软工计原联合项目

调试文档

NonExist 组
张钰晖，杨一滨，周正平

目录

1 文档说明 **2**

2 仿真调试 **3**

2.1 简介概述 3

2.2 具体实现 3

2.2.1 搭建仿真平台 3

2.2.2 初始化 9

2.2.3 时钟单步调试 9

2.2.4 时钟快速调试 9

3 硬件调试 **10**

3.1 简介概述 10

3.2 具体实现 10

4 输出调试 **12**

4.1 简介概述 12

4.2 具体实现 12

Chapter 1

文档说明

写码一小时，调试一整天。在真正的整个开发过程中，开发、测试、调试的时间几乎相等，实现 CPU 的过程中 Bug 是不可避免的，硬件调试也是非常痛苦的，不像软件开发可以轻易的输出中间结果，硬件出了问题没有任何方法可以轻松的 print 到屏幕上，因此调试非常重要。

本文档是 NonExist 组软工计原联合项目的调试文档，本文档主要描述了在实现 CPU 的过程中我们如何进行调试。

调试文档将项目分成了以下部分：

1. **仿真调试**：分阶段用 Verilog 实现 SOPC 进行软件仿真调试。
2. **硬件调试**：连接在板子上手按时钟进行调试。
3. **输出调试**：用串口输出信息进行调试。

调试文档每个章节遵从以下介绍流程：

1. **简介概述**：简要介绍调试的方法。
2. **具体实现**：详细介绍调试的方法和注意事项。

希望本文档能给读者带来裨益。

Chapter 2

仿真调试

2.1 简介概述

在开发的任何时间周期内，尤其是开发前期阶段，仿真调试都起到了至关重要、不可替代的作用。这里的仿真主要是指功能仿真，不考虑硬件的时序约束。

仿真调试有以下优点：

（1）无需编译。在开发后期，无论多么微小的改动，Vivado 在高性能笔记本上完整编译一次（合成、实现、生成比特流）至少需要 10 分钟！尤其是编译到一半发现代码写错重新编译是最令人绝望的事。而仿真无需编译，如果无法仿真，报错信息相比编译报错信息非常友好。

