计算机组成原理实验阶段二讲义

计算机组成原理实验课授课团队

2021-11-20

献给……

目录

表格

插图

单周期 CPU 设计阶段二

0.1 任务与实践

本实践任务要求：1. 在阶段一完成的 CPU 基础上，增加 sub.w、slt、sltu、slli.w、srli.w、srai.w、lu12i.w、and、or、nor、xor、beq、b、bl 和 jirl 指令的支持。2. 运行 func 测试通过仿真和上板验证。

本实践任务需要使用的环境为 lab3.zip，提供了 CPU 设计实验开发环境（CPU_CDE），包含了 Vivada 工程文件、全部的 RTL 文件和生成好的 IP，读者需要调试的 CPU 设计源码位于 mycpu_verify/rtl/myCPU 中。CPU_CDE 的目录结构和使用方法参见本章第0.2节。

请参考下列步骤完成实践任务一：

1. 将实验环境发布包解压到路径上无中文字符的目录里，实验环境为 CPU_CDE2。
2. 打开 gettrace 工程（CPU_CDE/gettrace/gettrace.xpr）。CPU_CDE 里的 Vivado 工程是使用 Vivado2019.2 创建的，如果使用更高版本的 Vivado 打开，请参考本章第0.2.4.1小节进行工程和 IP 核升级。
3. 确定 gettrace 工程中 soc_lite_top.v 中 INST_COE 宏定义指向的是对应 func 的 mif 文件（CPU_CDE2/func/obj/inst_ram.mif）。
4. 运行 gettrace 工程的仿真（进入仿真界面后，直接点击 run all 等待仿真运行完成），生成新的参考 trace 文件 golden_trace.txt（CPU_CDE/gettrace/golden_trace.txt）。要等仿真运行完成，golden_trace.txt 才有完整的内容。
5. 打开 myCPU 工程（CPU_CDE2/mycpu_verify/run_vivado/mycpu_prj1/mycpu.xpr）。如需要，请参考本章第0.2.4.1小节进行工程和 IP 核升级。
6. 运行 myCPU 工程的仿真（进入仿真界面后，直接点击 run all），开始 debug。
7. myCPU 仿真通过后，综合实现后生成 bit 流文件，进行上板验证。

0.2 CPU 设计开发环境（CPU_CDE）

本小节介绍 CPU 的设计所使用的开发环境——CPU_CDE。

0.2.1 快速上手 CPU 设计的开发环境

在使用 CPU 设计开发环境时，主要包括以下几项工作。

0.2.1.1 解压环境

将提供的开发环境压缩包解压到一个路径中没有中文字符的位置上，并且要确保能在这个位置运行 Vivado 软件。

0.2.1.2 设计自己的 CPU

用你习惯使用的文本编辑器[1]将所设计的 CPU 的 Verilog 代码描述出来。重点注意顶层模块的模块名和接口信号必须按照规定要求定义。

0.2.1.3 集成自己的 CPU

将写好的 CPU 的 Verilog 代码拷贝到 mycpu_verify/rtl/myCPU/目录下。

0.2.1.4 编译测试程序

如果是在 Windows 操作系统下面运行 Vivado：先确保你的虚拟机中运行着一个已经安装了 LoongArch-GCC 交叉编译工具的 Linux 操作系统，将 func 目录设置为虚拟机共享目录[2]。在虚拟机的 Linux 操作系统中进入 func 目录，先运行 make clean，再运行 make。回到 Windows 操作系统下，确认 func/obj/整个目录下的内容确实是最新编译更新的。

如果是在 Linux 操作系统下运行 Vivado：先确保你的 Linux 系统已经安装了 LoongArch-GCC 交叉编译工具。进入 func 目录先运行 make clean，再运行 make 就可以了。

有关 LoongArch-GCC 交叉编译工具的安装，请参看0.2.2节内容。

0.2.1.5 生成比对 Trace

进入 gettrace/目录，打开 Vivado 工程 gettrace.xpr，进行仿真，生成参考结果 golden_trace.txt。重点注意此时 inst_ram 加载的确实是前一个步骤编译出的结果。要等仿真运行完成，golden_trace.txt 才有完整的内容。

[1] Vivado 中集成的文本编辑器功能比较简单，强烈建议各位使用一个专门用于代码开发的文本编辑软件来编写代码。

[2] 针对本书中实验所涉及的程序编译工作，Windows 操作系统自带的 Windows Subsystem for Linux 就已经完全够用。wsl 无需进行此操作，即可通过访问/mnt/XXX 完成对 Windows 系统下 XXX 目录的访问。

0.2.1.6 仿真验证自己的 CPU

进入 mycpu_verify/run_vivado/mycpu_prj1/目录，打开 Vivado 工程 mycpu.xpr，再次确认 mycpu_verify/rtl/myCPU/目录下的设计代码是你准备验证的版本，确认无误后将它们作为设计文件添加到工程中，开始仿真。观察仿真 log 输出以确定是否出现结果异常。如果结果异常，则进行调试直至功能仿真通过。重点注意此时 inst_ram 和 data_ram 中加载的内容确实是需要进行的测试程序编译出的结果[3]，即 func/obj/目录下的内容。

0.2.1.7 上板验证自己的 CPU

若仿真结果正常即可进入上板检测环节。回到 mycpu 这个工程中，进行综合实现，成功后即上板进行检测，观察实验箱上数码管显示结果是否与要求的一致。若一致则此次实验成功；否则转到下面的调试阶段进行问题排查。

0.2.1.8 调试自己的 CPU

请按照下列步骤排查，然后重复仿真验证、上板验证、调试三个过程直至正确。

1. 复核生成、下载的 bit 文件是否正确。
   - 如果判断生成的 bit 文件不正确，则重新生成 bit 文件。
   - 如果判断生成的 bit 文件正确，转步骤 2
2. 复核仿真结果是否正确。
   - 如果仿真验证结果不正确[4]，则回到前面仿真验证步骤。
   - 如果仿真验证结果正确，转步骤 3。
3. 检查实现（Implementation）后的时序报告（Vivado 界面左侧“IMPLEMENTATION”→“Open Implemented Design”→“Report Timing Summary”）。
   - 如果发现时序不满足，则在 Verilog 设计里调优不满足的路径，然后回到前面的仿真验证环节依序执行各项操作；或者降低 SoC_lite 的运行频率，即降低 clk_pll 模块的输出端频率，然后回到前面上板验证环节依序执行各项操作。
   - 如果实现时时序是满足的，转步骤 4。
4. 认真排查综合和实现时的 Warning。
   - Critical warning 是强烈建议要修正的，warning 是建议尽量修正的，然后回到前面上板验证环节依序执行各项操作。
   - 如果没有可修正的 Warning 了，转步骤 5。

[3]实验者有时候会出现修改了测试程序的源代码但是忘记编译的操作疏忽。若怀疑此处，建议检查一下 func/obj/目录下生成出的 *.coe 和 *.mif 的时间是否晚于 func/src/目录下源代码文件的最后修改时间。

[4]如果仿真验证都没有通过就上板，我们只能表扬你勇气可嘉。

5. 人工检查 RTL 代码，避免多驱动、阻塞赋值乱用、模块端口乱接、时钟复位信号接错、模块调用处的输入输出接反，查看那些从别处模仿来的“酷炫”风格的代码，查找有没有仿真时被 force 住的值导致仿真和上板不一致……如果怎么看代码都看不出问题，转步骤 6。
6. 参考附录 C 第 1 节“使用 Chipscope 在线调试”进行板上在线调试；如果调试了半天仍然无法解决问题，转步骤 7。
7. 反思。真的，现在除了反思还能干什么?

根据我们的教学和培训经验，在此重点提醒读者，很多“仿真通过，上板不过”都是以下问题之一导致的：

1. 多驱动。
2. 模块的 input/output 端口接入的信号方向不对。
3. 时钟复位信号接错。
4. 代码不规范，阻塞赋值乱用，always 语句随意使用。
5. 仿真时控制信号有“X”。仿真时，有“X”调“X”，有“Z”调“Z”。特别是设计的顶层接口上不要出现“X”和“Z”。
6. 时序违约。
7. 模块里的控制路径上的信号未进行复位。

0.2.2 LoongArch-GCC 交叉编译工具的安装

LoongArch-GCC 交叉编译工具的下载方式为：打开链接

http://114.242.206.180:24989/nextcloud/index.php/s/7xXTPkWg6Jn5KLW ，

选择 install.tar.gz 压缩包，右键然后点击 Download。请注意最终要将该压缩包存于 Linux 操作系统自身的文件系统中。

具体安装步骤如下：

（1）打开一个 terminal，进入 install.tar.gz 所在目录，进行解压：

```
[abc@www ~]$ sudo tar -xvf install.tar.gz -C /
```

（2）确保目录“/install/bin”存在，随后执行：

```
[abc@www ~]$ echo  “export PATH=/install/bin:$PATH”  >> ~/.bashrc
```

（3）安装依赖库：

```
[abc@www ~]$ sudo apt-get install libisl-dev libmpfr-dev
```

（4）重新打开一个 terminal，输入 loongarch32 然后敲击 tab 键，如果能够-unknown-之类的补全，就说明工具链已经安装成功。此时可以编写一个 hello.c 然后用工具链进行编译看其是否可以工作。

```
[abc@www ~]$ loongarch32-unknown-elf-gcc hello.c
```

（5）如果编译出现错误，基本上是依赖库的问题，可运行如下命令查看缺少的库文件。

```
[abc@www ~]$ ldd /install/libexec/gcc/loongarch32-unknown-elf/8.3.0/cc1
```

