mtc0）不会报错，否则会触发异常。

5) 进入中断或异常时，EXL 位会被置为 1，即进入内核态

6) 中断屏蔽的条件为 IE=1 且 EXL=0 且 ERL=0 且 DM=0，其中 ERL 和 DM 几乎 不会去管。这里的中断指的是计数器等硬件中断，不包括 TLB 异常这些。

7) eret 指令会将 EXL 清零。

通过以上信息，我们可以设计 Status 的状态变迁如下：

1) 进入内核 main 时，UM=0, EXL=0,CU0=0,IE=1：内核态，开启中断，可以访问 CP0（这里其实可以 UM=1,EXL=1，等到内核的所有初始化结束后，将要进入第一次用户进程调度时，令 EXL=0，开启时间中断）

2) 每一个异常或中断处理结束后，令UM=1,IE=1,CU0=0,如果不是嵌套异常，则 使用 ERET 将 EXL 清零，返回用户态，如果时嵌套异常，则使用 J 指令，保持 EXL=1，内核态。 按照以上设计，用户程序始终处在用户态，并且 CU0=0，其行为不能损害 OS。

### 3.3.1初始化TLB

在初始化TLB之前，首先应读取Config寄存器和Config1寄存器，来读取TLB的size。然后对每一对虚拟地址入口，写入不同的物理页帧号。同时需要设置的寄存器有EntryLo0、EntryLo1、EntryHi、Index、Pagemask。应当注意的是，应保证每个物理页帧号无冲突，建议使用8k作为间隔。

初始化TLB的**汇编**函数模板如图3-7所示，请读者根据自己的理解，将其补充完整。

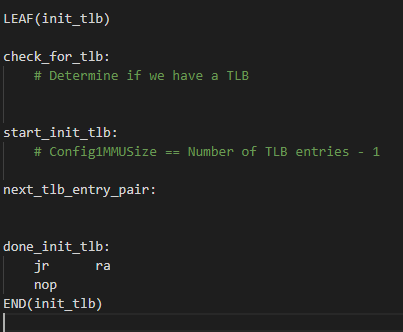


图3-7

### 3.3.2 Load elf文件

在加载elf文件到内存之前，首先需要了解elf文件的结构。我们编译好的elf文件由四个主要部分构成elf文件头Elf32\_Ehdr、程序头表Elf32\_Phdr、段表Elf32\_Shdr、符号表Elf32\_Sym。这些数据结构可以在 ./fs/elf.h中找到定义。

文件头Elf32\_Ehdr位于elf文件的起始位置，它包含整个文件的静态信息。其结构如图3-8所示。

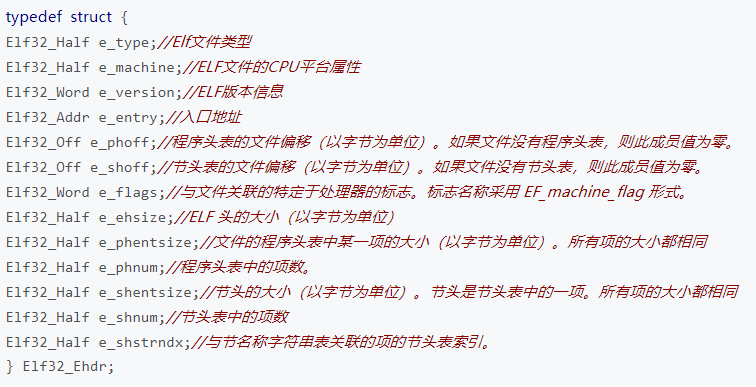


图3-8

程序头表Elf\_Phdr描述与程序执行直接相关的目标文件结构信息。用来在文件中定位各个段的映像。同时包含其他一些用来为程序创建映像所必须的信息。其数据结构如图3-9所示。

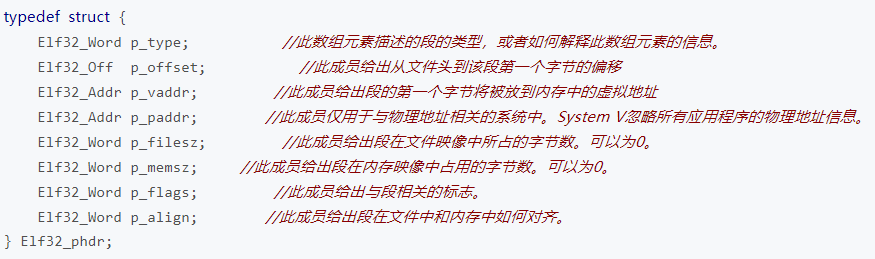


图3-9

本教程中使用上述两个elf文件的结构完成将elf镜像解析成可执行的二进制指令并加载到RAM中。

首先，在加载elf文件之前，elf文件已经被我们从SD卡通过FAT32文件系统的读操作完整的加载到内存的一块缓冲区中。这部分代码已经给出，如图3-10所示。

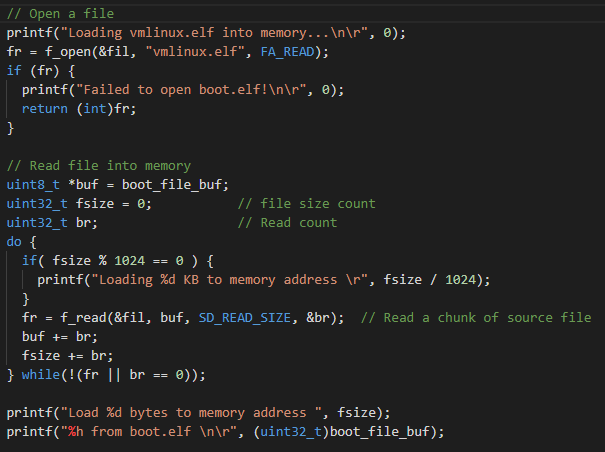


图3-10

那么这个缓冲区的起始地址就是elf文件头Elf32\_Ehdr的地址，根据该地址，我们可以得到e\_phoff，即程序头表的偏移地址（相对于起始地址）；以及e\_phnum，即程序头表的个数。接着我们可以根据这两个参数，顺序读取每个程序头表的信息：需要读取该段的物理地址、偏移和该段的字节数。使用提供的memCpy函数将各个段加载到其物理地址中。值得注意的是，如果内存映像字节数大于文件映像的字节数，那么超出的部分内存应该被置为0。

在学习了elf文件的格式后，读者需要根据如图3-11的函数模板完成load\_elf函数。

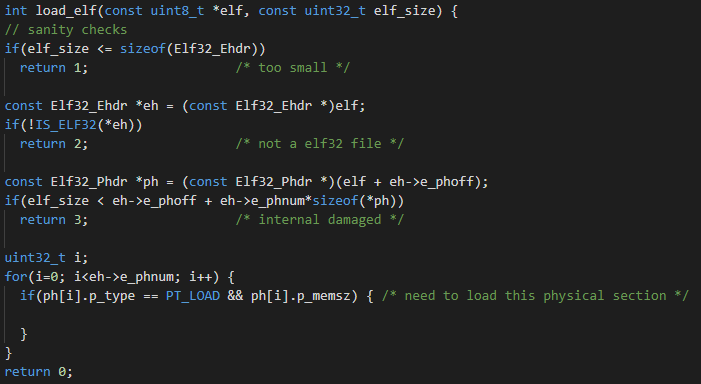


图3-11

在加载elf到内存中后，就可以跳转到入口地址执行了。读者可以自行编写代码，编译生成elf镜像文件，保存到SD卡。然后尝试其是否可以正确的被执行。

需要注意的是，SD卡中的elf的程序虚拟地址应保持在kseg0段，也就是内核所在的虚拟地址。因为该虚拟地址是不需要TLB映射的。如果读者想尝试将代码映射到用户段虚拟地址，则需要建立页表映射机制来进行TLB的填写。这就涉及到更进一步的页式内存管理的操作系统开发，读者将会在后续实验中去探索。

# 第四章 异常处理的设计与实现

异常处理是操作系统的核心之一。在本MIPS体系结构之中，我们使用广延意义上的异常：一方面指涉由外部硬件信号触发的中断，如定时器发出的时间片中断，用来完成round robin进程切换；另一方面指一些软件异常，比如地址越界或者进行系统调用等等需要陷入内核态的行为，统一作为第二类异常。之所以做这样的区分，是因为前者会带来中断屏蔽，不可能嵌套，而后者可能会产生中断嵌套，因此提醒大家需要根据这两种类型设计不同的保存上下文寄存器的方法。

值得注意的是，之所以把本章列在靠前的位置，是因为异常并不是一个独立的东西，操作系统本质上是一个异常/中断 驱动的系统，因此各种异常起到一个函数头的作用，由此引出之后一系列函数的执行，因此本章放在bootload完成之后，有着统领整个操作系统运行的重要意义，后面的章节就与之呼应。

## 4.1 上下文的保存与恢复

本部分代码需要直接操作寄存器，因此使用汇编语言完成。提供的代码在./lib/genex.S 文件中。需要注意两种类型的异常：对于时间中断(SAVE\_TF 和 env\_pop\_tf函数)，时间中断时，需要将上下文（寄存器内容）保存到每个进程独有的空间去，这个上下文可能在下次这个进程被调度时才会被取出来。因此我们选择将其存在了 内核中的 ENVS 的当前进程的内存中去，并没有存在 0X8000 0000 以下地址，是 为了避免中间触发缺页异常。并且保存上下文后我们会将 SP 指针指向一个固定 位置：0X8040 0000,这是内核的栈的地址，这一举措也是为了避免访问进程自己的 不在内核空间的栈，避免触发缺页异常。

对于TLB等异常（SAVE\_ALL和 ret\_from\_exception函数），它们所保存的上下文在异常处理结束时就会被恢复回去，因此我们采 用了内核的栈（0X8040 0000）来保存这些信息。同时考虑到这些异常会嵌套，因 此我们需要在进入异常时判断是否是嵌套异常，如果不是的话，就把 sp 从用户的 栈转到内核的栈，如果是的话，就保持 sp 不变。 需要注意的是，Mips 的 cp0\_EPC 寄存器不能保存嵌套的触发中断的地址， cp0\_nestedEPC 可以。我们目前是采用判断 nestedEPC 寄存器是否在 0X8000 0000 以上这个方法来判断是否是嵌套的，可以兼容内核初始化时的情况。 另外在中断返回时，嵌套中断需要用 j，非嵌套需要用 eret，因为 eret 会将 CP0\_status 的 EXL 位置零。

