版本历史

文档更新记录	文档名:	Lab02_多周期 CPU 存储器替换
	版本号	V0.1
	创建人:	计算机体系结构研讨课教学组
	创建日期 :	2017-09-14

更新历史

序号	更新日期	更新人	版本号	更新内容
1	2017/09/14	邢金璋	V0.1	初版。

文档信息反馈: xingjinzhang@loongson.cn

1 实验二 多周期 CPU 存储器替换

在学习并尝试本章节前，你需要具有以下环境和能力：

(1) 装有 Vivado2017.1 或 Vivado2017.2 的电脑一台。

(2) 熟悉 Vivado 工具，并能初步使用。

参考文档“A04_Vivado2017.1 使用说明”。

(3) 熟悉龙芯体系结构实验箱（Artix-7）。

参考文档“A01_龙芯体系结构教学实验箱（Artix-7）介绍”。

(4) 已独立完成“实验一 同步存储器”实验。

通过本章节的学习，你将获得：

(1) 进一步的实验环境（Vivado 工具、龙芯体系结构实验箱）的熟悉。

(2) 简化的 SoC 系统的认识。

(3) 我们提供的 CPU 设计验证平台的熟悉和使用

在本章节的学习过程中，你可能需要查阅：

(1) 文档“A05_“体系结构研讨课”MIPS 指令系统规范”。

(2) 文档“A06_CPU 设计验证平台使用说明”。

1.1 实验目的

1. 进一步熟悉实验平台。
2. 迁移组成原理的多周期实验到本学期实验平台上。
3. 熟悉并运用 verilog 语言进行电路设计。
4. 为后续实验打好基础。

1.2 实验设备

1. 装有 Xilinx Vivado 的计算机一台。
2. 龙芯体系结构教学实验箱（Artix-7）一套。

1.3 实验任务

1. 迁移组成原理研讨课上完成的多周期 CPU 设计到本学期实验平台上。
2. 要求多周期 CPU 支持以下 15 条机器指令：LUI、ADDU、ADDIU、BEQ、BNE、LW、OR、SLT、SLTI、SLTIU、SLL、SW、J、JAL、JR。
3. 要求多周期 CPU 对外访存接口为取指、数据访问分开的两个 SRAM 接口，且 SRAM 接口是同步读、同步写的。
4. 要求多周期 CPU 顶层连出部分 debug 信号，以供验证平台使用。
5. 要求多周期 CPU 复位从虚拟地址 0xbfc00000 处取指。
6. 要求多周期 CPU 虚实地址转换采用：虚即是实，也就是转换过程不需要对虚拟地址作变换。
7. 要求多周期 CPU 将每条指令对应的 PC 每个周期都带下去，一直带到写回级（最后一级），这个信息，暂

时 mycpu 里不会用到，但验证平台会使用到。

8. 要求实现 MIPS 架构的延迟槽技术，可以认为软件在延迟槽放置的永远为 NOP 指令。
9. 要求多周期 CPU 只实现一个操作模式：核心模式，不要求实现其他操作模式。
10. 不要求支持例外和中断。
11. 不要求实现任何系统控制寄存器。
12. 要求将多周期 CPU 嵌入到提供的一个简化系统 SoC_lite 上。完成仿真和上板功能验证。