通常所缺的依赖库文件是 libmpfr.so.4 和 libisl.so.15，此时有两种建议的解决方式。

方式 1

从链接 http://114.242.206.180:24989/nextcloud/index.php/s/7xXTPkWg6Jn5KLW 所在页面下载 lib.tar，解压后执行如下命令将库文件拷贝到指令目录（同 ldd 打印中其他库文件所在位置相同）

```
[abc@www ~]$ cp ./lib/libmpfr.so.4 /usr/lib/x86_64-linux-gnu/
[abc@www ~]$ cp ./lib/libmpfr.so.6 /usr/lib/x86_64-linux-gnu/
[abc@www ~]$ cp ./lib/libisl.so.15 /usr/lib/x86_64-linux-gnu/
```

方式 2

在/usr/lib/x86_64-linux-gnu/目录下建立所缺库文件名称的软链接，让其链接到这个库文件的更高（最新）版本。假设当前系统中存自库文件 libisl.so.19 和 libmpfr.so，那么可以执行如下命令：

```
cd /usr/lib/x86_64-linux-gnu/
sudo ln -s libisl.so.19 libisl.so.15
sudo ln -s libmpfr.so libmpfr.so.4
```

因为大部分高版本的库都会兼容低版本的库，所以上述操作风险不大（但也不排除）。不过，不同操作系统或者不同的版本，上述库文件的版本可能有差异，在进行软链接之前，先查看一下在/usr/lib/x86_64-linux-gnu/路径下的 libisl.so.xx 的版本号，然后合理替换，libmpfr.so 也同理，先查看一下有没有 libmpfr.so 这个文件。

0.2.3 CPU 设计开发环境（CPU_CDE）的组织结构介绍

整个 CPU 设计实验开发环境（CPU_CDE）的目录结构及各部分功能简介如下所示。其中只有 mycpu_verify/rtl/myCPU/目录中的内容才是需要大家自行开发的，其余部分都已经开发好了。

```
|-gettrace/              生成参考trace的部分。
|  |--src/               设计代码目录。
|  |  |--tb_top.v        仿真顶层，该模块会抓取debug信息生成到golden_trace.txt中。
|  |  |--soc_lite_top.v  SoC_lite的顶层文件。
|  |  |--myCPU/*         产生比对trace的单周期处理器核设计。
|  |  |--CONFREG/        confreg模块，用于访问CPU与开发板上数码管、拨码开关等外设。
|  |  |--BRIDGE/         1×2的桥接模块，CPU的data sram接口同时访问confreg和data_ram。
|  |--gettrace.xpr       Vivado工程文件。
|  |--golden_trace.txt   运行func测试程序所生成的参考trace。
|
|--func/                 实验任务所用的功能验证测试程序。
|  |--include/           功能验证测试程序共享的头文件所在目录。
|  |  |--sysdep.h        一些GCC通用的宏定义的头文件。
|  |  |--asm.h           LoongArch汇编需用到的一些宏定义的头文件，比如LEAF(x)。
|  |  |--regdef.h        LoongArch32 ABI下，32个通用寄存器的汇编助记定义。
|  |  |--cpu_cde.h       SoC_Lite相关参数的宏定义，如访问数码管的confreg的基址。
|  |  |--inst_test.h     各功能测试点的验证程序使用的宏定义头文件
|  |--inst/              各功能测试点的汇编程序文件。
|  |  |--Makefile        子目录里的Makefile，会被上一级目录中的Makefile调用。
|  |  |--n*.S            各功能测试点的验证程序，汇编语言编写。
|  |--obj/               功能验证测试程序编译结果存放目录
|  |  |--*               详见后面小节的说明。
|  |--start.S            功能验证测试的引导代码及主函数。
|  |--Makefile           编译功能验证测试程序的Makefile脚本
|  |--bin.lds            编译bin.lds.S得到的结果，可被make reset命令清除
|  |--convert.c          生成coe和mif文件的处理工具的C程序源码
|  |--Readme_first.txt   简单说明文件。
|--*                     其他功能或性能测试程序。
|
|-mycpu_verify/          读者实现的CPU的验证环境
   |--rtl/               SoC_lite设计代码目录。
   |  |--soc_lite_top.v  SoC_lite的顶层文件。
   |  |--myCPU/*         自己实现的CPU的RTL代码。
   |  |--CONFREG/        confreg模块，用于访问CPU与开发板上数码管、拨码开关等外设。
```

```
|  |--BRIDGE/          1×2的桥接模块， CPU的data sram接口同时访问confreg和data_ram。
|  |--xilinx_ip/       定制的Xilinx IP，包含clk_pll、inst_ram、data_ram。
|--testbench/          功能仿真验证平台。
|  |--mycpu_tb.v       功能仿真顶层，该模块会抓取debug信息与golden_trace.txt进行比对。
|--run_vivado/         Vivado工程的运行目录。
   |--soc_lite.xdc     Vivado工程设计的约束文件。
   |--mycpu_prj1/      创建的第一个Vivado工程，名字为mycpu_prj1。
   |   |--*.xpr        Vivado创建的工程文件，可直接打开。
   |--*                创建的其他Vivado工程。
```

0.2.3.1 验证环境模拟的计算机硬件系统

单纯实现一个 CPU 没有什么实际使用意义，通常我们需要基于 CPU 搭建出一个计算机硬件系统。本学期的实验课程中，我们也是基于实现的 CPU 搭建出一个计算机硬件系统，然后通过在这个计算机硬件系统上运行测试程序来完成 CPU 的功能验证。

在引入 AXI 总线接口设计之前，我们将采用一个简单的计算机硬件系统。这个硬件系统将通过 FPGA 开发板实现。其核心是在 FPGA 芯片实现的一个片上系统（System On Chip，简称 SoC）。这个 SoC 芯片通过引脚连接到电路板上的时钟晶振、复位电路，以及 LED 灯、数码管、按键这些外设接口设备。SoC 芯片内部也是一个小系统，其顶层为 SoC_Lite，内部结构示意如图1所示，对应的 RTL 代码均位于 mycpu_verify/rtl/目录下。我们重点关注 SoC_Lite 这个小系统。可以看到 SoC_Lite 的核心是各位将要实现的 CPU——mycpu。这个 CPU 与指令 RAM（iram）和数据 RAM（dram）进行交互完成取指和访存。除此而外这个小系统中还包含 PLL、confreg 等模块。

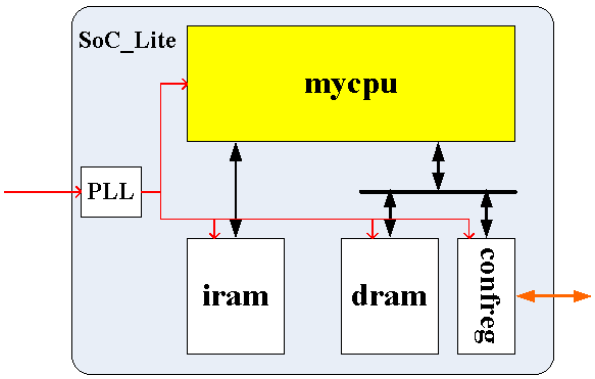

图 1: 用于验证 mycpu 的简单硬件系统

指令 RAM 和数据 RAM 和 CPU 之间的关系大家已经清楚了。这里简单解释一下 PLL、confreg 以及 mycpu 与 dram、confreg 之间的“一分二”功能。

我们开发板上给 FPGA 芯片提供的时钟（来自于时钟晶振）主频是 100MHz。如果直接用这个时钟作为 SoC_Lite 中各个模块的时钟，则意味着 mycpu 至少要能达到 100MHz 的主频。这对于初学者来说这可能是个比较严格的要求，因此我们添加了一个 PLL IP，将其输出时钟作为

mycpu 的时钟输入。这个 PLL 以 100MHz 输入时钟作为参考时钟，输出时钟频率可以配置为低于 100MHz。

confreg 是 “configuration register” 的简称，是 SoC 内部的一些配置寄存器。实验所搭建的 SoC 系统中，mycpu 是通过访问 confreg 来驱动板上的 LED 灯、数码管，接收外部按键的输入。简要解释一下这个操控的机理：外部的 LED 灯、数码管以及按键都是导线直接连接到 FPGA 的引脚上的，通过控制 FPGA 输出引脚上的电平的高、低就可以控制 LED 灯和数码管，同样的一个按键是否按下也可以通过观察 FPGA 输入引脚上电平的变化来判断。而这些 FPGA 引脚又进一步连接到 confreg 中某些寄存器的某些位上。所以 mycpu 可以通过写 confreg 寄存器就可以控制输出引脚的电平进而控制 LED 灯和数码管，也可以通过读 confreg 寄存器来知晓连接到按键的引脚是高电平还是低电平。

mycpu 和 dram、confreg 之间有一个 “一分二” 部件。这是因为在 LoongArch 指令系统架构下，所有 I/O 设备的寄存器都是采用 memory mapped 方式访问的。我们这里实现的 confreg 也不例外。Memory mapped 的访问方式意味 I/O 设备中的寄存器各自都有一个唯一内存编址，所以 CPU 可以通过 load、store 指令对其进行访问。不过 dram 作为内存也是通过 load、store 指令进行访问的。那么对于一条 load 或 store 指令来说，如何知晓它访问的是 confreg 还是 dram? 我们在设计 SoC 的时候用地址对其进行区分。因此在设计 SoC 的数据通路时就需要在这里引入一个 “一分二” 部件，它的选择控制信号生成是通过对访存的地址范围进行判断而得到的。

提醒读者，因为整个 SoC_Lite 的设计都要实现到 FPGA 芯片中，所以在进行综合实现的时候，你所选择的顶层应该是 soc_lite，不是你自己写的 mycpu。

myCPU 的顶层接口

为了让各位读者设计的 CPU 能够直接集成到本书所提供的 CPU 实验环境中，要对 CPU 的顶层接口做出明确的规定。myCPU 顶层接口信号的详细定义如表1所示。

