1 Trace 比对机制使用说明

通过这一讲，希望你们能够：

（1）基于所提供的 Trace 比对机制迅速开展 CPU 实验；

（2）了解 CPU 实验的一般性开发过程；

（3）了解我们所提供的 CPU 实验开发环境。

在编写本讲内容的时候，我们假定各位同学已经：

（1）基本会用 Verilog 写一个简单的数字电路设计；

（2）基本会用 Vivado 工具进行 FPGA 工程开发，知道“仿真-综合-实现-下载（上板）”这一过程；

（3）知道汇编语言是什么；

（4）知道 Linux 是什么，知道最常用的 Shell 命令，如查看文件、进入某个目录、执行程序之类的；

（5）知道基于 GCC 工具链编译的一般过程，即一个 C 或者汇编程序是如何一步步转化为一个可执行文件的。

上述内容如果你还不怎么了解，那么还是**强烈建议**你先去学习（复习）一下。

1.1 快速上手

(1) 【解压环境】将大赛发布包的功能测试目录 func_test/放到一个路径中没有中文字符的位置上，要确保你在这个位置下能运行 Vivado。进入 func_test/目录。

(2) 【设计 myCPU】用你习惯使用的文本编辑器（Vivado 中集成的文本编辑功能实在是不怎么样）将处理器核的 Verilog 代码编写好，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。

(3) 【myCPU 加入】如果 myCPU 为 SRAM 接口，则将写好的 CPU 代码拷贝到 soc_sram_func/rtl/myCPU/目录下；如果 myCPU 已封装为 AXI 接口，则将写好的 CPU 代码拷贝到 soc_axi_func/rtl/myCPU/目录下；

(4) 【编译 Func】如果你是在 Windows 下面运行 Vivado：将 soft/func 目录整体拷贝到一个已经安装了 MIPS-GCC 交叉编译工具的 Linux 环境中(或者将 soft/func 目录设置为虚拟机共享目录)，进入 soft/func 目录，先运行 make reset，再运行 make。将当前 soft/func/obj /整个目录的内容覆盖步骤（1）解压所在位置下的 soft/func/obj/目录。（有关 func 编译的内容详见本章 1.3.5 到 1.3.7 节）。

如果你是在 Linux 下运行 Vivado：先确保你的 Linux 系统已经安装了 MIPS-GCC 交叉编译工具。然后进入 func 目录先运行 make reset，再运行 make 就可以了。

(5) 【生成 Trace】进入 cpu132_gettrace/run_vivado/cpu132_gettrace/目录，打开 Vivado 工程 cpu132_gettrace，进行仿真，生成参考结果 golden_trace.txt。重点注意此时 inst_ram 加载的是第（4）步编译出的结果，即 soft/func/obj/目录下的 inst_ram.coe。（参考模型生成 Trace 参见本章 1.3.4 节）

(6) 【myCPU 仿真】进入 soc_sram_func/run_vivado/mycpu/或者 soc_axi_func/run_vivado/mycpu/目录，打开

Vivado 工程 mycpu，将你在第（2）步新加的文件添加到工程中，进行仿真。看仿真输出 log 是否与给出的正确 log 一致。如果结果异常，则进行调试直至通过。重点注意此时 inst_ram 加载的是第（4）步编译出的结果，即 soft/func/obj/目录下的 inst_ram.coe。

(7) 【myCPU 上板】回到第（6）步打开的 mycpu 这个工程中，进行综合实现，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。如果结果与要求的一致，则 myCPU 验证成功至此结束，恭喜你；否则转到第（8）步进行问题排查。

(8) 【反思】请按照下列步骤排查，重复第（7）、（8）步直至正确。

a) 复核生成、下载的 bit 文件是否正确。

i. 如果判断生成的 bit 文件不正确，则重新生成 bit 文件。

ii. 如果判断生成的 bit 文件正确，转 b）

b) 复核仿真结果是否正确。

i. 如果判断仿真结果不正确，则回到前面步骤（6）。

ii. 如果判断仿真结果正确，转 c）。

c) 检查实现时的时序报告（Vivado 界面左侧“IMPLEMENTATION”→”Open Implemented Design”→”Report Timing Summary”）。

i. 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，即降低 clk_pll 模块的输出端频率，做完这些改动后，回到前面步骤（7）。