1.4 实验环境

本次实验使用的 SoC_lite 架构如下图：

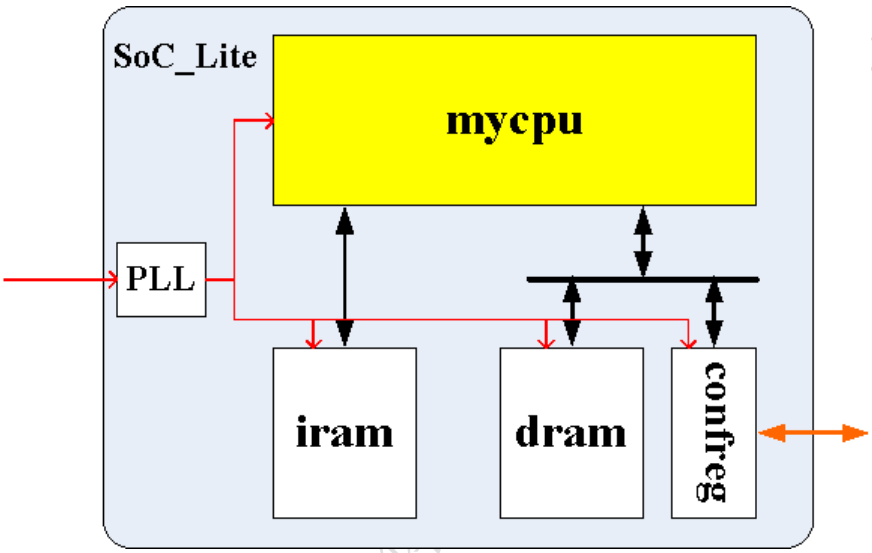

图 1-1 SoC_lite 架构

其中的 mycpu 为需要大家实现的，其实现基础为上学期组成原理研讨课的多周期 CPU，上图中其余模块均由我们提供。

除了提供以上环境外，我们还使用以上环境和龙芯开源 gs132 处理器核搭建了一个 CPU 设计的验证平台。该验证平台的搭建思路是：gs132 和 mycpu 运行统一程序，对比每条指令执行结果，一旦发现不一致则说明 mycpu 执行出错。该平台具体实现是：

(1) 使用 gs132 运行 func 抓取写回级（多周期最后一级）的 PC、写寄存器号、写数据作为参考结果存放在 trace_ref.txt 中；

(2) mycpu 时，运行同一 func，读取 trace_ref.txt，同时抓取 mycou 的取写回级的 PC、写寄存器号、写数据一一与之比对，一旦发现不一致，则说明 mycpu 执行出错，且能迅速定位出错的地方。

具体验证平台的使用，请参考文档“A06_CPU 设计验证平台使用说明”。

本次实验环境目录结构如下：红色部分为本次实验需要大家自行实现的，黑色部分为 lab2 提供的实验环境。整个目录已发布到课程网站上 lab2 目录下。

|-cpu132_gettrace/	目录，验证平台之生成参考 trace 部分。
| |--rtl/	目录，SoC_lite 的源码。
| | |--soc_lite_top.v	SoC_lite 的顶层。
| | |--CPU_gs132/	目录，龙芯开源 gs132 源码，对其顶层接口做了修改。
| | |--CONFREG/	目录，confreg 模块，连接 CPU 与开发板上数码管、拨码开关等 GPIO 类设备。
| | |--BRIDGE/	目录，bridge_1x2 模块，CPU 的 data sram 接口分流去往 confreg 和 data_ram。

| | |--xilinx_ip/	目录，Xilinx IP，包含 clk_pll、inst_ram、data_ram。
| |--testbench/	目录，仿真文件。
| | |--tb_top.v	仿真顶层，该模块会抓取 debug 信息生成到 trace_ref.txt 中。
| |--run_vivado/	目录，运行 Vivado 工程。
| | |--soc_lite.xdc	Vivado 工程设计的约束文件
| | |--cpu132_gettrace/	目录，Vivado2017.1 创建的 Vivado 工程，名字就叫 cpu132_gettrace
| | | |--cpu132_gettrace.xpr	Vivado2017.1 创建的 Vivado 工程，可使用 Vivado2017.1 或 2017.2 打开
| |--trace_ref.txt	该验证平台运行测试 func 生成的参考 trace。lab2 发布包已包含该文件。
|	
|-func/	目录，验证平台之测试 func 交叉编译部分。lab2 发布包已包含编译结果。
| |--include/	目录，mips 编译所有头文件
| | |--asm.h	MIPS 汇编需用到的一个宏定义的头文件，比如 LEAF(x)。
| | |--regdef.h	MIPS 汇编 32 个通用寄存器的助记符定义。
| |--inst/	目录，各条指令的验证汇编程序
| | |--Makefile	子目录里的 Makefile，会被上一层次的 Makefile 调用。
| | |--inst_test.h	各条指令的验证程序使用的宏定义头文件
| | |--n*.S	各条指令的验证程序，汇编语言编写。
| |--obj/	目录，func 编译结果
| | |--sim_test_ram/	目录，用于仿真的编译结果
| | |--syn_test_ram/	目录，用于综合实现的编译结果
| |--Makefile	编译脚本 Makefile
| |--start.S	func 的主函数
| |--bin.lds.S	交叉编译的链接脚本
| |--convert.c	生成 coe 和 mif 文件的本地执行程序源码。
| |--rules.make	子编译脚本，被 Makefile 调用
|	
|-mycpu_verify/	目录，自实现 CPU 的验证环境
| |--rtl/	目录， SoC_lite 的源码。
| | |--soc_lite_top.v	SoC_lite 的顶层。
| | |--myCPU /	**目录，自实现 CPU 源码。**
| | |--CONFREG/	目录，confreg 模块，连接 CPU 与开发板上数码管、拨码开关等 GPIO 类设备。
| | |--BRIDGE/	目录，bridge_1x2 模块，CPU 的 data sram 接口分流去往 confreg 和 data_ram。
| | |--xilinx_ip/	目录，Xilinx IP，包含 clk_pll、inst_ram、data_ram。
| |--testbench/	目录，仿真文件。
| | |--mycpu_tb.v	仿真顶层，该模块会抓取 debug 信息与 trace_ref.txt 进行比对。
| |--run_vivado/	目录，运行 Vivado 工程。
| | |--soc_lite.xdc	Vivado 工程设计的约束文件
| | |--mycpu/	目录，Vivado2017.1 创建的 Vivado 工程，名字就叫 mycpu
| | | |--mycpu.xpr	Vivado2017.1 创建的 Vivado 工程，可使用 Vivado2017.1 或 2017.2 打开