## 4.2 异常类型判断

在学习中断的时候我们知道，处理中断一种比较简洁的方式就是使用中断向量表来跳转到异常处理程序。每当有异常中断发生的时候，操作系统会自动进入一个用于判断异常中断类 型的程序，这个程序的作用就是检测发生了哪种异常，并调用相应的异常中断处 理程序。对于一般处理器而言，这个程序会被要求放在固定的某个地址上，以保证处理器能够在检 测到异常时正确跳转到那里，在我们设计中这个固定地址为 0x80000000。需要同学们完成的代码块在 ./boot/start.S 中 ,是用汇编实现。

\_mips\_general\_exception: # 中断处理地址

sw sp, kernel\_sp

mfc0 k1,CP0\_CAUSE

la k0, exception\_handlers

andi k1,0x7c #取出中断号

addu k0,k1

lw k0,(k0) #把内存中 k0 地址处的值赋给 k0

nop

jr k0 #跳到对应中断处理地址

nop

主要工作是从 cp0\_cause 寄存器中取出第 2-6 位的 Exc Code 及中断号，将其 作为索引从 exception\_handlers 数组作为基址中找到对应的中断处理函数，最后跳 转到对应的中断处理函数中，从而完成了异常中断的响应。

## 4.3 中断向量表

上一段所提到的 exception\_handlers 正可以被理解为中断向量表，其中存储着我们 所实现的中断处理程序入口函数地址，本部分代码在 lib/traps.c 中，专门用来处理不同类型的异常：

void trap\_init() {

int i;

for (i = 0; i < 32; i++) {

set\_except\_vector(i, handle\_reserved);

}

set\_except\_vector(0, handle\_int);

set\_except\_vector(1, handle\_mod);

set\_except\_vector(2, handle\_tlb);

set\_except\_vector(3, handle\_tlb);

set\_except\_vector(4, handle\_addr);

set\_except\_vector(5, handle\_addr);

set\_except\_vector(8, handle\_sys);

set\_except\_vector(24, handle\_mult\_tlb);

}

本操作系统主要实现以下几种异常处理：

0 号异常对应的处理函数为时间中断处理函数 handle\_int；

2 号异常对应的处理函数为 tlb 缺失中断处理函数 handle\_tlb；

3 号异常对应的处理函数为 tlb 缺失中断处理函数 handle\_tlb；

4、5 号异常对应的处理函数为访问地址越界中断处理函数 handle\_addr；

8 号异常对应的处理函数为系统调用处理函数 handle\_sys；

24 号异常对应的处理函数为 tlb 页表项重复处理函数 handle\_mult\_tlb。

其中有关TLB快表的相关内容在前面内存管理模块讲过了，没有在异常处理模块中重复。

## 4.4 时钟中断（handle\_int）

时钟中断和操作系统进程调度的时间片轮转调度算法有着密切关系。操作系统内核是只有通过定时器产生的时钟中断才能够知晓一个进程的时间片结束了。当时钟中断产生时，当前运行的进程就会被挂起，将上下文保存到当前进程的 env\_tf 当中，然后操作系统需要在 env\_runnable\_list 中选取下一个合适的进程来运行，即调用sched\_yield（） 函数。handle\_int 的函数体代码实现在 ./lib/genex.S 汇编文件中，下面做一些代码提示：

NESTED(handle\_int, TF\_SIZE, sp)

.set at

nop

SAVE\_TF

.set noat # 关闭关于 at 寄存器的警告

mfc0 t0, CP0\_CAUSE # 取出上一次 exception 的 cause

mfc0 t2, CP0\_STATUS # 取出 Processor status

and t0, t2

andi t1, t0, STATUSF\_IP0 # t0 和立即数 0X400 相与，取出 t0 第10 位，结果存t1

j timer\_irq

nop

END(handle\_int)

其中 timer\_irq 用来清空计时器并调用 sched\_yield 函数。需要注意的是，当触发时间中断时我们需要保存当前的上下文，这里我们使用 SAVE\_TF 宏，已经为同学们在 inc/stackframe.h 头文件中定义好了。SAVE\_TF 首先读取 curtf，这个其实就是 curenv->env\_tf，为了写汇编方便，我们在切换进程时会把这个指针存到内存固定地址 curtf 上。若是第一次从内核进时间中断，即没有任何进程，curtf 会为 0，根据代码跳转，并不保存上下文。若是之后的正常时间中断则从&curenv->env\_tf 开始作为基址，一次存入需要保存的上 下文到内核的进程管理信息中，方便下一次上下文恢复。

## 4.5 系统调用（handle\_sys）

系统调用只要使用汇编指令 syscall 即可触发（./ushell/user/syscall\_wrap.S）， 不会显式地传递任何参数，但你可以把参数放在寄存器里，然后 syscall 直接跳到处理函数，你就可以进行处理。系统调用应该在用户的程序里被调用，在内核状态中被处理。目前本操作系统采取的机制是使用 a0 寄存器传递一个系统调用号，通过这个系统调用号来进行不同处理。本操作系统支持的系统调用通过C语言的形式，为大家在./lib/syscall\_all.c中都定义好了函数，系统调用的执行流程需要大家在 . /lib/syscall.S 中使用汇编实现，整个逻辑可以总结如下：

首先调用 SAVE\_ALL 函数用来保护上下文，然 后判断系统调用号 t1 是否合法，合法后将 epc+4 确保系统调用结束能够回到触发 中断的下一条指令避免死循环，取 sys\_call\_table 作为基址后加上系统调用号 t1 作 为偏移，从内存中获取系统调用处理函数的入口地址存到 t2 中跳转过去处理，系 统 调 用 处 理 结 束 后 调 用

ret\_from\_exception ， 在 ret\_from\_exception 中调用 RESTORE\_ALL 恢复上下文再回到触发中断的地址。 其中 SAVE\_ALL 和 RESTORE\_ALL 这两个功能函数需要转到./ inc/stackframe.S 汇编文件中实现。

具体而言，SAVE\_ALL 中首先需要判断是否是在内核状态触发嵌套中断，或是嵌套中断则不 修改 sp 继续在当前栈顶将上下文压栈，若不是嵌套中断则将 sp 置为内核处理中 断栈顶 ex\_stack\_top。然后在栈顶开辟一段空间用来存放上下文 trapframe 完成上下文的保护。RESOTRE\_ALL 则是需要从当前栈顶依次恢复上下文状态，同时注意 sp 的恢复和cp0\_status 状态的清空。

值得注意的是需要增加系统调用时，需要更改\_\_NR\_SYSCALLS 的大小，它是用于判断系统调用号是否越界的 ， \_\_SYSCALL\_BASE 这个函数数其实没有用，另外要注意 Aurora/lib/syscall.S 中的 syscalltable 系统调用处理函数名要按顺序，因为系统调用号直接严格地对应内存地址。

## 4.6 访问地址越界中断（handle\_addr）

该异常中断主要触发于当用户态进程想要访问内核地址，及 0x80000000 以上时会 自动触发地址访问越界，在我们的设计中当触发这个中断时将自动释放当前运行的进程 curenv，在初始化时将每一个 env 的 ra 都设置为 0x90000000，这样进程运行结束时回返回 ra 从而触发 handle\_addr，实现自动释放运行结束的进程。 handle\_addr 具体实现请同学们在 lib/genex.S 汇编文件中完成：

NESTED(handle\_addr, TF\_SIZE, sp)

nop

la sp,ex\_stack\_top

.set at #开启 at 寄存器警告

mfc0 t0, CP0\_CAUSE # 取出上一次 exception 的 cause

mfc0 t2, CP0\_STATUS # 取出 Processor status

and t0, t2

andi t1, t0, STATUSF\_IP0 # t0 和立即数 0X400 相与，取出 t0 第 10 位，结果存 t1

j print\_addr\_error

nop

END(handle\_addr)

当触发 handle\_addr 时因为该进程即将被释放，需要首先将 sp 切换为内核处理 异常的栈顶，判断中断有效后在 print\_addr\_error 中调用 env\_free 函数从而实现我 们所设定的功能。

## 4.7 TLB异常处理

TLB 异常是操作系统最常见的异常，每次对 TLB 中缺失的虚拟地址的访问都会导致TLB 异常，这也是 MIPS提供 TLB 异常专用入口 0x8000\_0000 的原因。

首先在 MIPS32 体系结构中，CPU 访问到的虚拟地址导致 TLB 异常后，CPU 会自动

进入内核态并且屏蔽中断，然后转到 TLB 专用异常向量入口 0x8000\_0000，因此 TLB 异常的处理函数应该放在这个入口处。在进入 TLB 异常处理函数后，内核首先需要查询页目录，找到虚拟地址的页表项，然后根据页表项中的有效位 V 判断是否发生页缺失。如果没有对应的页表项，或者有效位 V = 0，那么就说明没有与虚拟地址对应的物理页存在。在