ii. 如果实现时时序是满足的，转 d）。

d) 认真排查综合和实现时的 Warning，特别是 Critical Warning

i. 如果有尚未修正的 Warning，尽量修正它们了，然后回到前面步骤（7）。

ii. 如果没有可再修正的 Warning 了，转 e）。

e) 排查 RTL 代码规范，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。

f) 如果你会使用 Vivado 的逻辑分析仪进行板上在线调试，那么就去调试吧；如果调试了半天仍然无法解决问题，转 g）

g) 反思。真的，现在除了反思还能干什么？

1.2 CPU 实验的一般性开发过程

CPU 实验的一般性开发过程：

第 1 步，想清楚你的设计，用 Verilog 描述出来，把写好的 Verilog 代码拷贝到所提供实验环境的指定位置。

第 2 步，进行功能仿真，一遍一遍地修改你的设计，直到仿真通过。

第 3 步，进行综合实现，一遍一遍地修改你的设计和约束，直到所有的 Error、Critical Warning 和 Timing Violation 都不出现。

第 4 步，将生成的 bit 流文件下载到 FPGA 上进行实际演示操作，如果出现异常，首先确保你第 2 步和第 3 步真的做对了。

1.3 Trace 比对机制

1.3.1 目录组织与结构

目录为大赛发布包的 func_test/目录，整个目录结构及各主要部分的功能如下：红色部分为大家自实现的 CPU，黑色部分是我们已经搭建好的其余部分。

|-cpu132_gettrace/	目录，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，Vivado2018.1 创建的 Vivado 工程，名字就叫 cpu132_gettrace
| | | |--cpu132_gettrace.xpr	Vivado2018.1 创建的 Vivado 工程
| |--trace_ref.txt	该开发环境运行测试 func 生成的参考 trace。
|	
|-soft/	目录，功能测试 func。
| |--func/	目录，功能测试程序。
| | |--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。
| | | |--cpu_cde.h	使用到的宏定义，如数码管基址
| | |--inst/	目录，各功能测试点的验证汇编程序
| | | |--Makefile	子目录里的 Makefile，会被上一层次的 Makefile 调用。
| | | |--inst_test.h	各功能测试点的验证程序使用的宏定义头文件
| | | |--n*.S	各功能测试点的验证程序，汇编语言编写。
| | |--obj/	目录，func 编译结果
| | | |--*	参见 1.3.7 节。
| | |--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源码

| | |--bin.lds	bin.lds.S 的交叉编译结果，可被 make reset 命令清除
| | |--convert.c	生成 coe 和 mif 文件的本地执行程序源码
| | |--convert	convert.c 的本地编译后的可执行文件，可被 make reset 命令清除
| | |--rules.make	子编译脚本，被 Makefile 调用
| |--*	目录，其他功能或性能测试 func。
|	
|-soc_sram_func/	目录，自实现 CPU（SRAM 接口）的功能测试环境
| |--rtl/	目录， SoC_lite 的源码。
| | |--soc_lite_top.v	SoC_lite 的顶层。
| | |--myCPU /	**目录，自实现 CPU（SRAM 接口）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_prj1/	目录，Vivado2018.1 创建的 Vivado 工程 1
| | | |--*.xpr	Vivado2018.1 创建的 Vivado 工程，可直接打开
|-soc_axi_func/	目录，自实现 CPU（AXI 接口）的功能测试环境，与 soc_sram_func/类似

1.3.2 SoC_SRAM_Lite

当自实现 CPU 为 SRAM 接口时，运行的功能测试环境为 SoC_SRAM_Lite，CPU132_gettrace 也是该环境。

目录为发布包 func_test/soc_sram_func/。

SoC_SRAM_Lite 使一个简单的硬件系统。这个系统里面有你们设计的 CPU（mycpu），供 CPU 访问的指令 RAM（iram）和数据 RAM（dram），用于显示和输入的 LED 灯、数码管、按键，以及时钟、复位等。这个硬件系统是通过我们的 FPGA 实验板实现的。其中 LED 灯、数码管、按键以及时钟晶振是焊在 PCB 板上的，其余部分都实现在 FPGA 中。请注意，FPGA 中所实现的内容也可以看作是一个系统（System），而这些内容都实现在一个芯片（Chip）内部，我们通常将这里在 FPGA 中所实现的内容称为 SoC（System On Chip）。图 1-1 给出了整个简单硬件系统的结构示意图，中间 SoC_SRAM_Lite 区域内包含的就是最终实现在 FPGA 中的内容。可见 SoC_SRAM_Lite 中除了 mycpu 以外还有其它模块。整个 SoC 的设计代码位于 soc_sram_func/rtl/目录下。