表 1: myCPU 顶层接口信号的描述

名称	宽度	方向	描述
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
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
data_sram_wen	4	output	RAM 字节写使能信号，高电平有效

名称	宽度	方向	描述
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_wb_pc	32	output	写回级（多周期最后一级）的 PC，需要 myCPU 里将 PC 一路传递到写回级
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
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

0.2.3.2 功能仿真验证

数字电路的功能验证是为了检查所设计的数字电路在功能上是否符合设计目标。简单来说，就是检查设计的电路功能对不对。读者应该都开发过 C 语言程序，都知道写完的程序要测试一下正确性。我们这里说的功能验证与软件开发里面的功能测试的意图是一样的。但是我们用“验证（Verification）”这个词，是为了避免和本领域另一个概念“测试（Test）”相混淆。在我们集成电路设计领域，测试通常指检查生产出的电路没有物理上的缺陷和偏差，能够正常体现设计所期望的电路行为和电气特性。

所谓数字电路的功能仿真验证，就是用（软件模拟）仿真的方式而非电路实测的方式进行电路的功能验证。图2给出了数字电路功能仿真验证的一个基本框架。

在这个基本框架中，我们给待验证电路（DUT）一些特定的输入激励，然后观察 DUT 的输出结果是否如我们预期。所谓“不管白猫、黑猫，只要抓到老鼠就是好猫”，这里“老鼠”就是激励，期望结果是“抓到老鼠”，只要被验证对象在这个激励下得到所期望的结果，哪怕它明明是只黄鼠狼，我们也认为它是一只好猫。

我们给 CPU 设计进行功能仿真验证时，沿用的依然是上面的思路，但是在输入激励和输出结果检查两方面的具体处理方式与简单的数字电路设计存在区别。简单数据电路的功能仿真验证，通常是产生一系列变化的激励信号，输入到被验证的电路的输入端口上，然后观察电路输出端口的信号判断结果是否符号预期。然而对于 CPU 来说，其输入输出端口只有时钟、复位和 I/O，采用这种直接驱动和观察输入输出端口的方式，验证效率太低。

我们采用测试程序作为 CPU 功能验证的激励。即输入激励是一段测试指令序列，通常是用汇编语言或 C 语言编写，用编译器编译出来的机器代码。我们通过观察测试程序的执行结果是否符合预期，来判断 CPU 功能是否正确。这样做验证的效率是大幅度提高了，但是验证过程中出错后定位出错点的调试难度也相应提升了。考虑到初学者的调试能力尚在建立过程中，我们提供了一套基于 trace 比对的调试辅助手段，用以帮助在调试过程中更加快速的定位。

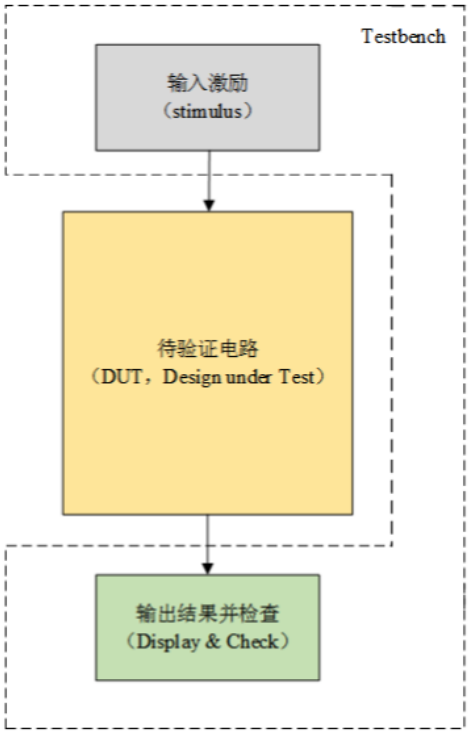

图 2: 功能仿真验证的基本框架

（1）基于 trace 比对的调试辅助手段

读者在调试 C 程序的时候应该都使用过单步调试这种调试手段。在“慢动作”运行程序的每一行代码的情况下，能够及时看到每一行代码的运行行为是否符合预期，从而能够及时定位到出错点。我们在实验开发环境中提供给读者的这套基于 trace 比对的调试辅助手段，借鉴的就是这种“单步调试”的策略。

其具体实现方式是：我们先用一个已知的功能上是正确的 CPU 运行一遍测试指令序列，将每条指令的 PC 和写寄存器的信息记录下，记为 golden_trace；然后在验证 myCPU 的时候运行相同的指令序列，在 myCPU 每条指令写寄存器的时候，将 myCPU 中的 PC 和写寄存器的信息同之前的 golden_trace 进行比对，如果不一样，那么立刻报错并停止仿真。

对 LoongArch 指令熟悉的读者可能马上就会问：有些转移指令和 store 指令不写寄存器，上面的方式没法判断啊？这些读者提的问题相当对。不过一旦分支跳转的不对，那么错误路径上第一条会写寄存器的指令的 PC 就会和 golden_trace 中的不一致，就会报错并停下来。store 指令执行错了，后续从这个位置读数的 load 指令写入寄存器的值就会与 golden_trace 中的不一致，也会报错并停下来。虽然报错的位置稍微有些靠后，但总体上还是有规律可循的。分支指令和 store 指令的及时报错不是不可以实现，只不过它会进一步增加 myCPU 上调试接口的复杂度，也会在一定程度上限制 myCPU 实现分支指令和 store 指令的自由度，权衡利弊后，我们采取了用少量投入解决大部分问题的设计思路。

上面我们介绍了利用 trace 进行功能仿真验证错误定位的基本思路，下面我们再具体介绍一下如何生成 golden_trace，以及 myCPU 验证的时候是如何利用 golden_trace 进行比对的。

（2）利用参考模型生成 golden_trace

功能验证程序 func 编译完成后，就可以使用验证平台里的 gettrace 工程运行仿真生成参考 trace 了。gettrace 这个工程中里所用的 SoC_Lite 和验证 myCPU 所用 SoC_Lite 架构几乎一样，主要区别就是里面所用的处理器核不一样。

仿真顶层为 gettrace/src/tb_top.v，与抓取 golden_trace 相关的重要代码如下：

```
......
`define TRACE_REF_FILE "../../../../golden_trace.txt"  //参考trace的存放目录
`define END_PC 32'h1c000100                //func测试完成后会32'h1c000100处死循环
......
assign debug_wb_pc       = soc_lite.debug_wb_pc;
assign debug_wb_rf_wen   = soc_lite.debug_wb_rf_wen;
assign debug_wb_rf_wnum  = soc_lite.debug_wb_rf_wnum;
assign debug_wb_rf_wdata = soc_lite.debug_wb_rf_wdata;
......
// open the trace file;
integer trace_ref;
initial begin
    trace_ref = $fopen(`TRACE_REF_FILE, "w");   //打开trace文件
end

// generate trace
always @(posedge soc_clk)
begin
    if(|debug_wb_rf_wen && debug_wb_rf_wnum!=5'd0)    //trace采样时机
    begin
        $fdisplay(trace_ref, "%h %h %h %h" , `CONFREG_OPEN_TRACE ,
            debug_wb_pc, debug_wb_rf_wnum, debug_wb_rf_wdata_v);   //trace采样信号
    end
end
......
```

Trace 采样的信号包括：

1. CPU 写回级（Write Back，WB）的 PC，要求大家将每条指令的 PC 一路带到写回级。
2. 写回级的写回使能。
3. 写回级的写回的目的寄存器号。
4. 写回级的写回的目的操作数。

显然并不是每时每刻 CPU 都有写回，因此 Trace 采样需要有一定的时机：wb 级写通用寄存器堆信号有效，且写回的目的寄存器号非 0。大家可以思考下，为什么此处判断写回目的寄存器非 0 时才采样?