这种情况下，需要调用page\_insert 函数来进行页插入，当完成了虚拟地址和物理页的映射后，即可根据返回的物理地址进行 TLB 重填。

图4-1为TLB异常处理流程：

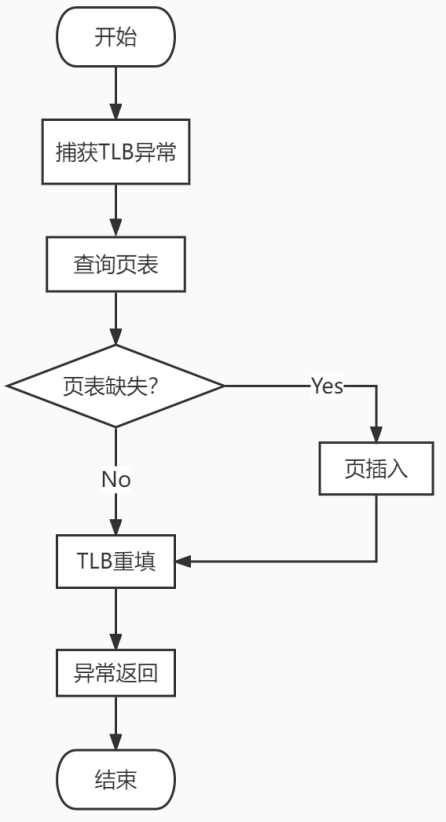


图4-1 TLB异常

# 第五章 内存管理的设计与实现

## 5.1 理解虚拟地址

对于操作系统而言，最重要的概念之一就是程序地址空间。图4-1详细的揭示了本操作系统中，4G大小的虚拟地址是如何给各部分划分的。

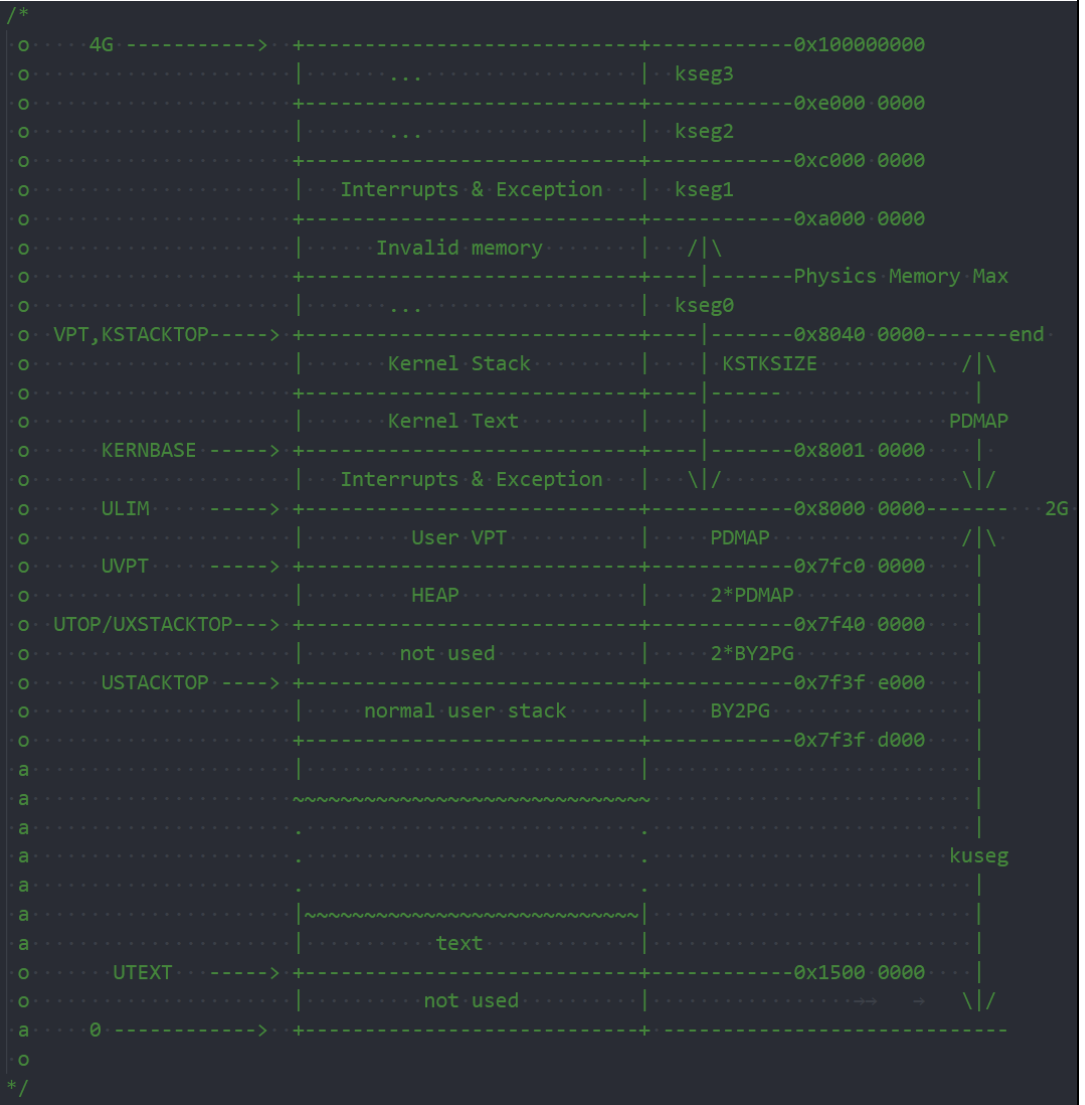


图5-1 虚拟地址分配

注意到：，0X 8000 0000 以上地址是所有进程相同的，其以下的地址是每个进程独有的，下面对每部分进行说明：

1) 0X bfc0 0000：Mips Core 重启时会从处开始执行，因此这部分需要有 bootloader 的代码，bootloader 执行完之后进入 OS 内核；

2) Kernel Stack: 内核在初始化时以及异常处理时所用的栈；

3) Kernel text：内核代码段；

4) Interrupt&Exception：中断和异常的处理函数，Mips Core 在触发中断和异常时 会固定跳到 0X8000 0000 进行处理；

5) User VPT：映射为了当前进程的二级页表，这是先前留下的设计，暂时没有什么用； 6) HEAP: 堆区，向上生长，用于进程申请内存的系统调用，在共享内存分配时会用上；

7) Normal user stack：用户的栈区，向下生长。

8) text：用户的代码段，根据链接脚本，代码会从这里开始放置。

* kuseg是用户内核段，该虚拟内存段主要用于用户模式的访问，可以通过TLB映射到任意物理地址。
* kseg0是内核段，该段地址不经过TLB映射，物理地址是虚拟地址直接减去0x8000\_0000，也就是说，虚拟地址0x8000\_0000直接映射物理地址0x0。
* kseg1也是内核段地址，该区域也不经过TLB映射，不经过cache缓存，因此适合把外设（即外设的控制寄存器）映射到这个段地址中去，该段物理地址为虚拟地址减去0xA000\_0000，也就是说0xA000\_0000的虚拟地址映射的物理地址也是0x0。

在理解了虚拟地址的分配过后，还需要理解硬件对应的设计：根据硬件设计，MMU 会将 0X8000 0000 以上地址直接映射到 0X0000 0000， 即将最高位置为 0，不经过 TLB，具体设计可见 Manual 4.4 章。0X 8000 0000 以下地址目前设计为经过 4K 为一页的 TLB 映射。利用这个设计，我们在 GDB 调试时想访问某个物理地址也可直接将其最高位置为 1。Aurora内核和异常处理的代码和数据全部设计为在 0X 8000 0000 以上，用户态的程序访问这部分空间会导致一个异常，以此来保护操作系统。同时这些代码运行不会触发 TLB 异常，极大避免了嵌套异常的出现。

## 5.2 页式内存体系

在MIPS CPU中，我们使用分页式的内存管理体系，每页为4KB。

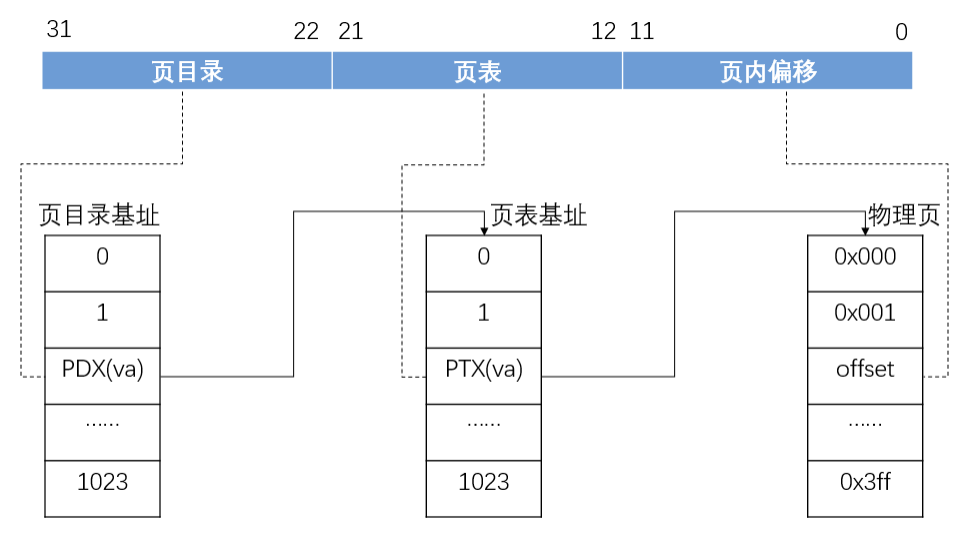


图5-2 二级页表地址翻译

本操作系统采用二级页表结构，如图 4-2 所示，PDX 是 Page Directory Index，是一级页表中的偏移，PTX 是 Page Table Index，是二级页表中的偏移。首先页目录基址 pgdir\_addr 已知（本文是放在虚拟地址 0x8040\_0000），根据虚拟地址中的页目录索引字段 PDX(va)，对地址 pgdir\_addr + PDX(va)解引用就可以得到所在页表的 基址 pgtable\_addr。然后取出虚拟地址中的页表索引字段 PTX(va)，基于页表索引段在页表中查找物理页帧号,也就是对 pgtable\_addr + PTX(va)进行解引用就可以得到物理页帧号。在拿到物理页帧号后，与虚拟地址中的页内偏移字段 offset 拼接即可得到完整的物理地址。 若页目录或页表项中对应的页表项为空，或者页表中有效位 V = 0，即发生页缺失错误，需要进行页面插入处理。

同学需要根据写好的注释将每个函数核心补充完整（此模块代码在附件./mm/pmap.c中）：这个文件是整个工程中比较困难的一部分，由于函数数目较多，所以给大家做了一个函数运行逻辑的导图，如图所示