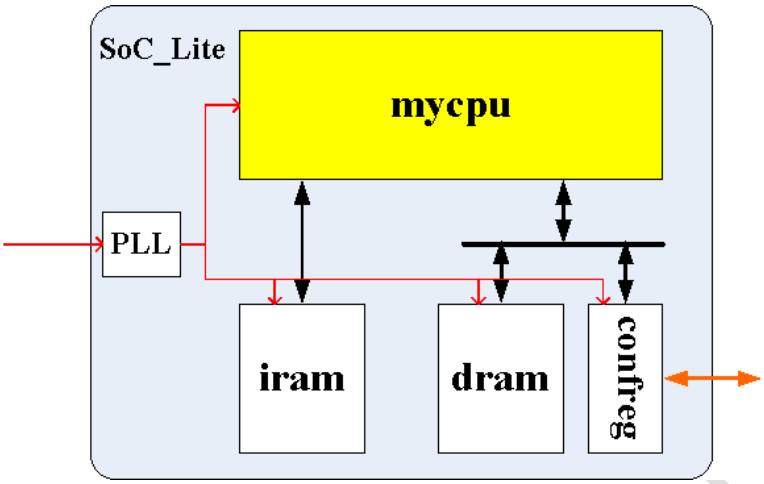

图 1-1 基于 mycpu 的简单硬件系统 SoC_SRAM_Lite

SoC_SRAM_Lite 中的 mycpu、iram 和 dram 各自功能是自明的。这里简单解释一下 PLL、confreg 以及 mycpu 与 dram、confreg 之间的二选一。

我们实验板上给 FPGA 芯片提供的时钟（来自于晶振）是 100MHz，如果直接用这个时钟作为 SoC_Lite 中各个模块的时钟，我们担心同学们设计的 mycpu 无法达到这个频率，所以调用 Xilinx 的 PLL IP，以 100MHz 输入时钟作为参考时钟，生成出一个频率低一些的时钟。

confreg 是 configuration register 的简称，是 SoC 内部的一些配置寄存器，实验中我们用来操控板上的 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直接连接到 FPGA 的引脚上的，通过控制 FPGA 引脚上的电平的高、低就可以控制 LED 灯和数码管，而按键是否按下也可以通过观察 FPGA 引脚上电平的变化来判断。这些 FPGA 引脚又进一步连接到 confreg 中的某些寄存器上。所以 CPU 可以通过观察、操控这些寄存器来实现对于 FPGA 相关引脚的观察、操控。

mycpu 和 dram、confreg 之间有一个二选一，这是因为 confreg 中的寄存器是 memory mapped，也就是说从 CPU 的角度来看，confreg 也是内存。dram 是数据 RAM，自然是内存。那么 confreg 和 dram 如何区分？用地址高位进行区分，所以就有了一个二选一。