（3）使用 golden_trace 监控 myCPU

myCPU 功能验证所使用的 SoC_Lite 与 gettrace 工程中的架构一致，但 testbench 就有所不同了，见 mycpu_verify/testbench/mycpu_tb.v，重点部分代码如下：

```
......
`define TRACE_REF_FILE "../../../../../../../gettrace/golden_trace.txt"
                                                    //参考trace的存放目录
`define CONFREG_NUM_REG soc_lite.confreg.num_data  //confreg中数码管寄存器的数据
`define END_PC 32'h1c000100                        //func测试完成后会在32'h1c000100处死循环
......
assign debug_wb_pc          = soc_lite.debug_wb_pc;
assign debug_wb_rf_wen      = soc_lite.debug_wb_rf_wen;
assign debug_wb_rf_wnum     = soc_lite.debug_wb_rf_wnum;
assign debug_wb_rf_wdata    = soc_lite.debug_wb_rf_wdata;
......
//get reference result in falling edge
reg [31:0] ref_wb_pc;
reg [4 :0] ref_wb_rf_wnum;
reg [31:0] ref_wb_rf_wdata_v;
always @(negedge soc_clk)             //下降沿读取参考trace
begin
    if(|debug_wb_rf_wen && debug_wb_rf_wnum!=5'd0 && !debug_end && `CONFREG_OPEN_TRACE)
    //读取trace时机与采样时机相同
    begin
        $fscanf(trace_ref, "%h %h %h %h" , trace_cmp_flag ,
                ref_wb_pc, ref_wb_rf_wnum, ref_wb_rf_wdata); //读取参考trace信号
    end
end

//compare result in rsing edge
always @(posedge soc_clk)                //上升沿将debug信号与trace信号对比
begin
    if(!resetn)
    begin
        debug_wb_err <= 1'b0;
    end
```

```
    else if(|debug_wb_rf_wen && debug_wb_rf_wnum!=5'd0 && !debug_end && `CONFREG_OPEN_TRACE
    //对比时机与采样时机相同
    begin
        if (  (debug_wb_pc!==ref_wb_pc) || (debug_wb_rf_wnum!==ref_wb_rf_wnum)
            ||(debug_wb_rf_wdata_v!==ref_wb_rf_wdata_v) )    //对比时机与采样时机相同
        begin
            $display("--------------------------------------------------------------");
            $display("[%t] Error!!!",$time);
            $display("    reference: PC = 0x%8h, wb_rf_wnum = 0x%2h, wb_rf_wdata = 0x%8h",
                     ref_wb_pc, ref_wb_rf_wnum, ref_wb_rf_wdata_v);
            $display("    mycpu    : PC = 0x%8h, wb_rf_wnum = 0x%2h, wb_rf_wdata = 0x%8h",
                     debug_wb_pc, debug_wb_rf_wnum, debug_wb_rf_wdata_v);
            $display("--------------------------------------------------------------");
            debug_wb_err <= 1'b1;         //标记出错
            #40;
            $finish;              //对比出错，则结束仿真
        end
    end
end
......
//monitor test
initial
begin
    $timeformat(-9,0," ns",10);
    while(!resetn) #5;
  $display("==============================================================");
    $display("Test begin!");
    while(`CONFREG_NUM_MONITOR)
    begin
        #10000;        //每隔10000ns，打印一次写回级PC，帮助判断CPU是否死机或死循环
        $display ("        [%t] Test is running, debug_wb_pc = 0x%8h", debug_wb_pc);
    end
end

//test end
wire global_err = debug_wb_err || (err_count!=8'd0);
always @(posedge soc_clk)
begin
```

```
    if (!resetn)
    begin
        debug_end <= 1'b0;
    end
    else if(debug_wb_pc==`END_PC && !debug_end)
    begin
        debug_end <= 1'b1;
      $display("================================================================");
      $display("Test end!");
      $fclose(trace_ref);
      #40;
        if (global_err)
        begin
            $display("Fail!!!Total %d errors!",err_count);    //全局出错，打印Fail
        end
        else
        begin
            $display("----PASS!!!");                  //全局无错，打印PASS.
        end
      $finish;
  end
end
......
```

0.2.3.3 func 程序说明

func 程序分为 func/start.S 和 func/inst/*.S，都是 LoongArch32 汇编程序:

1. func/start.S ： 主函数，执行必要的启动初始化后调用 func/inst/下的各汇编程序。
2. func/inst/*.S：针对每条指令或功能点有一个汇编测试程序。

主函数 func/start.S 中主体部分代码如下，分为三大部分，具体查看注释。

```
......
#以下是设置程序开始的LED灯和数码管显示，单色LED全灭，双色LED灯一红一绿。
    LI (a0, LED_RG1_ADDR)
    LI (a1, LED_RG0_ADDR)
    LI (a2, LED_ADDR)
    LI (s1, NUM_ADDR)
```

```
    LI (t1, 0x0002)
    LI (t2, 0x0001)
    LI (t3, 0x0000ffff)
    lu12i.w s3, 0
    NOP4

    st.w t1, a0, 0
    st.w t2, a1, 0
    st.w t3, a2, 0
    st.w s3, s1, 0
#以下是运行各功能点测试，每个测试完执行idle_1s等待一段时间，且数码管显示加1。
inst_test:
    bl n1_lu12i_w_test    #lu12i.w
    bl idle_1s

    bl n2_add_w_test   #add.w
    bl idle_1s
......
#以下是显示测试结果，PASS则双色LED灯亮两个绿色，单色LED不亮；
#Fail则双色LED灯亮两个红色，单色LED灯全亮。
test_end:
    LI  (s0, TEST_NUM)
    NOP4
    beq s0, s3, 1f

    LI (a0, LED_ADDR)
    LI (a1, LED_RG1_ADDR)
    LI (a2, LED_RG0_ADDR)

    LI (t1, 0x0002)
    NOP4

    st.w zero, a0, 0
    st.w t1, a1, 0
    st.w t1, a2, 0
......
```

每个功能点的测试代码程序名为 n#_*_test.S，其中“#”为编号，如有 15 个功能点测试，则从 n1 编号到 n15。每个功能点的测试，其测试代码大致如下。其中红色部分标出了关键的 3 处

代码。

```
.....
LEAF(n1_lu12i_w_test)
    addi.w   s0, s0 ,1          #加载功能点编号s0++
    addi.w   s2, zero, 0x0
    lu12i.w  t2, 0x1
    ###test inst
    addi.w   t1, zero, 0x0
    TEST_LU12I_W(0x00000, 0x00000)
    ......                      #测试程序，省略
    TEST_LU12I_W(0xff0af, 0xff0a0)
    ###detect exception
    bne      s2, zero, inst_error
    ###score ++                 #s3存放功能测试计分，每通过一个功能点测试，则+1
    addi.w   s3, s3, 1
    ###output (s0<<24)|s3
inst_error:
    slli.w   t1, s0, 24
    NOP4
    or       t0, t1, s3         #s0高8位为功能点编号，s3低8位为通过功能点数，
                                #相或结果显示到数码管上。
    NOP4
    st.w     t0, s1, 0          #s1存放数码管地址
    jirl     zero, ra, 0
END(n1_lu12i_w_test)
```

从以上可以看到，测试程序的行为是：当通过第一个功能测试后，数码管会显示 0x0100_0001，随后执行 idle_1s；执行第二个功能点测试，再次通过数码管会显示 0x0200_0002，执行 idle_1s……依次类推。显示，每个功能点测试通过，应当数码管高 8 为和低 8 位永远一样。如果中途数码管显示从 0x0500_0005 变成了 0x0600_0005，则说明运行第六个功能点测试出错。

最后，再来看下 idle_1s 函数的代码，其实使用一个循环来暂停测试程序执行的，如下：

```
idle_1s:
    LI(t0,SW_INTER_ADDR)
    LI(t1, 0xaaaa)
    #initial t3           //读取confreg模块里的switch_interleave的值
    ld.w    t2, t0, 0     #switch_interleave: {switch[7],1'b0, switch[6],1'b0...switch[0],1'b0}
    NOP4
    xor     t2, t2, t1    //拨码开关拨上为0，故要xor来取反
```

```
    NOP4
    slli.w  t3, t2, 9     #t3 = switch interleave << 9
    NOP4

sub1:
    addi.w  t3, t3, -1    //t3累减1

    #select min{t3, switch_interleave} //获取t3和当前switch_interleave的最小值
    ld.w    t2, t0, 0     #switch_interleave:{switch[7],1'b0,switch[6],1'b0...switch[0],1'b
    NOP4
    xor     t2, t2, t1
    NOP4
    slli.w  t2, t2, 9     #switch interleave << 9
    NOP4                  //以上ld.w-xor-slli.w三条指令再次获取switch_interleave
    sltu    t4, t3, t2    //无符号比大小，如果t3比switch_interleave 小则置t4=1
    NOP4
    bne     t4, zero, 1f  //t4!=0,意味着t3比switch_interleave大，则跳1f
    nop
    addi.w  t3, t2, 0     //否则，将t3赋值为更小的switch_interleave
    NOP4
1:
    bne     t3,zero, sub1 //如果t3没有减到0，则返回循环开头
    jirl    zero, ra, 0   //结束idle_1s
```

从以上代码可以看到，idle_1s 会依据拨码开关的状态设定循环次数。在仿真环境下，我们会模拟拨码开关为全拨下的状态，以使 idle_1s 循环次数最小。之所以这样设置，是因为 FPGA 运行远远快于仿真的速度，假设 CPU 运行一个程序需要 10^6 个 CPU 周期，再假设 CPU 在 FPGA 上运行频率为 10MHz，那其在 FPGA 上运行完一个程序只需要 0.1s；同样，我们仿真运行这个程序，假设我们仿真设置的 CPU 运行频率也是 10MHz，那我们仿真运行完这个程序也是只需要 0.1s 吗？显然这是不可能的，仿真是软件模拟 CPU 运行情况的，也就是它要模拟每个周期 CPU 内部的变化，运行完这一个程序，需要模拟 10^6 个 CPU 周期。我们在一台 2016 年产的主流 X86 台式机上进行实测发现，Vivado 自带的 Xsim 仿真器运行 SoC_lite 的仿真，每模拟一个周期大约需要 600us，这意味着 Xsim 上模拟 10^6 个周期所花费的实际时间约 10 分钟。

同一程序，运行仿真测试大约需要 10 分钟，而在 FPGA 上运行只需要 0.1 秒（甚至更短，比如 CPU 运行在 50MHz 主频则运行完程序只需要 0.02s）。所以我们如果不控制好仿真运行时的 idle_1s 函数，则我们可能会陷入到 idle_1s 长时间等待中；类似的，如果我们上板时设定 idle_1s 函数很短（比如拨码开关全拨下），则 idle_1s 时间太短导致我们无法看到数码管累加的效果

如果大家在自实现 CPU 上板运行过程中，发现数码管累加跳动太慢，请调小拨码开关代表的数值；如果发现数码管累加跳动太快，请调大拨码开关代表的数值。

0.2.3.4 编译脚本说明

功能仿真验证测试程序的编译脚本为验证平台目录下的 func/Makefile，对 Makefile 了解的可以去看下该脚本。该脚本支持以下命令：

- make help ： 查看帮助信息。
- make ： 编译得到仿真下使用的结果。
- make clean ： 删除 *.o，*.a 和./obj/目录。

0.2.3.5 编译结果说明

func 编译结果位于 func/obj/下，共有 3 个文件，各文件具体解释见表2。

表 2: 编译生成文件

文件名	解释
inst_ram.coe	重新定制 inst ram 所需的 coe 文件
inst_ram.mif	仿真时 inst ram 读取的 mif 文件
test.s	对 main.elf 反汇编得到的文件