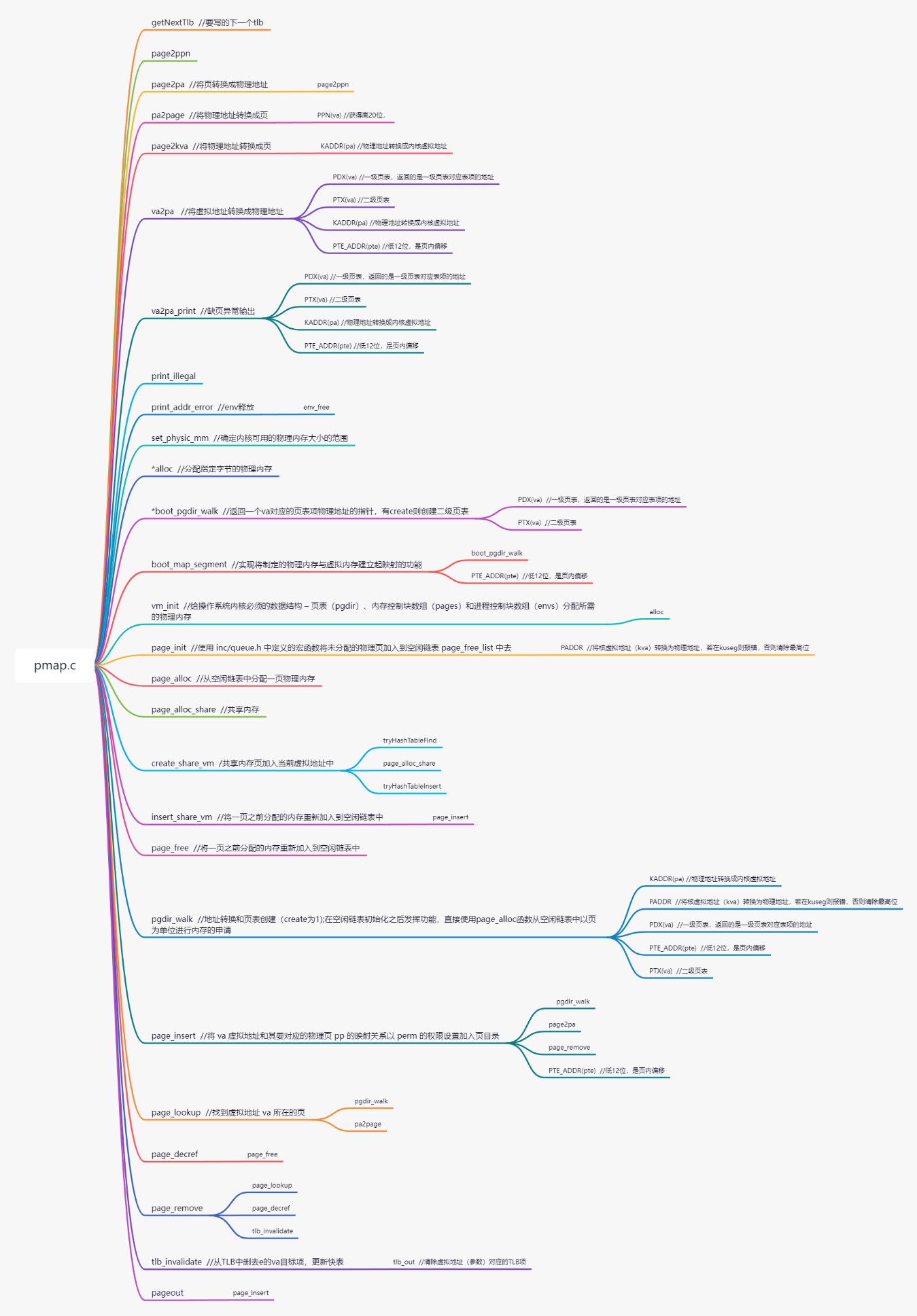


图5-3 pmap.c中函数相互关系

1）基本数据结构：

page：内存控制块，包含该页面被引用的次数，以及指向链表中的下一个节点的指针。 pages：内存控制块数组，包含所有内存控制块。

page\_free\_list：记录所有空闲物理内存的链表。

这些数据结构在系统启动时，需要被初始化和分配内存。page 可以根据在 pages 中的 索引，唯一定位一个物理页；同样的，一个物理页帧号也唯一映射一个 page 结构。这样， 就可以通过 page 结构来管理剩余的物理内存。

2）重要函数提示：

1. /\*函数功能\*/
2. u\_long page2pnn(**struct** Page \*pp) //
3. u\_long page2pa(**struct** Page \*pp) //将页转换成物理地址
4. **struct** Page \*pa2page(u\_long pa) //将物理地址转换成页
5. u\_long page2kva(**struct** Page \*pp) //返回页的虚拟地址
6. u\_long va2pa(Pde \*pgdir, u\_long va) //将虚拟地址转换成物理地址
7. u\_long va2pa\_print(Pde \*pgdir, u\_long va) //缺页异常输出
8. **void** print\_illegal(**int** num) //
9. **void** print\_addr\_error() //env释放
11. **void** set\_physic\_mm() //确定内核可用的物理内存大小的范围
12. **static** **void** \*alloc(u\_int n, u\_int align, **int** clear) //分配指定字节的物理内存
13. **static** Pte \*boot\_pgdir\_walk(Pde \*pgdir, u\_long va, **int** create) //返回一个va对应的页表项物理地址的指针，有create则创建二级页表
14. **void** boot\_map\_segment(Pde \*pgdir, u\_long va, u\_long size, u\_long pa, **int** perm) //实现将制定的物理内存与虚拟内存建立起映射的功能，perm 实际上是 PTE\_R 修改位
15. **void** vm\_init() //给操作系统内核必须的数据结构 – 页表（pgdir）、内存控制块数组（pages）和进程控制块数组（envs）分配所需的物理内存.
16. **void** page\_init(**void**) //使用 inc/queue.h 中定义的宏函数将未分配的物理页加入到空闲链表 page\_free\_list 中去
17. **int** page\_alloc(**struct** Page \*\*pp) //从空闲链表中分配一页物理内存
18. **int** page\_alloc\_share(**struct** Page \*\*pp) //共享内存
19. **struct** Page\* create\_share\_vm(**int** key, **size\_t** size) //共享内存页加入当前虚拟地址中
20. **void**\* insert\_share\_vm(**struct** Env \*e, **struct** Page \*p) //将一页之前分配的内存重新加入到空闲链表中
21. **void** page\_free(**struct** Page \*pp) //将一页之前分配的内存重新加入到空闲链表中
22. **int** pgdir\_walk(Pde \*pgdir, u\_long va, **int** create, Pte \*\*ppte) //地址转换和页表创建（create为1）；在空闲链表初始化之后发挥功能，直接使用page\_alloc函数从空闲链表中以页为单位进行内存的申请
23. **int** page\_insert(Pde \*pgdir, **struct** Page \*pp, u\_long va, u\_int perm) //将 va 虚拟地址和其要对应的物理页 pp 的映射关系以 perm 的权限设置加入页目录
24. **struct** Page \*page\_lookup(Pde \*pgdir, u\_long va, Pte \*\*ppte) //找到虚拟地址 va 所在的页
25. **void** page\_decref(**struct** Page \*pp) //
26. **void** page\_remove(Pde \*pgdir, u\_long va) //解除虚拟地址 va 与其对应的页的映射关系，并且将对应的内存控制块重新插入到空闲链表中去
27. **void** tlb\_invalidate(Pde \*pgdir, u\_long va) //从TLB中删去e的va目标项，更新快表

3）由于本系统由C语言和汇编语言共同完成，因此我们需要关注一下二者之间的接口，统一他们的名字以便不会在链接的时候出错，下面罗列以下汇编接口：

1. //汇编部分
2. **extern** **int** mips\_tlb\_size (**void**); //返回TLB的大小
3. **extern** **void** mips\_tlbinval (tlbhi\_t hi); // 探测TLB以查找与hi匹配的条目，如果存在，则使其无效。
4. **extern** **void** mips\_tlbinvalall (**void**); //使得整个TLB无效
5. **extern** **void** mips\_tlbri2 (tlbhi\_t \*phi, tlblo\_t \*plo0, tlblo\_t \*plo1, unsigned \*pmsk, **int** index); //读取由索引指定的TLB条目，并分别返回\*phi、\*plo0、\*pl1和\*pmsk中的EntryHi、EntryLo0、EntryO1和PageMask部分。
6. **extern** **void** mips\_tlbwi2 (tlbhi\_t hi, tlblo\_t lo0, tlblo\_t lo1, unsigned msk, **int** index); //将hi、lo0、lo1和msk写入索引指定的TLB条目。
7. **extern** **void** mips\_tlbwr2 (tlbhi\_t hi, tlblo\_t lo0, tlblo\_t lo1, unsigned msk); //将hi、lo0、lo1和msk写入随机寄存器指定的TLB条目。
8. **extern** **int** mips\_tlbprobe2 (tlbhi\_t hi, tlblo\_t \*plo0, tlblo\_t \*plo1, unsigned \*pmsk); //探测TLB中与hi匹配的条目，并返回其索引，如果未找到，则返回-1。如果找到，则条目的EntryLo0、EntryLo1和PageMask部分也分别在\*plo0、\*plo1和\*pmsk中返回
9. **extern** **int** mips\_tlbrwr2 (tlbhi\_t hi, tlblo\_t lo0, tlblo\_t lo1, unsigned msk); //探测TLB中与hi匹配的条目，如果存在，则重写该条目，否则更新随机条目。更新TLB的安全方法。

注意MIPS体系结构下内存分配的过程，几点提示：

1. 在链接脚本中，内核的代码段和数据段会从 0X8000 0000 开始分配，然后把栈顶设在 0X80400000，所以栈的范围从0X80400000 到大约 0X80020000， 0X80400000 这个位置也被记为符号 end，在之后内存分配使用。
2. 在函数vm\_init()中会使用 alloc()开始分配空间给 envs(进程管理信息)和 pages(页表 管理信息)，pages 会给每一个 4K 大小的物理页建立一个管理信息。alloc()从 end开始，随意指定内存大小进行分配，每次分配的内存位置会记录在一个 freemem 的全局变量里。
3. 分配完之后，由于 freemem 分配的内存都是在 0X8000 0000 以上，其物理地址 是直接映射的，因此可以得到物理地址，这部分物理内存对应的 pages 管理信 息会被标记为已用，剩下的物理内存会被标记为未用，并加入到一个 page\_free\_list 中。
4. 之后因为页表管理信息已经建立起来，因此不再使用 alloc()分配，而是使用 page\_alloc()进行按页的内存分配了。
5. 页page的数据结构在inc/pmap.h 文件中已经给大家定义好了：

struct Page {

struct Page\* pp\_link; /\* free list link \*/

u\_short pp\_ref;