（2）信息全部透明。可以单步调试，所有数据的信息全部可以通过仿真窗口完整的查看，比通过 LED 灯或者七段数码管显示友好太多。

（3）速度快。仿真过程中一次仿真几百万 ns 瞬间就能完成，更可以随时回退，重新仿真。

仿真调试有以下缺点：

（1）无时序约束。功能仿真不考虑时序，有可能因为时序的问题（如数据建立时间、数据保持时间没有考虑），仿真结果正确而实际运行不正确。

事实上，在笔者的开发过程中，绝大多数情况下，仿真和板子实际运行的结果是相同的。

仿真可以说是硬件调试界的 gdb，具有无可比拟的强大优势，硬件开发必须熟练掌握这一武器。

2.2 具体实现

2.2.1 搭建仿真平台

想要进行仿真调试，第一步便是搭建仿真平台，即搭建 SOPC。

笔者在整个开发阶段中共搭建了两个 SOPC。

初期阶段，SOPC 主要包含 CPU，配合上指令存储器、数据存储器，主要测试的是 CPU 能不能执行正确单条指令，代码如下，相信聪明的读者一定能一眼看懂。

```
`include "defines.v"

module openmips_min_sopc(
```

```
    input wire clk,
    input wire rst
);

    wire[`InstAddrBus] inst_addr;
    wire[`InstBus] inst;
    wire rom_ce;
    wire mem_we_i;
    wire[`RegBus] mem_addr_i;
    wire[`RegBus] mem_data_i;
    wire[`RegBus] mem_data_o;
    wire[3:0] mem_sel_i;
    wire mem_ce_i;
    wire[5:0] int;
    wire timer_int;

    assign int = {5'b00000, timer_int};

    openmips openmips0(
        .clk(clk),
        .rst(rst),
        .if_addr_o(inst_addr),
        .if_data_i(inst),
        .if_ce_o(rom_ce),
        .mem_we_o(mem_we_i),
        .mem_addr_o(mem_addr_i),
        .mem_sel_o(mem_sel_i),
        .mem_data_o(mem_data_i),
        .mem_data_i(mem_data_o),
        .mem_ce_o(mem_ce_i),
        .int_i(int),
        .timer_int_o(timer_int)
    );

    inst_rom inst_rom0(
        .addr(inst_addr),
        .inst(inst),
        .ce(rom_ce)
    );

    data_ram data_ram0(
        .clk(clk),
        .we(mem_we_i),
        .addr(mem_addr_i),
        .sel(mem_sel_i),
        .data_i(mem_data_i),
        .data_o(mem_data_o),
        .ce(mem_ce_i)
    );

endmodule // openmips_min_sopc
```

```
`include "defines.v"

module inst_rom(
```

```
    input wire ce,
    input wire[`InstAddrBus] addr,
    output reg[`InstBus] inst
);

    reg[`InstBus] inst_mem[0:`InstMemNum-1];

    initial $readmemh ("inst_rom.data", inst_mem);

    always @(*) begin
        if (ce == `ChipDisable) begin
            inst <= `ZeroWord;
        end else begin
            inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
        end
    end

endmodule // inst_rom
```

```
`include "defines.v"
module data_ram(
    input wire clk,
    input wire ce,
    input wire we,
    input wire[`DataAddrBus] addr,
    input wire[3:0] sel,
    input wire[`DataBus] data_i,
    output reg[`DataBus] data_o
);
    reg[`ByteWidth] data_mem0[0:`DataMemNum-1];
    reg[`ByteWidth] data_mem1[0:`DataMemNum-1];
    reg[`ByteWidth] data_mem2[0:`DataMemNum-1];
    reg[`ByteWidth] data_mem3[0:`DataMemNum-1];

    always @(posedge clk) begin
        if (ce == `ChipDisable) begin
            data_o <= `ZeroWord;
        end else if (we == `WriteEnable) begin
            if (sel[3] == 1'b1) begin
                data_mem3[addr[`DataMemNumLog2+1:2]] <= data_i[31:24];
            end
            if (sel[2] == 1'b1) begin
                data_mem2[addr[`DataMemNumLog2+1:2]] <= data_i[23:16];
            end
            if (sel[1] == 1'b1) begin
                data_mem1[addr[`DataMemNumLog2+1:2]] <= data_i[15:8];
            end
            if (sel[0] == 1'b1) begin
                data_mem0[addr[`DataMemNumLog2+1:2]] <= data_i[7:0];
            end
        end
    end

    always @(*) begin
        if (ce == `ChipDisable) begin
```

```
            data_o <= `ZeroWord;
        end else if (we == `WriteDisable) begin
            data_o <= {data_mem3[addr[`DataMemNumLog2+1:2]], data_mem2[addr[`DataMemNumLog2+1:2]],
    data_mem1[addr[`DataMemNumLog2+1:2]], data_mem0[addr[`DataMemNumLog2+1:2]]};
        end else begin
            data_o <= `ZeroWord;
        end
    end

endmodule // data_ram
```