最后再强调一点，SoC_SRAM _Lite 是进行 FPGA 综合实现时的顶层。

1.3.3 SoC_AXI_lite

当自实现 CPU 封装为 AXI 接口时，运行的功能测试环境为 SoC_AXI_Lite。

目录为发布包 func_test/soc_axi_func/。

组织结构和 SoC_SRAM_Lite 类似，只是外部存储器不再分为 iram 和 dram，而是只有一个存储单元 axi_ram。

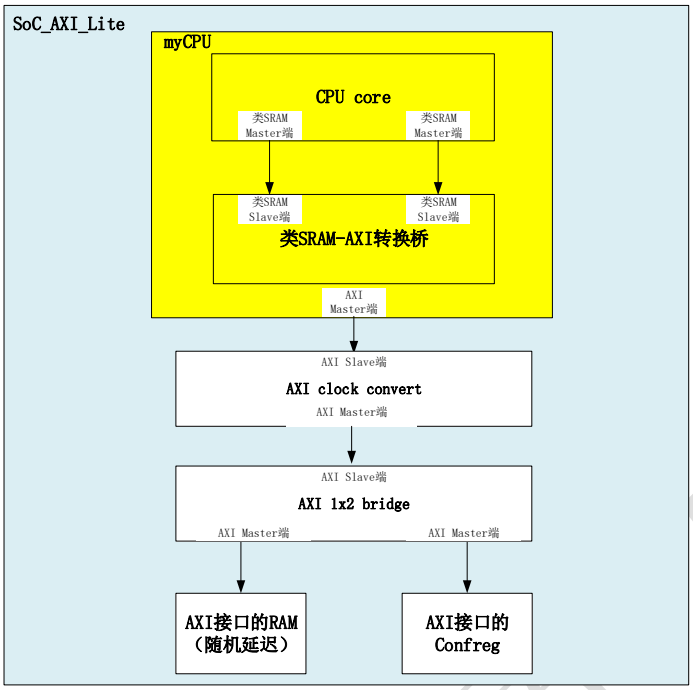

图 1-2 基于 mycpu 的简单硬件系统 SoC_AXI_Lite

1.3.4 功能仿真验证

说到 Verilog 设计的功能仿真验证，大多数讲 Verilog 的书籍都会给出如图 1-3 的示意图。

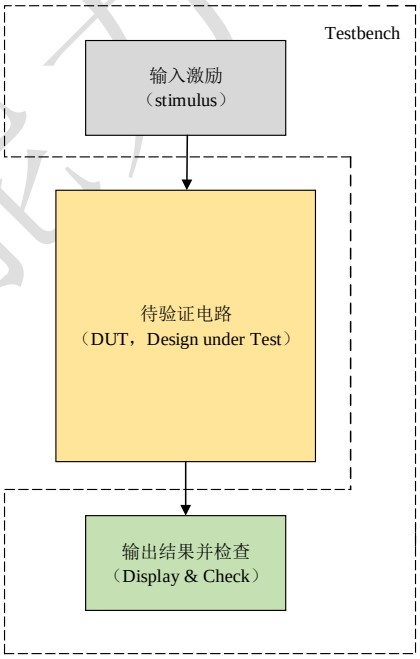

图 1-3 功能仿真验证

这个图其实也蛮好理解的，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。

我们给 CPU 设计进行功能测试时，沿用的依然是上面的思路，只不过输入激励和输出结果检查与普通的数字

电路设计不太一样。这里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。输出结果是什么呢？如果我们用整个执行序列最终的结果作为检查的依据，是可以的，但是出错的定位就比较困难。同学们都开发过 C 程序，调试的时候应该都用过单步调试，这种调试方式能够快速的定位出出错的位置。我们提供给同学们的实验开发环境就使用了这种“单步调试”的策略。具体来说，我们先找一个已知的功能上是正确的 CPU（开源的龙芯 GS132 处理器），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；然后我们在验证 myCPU 的时候，也跑同样的指令序列，在 myCPU 每条指令写寄存器的时候，将设计中的 PC 和写寄存器的信息同之前的 golden_trace 进行比对，如果不一样，那么立刻报错停止仿真。

对 MIPS 指令熟悉的同学可能马上就会问：分支指令和 store 指令不写寄存器，上面的方式没法判断啊？这些同学提的问题相当对。但是大多数情况下，分支和 store 指令执行的错误会被后续的写寄存器的指令判断出来。例如，如果分支跳转的不对，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，也会报错并停下来。store 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，也会抱错并停下来。我们这样设计 trace，是在报错的及时性和 CPU debug 接口的复杂度之间作了权衡。

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。

参考模型生成 golden_trace

功能测试程序 func 编译完成后，就可以使用验证平台里的 cpu132_gettrace 运行仿真生成参考 trace 了。cpu132_gettrace 也支持综合实现，然后上板运行，上板运行效果就是大家自实现 myCPU 上板运行的正确的效果。

cpu132_gettrace 里是采用 gs132 搭建的 SoC_Lite，见图 1-1。仿真顶层为 cpu132_gettrace/testbench/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'hbfc00100                //func 测试完成后会 32'hbfc00100 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 采样的信号为：

(1) CPU 写回级（最后一级，记为“wb”）的 PC，因而要求大家将每条指令的 PC 一路带到 wb 级。

(2) wb 级的写回使能。

(3) wb 级的写回的目的寄存器号。

(4) wb 级的写回的目的操作数。

显然并不是每时每刻，CPU 都是有写回的，因而 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。

myCPU 功能测试使用 golden_trace

myCPU 功能验证所使用的 SoC 架构是图 1-1 展示的 SoC_SRAM_Lite 或者是图 1-2 展示的 SoC_AXI_Lite，testbench 就与 cpu132_gettrace 里的有所不同了，见 mycpu_verify/testbench/mycpu_tb.v，重点部分代码如下：

```
……
`define TRACE_REF_FILE "../../../../../../../cpu132_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'hbfc00100                //func 测试完成后会 32'hbfc00100 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_v); //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
……
```

由于验证平台的 trace 比对时机同采样时机，都是要求指令具有写回且写回目的寄存器号非 0，才进行比对。所有存在以下两种情况，CPU 出错，而 trace 比对无法发现：