};

其中pp\_link 即用来串起 page\_free\_list 的指针，每次page\_alloc()都需从 page\_free\_list 取出一个未用过的物理页，然后通过 page\_insert()将其与某个虚拟地 址建立起映射，这过程中会将 pp\_ref++，表示这个物理页被一个进程引用了。在 释放进程时，需要遍历一个进程拥有的所有物理页，将其 pp\_ref--，如果减为了0， 就需要将其回收，并重新加入 page\_free\_list。

1. 每个进程一开始一级页表都是空的，随着每次访问一个不越界的地址，就会触发 TLB 异常，然后异常处理会 alloc 一个物理页，并将其映射到访问的虚拟地址上，进程的虚拟页表会这样一步步的完善。

## 5.3 TLB作为页表的缓存

由于页表太大，全部用硬件实现十分不经济。而只放在内存中，每次访存的开销会使 系统性能非常低。因此一种 CPU 可以直接访问的硬件 TLB 就诞生了。 TLB（Translation Lookaside Buffer，翻译后备缓冲器，也叫做快表），是 MIPS32 体系 结构中完成映射段的物理地址到虚拟地址的映射的硬件，可以视为页表的 cache。与 TLB 相关的寄存器有 EntryLo0、EntryLo1、EntryHi 以及 PageMask 等寄存器，TLB 表项中每一项拥有一个 VPN（虚拟页号）和一个 PFN（物理页帧号）。 当程序访问一个虚拟地址时，会先和 TLB 中的每一个 VPN 相比较，同时比较 ASID（在 EntryHi 寄存器中指定），并且对标志位也进行匹配。如果匹配成功，则返回对应的 PFN。 如果全部匹配失败，那么会出发 TLB 未命中异常。操作系统内核应查询页表，进行 TLB 重填。microAptiv UP 核拥有 16 个页表项。TLB 的地址转换过程如图 4-3 所示。

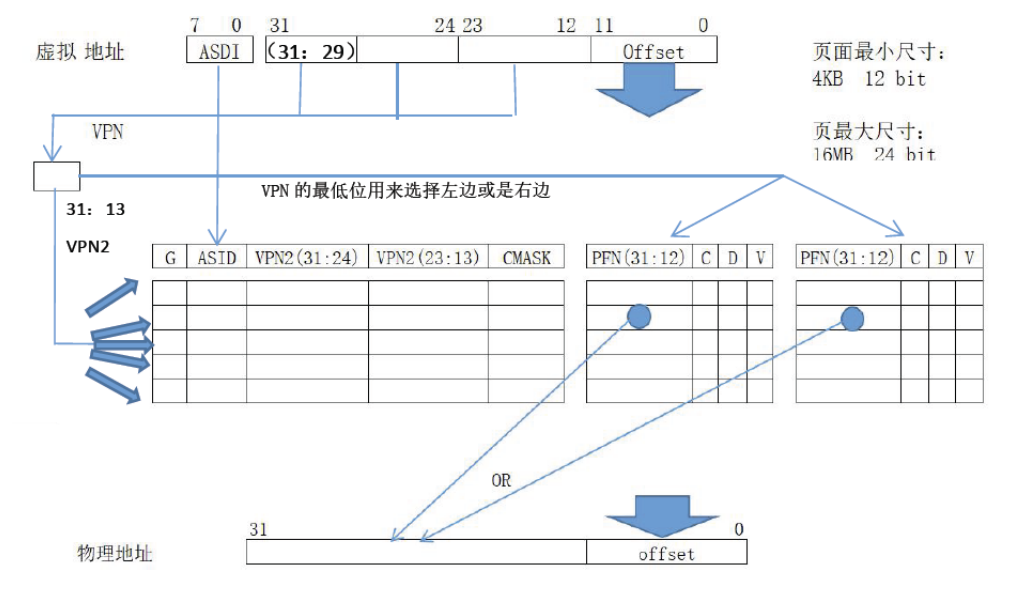


图4-3 TLB硬件机制

其中，各个符号的含义如下： CMASK：页掩码，决定页面大小和用于比较的 VPN2 的位数。 G：全局位，被置位时忽略 ASID。 ASID：地址空间标志，用于地址空间的隔离，在本文中被定义为进程 ID。 VPN：虚拟页号。 PFN：物理帧号。 D：可写位，D = 1 时该页可写。 V：有效位，V = 1 时该页有效。 C：缓存性质。

值得注意的是，TLB异常是最为常见的异常，因此在这里单独列出，TLB 的信息可参考 Manual p85，TLBWI 的信息可参考 Manual p346，关于异常的综述请回看“异常处理”章节。下面强调几个注意点：

1）Mips 的 TLB 会把两个映射项存在同一个 TLB 项中。以 4K 一页为例，虚拟地 址本应是一个 22 位的虚拟页号，Mips TLB 却只会用 21 位，然后这个虚拟地 址对应的 TLB 项存两项物理地址，分别对应最后一位为 0 和 1。

2）虚拟地址的读写是通过 CP0\_EntryHi 寄存器，只有前 21 位代表地址，最后 8 位是 ASID，表示这个项归哪个进程所有；物理地址的读写是通过 CP0\_Entrylo0 和 CP0\_Entrylo1 寄存器，只有前 22 位代表地址，后面 12 位也是管理信息， 比如有个 valid 位。

3）平常输入一个 0X8000 0000 以下的虚拟地址，TLB 会结合 EntryHi 寄存器指定 的 8 位 ASID (Manual p150) ，查询 TLB 中有没有对应的虚拟页号和 ASID。 因此切换进程时必须向 EntryHi 寄存器填入 ASID 的值，以区分不同进程的寻 址空间。

4）当找不到对应虚拟页号和 asid 的 tlb 项，或者找到了但是 valid 位为 0，就会触 发异常，异常处理时无法辨别出属于前面所说的两种情况，触发异常时 EntryHi 寄存器会有查询失败的虚拟地址，EnrtLo0 和 EntryLo1 的值没有意义。

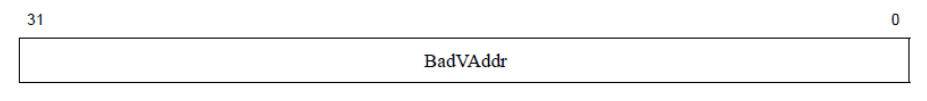
5）TLBWI 这条指令会将 EntryHi 寄存器的值作为虚拟地址，EntryLo0，Entrylo1 的值作为两个物理地址，写入 CP0\_index 寄存器所指定序号的 tlb 中去，每次 都必须两个一块写。TLBP 指令可查询是否有 EntryHi 寄存器对应的 tlb 项，并 将结果填到 index 寄存器中。

6）不可存在两个具有相同的 EntryHi 值的 TLB 项，即不可两个 tlb 项前 21 位虚 拟页号相同，否则在写时会触发异常（Manual p169 p118）,因此相邻两个 4k 页 必须存在同一个 tlb 项中。

1. 与TLB相关的几个寄存器

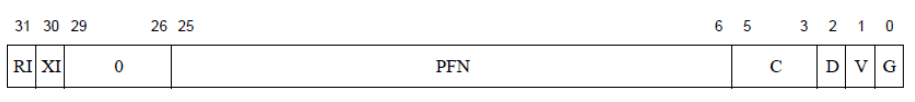
**①BadVAddr**

BadVAddr 寄存器各比特的含义如图\*\*所示。该寄存器自动保存最近一次地址相关异常的虚拟地址。在发生 TLB 相关异常时，需要查寻该寄存器，来获取导致 TLB 异常的虚拟地址，来进行 TLB 重填操作。



②**EntryLo0、EntryLo1**

EntryLo0、EntryLo1 两个寄存器也是与 TLB 有关的寄存器，EntryLo0 存放偶数页的入口物理地址，EntryLo1 存放奇数页的入口物理地址。



RI，XI：读禁止和执行禁止，如果该位被设置位 1，那么即使 V = 1，对于该页的读取也会导致 TLB Invalid 异常。

29:26 ：强制写入 0

PFN：Page Frame Number，即物理页帧号。对应物理地址的 31:12 位。

C：描述页面的缓存一致性属性，对于 microAptiv UP 核来说，2为不可缓存，3为可缓存。

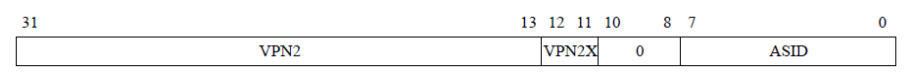
D：脏位，或者叫做写使能位。D = 0 时，不允许写入该页，否则会引起 TLB 修改异常。

V：有效位，表明当前 TLB 映射是否有效。V = 1，表示允许访问该页，V = 0，对该页面的访问会导致 TLB 无效异常。

G：全局位，如果 G = 1，那么 TLB 表项匹配时会忽略 ASID 域。

**③EntryHi**

EntryHi 是与 EntryLo0、EntryLo1 功能类似的寄存器，也是用于对 TLB 读、写操作的寄存器。每当 TLB 发生异常时，需要将发生异常的虚拟地址的 31:13 位写入VPN2。