后期阶段，SOPC 主要包含了内含 CPU 的 thinpad_top，thinpad_top 是硬件端的顶层文件，配合上模拟硬件实现的 Flash、RAM，实现过程中严格按照硬件的实际情况实现，保证接口、大小端、编址方式全部和硬件相同，这样可以保证，不考虑时序的前提下，仿真的结果和硬件结果完全相同。

```
`include "defines.v"
`timescale 1ns/1ps

module thinpad_min_sopc();

    wire [31:0] ram_data;
    wire [19:0] ram_addr;
    wire [3:0] ram_be_n;
    wire ram_ce_n;
    wire ram_we_n;
    wire [31:0] ext_ram_data;
    wire [19:0] ext_ram_addr;
    wire [3:0] ext_ram_be_n;
    wire ext_ram_ce_n;
    wire ext_ram_we_n;
    wire [5:0] touch_btn;
    wire [22:0] flash_addr;
    wire [15:0] flash_data;

    reg clk;
    reg clk_25;
    reg CLOCK_11059200;
    reg rst;

    initial begin
        clk = 1'b0;
        forever #10 clk = ~clk;
    end
    initial begin
        clk_25 = 1'b0;
        forever #20 clk_25 = ~clk_25;
    end
    initial begin
        CLOCK_11059200 = 1'b0;
        forever #45.211227 CLOCK_11059200 = ~CLOCK_11059200;
    end

    initial begin
        rst = 1'b1;
        #195 rst = 1'b0;
```

```
        #500000000 $stop;
    end

    assign touch_btn = {rst, 5'b00000};

    thinpad_top thinpad_top0(
        .clk_in(clk),
        .clk_uart_in(CLOCK_11059200),
        .touch_btn(touch_btn),
        .base_ram_data(ram_data),
        .base_ram_addr(ram_addr),
        .base_ram_be_n(ram_be_n),
        .base_ram_we_n(ram_we_n),
        .base_ram_ce_n(ram_ce_n),
        .ext_ram_data(ext_ram_data),
        .ext_ram_addr(ext_ram_addr),
        .ext_ram_be_n(ext_ram_be_n),
        .ext_ram_we_n(ext_ram_we_n),
        .ext_ram_ce_n(ext_ram_ce_n),
        .flash_data(flash_data),
        .flash_a(flash_addr)

    );

    flash flash0(
        .clk(clk),
        .a(flash_addr[22:1]),
        .data(flash_data)
    );

    ram ram0(
        .clk(clk),
        .we(ram_we_n),
        .addr(ram_addr),
        .be(ram_be_n),
        .data(ram_data),
        .ce(ram_ce_n)
    );

    ram ram1(
        .clk(clk),
        .we(ext_ram_we_n),
        .addr(ext_ram_addr),
        .be(ext_ram_be_n),
        .data(ext_ram_data),
        .ce(ext_ram_ce_n)
    );

endmodule // openmips_min_sopc
```

```
module ram(
    input wire clk,
    input wire ce,
    input wire we,
    input wire oe,
```

```
    input wire[19:0] addr,
    input wire[3:0] be,
    inout wire[31:0] data
);
    reg[31:0] data_ram[0:1048575];

    // initial $readmemh ("ram.data", data_ram);

    assign oe = 1'b0;

    always @(posedge clk) begin
        if (ce == 1'b1) begin

        end else if (we == 1'b0) begin
            if (be[3] == 1'b0) begin
                data_ram[addr][7:0] <= data[31:24];
            end
            if (be[2] == 1'b0) begin
                data_ram[addr][15:8] <= data[23:16];
            end
            if (be[1] == 1'b0) begin
                data_ram[addr][23:16] <= data[15:8];
            end
            if (be[0] == 1'b0) begin
                data_ram[addr][31:24] <= data[7:0];
            end
        end
    end

    assign data = we? { data_ram[addr][7:0], data_ram[addr][15:8], data_ram[addr][23:16], data_ram[addr
    ][31:24] }: 32'bzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz;

endmodule // data_ram
```

```
module flash(
    input wire clk,
    input wire ce,
    input wire we,
    input wire oe,
    input wire rp,
    input wire byte,
    input wire[21:0] a,
    inout wire[15:0] data
);
    reg[15:0] data_flash[0:4194303];

    initial $readmemh ("flash.data", data_flash);

    assign rp = 1'b1;
    assign byte = 1'b1;
    assign we = 1'b1;
    assign oe = 1'b0;
    assign ce = 1'b0;

    assign data = { data_flash[a][7:0], data_flash[a][15:8] };
```

```
endmodule // data_flash
```

2.2.2 初始化

初始化主要有以下内容：

设置仿真时间上界。

设置 clk 时钟信号，这里时钟频率可以是任意的，因为功能仿真并没有考虑时序。

设置 rst 重置信号，这里的重置信号在某一时间从 1 跳变为 0，CPU 开始运行。

通过 initial 命令进行信号的初始化。

通过文件配合 initial 命令对存储器进行数据的初始化。

2.2.3 时钟单步调试

在搭建仿真平台并初始化后，下一步便是打开进入激动人心的仿真了！

先重置到 0ns，然后根据仿真设置的 clk 时钟频率，例如 25MHz 的时钟频率就对应 40ns 一个时钟周期，开始进行单步调试，单步调试时可以通过左侧的窗口详细完整的看到所有寄存器的数据信息。

通过跟踪这些状态信息，读者就可以清楚地定位到某一条指令执行是否正确，某一模块实现是否存在异常。

总体而言，仿真调试好比一个图形化的 gdb，可以帮助开发人员在任何阶段快速定位到错误信息。

2.2.4 时钟快速调试

本部分调试适用于开发中后期，待测试指令条数非常多时。

例如运行功能测例时，每通过一个指令 19 号寄存器值加 1，由于测试指令有很多很多条，可能执行 10000 条指令才能出现想看到的结果（寄存器值变化），对此可以根据仿真设置的 clk 时钟频率，快速推进十万个时钟周期。

如果结果和预期不符，利用二分法找出错误位置，再重复时钟单步调试过程。

Chapter 3

硬件调试

3.1 简介概述

只有做了充足的仿真，确保仿真结果运行完全正确，笔者才推荐进入这一阶段，硬件调试。

顾名思义，硬件调试，就是在硬件上的调试。

先将数据信息烧入到 Flash 和 RAM 中，利用 Vivado 编译生成出的 bitstream 文件烧入 FPGA，注意这一步顺序不能相反，否则需要重新烧入 FPGA。