0.2.3.6 测试程序的装载

我们开发的测试程序用 GCC 工具编译之后形成一个 ELF 格式的可执行文件——main.elf。那么我们是要在自己开发的 CPU 上直接运行这个 ELF 格式的可执行文件吗？显然不是。我们测试的环境俗称“裸机”，它是一台没有运行任何操作系统或者监控环境的单纯的硬件系统，所以文件系统、可执行程序的加载器等软件统统都没有。

我们其实真正需要的其实是 main.elf 的代码和初始数据[5]。我们只要能把这些代码和初始数据提取出来，将代码放到指令 RAM 中，将初始数据放到数据 RAM，那么就可以把 CPU 跑起来了。所以我们用工具链中的 objcopy 工具，将 main.elf 文件中的.text 段提取出来生成二进制格式的纯数据文件 main.bin，将 main.elf 文件中的.data 段提取出来生成二进制格式的纯数据文件 main.data。

接下来就是把这些信息怎么“装”到 RAM 中去了。我们利用的是 Xilinx FPGA 中 block RAM IP 的初始内容加载功能。该功能需要将加载的内容按照规定的格式生成文本文件。于是我们进一步将前面得到的 main.bin 和 main.data 转换为所需的文本文件。每个二进制纯数据文件都生成一个.coe 后缀的文件和一个.mif 后缀的文件。这两个文件的数据内容其实完全相同，只是

[5]后期的某些实验中，测试程序需要一定量的输入数据，如果采用立即数加载的方式产生将耗费较多的 CPU 执行时间，届时我们会将这些输入数据以赋了初值的全局静态变量的形式传給测试程序。这些全局变量的初值将记录在 ELF 文件的只读数据段中。

文档的其它格式信息存在差异。coe 文件是用于生成上板配置文件的，而 mif 文件是用于功能仿真的。我们建议读者在调试过程中不要通过直接修改 coe 文件或是 mif 文件的方式来调整测试激励，除非你真的搞清楚了你的修改会影响哪个环节的验证结果。

0.2.3.7 仿真验证结果判断

仿真结果正确判断有两种方法。

第一种方法，也是最简单的，就是看 Vivado 控制台打印 Error 还是 PASS。正确的控制台打印信息如图3。

图 3: 仿真验证通过的控制台打印信息

第二种方法，是通过波形窗口观察程序执行结果 func 正确的执行行为，抓取 confreg 模块的信号 led_data、led_rg0_data、led_rg1_data、num_data：1. 开始，单色 LED 写全 1 表示全灭，双色 LED 写 0x1 和 0x2 表示一红一绿，数码写全 0；2. 执行过程中，单色 LED 全灭，双色 LED 灯一红一绿，数码管高 8 位和低 8 位同步累加；3. 结束时，单色 LED 写全 1 表示全灭，双色 LED 均写 0x1 表示亮两绿，数码管高 8 位和低 8 位数值（十六进制）相同，对应测试功能点数目。

0.2.3.8 FPGA 上板验证结果判断

在 FPGA 上板验证时其结果正确与否的判断也只有一种方法，func 正确的执行行为是：

1. 开始，单色 LED 全灭，双色 LED 灯一红一绿，数码管显示全 0；
2. 执行过程中，单色 LED 全灭，双色 LED 灯一红一绿，数码管高 8 位和低 8 位同步累加；
3. 结束时，单色 LED 全灭，双色 LED 灯亮两绿，数码管高 8 位和低 8 位数值相同，对应测试功能点数目。

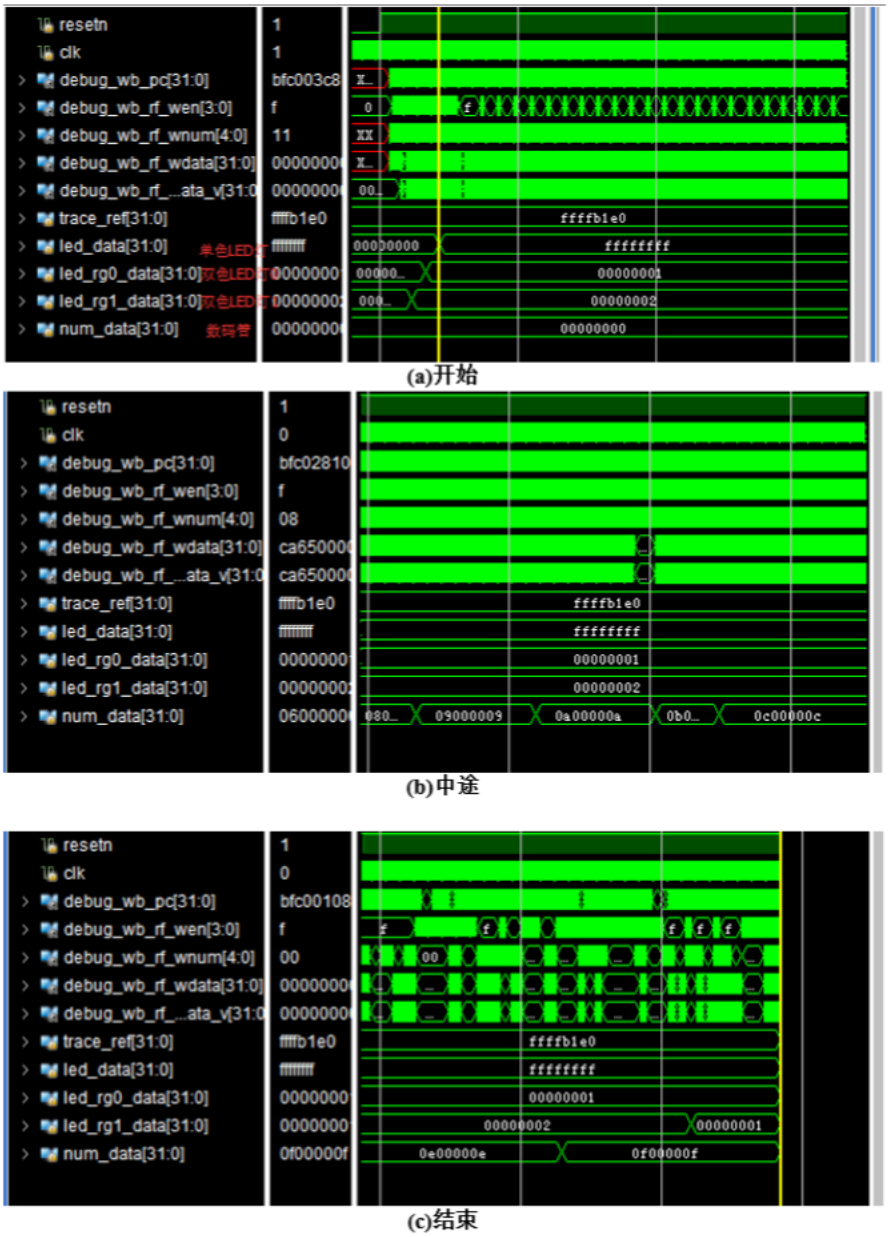
resetn
clk
debug_wb_pc[31:0]
debug_wb_rf_wen[3:0]
debug_wb_rf_wnum[4:0]
debug_wb_rf_wdata[31:0]
trace_ref[31:0]
led_data[31:0]
led_rg0_data[31:0]
led_rg1_data[31:0]
num_data[31:0]
ffffb1e0
00000000
ffffffff
00000001
00000002
(a)开始
bfc02810
ca650000
09000009
0a00000a
0c00000c
(b)中途
bfc00108
0e00000e
0f00000f
(c)结束

图 4: 正确的仿真波形图

如果 func 执行过程中出错了，则数码管高 8 位和低 8 位第一次不同处即为测试出错的功能点编号，且最后的结果是单色 LED 全亮，双色 LED 灯亮两红，数码管高 8 位和低 8 位数值不同。

最后 FPGA 验证通过的效果，类似图5，数码管高 8 位和低 8 位显示为运行的功能点数目。

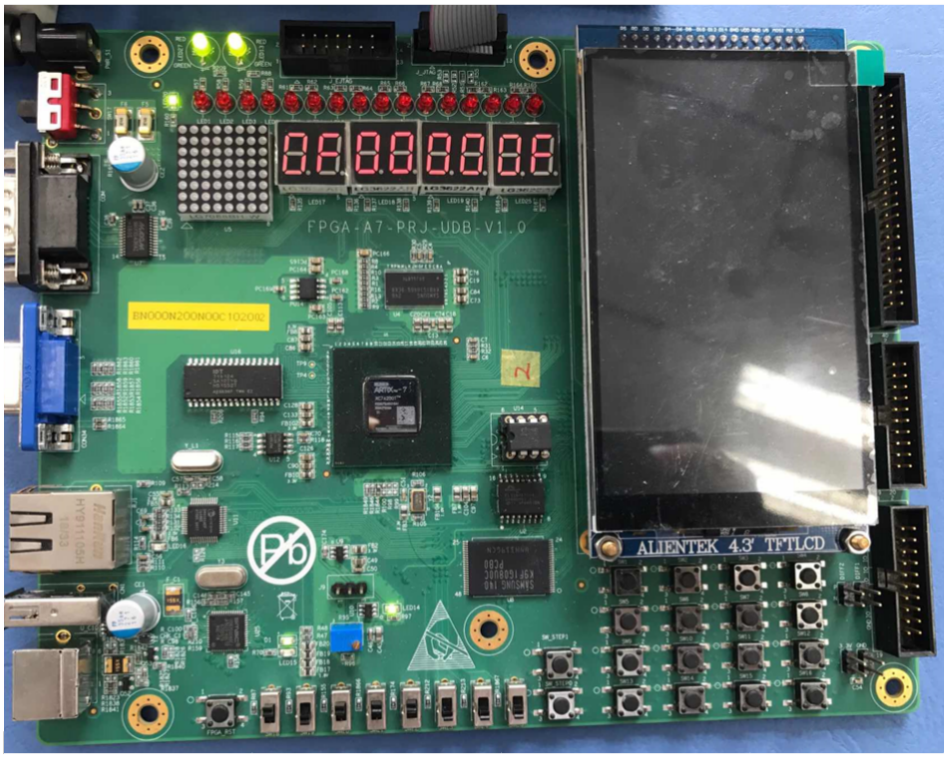

图 5: 上板验证正确效果图

0.2.4 CPU 设计开发环境使用进阶

0.2.4.1 升级工程和 IP 核

本 CPU 设计实验开发环境（CPU_CDE）是在 Vivado2019.2 中创建的，如果使用更高版本的 Vivado 打开，需要对工程和 IP 核进行升级。注意：Vivado 不支持向前兼容，也就是低版本 Vivado 无法修改高版本 Vivado 创建的工程。