VPN2：发生异常的虚拟地址的 31:13bit

VPN2X：本文中为常量 0

ASID：表示地址空间标志，在对 TLB 进行读操作时，由硬件写入；写操作时，由操作系统软件写入。用于地址空间的隔离。

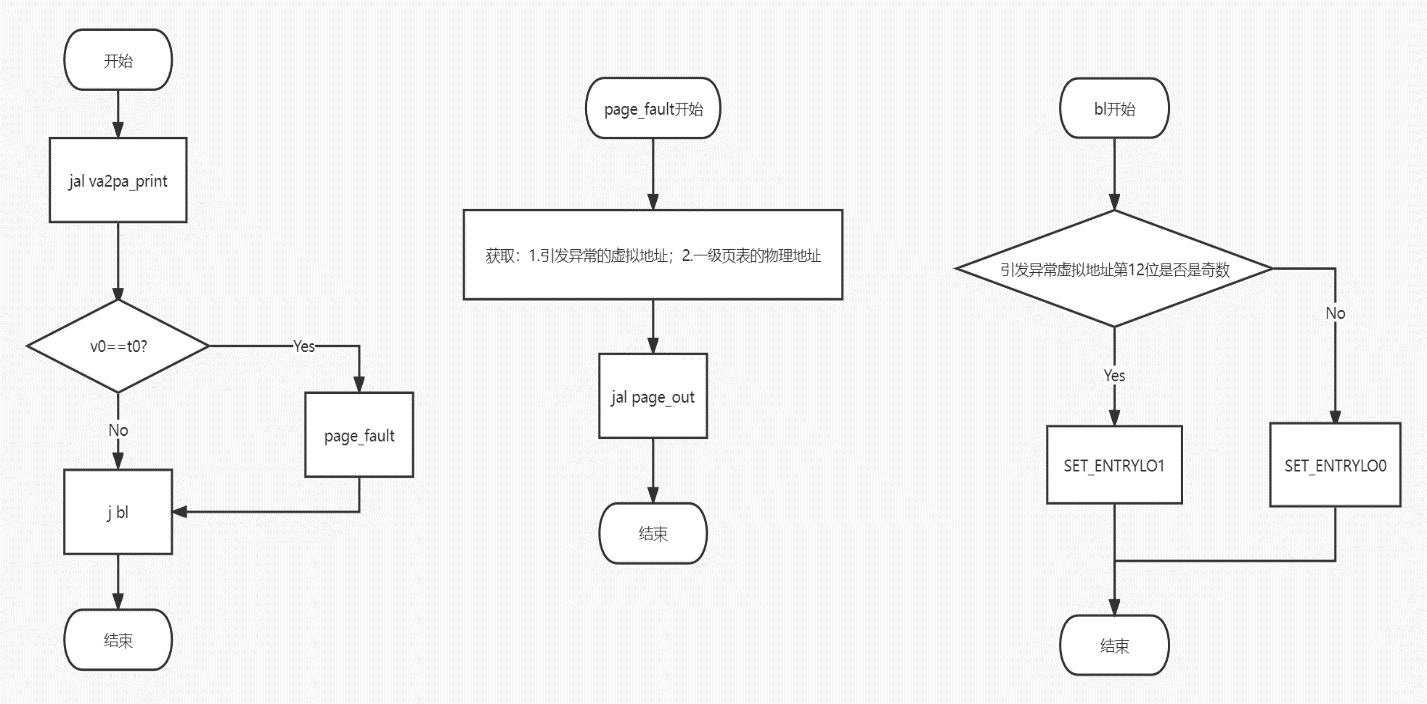
下面给出TLB 重填的伪代码设计供参考，TLB相关的头文件已经给大家定义好（inc/tlbop.h），大家需要完善的**汇编**文件为mm/m32tlb\_ops.s。图4-4展示了处理TLB异常的汇编文件逻辑，需要填写的代码量很少，重点是要学

图4-4 TLB异常处理汇编逻辑

生理解这个流程。

# 第六章 进程管理的设计与实现

进程是 CPU 执行的基本单元，也是 CPU 分配资源的基本单元。本文设计并实现了基 本的进程管理模块，该模块包括进程控制块（PCB）、进程创建、进程撤销等算法，以及一个基于时间片轮转(round-robin, RR)的进程调度算法。本模块需要完成的源码框架定义在 ./env 文件夹下。

## 6.1 进程控制块（PCB）与内存初始化

关于PCB的结构体和进程管理模块中需要用的成员函数，作为规范已经给大家在 ./inc/env.h 头文件中定义好了，下面对结构体Env进行简要的说明：

struct Env {

struct Trapframe env\_tf; // Saved registers

struct Env\* env\_link; // Free list

u\_int env\_id; // Unique environment identifier

u\_int env\_parent\_id; // env\_id of this env's parent

u\_int env\_status; // Status of the environment

Pde \*env\_pgdir; // Kernel virtual address of page dir

u\_int env\_cr3; u\_int env\_pri; u\_int heap\_pc; uint32\_t va;

};

其中 Trapframe env\_tf 用来存储进程的上下文，包括 32 个通用寄存器， cp0\_cause 寄存器，cp0\_epc 寄存器等，由于每个进程我们分配独立的用户栈，因 此在用户栈中我们取一段连续的空间(&env\_tf)用来存储当前进程的上下文。 Pde \*env\_pgdir 用来存储当前进程的页表的虚拟地址。

在本OS进程管理设计中，我们将存放进程控制块的物理内存在系统启动后就要求被分配好，并且这块内存不可以被换出。因此在系统启动之后就要为进程控制块数组 envs 分配好内存。由于这部分与内存管理模块有交叉，因此函数体部分的代码实现在(./mm/pmap.c)文件中完成：

// 初始化所有 env，链到 env\_free\_list 上

void env\_init(void) {

int i;

for (i = NENV - 1; i >= 0; i--) {

envs[i].env\_id = 0XFFFFFFFF;

envs[i].env\_status = ENV\_FREE;

envs[i].env\_link = env\_free\_list;

env\_free\_list = &envs[i]; e

nvs[i].heap\_pc=UTOP;

}

}

其中的 struct Env \*envs = NULL; // All environments

envs 用来包含我们所有的进程控制块。当我们为进程控制块数组 envs 初始化后，还需要把空闲的 env 控制块按照链表形式串起来，因此我们定义了 env\_free\_list：

struct Env \*env\_free\_list = NULL; // Free list

它使用链表的形式这样方便我们后续申请创进程控制块 Env。因为一开始所有的进程控制块都是空闲的，因此在初始化时需要将它们依次指向下一个并放到 env\_free\_list 上。

## 6.2 进程创建

本部分，我们需要利用空闲进程链表 env\_free\_list 来进行进程的分配。需要完成的函数在./env/env.c 文件中，下面对于核心函数env\_alloc给出一些提示：

int env\_alloc(struct Env \*\*new, u\_int parent\_id) {

int r;

struct Env \*e; /\*Step 1: Get a new Env from env\_free\_list\*/

e = env\_free\_list;

if (e == NULL) { return -E\_NO\_FREE\_ENV; }

env\_setup\_vm(e);

e->env\_id = mkenvid(e);

e->env\_parent\_id = parent\_id;

e->env\_status = ENV\_RUNNABLE;

e->env\_tf.cp0\_status = 0x10007c01;

e->env\_tf.regs[29] = USTACKTOP; //栈顶

e->env\_tf.regs[31] = 0x90000000; //返回地址（指向结束的系统调用）

e->env\_runs = 0;

env\_free\_list = env\_free\_list->env\_link;

\*new = e;

return 0; }

首先我们从 env\_free\_list 中取出第一个空闲 PCB 块，调用 env\_setup\_vm 函数 来初始化这个新 env 的一级页表，因为每一个进程都需要有自己独立的虚拟地址 空间因此初始化一些进程控制信息如用户栈顶 sp，用户堆地址，同时初始化 env\_status 为 ENV\_RUNNABLE，进程释放的返回地址 0x90000000，我们设计是 由于用户态访问 80 以上地址触发访问越界并释放当前进程。

在进程分配结束后我们需要为该进程分配空间来读取程序代码，这里我们借 助了 load\_icode 和 load\_elf\_mapper 来实现这一个需求。读者根据以下提示完成env.c 中对应函数的核心代码。

static void load\_icode(struct Env \*e, char \*elf\_name) 该函数首先就为新建的进程分配一个物理页，并映射到它的栈的地址上去，这 个栈在这里初始化只是少了一个 tlb 异常而已，可删去。接着这个函数会将完整的 二进制镜像加载到进程的用户内存中去。目前设计上会需要将完整的 elf 通过文件系统读到内存中一个固定地址（boot\_file\_buf 是个固定值），然后根据这部分内存的内容，读出 elf 的管理信息，再将实际的代码存到 elf 指定的进程虚拟地址空间中去。最后我们需要将 elf 指定的的代码入口地址 entry\_point 存在当前进程 env 的 env\_tf.cp0\_epc 当中，作为后续代码运行的起始 pc 地址。

uint32\_t load\_elf\_mapper(char \*elf\_name, struct Env \*e) 该函数是内核加载 elf 函数，函数首先就需要把所有的二进制内容即 ELF 文 件内容全部都加载到内存当中，接着分配页到内存的段中去，如果该段在文件中 的内容大小达不到该段在内存中所应有的大小，那么余下的部分用 0 来填充。需 要注意的是，在其中调用 load\_elf\_sd 从 sd 卡中读取 elf 之前需要执行：

lcontext(e->env\_pgdir,0);

set\_asid(GET\_ENV\_ASID(e->env\_id));

这是因为在加载 elf 进内存时会触发缺页中断，缺页中断会填当前调用进程的 asid 和页表基址进 tlb 页表项，因为我们需要在 load\_elf\_sd 之前将进程的 asid 和 页表基址进 tlb 页表项切换成装载 elf 进程的值，在 load 结束后恢复回之前的值。

有了上面的函数后我们就能通过调用：

void env\_create\_priority(char \*binary, int priority) 来实现完整的进程创建，分配并设置新的进程控制块，并将二进制文件名 binary 所对应的代码载入到新进程的用户虚拟地址空间。 在创建结束后我们需要把新创建的进程加入等待队列来等待系统的调度，在 这里我们设计的数据结构 env\_runnable\_list 为环状链表，具有头和尾两个指针，具体如下：

struct Env \*env\_runnable\_head = NULL; // Runnable ring head

struct Env \*env\_runnable\_tail = NULL; // Runnable ring tail

其中 env\_runnable\_tail 指向 env\_runnable\_head 从而完成环状链表，采用环状链表是为了方便我们后面所选择的多优先级轮转调度算法，指针只需向后指，无需过多判断头和尾。

## 6.3 进程调度

首先我们要能够使一个进程运行起来，因此需要一个单独的函数入口来使内核运行某一个进程控制块，在运行某一进程前我们需要内核保存当前进程上下文，然后就是加载我们需要运行的进程上下文，然后运行该进程。函数具体实现在./env/env.c中：

void env\_run(struct Env \*e) {

curenv = e;

curenv->env\_runs++;

lcontext(curenv->env\_pgdir,&(curenv->env\_tf)); set\_asid(GET\_ENV\_ASID(curenv->env\_id));

env\_pop\_tf(&(curenv->env\_tf));