相比仿真调试，硬件调试的过程非常痛苦，痛苦之处有以下三点：

（1）编译慢。无论多么细微的改动，完整的编译需要合成、实现、生成比特流三步，编译的速度和 CPU 核心数没有关系，只与单核主频有关，即使是高性能笔记本电脑，编译一次也需要整整 10 分钟。

（2）不灵活。如果接 Thinpad 板子上的 clk 时钟，clk 时钟没有方式能到一个固定时间停止；如果手按时钟，仅适用于指令数量极少的时期，如果指令数目很多，很可能按一天也按不到想要的结果，更何况如果一不小心到了结果多按一次，唯一的解决办法只有按下 rst，重新来过。

（3）信息不透明。想看到某一个数据信息，唯一的方法便是通过 Thinpad 开发板上 16 个 LED 灯和七段数码管显示出来，如果你想看另一个数据信息，请重新编译，重新来过。

但是即使硬件调试有这么多缺点，硬件调试也是不可缺少的必经过程，因为功能仿真无法解决时序问题，硬件运行正确才是我们最终的目标。

3.2 具体实现

接上板子降频后的 clk 时钟后（由于硬件延迟时间所限，一般 CPU 无法运行在板子自带时钟 50MHz），由于做了充足的仿真测试，你的硬件测试结果应该和仿真结果一致。

如果不一致，推荐读者先对 clk 时钟降频再次测试，例如降频至 1MHz，一般错误原因均为时序原因，如果降频后结果正确，说明是硬件条件限制导致主频不能过高，你只能选择优化 CPU 的代码。

如果降频仍然错误。那么只能进行硬件调试了。

硬件调试的方法是：

（1）CPU 输入时钟改为 clk 按键

（2）将你想看到的调试信息，例如当前指令地址、当前指令内容等，通过 MMU 地址映射输出到 LED 灯和七段数码管中。

（3）精简数据，由于手按时钟不可能按太多次，尽可能保留精简的指令，例如功能测例只放入一组测例的一小部分等，精简后将数据拷入 RAM 和 Flash

（4）编译并烧入 FPGA

接下来，手按时钟，根据 LED 信息和七段数码管信息分析错误原因。

Chapter 4

输出调试

4.1 简介概述

如果读者已经把串口调试通过，可以通过串口收发信息，那么可以通过修改软件直接将数据发送至串口查看硬件信息。

听起来非常棒，就像是调试软件中可以向屏幕 print 信息一样，但事实上，现实并没有那么美好。

（1）功能测例本身不支持向串口发送信息，很难重构功能测例的代码实现向串口发送信息。更何况功能测例是测试均为最基本的指令，向串口发送数据你至少要保证基本的指令要实现正确、MMU 要实现正确等等。

（2）监控程序支持向串口发送信息，但非常不完善，不仅是本身有 bug，而且几乎不能向串口发送任何有用的错误信息，如果你的监控程序可以向串口正常发送信息了，那么你大概率监控程序已经调试通过了。

只有 ucore 能向串口发送出非常有用的信息，告诉读者哪一部分代码执行错误了，所以，读者看到这一章节时，已经基本到达了调试尾声了。

值得庆祝的是，这部分调试比之前两部分调试友善的多，因为 ucore 非常完善，可以修改 ucore，通过 ucore 自带的 cprintf 语句轻松的向串口发送各种信息，事实上，进入这一部分，就与正常写软件调试相差无几了。

4.2 具体实现

根据前面的假定，认为读者本阶段在调试 ucore 操作系统，同时认为读者前面阶段已经进行了充足的测试，能够完美运行功能测例和监控程序。

在将 ucore 烧入 Flash，将硬件烧入 FPGA 后，ucore 操作系统会开始启动。

如果 ucore 没有正常启动，ucore 会在串口输出的错误信息，同时进入内核调试程序，内核调试程序无太大用处。输出的错误信息中内含错误位置，定位到 ucore 操作系统对应的位置上，根据函数调用过程加入 cprintf 输出一些变量信息，cprintf 用法和 C 语言 printf 用法无太大差异，详情可阅读操作系统文件。

重新编译 ucore，重新烧入 Flash，无需重新编译烧入 FPGA，通过不断定位找出错误位置，思考

硬件实现为什么错误。

在本阶段，编译 ucore 时间仅仅几秒钟，更可以通过 cprintf 自由的输出各种信息，相比之下调试可以算是非常友好。

最后祝读者调试顺利！