1.5 实验说明

1.5.1 myCPU 实现指令集

要求实现以下 15 条指令：LUI、ADDU、ADDIU、BEQ、BNE、LW、OR、SLT、SLTI、SLTIU、SLL、SW、J、JAL、JR。

关于上述指令的详细定义可以参考文档“A05_“体系结构研讨课”MIPS 指令系统规范”。

1.5.2 myCPU 顶层接口

本实验要求 myCPU 按以下接口封装，这样可以直接套用图 1 中的 SoC_lite 架构。

表 1-1 myCPU 顶层接口信号描述

名称	宽度	方向	描述
时钟与复位			
clk	1	input	时钟信号，来自 clk_pll 的输出时钟
resetn	1	input	复位信号，低电平同步复位
取指端访存接口			
inst_sram_en	1	output	ram 使能信号，高电平有效
inst_sram_wen	4	output	ram 字节写使能信号，高电平有效
inst_sram_addr	32	output	ram 读写地址，字节寻址
inst_sram_wdata	32	output	ram 写数据
inst_sram_rdata	32	input	ram 读数据
数据端访存接口			
data_sram_en	1	output	ram 使能信号，高电平有效
data_sram_wen	4	output	ram 字节写使能信号，高电平有效
data_sram_addr	32	output	ram 读写地址，字节寻址
data_sram_wdata	32	output	ram 写数据
data_sram_rdata	32	input	ram 读数据
debug 信号，供验证平台使用			
debug_wb_pc	32	output	写回级（多周期最后一级）的 PC，因而需要 mycpu 里将 PC 一路带到写回级
debug_wb_rf_wen	4	output	写回级写寄存器堆(regfiles)的写使能，为字节写使能，如果 mycpu 写 regfiles 为单字节写使能，则将写使能扩展成 4 位即可。
debug_wb_rf_wnum	5	output	写回级写 regfiles 的目的寄存器号
debug_wb_rf_wdata	32	output	写回级写 regfiles 的写数据

对于多周期CPU的对外访存接口，接口为取指、访存分开的两个SRAM接口。分别连接到对应的RAM IP中。

(1) 所有取指都从取指端 inst_sram 发出，去往 inst_ram IP，该 RAM 为一个读写端口的 Xilinx IP，但只使用其读功能，因而 CPU 出来的 inst sram 接口的写使能(wen)和写数据(wdata)时钟接 0 即可。

(2) 所有数据访问都从数据端 data_sram 发出，去往 data_ram IP，该 RAM 为一个读写端口的 Xilinx IP，其写使能设置为字节写使能。

(3) 需要注意，计算机里是以字节寻址，寻址一个字（4 字节，32bit）的数据，需要访问 4 个地址，比如 0x00000000~0x00000003，但在 RAM 里是按数据宽度寻址的，我们生成的 RAM 数据宽度为一个字，寻址一个字的数据，只需要访问一个地址，比如 0。因而在 RAM 和 CPU 的地址接口连接时，需要注信号的地