}

我们需要先 lcontext 并设置 asid 来确保 TLB 查找时能够找到要运行 进程的页表项，即切换到 e 的地址空间，然后调用 env\_pop\_tf 函数恢复进程的上下文。

值得注意的是，lcontext 函数，set\_asid 函数，env\_pop\_tf 函数需要同学们在 env/env\_asm.S 中汇编实现，其中 env\_pop\_tf 函数为重点，主要功能是确保我们能够加载新的进程的状态：

LEAF(env\_pop\_tf)

.set noreorder

move k0, a0 # k0 = &(curenv->env\_tf)

lw k1, TF\_CAUSE(k0)

mtc0 k1, C0\_CAUSE

lw v1, TF\_LO(k0)

mtlo v1

lw v0, TF\_HI(k0)

mthi v0

lw v1, TF\_EPC(k0)

# li v1,0x15001f90

mtc0 v1, C0\_EPC

ehb

lw $31,TF\_REG31(k0)

...

lw $1,TF\_REG1(k0)

mfc0 k0,C0\_STATUS

li k1,0x11 #设为 USER MODE,开启中断

or k0,k1

li k1,0xEFDFFFFF

and k0,k1

mtc0 k0,C0\_STATUS

ehb

eret #回 epc

nop

END(env\_pop\_tf)

首先加载要恢复进程 curenv 的上下文地址&env\_tf，依次从该基址加载各个通 用寄存器及 CP0 的状态寄存器，通过加载 epc 恢复新进程的代码继续执行。在完成了单个进程的运行过后，我们才可以考虑多进程之间调度的问题，本教学用操作系统实现了一个时间片固定的多优先级调度，每到固定时间，就会执行下一个具有最高优先级的进程。

本部分需要大家完善的代码在./env/sched.c 文件中，下面分析其核心函数void sched\_yield()：每一次进 行时钟中断时，都会跳转到该函数进行进程的调度。令亟待调度运行的进程控制 块指针为 Env\* e，该函数先取当前运行的进程控制块指针 curenv。若是第一次进 行进程调度，则 curenv 为空指针，则可以将 e 赋值为待调度进程队列 env\_runnable 的队首 env\_runnable\_head，然后就可以直接调用 env\_run 函数进行进程调度，e 就 会作为本系统此次启动后的第一个进程运行。由于在初始化的步骤中，会先创建 进程再初始化时钟中断，所以管理所有 ENV\_RUNNABLE 状态的 env\_runnable 队 列不应为空。第一次进入时钟中断进行调度的代码为：

struct Env \*e=curenv;

if(curenv==NULL) //第一次进时间中断

{ e=env\_runnable\_head;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* first sched \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \n");

}

若当前不是第一次进入时间中断进行进程调度，则需要根据优先级进行调度。 先通过一次对 env\_runnable 的遍历得到当前运行优先级最高的进程的优先级级 别，然后再遍历到下一个为此优先级级别的进程（从当前 curenv 遍历到下一个目 的为实现同为最高优先级的进程之间的轮转）。该部分代码如下：

int highestPt=0;

struct Env \*tempE=env\_runnable\_head;

do{

if(tempE->env\_pri>highestPt)highestPt=tempE->env\_pri;

tempE=tempE->env\_link;

}while(tempE!=env\_runnable\_head);

do{

e=e->env\_link;

}while(e->env\_pri<highestPt);

## 6.4 进程释放

对于已经执行完毕的进程，我们需要释放它所占用的程序地址空间，不管是虚拟地址还是物理地址，本部分需要同学们完成的函数也在env.c 源文件中。下面给出几点提示：

for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {

/\* Hint: only look at mapped page tables. \*/

if (!(e->env\_pgdir[pdeno] & PTE\_V)) { continue; }

/\* Hint: find the pa and va of the page table. \*/

pa = PTE\_ADDR(e->env\_pgdir[pdeno]); //物理地址

pt = (Pte \*)KADDR(pa); //虚拟地址

/\* Hint: Unmap all PTEs in this page table. \*/

for (pteno = 0; pteno <= PTX(~0); pteno++) {

if (pt[pteno] & PTE\_V) {

page\_remove(e->env\_pgdir, (pdeno << PDSHIFT) | (pteno << PGSHIFT)); } }

/\* Hint: free the page table itself. \*/

e->env\_pgdir[pdeno] = 0;

page\_decref(pa2page(pa)); }

/\* Hint: free the page directory. \*/

pa = e->env\_cr3;

e->env\_pgdir = 0;

e->env\_cr3 = 0;

page\_decref(pa2page(pa));

首先我们释放一个进程的同时需要释放它所分配的所有内存页来提高操作系 统的可用性，在这里我们首先遍历该进程的一级页表，当找到有映射的页表项时， 接着找到页表项对应的物理地址和虚拟地址，然后遍历调用 page\_remove 函数清空二级页表，接着调用 page\_decref 函数释放页表本身，再类似的把一级页表和页目录释放。 其中 page\_decref 函数在释放页前会判断待清空页的引用数 pp\_ref 是否大于 1， 因为若引用数大于 1 则代表还有其他进程同时也在页表中存在对这个页的使用， 如共享内存当中两个进程 A 和 B 同时具有对同一张页的访问权限，在释放掉进程 A 时应同时释放页导致进程 B 无法继续访问该内存空间。

在释放掉内存空间后，需要将 env 的状态置为 ENV\_FREE，并且将被释放掉的 进程加入到空闲链表 env\_free\_list 当中。 在调用 env\_free 时我们需要判断被释放掉的进程是否为当前运行的进程，若是 的话我们还需要查找 env\_runnable\_list 中的下一个进程，重置时间中断后调用 env\_run 函数将 cp0 使用权交给下一个进程。

## 6.5 线程创建

类似于进程，我们在./env/env.c 源码文件中设计一个void pthread\_create(void \*func, int arg) 函数来创建线程。其中 func 用来接收需要运行的函数地址，arg 用来接收函数参数。首先调用 env\_alloc 函数分配一个线程控制块 env，调用 copy\_curenv 函数将新创建的线程与当前运行的进程设定一些共享的资源如页表， 在设定完基本参数后将新创建的线程 env 加入 env\_runnable\_list 当中等待内核的调度。

void copy\_curenv(struct Env \*e, struct Env \*env\_src, void \*func, int arg) 函数用来设定线程的一些参数如代码入口地址为 func，并使线程与创建该线程的进程共享页表。核心部分提示如下：

for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {

if (!(env\_src->env\_pgdir[pdeno] & PTE\_V)) {

e->env\_pgdir[pdeno] = 0;

continue; }

e->env\_pgdir[pdeno] = env\_src->env\_pgdir[pdeno]; //拷贝二级页表地址

printf("content:0x%x\n",e->env\_pgdir[pdeno]);

pa = PTE\_ADDR(env\_src->env\_pgdir[pdeno]); //源二级页表物理地址

pt = (Pte \*)KADDR(pa); //源二级页表虚拟地址

pa2page(pa)->pp\_ref++; //增加二级页表物理引用

for (pteno = 0; pteno <= PTX(~0); pteno++) {

if (pt[pteno] & PTE\_V) {

int pa\_tmp = PTE\_ADDR(pt[pteno]);

pa2page(pa\_tmp)->pp\_ref++;

} }

}

同 env\_free 函数的思想遍历一级页表并查找有效页表项，拷贝二级页表地址， 我们设计的是线程与进程共享二级页表，因此查找二级页表的物理地址和虚拟地址，增加二级页表物理引用。同时因为线程应具有自己独立的栈空间，因此在拷贝完一级页表后我们需要其中清空栈地址，这样线程就拥有了自身独立的栈。线程创建出来之后，我们就把它当进程一样管理，其释放、调度都与进程相同。

## 6.6 设备管理

由于本操作系统的文件系统部分并不是Linux式的抽象文件系统，因此外设也不能够像Linux一样被视为抽象文件来处理，因此需要构建一个设备管理相关的程序，由于进程往往要使用设备，因此这一部分放在进程管理中是合理的，不同进程对于设备存在争抢也是要考虑的问题。

本OS对IO设备进行统一管理，并为此设置一个专门的数据结构 rt\_device，用它来记录设备的属性，对设备进行 I/O 操作的统一接口，以及在资源分配过程中需要使用的申请和释放接口，接口使用函数指针，在系统初始化设备的时候进行注册。其属性已经在头文件 ./inc/rtThread.h 中为大家定义好了：

struct rt\_device

{  uint16\_t               flag;          /\* 设备参数 \*/

    uint16\_t               open\_flag;     /\* 设备打开标志 \*/

    uint8\_t                ref\_count;     /\* 设备被引用次数 \*/

    uint8\_t                device\_id;     /\* 设备 ID,0 - 255 \*/

    uint8\_t                device\_num;    /\* 设备数量\*/

    //rt\_device\_read;

    bool (\*rt\_device\_read)(char \*);

    //rt\_device\_write;

    bool (\*rt\_device\_write)(char \*);

    bool (\*rt\_device\_write\_byte)(char \*,u32 i);

    bool (\*rt\_device\_write\_by\_num)(char \*, u32 i);

    bool (\*rt\_device\_read\_by\_num)(char \*, u32 i);

    bool (\*rt\_require\_device)(u32 i);

    bool (\*rt\_release\_device)(u32 i);

}；

需要同学们补充完整的函数代码在 ./lib/rtThread.c 中，下面对该文件中的几个难点进行阐释：

bool rt\_device\_init(u32 device\_id, bool (\*rt\_device\_read)(char \*),…, u32 num,…){

/\*给设备的函数指针赋予相对应的具体的 I/O 操作\*/

device\_list[device\_id].rt\_device\_read = rt\_device\_read;

/\*银行家算法的部分参数初始化\*/

}

在做整个操作系统实验的时候，同学们一定要注意不同模块间的交互关系，比如在./init/init.c 进行设备初始化的时候 device\_init()，是调用本文件中的 rt\_device\_init() 函数来注册所有的设备。在进行 I/O 操作时，调用统一的接口，只需要传递该设备的 ID 和内容即可。由设备管理器负责根据 ID 索引找到已经被注册的设备，并调用具体的 I/O 函数：

bool rt\_device\_write(u32 device\_id, char \* buf)