高版本 Vivado 打开低版本的工程，升级工程和 IP 核的方法如下：

（1）高版本 Vivado 打开低版本的工程，会弹出如图6界面，选择第一个选项 “Automatically Upgrade...”，点击 “OK” 进行升级。

（2）如果你的工程里包含 Vivado 定制的 IP 核，则会提示图7左侧的提醒，选择 “Report IP Status”，则会显示 IP 核状态，如图7右侧：三个 IP 和显示红色的锁标记，说明该 IP 核目前被锁住，无法修改。

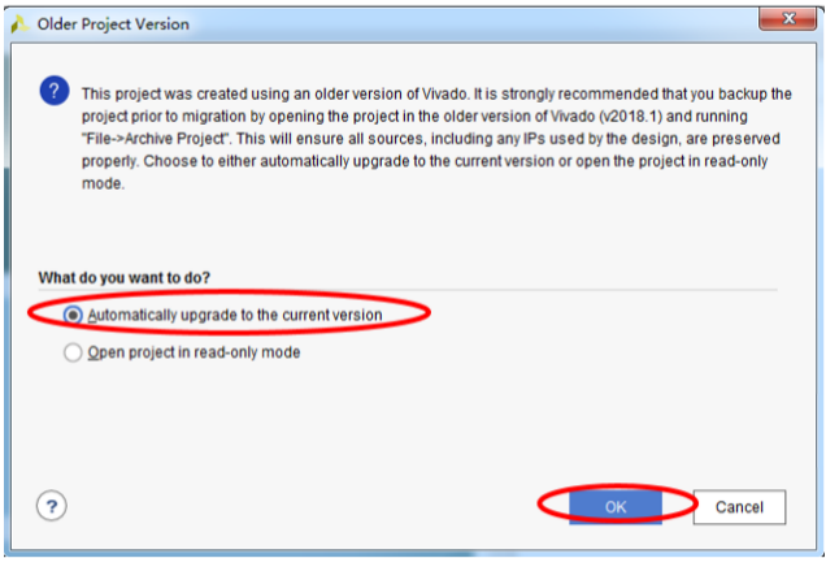

图 6: 低版本工程升级

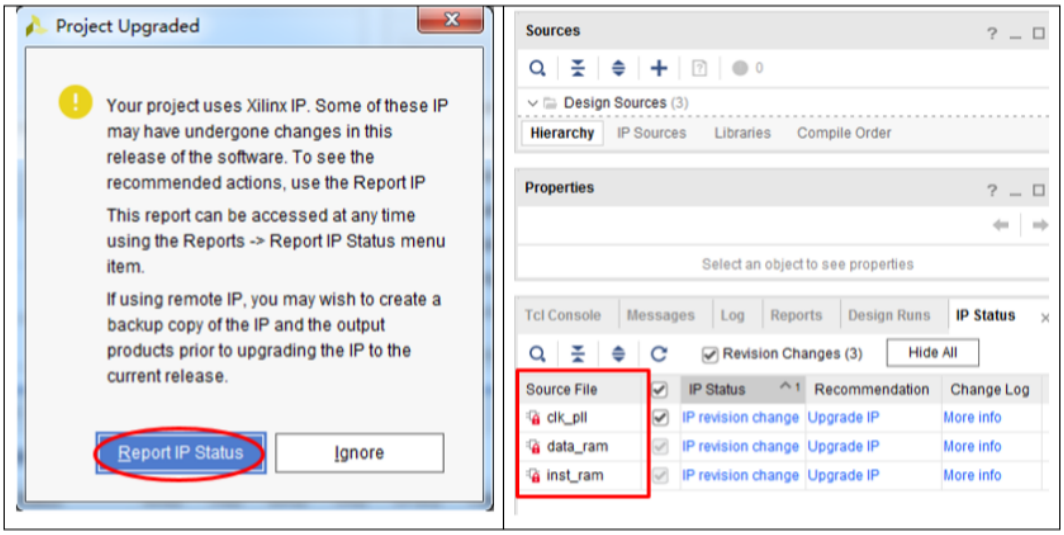

图 7: 显示 IP 核被锁住

(3) 需要对锁住的 IP 核依次进行升级，才能在此版本 Vivado 里修改这些 IP 核。在 Sources 窗口中找到要升级的 IP，右键后点击 “Upgrade IP…”，如图8。

(4) 之后，会弹出如图9所示界面，选择第二个选项 “Continue with Core…”，点击 “OK”。

(5) 在弹出的 Generate Output Product 窗口中（如图10所示）点击 Generate（根据需要选择 global 或 ooc 模式），完成升级。

0.2.4.2 重新生成 golden_trace.txt

当更新 func 后，记得在 gettrace 里重新生成 golden_trace.txt，其的步骤如下：

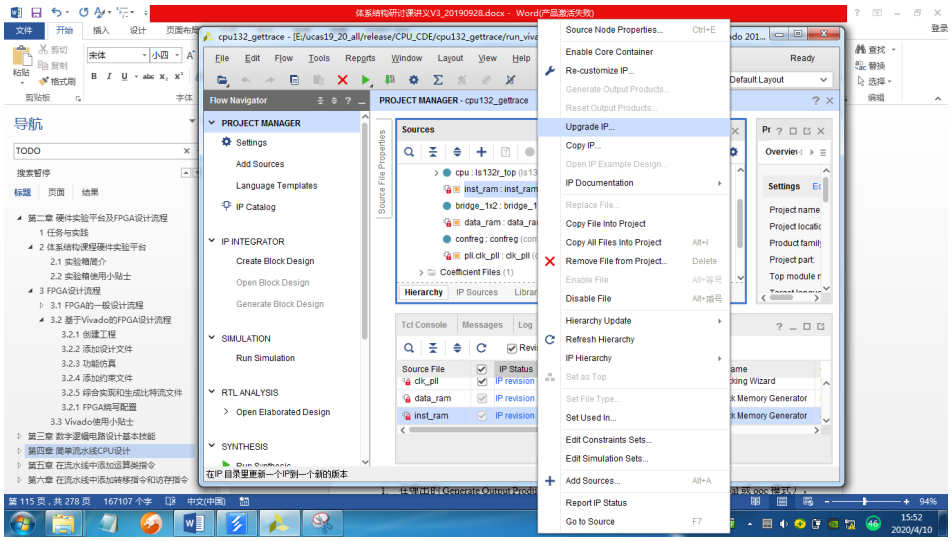

图 8: 右键 IP 核选择 “Upgrade IP...”

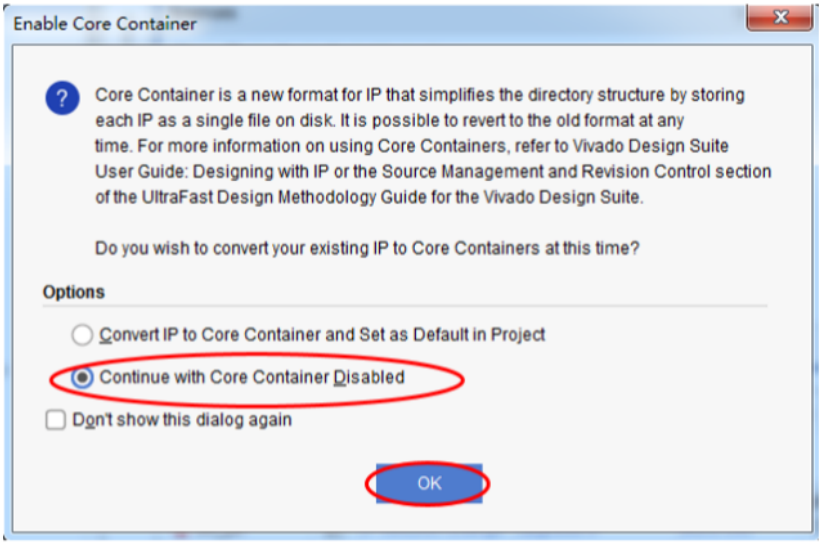

图 9: 将 IP 核升级

1. 打开 gettrace 工程，确保 soc_lite_top.v 中 INST_COE 宏定义指向更新后 func 生成的 mif 文件。
2. 运行仿真，仿真结束后 gettrace 目录下的 golden_trace.txt 会被更新。

0.3 CPU 设计实验功能仿真调试技术

上一章我们已经介绍过数据逻辑电路设计在功能仿真过程中常用的一些调试方法和技术。在这一节中，我们将结合 CPU 这种具体类型的数字逻辑电路以及我们所提供的实验开发环境，再来介绍一些关于这方面的功能仿真调试技术。这里我们仍然以观察仿真波形作为 CPU 功能仿真