址，假设 RAM 地址宽度为 w，则地址信号对其关系为：ram_addr[w-1:0] <-> cpu_addr[w+1:2]。在本实验中，inst_ram 地址宽度为 18 位，故对应 cpu_inst_addr[19:2]；data_ram 地址宽度为 16 位，对应 cpu_data_ram[17:2]。具体可以参考 soc_lite_top.v 里 RAM 实例化处的代码。

1.5.3 myCPU 复位 PC

要求多周期 CPU 复位从 0xbfc00000 处取指（MIPS 架构规范要求如此，以后解释）。由于本实验 inst 和 data ram 地址宽度有限，并没有使用到 32 位地址的高 12 位，因而 CPU 复位 PC 为 0xbfc00000 和为 0x00000000 效果一样，但我们还是要求尽量符合规范。

13. 要求将多周期 CPU 嵌入到提供的一个简化系统 SoC_lite 上。完成仿真测试和上板测试。所有测试只对第 1 点中的 15 条指令进行测试。

1.5.4 myCPU 虚实转换

MIPS 常用虚实转换机制包括：固定地址映射（FMT）和 TLB 机制。但本次实验均不要求实现这两种机制，我们默认虚地址即是实地址。

1.5.5 myCPU 延迟槽实现要求

本次实验对延迟槽实现的要求，延续组成原理研讨课的要求，即认为软件程序有延迟槽，但延迟槽里存放的指令永远为 NOP 指令。

1.5.6 myCPU 操作模式

MIPS 架构 CPU 操作模式有核心模式、监管模式和用户模式。但本次实验只要求实现核心模式，即认为程序用于运行与核心模式。

1.5.7 myCPU 不要求实现项

本次实验不要求是实现中断和例外，不要求实现任何系统控制寄存器。

1.5.8 myCPU 验证

验证时，通常步骤为：

(1) 在 Linux 下使用交叉编译工具编译测试 func，得到编译结果。（lab2 发布包里，已包含 func 编译结果，可跳过该步骤。因而本次实验可以不需要安装 Linux 虚拟机和交叉编译工具链，但建议大家还是尽早装好虚拟机，后续实现会用到的。）

(2) Vivado 仿真运行 cpu132_gettrace，inst_ram 加载 func 用于仿真的编译结果，生成参考结果 trace_ref.txt。（lab2 发布包里，已生成好 trace_ref.txt，可跳过该步骤。）

(3) Vivado 仿真运行 mycpu，inst_ram 加载 func 用于仿真的编译结果，参考仿真结果是 error 还是 pass。

(4) 第(3)步中，如果 error，则进行仿真 debug，重复第(3)、(4)步。

(5) 第(3)步中，如果 pass，则 Vivado 综合实现 mycpu，此时 inst_ram 需要加载 func 用于综合实现的编译结果，进行上板验证，观察实验箱上数码管显示结果，判断是否正确。

(6) 第(5)步中，如果判断上板正确，则 myCPU 验证成功，结束。

通常仿真验证通过，上板验证也会通过，也就是到上述第（6）点就结束了。但如果在上述第（5）点中，判断上板不正确，则出现了“仿真通过，上板失败”的情况，此时通常是时序不满足或者代码不规范导致的，请按以下步骤去排查。

(1) 复核生成、下载的 bit 文件是否正确。

(2) 第(1)步中，如果判断生成的 bit 文件不正确，则重新生成 bit 文件。

(3) 第(1)步中，如果判断生成的 bit 文件正确，则复核仿真结果是否正确。

(4) 第(3)步中，如果判断仿真结果不正确，则重新仿真 debug。

(5) 第(3)步中，如果判断仿真结果正确，则检查实现时的时序报告（Vivado 界面左侧“IMPLEMENTATION”->”Open Implemented Design”->”Report Timing Summary”）。

(6) 第(5)步中，如果发现实现时时序不满足，则在 Verilog 设计里调优不满足的路径，或者降低 SoC_lite 的运行频率，也就是降低 clk_pll 模块的输出端频率。

(7) 第(5)步中，如果实现时时序是满足的，则认真排查综合和实现时的 Warning，将能修正的 Warning 都修正了。

(8) 第(7)步后，重新仿真通过后，重新生成 bit 文件上板依然不通过，则使用 Vivado 的逻辑分析仪进行板上在线调试。Vivado 在线调试方法说明文档以后再补充，希望大家不会到这一步。

CPU 设计验证平台的详细使用，请参考文档“A06_CPU 设计验证平台使用说明”。