{

if(device\_list[device\_id].rt\_device\_write==NULL)

return false;

return device\_list[device\_id].rt\_device\_write(buf);

}

在操作系统课程中，我们学习了避免死锁的银行家算法来控制资源，那么我们需要在FPGA板子上实践它！下面给大家提供了模拟的思路：由于我们使用的开发板的 I/O 资源只有一套 LED 和数码管，我们模拟银 行家算法最好需要每类资源有多个，所以我们按照一个字节为一组，将数码管看成独立的 4 个资源，LED 看成三个资源。 经过实验，数码管和 LED 都不能实现一次写入一个字节的功能，所以我们在物理层面上还是写入了 u32，通过算法实现更新指定某组数码管或者 LED 的操作。 具体来说，就是获取要写第 i 组，将要写的数左移 i\*8 位得到 tmp，再取原来的数 ori，将相应位置的字清零，再与 tmp 或来更新，实现更新单个字节的功能：

bool rt\_leds\_write\_byte(char \*v,u32 i){

if(i>2) return false;

u32 tmp = (u32)\*v;

tmp = tmp << (i\*8);

u32 tmp2 = ~(0xff << (i \* 8));

u32 ori = mips\_get\_word( LEDS\_ADDR, NULL);

mips\_put\_word( LEDS\_ADDR, tmp | (ori & tmp2) );

return true; }

接着，我们利用一些基础的数据结构来实现银行家算法：

static int all\_devices[NUMBER\_OF\_RESOURCES];                     //设备数量数组

static int available[NUMBER\_OF\_RESOURCES];                 //剩余可用设备数量

static int maximum[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

 //进程要求的最大数量

static int allocation[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

//已经分配给进程的数量

static int need[NUMBER\_OF\_CUSTOMERS][NUMBER\_OF\_RESOURCES];

//进程还需要的数量

static int status[NUMBER\_OF\_CUSTOMERS];                 //进程是否完成 1 为完成

static int asid\_list[NUMBER\_OF\_CUSTOMERS];              // asid to index mapping

在函数bool rt\_require\_device(u32 device\_id, u32 request\_num) 体的编写过程中，要主要遵循以下过程：

1. 若请求的资源大于自己剩余需要的，请求失败如果请求的资源大于剩余可用的资源，则请求失败。
2. 判断分配后，剩下的进程能否正常执行下去。
3. 检查当前进程能否一次获取所需资源，mark=0 为否
4. 分配后还是安全的，则执行资源分配
5. 分配后不安全，则跳过资源分配部分，直接释放信号量。

对于 led 资源，led\_status 记录使用此资源的进程号（static u32 led\_status[3]）

/\*申请资源，首先获取当前进程的进程号，从 0 号编组开始遍历，如果该组 LED 灯没有被占用，则当前进程获取此资源，直到获取所有资源。\*/

bool rt\_leds\_require(u32 num) {

u32 tmp = get\_asid() & 0xff;

int count = 0;

for(int i = 0; i < 3 && count != num; i++) {

if (led\_status[i] == -1) {

led\_status[i] = tmp;

count ++; }

} }

/\*补充函数：释放所拥有的资源\*/

bool rt\_leds\_release(u32 num) {

u32 tmp = get\_asid() & 0xff;

int count = 0;

for(int i = 0; i < 3 && count != num; i++) //如果未释放全部，默认从最 低位开始释放

{ if (led\_status[i] == tmp){

led\_status[i] = -1;

count ++; }

} }

进程请求资源后，利用银行家算法判断是否可以分配相应的资源 由于进程号是一个不确定的数，因此需要进行相应的匹配算法对应其在需求矩阵等的行号。 用 asid\_list[]记录相应行号对应的进程号，索引即行号，辅助函数如下：

int getAsidIndex() {

/\*获取进程号\*/

int temp= 0xff & get\_asid();

/\*遍历 asid\_list[]，找到当前进程号对应的索引号\*/

/\*如果没有，则从头开始，找到一个还没有被匹配的索引，则注册到 List 中，并返回序列号\*/ }

//声明该进程需要的最大资源数量

bool rt\_claim\_device(u32\* require) {

int index = getAsidIndex(); //对进程匹配所在矩阵的行号

/\*如果请求的数量超过了设备的资源数量，则声明失败\*/

/\*注册的时候对最大需求矩阵和剩余需求矩阵进行初始化\*/

for(int i = 0; i < NUMBER\_OF\_RESOURCES; i ++) {

need[index][i] = require[i]; maximum[index][i] = require[i];

}

}

# 第七章 关于拓展任务

以上内容仅仅是操作系统能够运行起来的基本条件，但是作为一个高级的操作系统还是完全不够的，因此我们给同学们提供了一些拓展任务（不局限于此），大家可以将自己的操作系统不断完善，越来越接近商业化的OS。

## 7.1 文件系统的抽象

本操作系统file system模块(./fs 文件夹)采用的是FAT32 (file allocate table 32 bits) 类Unix文件系统，文件系统基本单 位为块（Block），块大小为 64B，文件系统大小不超过 32KB。底层对块的读写是通过对磁盘的读写封装而来的其中的代码量巨大，但涉及的都是底层驱动，比如磁盘驱动等等（本案例中使用SD卡作为存储器的话就是SPI mode驱动）。这部分大约有四千行代码，但底层关于磁盘的操作已经给出，并不需要学生编写，感兴趣的话可以进行自主阅读。换言之，想要拓展的同学们完成的是抽象文件系统的编写，从而实现现代文件系统的功能。

梳理清楚模块之间的相互引用是完成本操作系统实验的重点，文件系统这一模块由于代码量很大，因此这里和大家简单梳理一下各个模块交互中的逻辑关系。首先在./fs 文件夹中，ffconf.h是关于文件系统的一些配置，这个已经写死在了里面，不需要更改。然后是elf.c 文件是一小段关于从SD卡中加载elf可执行文件的代码，也已经帮大家写好。因此需要仔细研读的就是ff.h 头文件及其对应的ff.c 源码文件。这里面是文件系统的核心内容，因为它定义了文件管理的基本数据结构，比如block、table、directory等以及相应的功能函数。

在ff.c 源码文件中，我们发现其引入了一个’diskio.h’ 的文件，这是因为对于文件系统的操作是建立在对于磁盘块的读写封装之上的，而diskio.h及diskio.c 恰恰就是在第三章第二节中的驱动实验(./drivers 文件夹下有各种外设相关的驱动，从最简单的LED灯到复杂的SD存储卡驱动)要完成的内容之一。而在diskio.c 中我们又引入了 sd.h 这个头文件，这个sd就是指代SD卡，因为在我们的开发板上并没有真正意义上的的磁盘，而SD卡就是我们能够使用的存储介质，因此在第三章第二节那个驱动实验中，需要首先写好SD卡驱动(sd.c)，然后写好磁盘驱动(diskio.c)，最后才能够在文件系统中使用这些底层的函数。

之前我们从SD卡中读取操作系统内核代码的bootload那段其实已经用到了文件系统，但是那是未经过封装的，换言之就是进行了一些非常底层的操作把数据读出来，这是不符合现代文件系统理念的，因此同学们可以根据已经写好的控制物理层的函数API来实现一个由superblock、inode的现代文件系统。本教程提供的代码来源于FAT开源框架，关于FAT的官方说明文档在 <http://elm-chan.org/fsw/ff/00index_e.html> 中，里面对于每个函数接口都有详尽的解释，想做抽象文件系统的同学们应当首先仔细阅读此网站。

## 7.2 VGA显示屏

由于我们的FPGA板子有VGA接口，因此显示功能也是可以完成的，这部分需要大家先在vivado里面进行VGA硬件控制器的设计，这部分对同学们的时序要求比较高，软件方面不需要太多的代码，因此也是值得尝试的。

## 7.3 页置换

因为在先前的内容中我们已经帮大家打通了SD卡，SD卡就是作为我们这个SOC的磁盘，因此可以实现虚拟内存的页置换功能，这一部分是之前没有同学做过的，但与此同时会增加不少用于管理磁盘和内存之间交互的数据结构，这个需要大家仔细地去思索，也是非常有意义的。

## 7.4 加载器的动态链接

静态加载器我们已经在bootload那一章节中完成了，它的任务就是将一段二进制代码从磁盘调入到内存中。但是高级编译技术会帮助我们使用动态链接库，而运行时的动态链接的功能就是操作系统负责的，因此可以对于load模块进行拓展以支持动态链接功能。

# 附录 如何运行整个工程？

完成了上述所有实验之后，他们只是相互独立的代码文件，如何将这些代码整合成一个可用的操作系统，并且在可重构芯片上运行起来是我们最后关注的问题！

首先，你需要确保全流程工具链的完整：

a) 默认配置安装 OpenOCD-0.9.3

b) 连 接 板 子 后 安 装 ZaDig.exe 下 拉 框 选 择 BUSBLASTERv3c

(INTERFFACE0)，绿色箭头右侧选择 libusbK (v3.0.7.0)

c) 安装 mipssdk. V.2.0.0，勾选所有组件安装

d) 安装 putty

e) 安装 Vivado2017.4

其次，打开 Vivado，上方选择 Flow>Open Hardware Maneger，然后选择 Open Target>Auto connect ， 然后 program device ， 将之前做硬件实验生成好的xxx.bit 位流文件烧进FPGA芯片；

接着，打开串口助手Putty，设置串行，波特率115200.

然后，在 Aurora/ushell 中编辑自己想要的用户程序，并在 ushell 目录下 make,编译 出 elf 文件，或者可以直接使用 Aurora/ushell/elf 文件夹中提供的文件，其运行内容为 Aurora/ushell/user/shell.c 中的同名函数。将 elf 文件存入 sd 卡中，sd 卡插到板子上，文件名不可超过 8 个字符。Aurora/init/init.c 的 sys\_init()中选择想打开的 elf 文件，然后在 aurora 目录 下 make，然后运行 Aurora/run.bat。

最后在打开的 gdb 窗口中输入 load vmlinux.elf ,然后按 c 即可运行内核，也可在 load 之后输入 file vmlinux.elf 载入符号表，加断点，进行 gdb 调试。注意：每次修改代码后，需要重新 make，并重新烧 bit 文件。