计算机系统结构课程设计实验报告

——MIPS 指令集流水线 CPU 改造

同济大学
TONGJI UNIVERSITY

院　　系　电子与信息工程学院
专　　业　计算机科学与技术
姓　　名　庄镇华
学　　号　1853790
题　　目　MIPS 指令集流水线 CPU 改造
指导老师　郭玉臣
联系方式　17721295617
完成日期　2021.04.25

目录

一、实验环境与实验内容

1.1 实验环境

1.1.1 VIVADO 的安装

a.关闭防火墙软件，访问 https://www.xilinx.com/support/download.html，下载 vivado 在线安装文件（或者下载完整安装包），运行安装程序，注册（登录）Xilinx 账号。

b.打开在线安装程序，输入注册账号以及选择下载位置，将安装文件放在英文目录下，点击 next，看到软件下载信息，点击 download 开始下载。

c.下载完成打开下载目录，双击 xsetup.exe 进行安装。

d.选择 vivado HL Design Edition。

e.选择安装位置，注意不要使用中文以及带空格的目录。

f.点击 next，看到安装信息，开始安装。

g.证书的安装，选择左侧的 load license，然后点击 copy license，选择证书， 安装结束。

1.1.2 MODELSIM 的安装

a. 访问 https://www.mentor.com/products/fpga/download/modelsim-pe-simulator-download 下载 modelsimPE 安装包。

b.关闭防火墙软件，执行安装程序，点击 NEXT。

c.进入安装界面，此处可更改默认的安装路径。

d.系统提示安装路径不存在，是否建立新的安装路径，点击 Yes。

e.安装 Key Driver，点击 Yes（win10 系统选择 No）。

f.安装完 License 之后重启。

g.建系统变量，变量名为 MGLS_LICENSE_FILE。

1.1.3 VIVADO 和MODELSIM 软件的关联

a.打开 Vivado 2016.2 Tcl Shell。

b.输入命令：compile_simlib -directory <保存编译生成的库文件的路径> -simulator modelsim -simulator_exec_path <Modelsim 可执行文件的路径>

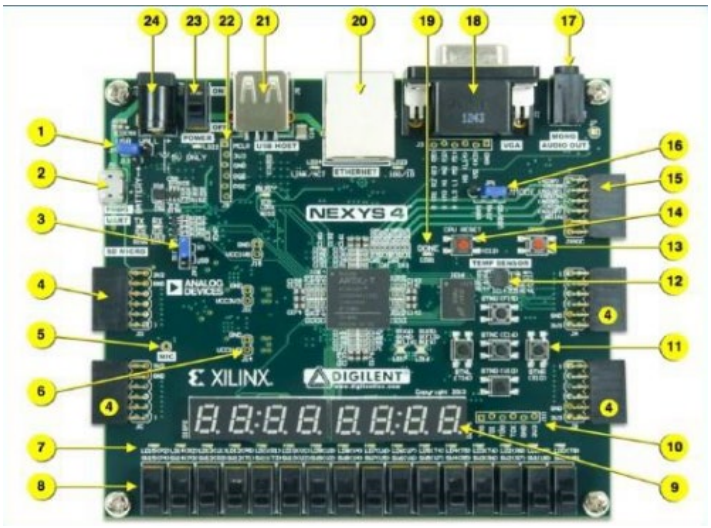
c.编译结束，查看 D:/xilinx_sim_lib 文件夹，已经生成了库文件。

d. 设置关联，打开 Vivado。Tools-->options-->General 选择 modelsim 的安装路径。在工程中对仿真工具进行配置，如下图。 选择 Simulation Setting-->Simulation-->将 Target simulation 设为 modelsim, Compiled library location 设为 D:/xilinx_sim_lib。

1.1.4 XILINX NEXYS 4 DDR 开发板

Nexys4 是 Digilent 公司的基于 Xilinx 公司最新的 Artix-7 系列 FPGA 的开发板。它可以通过 USB 供电、编程以及串口监视，只需要一根 micro USB 连接线就行，搭建工作平台还是很简单。Nexys4 是基于最新的 Xilinx7 系列 FPGA，板上核心器件是一块 Artix-7 FPGA(XC7A100T-1CSG324C)。