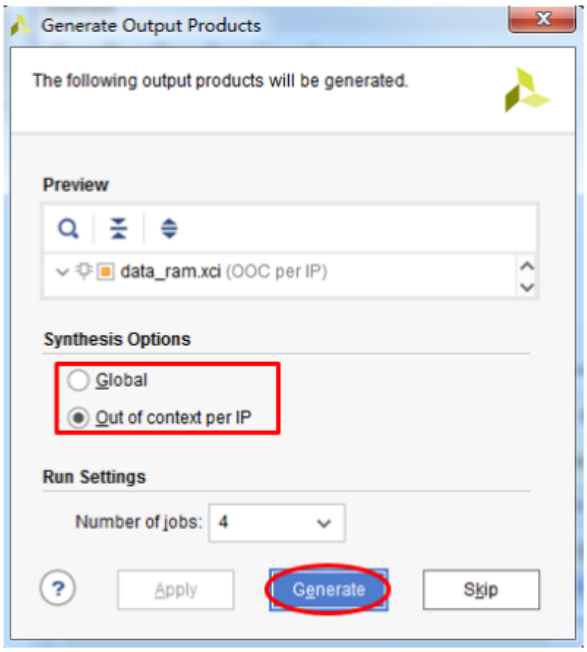

图 10: 完成 IP 核升级

验证调试的主要手段。在上一章中我们给出过观察数字逻辑电路功能仿真波形的基本思路，其核心就是“找到一个你能明确的错误点，然后从沿着设计的逻辑链条逆向逐级查看信号，直至找到源头。”CPU 也是数字逻辑电路，所以它的功能仿真验证调试同样遵循这个思路。因此我们主要结合 CPU 功能仿真验证的特殊性，总结出具体的调试方法。

0.3.1 为什么要用基于 trace 比对的调试辅助手段

过前面 CPU 设计实验环境的介绍，读者已经知道在对 CPU 进行功能验证时，是采用一段测试指令序列（或称为测试程序）作为 CPU 功能验证的激励的；验证通过与否，是通过观察测试程序的执行结果是否符合预期的。这种验证方式的优点是它既可以用于仿真验证也可以用于实际机器的测试，缺点是验证者看到出错现象时往往离出错源头相隔很远了。对于这里所说的缺点，大家可能没有直观的概念。我们接下来举例子说明一下。

譬如说，我们想通过测试程序验证 CPU 有没有把 add.w 指令实现对，那么很自然的会用这样的流程开发测试程序：首先，分别向两个寄存器存入操作数；然后执行一条 add.w 指令，将这两个寄存器作为源寄存器，同时向不同于 add.w 指令目的寄存器的另一个寄存器存入正确的结果；最后，将存有期望结果的寄存器和 add.w 指令的目的寄存器进行比较，如果不相等则调用打印函数向终端输出一条出错信息，否则继续进行后续其它测试。按照这个流程，假设 CPU 设计中出现了错误，导致 add.w 指令出错，那么等到验证人员从终端中看到出错信息，其实这中间 CPU 已经执行了比较指令和打印出错信息的函数的所有指令。如果整个系统的输出函数还有自己内部的软件缓存，那么出错信息真正写入到输出设备中的时间还要更靠后。也就是说，从 add.w 指令出错，到你看到提示出错，CPU 可能又执行了成百上千条指令。这会造成什么问题呢？就是你打

开的仿真波形后，从最后时刻开始向前的数百乃至数千个周期中 CPU 都是行为的正常，也就是说你查看这数百乃至数千个周期中的任何信号都是徒劳的。如果没有一种方法让你快速定位到出错的 add.w 指令的执行时刻，那么就会花费相当多无用的调试时间。

有读者会说，这个好办，我们在错误信息中把这是第几个测试程序的第几个测试用例打印出来，然后按照这个信息定位被测试的指令就可以了。这个方法不错，但是不能解决所有问题。因为你要知道现在出错不是软件写错了，而是 CPU 设计出错了，那么出错的原因就可能五花八门。还是刚才的例子，检验 add.w 指令执行正确的测试中可不是只有 add.w 指令，还有其它指令。一旦这些指令执行出错了，你还是看到提示说 add.w 指令测试出错了，但事实上并非如此。有的读者可能又会想，如果我们在测试里面用的其它指令都是前面的测试用例已经测试过的，那么就不应该会错在其它指令上了。我们只能说这是一个美好的愿望。我们暂且不论第一条指令的测试该如何写（这意味着只用一条指令写一段测试程序），就说已经测试过的指令后面再使用就不会出错这个假设也是不成立的。CPU 设计的调试有时候之所以困难就在于，同样一条指令，它在这种执行序列下可能会做对但是换一种执行序列就有可能会做错。因此一条指令在前面测试通过了未必在后面的的测试序列中就不出错。

上面说道了半天，其实就是告诉各位：**CPU 执行出错的错误源头，可以通过测试程序的逻辑路径传递很远之后才能被验证者发现**。采用测试程序测试 CPU 设计的验证方法，其调试过程中最大的工作量就是找到错误源头对应的那条指令。

解释到这里，也许读者就能真正理解，为什么我们要在实验开发环境中提供一套基于 trace 比对的调试辅助手段。因为通过这个手段，**在大多数情况下**，仿真验证平台都能在错误源头的那条指令刚刚执行完的时候立刻报错。具体到波形中，你们可以将 mycpu_tb 这个模块中 debug_wb_err 信号抓到波形中，它从 0 变为 1 的时刻，就是 trace 比对不通过的那条指令在写回级写寄存器的时刻。是不是特别简单？瞬间就定位到出错的指令了。

0.3.2 基于 trace 比对调试手段的“盲区”及其对策

不过 trace 比对机制也不是万能的，就是说还是存在 trace 比对无法及时发现的错误。我们总结了一下，主要有下面三种情况：

1. 被测 CPU 死机了。波形上体现为时钟信号还在随着时间增加而增长，但是 CPU 一直没有指令写回。
2. 被测 CPU 执行一段死循环，且这段死循环中没有写寄存器的指令。譬如像 “1: b 1b; nop” 这样的程序片段。
3. 被测 CPU 执行 store 指令出错。

针对前两种情况，我们在实验开发环境中引入了另一套监控机制：通过在验证平台 mycpu_tb 中每隔 10000ns 利用 verilog 的 $display 系统调用直接向终端输出 CPU 写回级的 PC。如果发现终端不再输出写回级 PC 或者输出的都是同一 PC，则说明 CPU 死机了；如果终端输出的 PC 具有很明显的规律，是在一个很小的范围内不停地重复，则说明陷入了死循环。这时候就需要你找到这串提示异常的输出的第一条，找到该条提示开头的仿真时间。你的波形是要从这个仿真时

间开始往前找（后面的就不用看了），找到第一个写回级有写寄存器信号的时刻。那么错误肯定是在这个时刻开始之后的某个时刻出现的。

对于第三种情况，执行出错的 store 指令会被后面访问相关地址[6]的 load 指令报出来，所以你会看到 trace 比对机制报错并停在一条 load 指令上。当你检查完这条 load 指令的执行过程，发现 load 指令自身的执行都没有问题，那么问题十有八九就是前面访问相关地址 store 指令执行出问题了。要么是该写这个地址的 store 指令把写入的数值搞错了，要么就是该向这个地址写入数值的 store 指令没有写进去，要么就是不该访问这个地址的 store 指令因为把地址搞错了错误地更改了这个位置的数值。具体是哪种原因，就需要结合程序的行为和波形的反馈来调试。应该说，这种 store 指令执行错的情况是最难调的。为了调好这类错误，就必须掌握接下来介绍的看反汇编的技能。

0.3.3 学会阅读汇编程序和反汇编代码

在进行 CPU 设计实验时，为了验证自己写的 CPU 的功能，我们需要让 CPU 运行一段验证程序。这种验证程序通常是由 C 语言或汇编语言编写的。C 语言编写比较方便、快捷，适合编写复杂的程序；汇编语言编写虽然稍微繁琐点，却可以明确控制指令顺序、编译结果。比如，有时候我们需要验证 CPU 对特点指令序列流的处理是否正确，此时就适合使用汇编语言编写。另外，C 语言中也可以内嵌汇编进行编程，这种汇编与 C 语言混合编程的写法适合提升程序的性能或实现特定的功能。

以下，我们将简单介绍 LoongArch32 汇编程序和反汇编代码的阅读，帮助读者初步阅读 LoongArch32 汇编程序和反汇编代码。想要了解具体的工具链编译流程，以及围绕 elf 文件的底层原理，可以参考《程序员的自我修养》一书。该书是以 x86 指令集作为介绍的，所以在具体的例子上会和 LoongArch32 有所差异，但是所阐述原理是一致的，只需要稍加变通完全可以应用到 Loongarch32 指令集上。同样的，《See MIPS Run》一书尽管针对的是 MIPS 指令集，但它仍然是介绍指令集相关底层编程模式和其上软件栈如何运行的经典入门著作，值得一读。

一般地，程序员编写的汇编程序，会将其后缀名记为 “.S”，比如我们经常在一个编译目录里看到了文件名 “start.S” 就是一个人为编写的汇编程序；对编译工具链自动生成的汇编/反汇编文件，将其后缀名记为 “.s”。

0.3.3.1 常见的汇编代码

常见的汇编代码格式如下：

```
##s4, exception pc
    .globl  _start
    .globl  start
    .globl  __main
```

[6]之所以用 “相关地址” 而不是 “相同地址”，是因为只要 store 写的地址区域和 load 读的地址区域有重叠就会有体现，而此时两条地址相关的 load、store 指令的地址未必严格相同。

```
_start:
start:
    li       $t0, 0xffffffff
    addi.w   $t0, $zero, -1
    b        locate

##avoid "j locate" not taken
    lu12i.w  $t0, -0x80000
    addi.w   $t1, $t1, 1
    or       $t2, $t0, $zero
    add.w    $t3, $t5, $t6
    ld.w     $t4, $t0, 0

##avoid cpu run error
    .org   0x0ec
    lu12i.w  $t0, -0x80000
```