(1) myCPU 死机，时钟没有写回；

(2) myCPU 执行一段死循环程序，且该死循环程序没有写回的指令，比如程序"1: b 1b; nop;"。

不过这两种情况在我们提供的验证平台上依然可以被发现，这是通过 mycpu_tb.v 中的"monitor test"机制实现的，就是每隔 10000ns，打印一次写回级 PC。如果发现 Vivado 控制台不再打印写回级 PC，或者打印的都是同一 PC，则说明 CPU 死机了；如果 Vivado 控制台打印的 PC 具有很明显的规律，不停重复打印，则说明陷入了死循环。借此，可以帮助大家判断 myCPU 是否执行 func 出错。

1.3.5 功能测试程序说明

功能测试程序位于发布包的 func_test/soft/目录下，分为 func/start.S 和 func/inst/*.S，都是 MIPS 汇编程序:

(1) func/start.S ：主函数，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，比如 lab2 里共有 15 条指令对应的汇编测试程序。

主函数 func/start.S 中主题部分代码如下，分为三大部分，具体查看红色注释。

```
……
#以下是设置程序开始的 LED 灯和数码管显示，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)
    lui s3, 0

    sw t1, 0(a0)
    sw t2, 0(a1)
    sw t3, 0(a2)
    sw s3, 0(s1)
#以下是运行各功能点测试，每个测试完执行 wait_1s 等待一段时间，且数码管显示加 1。
  inst_test:
```

```
  jal n1_lui_test    #lui
  nop
  jal wait_1s
  nop
  jal n2_addu_test   #addu
  nop
……
#以下是显示测试结果，PASS 则双色 LED 灯亮两个绿色；
#Fail 则双色 LED 灯亮两个红色。
test_end:
test_end:
    LI  (s0, 0x14)
    NOP
    NOP
    NOP
    beq s0, s3, 1f
    nop

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

    LI (t1, 0x0002)
    NOP
    NOP

    sw zero, 0(a0)
    sw t1, 0(a1)
    sw t1, 0(a2)
……
```

每个功能点的测试代码程序名为 n*_*_test.S，其中第一个“*”处为编号，共 89 个功能点测试，则从 n1 编号到 n89。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_lui_test)
  .set noreorder
  addiu s0, s0, 1                    #累加功能点编号
  addiu s2, zero, 0x0
  lui   t2, 0x1
  ###test inst
  addiu t1, zero, 0x0
  TEST_LUI(0x0000, 0x0000)
  …….#测试程序，省略
    TEST_LUI(0xf0af, 0xf0a0)
  ###detect exception
    bne s2, zero, inst_error
    nop
  ###score ++                    #s3 存放功能测试计分，没通过一个功能点测试，则+1
    addiu s3, s3, 1
  ###output (s0<<24)|s3
```

```
  inst_error:
    sll t1, s0, 24              #功能点编号移位到高 8 位
    or t0, t1, s3               #t1 高 8 位为功能点编号，s3 低 8 位为通过功能点数，相与结果显示到数码管上
    sw t0, 0(s1)                #s1 存放数码管地址
    jr ra
    nop
END(n1_lui_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 wait_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 wait_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。

1.3.6 编译脚本说明

func 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：

(1) make help ：查看帮助信息。

(2) make ：编译得到仿真下使用的结果。

(3) make clean ：删除*.o，*.a 和./obj/目录。

(4) make reset ：执行命令”make clean”，且删除 convert，bin.ld。

1.3.7 编译结果说明

func 编译结果位于 func/obj/下，共有 9 个文件，各文件具体解释见表 1-1，文件生成关系见图 1-4。

表 1-1 编译生成文件

文件名	解释
data_ram.coe	重新定制 data ram 所需的 coe 文件
data_ram.mif	仿真时 data ram 读取的 mif 文件
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
axi_ram.mif	复制自 inst_ram.mif
main.bin	编译后的代码段，可以不用关注
main.data	编译后的数据段，可以不用关注
main.elf	编译后的 elf 文件
test.s	对 main.elf 反汇编得到的文件

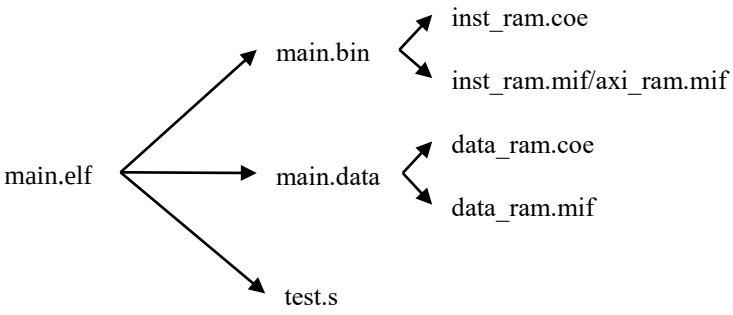

图 1-4 编译得到的文件生成关系