这段代码中有 4 类功能语句：

1. 注释代码：比如 “##…”，参见第0.3.3.2小节的介绍。
2. 标号（label）：比如 “_start:” 和 “start:”，参见第0.3.3.3小节的介绍。
3. 伪指令：比如 “.globl” 和 “.org”，参见第0.3.3.4小节的介绍。
4. 汇编指令：比如 “b locate”（locate 是一处标号）、“li”、“addi.w” 等等，参见第0.3.3.5小节的介绍。

0.3.3.2 注释代码

在 LoongArch32 汇编中，通常汇编器器支持两类注释代码：

1. #：表示从当前字符到行尾为注释；
2. /*…*/：同 C 语言语法，之间的内容为注释。

值得说明的是，“#” 还有其他用途，用于预处理的引导字符。比如 “#define”、“#if” 和 “#include”，它们都不是注释，而是预处理的指示语句，和 C 语言类似。

“#if” 配合 “#elif”、“#else” 和 “#endif” 可以实现条件编译的功能，如下：

```
run_test:
#if CMP_FUNC==1
    bl  shell1
#elif CMP_FUNC==2
```

```
    bl  shell2
#elif CMP_FUNC==3
    bl  shell3
#elif CMP_FUNC==4
    bl  shell4
#else
    b   go_finish
#endif
go_finish:
```

0.3.3.3 标号

标号用于代替该条汇编指令的起始地址，其实相当于定义了一个变量，该变量指向该汇编指令的起始地址。其定义格式是：“标号”+“:”。

标号的命名遵循汇编中的变量命名的格式，可以是 C 语言中任意合法的标识符（大小写字符、数字和下划线），同时还可以包含字符 “.”，比如 “.t0” 可以作为一个合法的标号。

有一类特殊的标号是数字标号，如下面示例中的 “1:” 和 “2:” 是数字标号是定义，“2b” 是对数字标号 “2” 的引用。

```
    slli.w   $t0, $t0, 16
1:
    addi.w   $t0, $t0, 1
2:
    addi.w   $t0, $t0, -1
    bne      $t0, $zero, 2b
    jr       $ra
```

数字标号具有局部标号的作用：引用的格式为 “数字标号 +‘f’ ” 表示在将来的程序（forward）中找寻最近的该数字标号；引用的格式为 “数字标号 +‘b’ ” 表示在过去的程序（backward）中找寻最近的该数字标号。比如：“1f” 表示在将来的程序中找寻最近的数字标号 “1”，也就是代码界面里向下找最近的 “1”；“2b” 表示在过去的程序中找寻最近的数字标号 “1”，也就是代码界面里向上找最近的 “2”，在上图中 “2b” 就是引用 “addi.w t0, -1” 这条指令处定义的标号 “2”。由于数字标号的引用遵循就近原则，所以数字标号可以重复定义、引用，而不会导致混乱，极大的方便了汇编程序的编写（程序员不需要为一些临时或简单的跳转标号而绞尽脑汁命名）。

0.3.3.4 伪指令

在 LoongArch32 汇编中还经常看到伪指令，比如 “.globl”、“.text”、“.org” 和 “.align” 等等。伪指令的功能是指导汇编器的工作。

- “.globl” 声明了该变量是全局变量，可以供其他文件引用。
- “.text” 告诉汇编器后续代码放在 “.text” 段（代码段），除非遇到其他说明。与 “.text” 类似的功能还是 “.section .rodata”（只读数据段）、“.data”（数据段）、“.section .sdata, “aw”” （小数据段，用于优化生成代码，s 前缀是 small 的意思）等等。
- “.org” 和 “.align” 指示了对齐格式：“.org n” 是表示从起始地址偏移 n，“.align n” 表示以 2n 字节对齐。比如：“.org 0x100” 表示要求地址低位为 0x100（也就是起始地址偏移 n）；“.align 2” 表示以 2^2 字节对齐，也就是要求地址末两位为 0。

0.3.3.5 汇编指令与机器指令

汇编指令是机器指令的超集，也就是一条机器指令一定也属于汇编指令。汇编指令是我们可以直接在汇编文件中编写的指令，作为汇编器的输入；机器指令则是作为汇编器的输出，每条机器指令对应一个独一的指令编码，供机器识别并执行。不同的汇编器支持的汇编指令可能有所不同，但是它们支持的同一架构版本的机器指令一定是相同的。

机器指令也是我们实现的 CPU 中直接支持的指令，我们在指令手册的指令列表中看到的指令都是机器指令。比如，“add.w” 指令就直接是一条机器指令，通常一条汇编指令对应一条机器指令。

但有些汇编指令是一条对应多条机器指令。为什么要有这类汇编指令？答案是为了汇编代码更好的可读性与易编写性。比如非常常用的加载立即数或者加载变量的地址的编程需求。如果立即数或者变量地址是一个 “不规则” 的 32 位数，一条机器指令显然就没办法将其编码进去，例如：汇编语句 “li $t0, 0x12345678$t0 寄存器加载入 0x12345678 这个 32 位二进制数，就对应于两条机器指令，见表3的第三种情况。而即便是有些情况下（如表3的前两种情况）能对应于一条指令，我们仍然倾向于用像 “li” 这样非机器指令的汇编指令，因为这样程序的行为更加一目了然。

表 3: 一条 “li” 汇编指令对应一或多条机器指令

汇编指令	**对应的机器指令**
li t0, 0x12340000	lu12i.w $t0, 0x12340
li t0, 0xfffff865	addi.w $t0, $zero, 0x865-0x1000
li t0, 0x12348765	lu12i.w $t0, 0x12348; ori $t0, $t0, 0x765

表3中有一点需要注意的是 addi.w 指令这里不能写作 “addi.w $t0, $zero, 0x865”。这是因为 addi,w 指令会将 0x865 当作有符号数，但这已经超出了 addi.w 立即数域的表示范围，想要表示 0x865 这个二进制数，就要先将其减去 0x1000 转换为有符号下的表示。通常我们把像 “li” 这样的在指令手册中并不存在，但汇编器可以识别出来并转换成相应具体机器指令序列（一条或多条）的汇编指令成为宏指令。而一旦遇到新的宏指令，要善于积累总结，不要因为指令手册上没有而

感到困惑。(思考：前面的示例中出现过一条指令 “jr $ra”，它是不是宏指令？如果是，又是对应哪条具体情况下的机器指令。)

0.3.3.6 通用寄存器的习惯命名和用法

我们实验中使用的 ABI 为 ilp32，其对通用寄存器的命名和使用约定如表4所示：

表 4: ilp32 ABI 通用寄存器命名约定及用法

寄存器编号	助记符	用法	函数调用过程中的保存情况
$r0	$zero	Constant zero	Unused
$r1	$ra	Return address	No
$r2	$tp	TLS	Unused
$r3	$sp	Stack pointer	Yes
$r4-$r11	$a0-a7	Argument register	No
$r12-$r20	$t0-$t8	Temporary register	No
$r21	$x	Reserved	Unused
$r22	$fp	Frame pointer	Yes
$r23-$r31	$s0-$s8	Subroutine register variables	Yes

0.3.3.7 反汇编命令和代码阅读

汇编或 C 语言编写程序，对其进行编译得到的可执行文件是 elf 格式的文件，对 elf 文件使用 objdump 命令可以进行反汇编，通常约定反汇编后的文件后缀为 “.s”。

比如，假设 elf 文件为 main.elf，对该文件进行反汇编可以在 linux 命令行执行以下命令：

```
[abc@www ~]$ loongarch32-unknown-elf-objdump -ald main.elf > test.s
```

或

```
[abc@www ~]$ loongarch32-unknown-elf-objdump -alD main.elf > test.s
```

上述命令中“loongarch32-unknown-elf-objdump”为交叉编译工具 objdump 的命令，“-ald”或“-alD”为命令参数，“main.elf”为 elf 文件的名字，“>”为重定向命令（表示将终端输出重定向到后续的文件 test.s 中），“test.s”为输出重定向后的文件名。

关于命令参数的具体含义如下：

1. -a：显示 elf 文件的文件头信息。
2. -l：显示源码中的行号，通常与-d 或-D 联合使用，并且需要在编译生成 elf 文件时使用编译参数-g（编译结果中保留调试信息）。
3. -d：只反汇编.text 段（代码段）的内容，比如数据段的内容就不会被反汇编。
4. -D：反汇编所有段的内容。

反汇编后的文件，打开的格式如图11。其中每条指令占据一行，主要包含指令地址、指令编码和指令汇编格式三部分信息。

```
Disassembly of section .text:
 指令地址                          指令汇编格式
00000000 <.L2^B1-0x8>:
   0: 0040c18c  slli.w  $r12,$r12,0x10
   4: 0280058c  addi.w  $r12,$r12,1(0x1)
      指令编码
00000008 <.L2^B1>:
.L2^B1():
   8: 1424690c  lu12i.w $r12,74568(0x12348)
   c: 039d958c  ori $r12,$r12,0x765
  10: 02a1958c  addi.w  $r12,$r12,-1947(0x865)
  14: 5ffff580  bne $r12,$r0,-12(0x3fff4) # 8 <.L2^B1>
  18: 4c000020  jirl  $r0,$r1,0
```

图 11: 用于验证 mycpu 的简单硬件系统

除了对 elf 文件反汇编，我们还可以使用 readelf（交叉编译工具则是 loongarch32-unknown-elf-readelf）命令对 elf 文件进行读取，显示 elf 文件的结构，比如.text 段（代码段）的起始地址和大小等等。