XC7A100T-1CSG324C 是 Artix-7 系列中资源比较丰富的一款芯片，多达 10 万逻辑单元(Logic Cells)同时集成了硬件 AXI IP 资源和模拟混合信号，支持 8 路 6.6G 收发器、930GMAC、13Mb 的 BRAM、1.2Gb/s LVDS 和 DDR3-1066 等。XC7A100T-1CSG324C 采用了 Chip scale 封装，尺寸只有 15mm×15mm，最大用户 IO 达到 210 个。



1.2 实验内容

1.2.1 目的

- 回顾 MIPS 五阶段流水线 CPU 的基本结构
- 学习教学用 CPU 各模块的编写以及模块之间的连接结构
- 加深对 Verilog 语言的理解以及对硬件开发的理解
- 为之后移植操作系统做准备工作

1.2.2 内容

总体实验步骤：

- 基本部分：

任务 1、MIPSCPU 的改造，适合我们的 N4 板，下板；

任务 2、μC/OS-II 或其它操作系统的移植。

- 扩展部分：

任务 3、应用程序开发，包括 GUI 程序等类别

本次实验是总体实验的第一步，实验可以采用三种方案：

- 《自己动手做 cpu》书上相关的路线，即改造 89 条指令 cpu（原有实验 cpu 基础上）、cpu 下板、移植 uc-OSII 操作系统；
- 按照已经给 45 条指令 cpu（这个可下载系统结构群中给出的 cpu），改造这个 cpu 下板，然后按照所给出操作系统移植；
- 自己选取 cpu 下载到我们的 N4 板，寻找合适操作系统移植；

本次实验我采用第一种方案，即改造 89 条指令 cpu，实现 CP0、异常处理，我主要参照了《自己动手写 CPU》的基础篇，扩展完善了早期自己设计的基于 MIPS 指令集的动态流水线 CPU，从指令集的指令条数、总线及总线控制器等方面完善，使中央处理器具备多于已有的 54 条 MIPS 指令集，并具备总线的管理和中断管理功能，以适应后期嵌入式操作系统和应用软件的运行。

1.2.3 测试

老师在系统结构群里给出的测试用例只针对第一种思路，第二、第三种思路的测试需要自己设计测试用例。

对于第一种思路，老师分享了大端和小端两种方式的测试用例。大端和小端模式是指数据在内存中的存储模式，它由 CPU 决定：大端模式是指将数据的低位字节排放在内存的高地址端，高位字节排放在内存的低地址端，而小端模式是指将低位字节排放在内存的低地址端高位字节排放在内存的高地址端。不难看出大端模式比较符合人的直观认识，但大小端模式各有优势：小端模式强制转换类型时不需要调整字节内容，直接截取低字节即可；大端模式由于符号位为第一个字节，很方便判断正负。

本次实验我采用第一种思路的大端模式，测试用例由李家麟同学提供并由老师分享。

二、实验过程与方法

2.1 实验过程

1. 建立新的 vivado 工程文件，将《自己动手写 CPU》第 11 章的所有文件加载入。该 CPU 工程文件主要分成三个部分：第一，头文件部分，第二，CPU 主体部分，第三，仿真部分。在载入到本地 vivado 工程中时，需要分清以上三个模块，分别载入。

2. 将测试指令用 Mars 软件转化为 16 进制 coe 文件，并配置 ip 核以便后续进行下板测试。

3. 修改底层模块，比如分支预测部分，在《自己动手写 CPU》的代码当中，连续两个分支指令，尤其是一条条件跳转指令，紧接着一条无条件跳转指令会出现一定的问题。如下图所示，bgtz 在计算出是否应当分支的时候，其目标地址已经被 j 指令的目标地址给覆盖掉，表现为 bgtz 指令无效。因此我们可以修改分支预测部分或者更新测试文件。

```
bgtz $10,TAG5
j test7_error
```

比如加上 break 指令，对于《自己动手写 CPU》的代码而言，是不存在 break 指令的（查阅了纸质书），代码中执行 break 的逻辑是走 inst_invalid 的路线。而这个操作会将 cause 寄存器赋值为 0x28，而非我们检验所需的 0x24。因此我们可以自行加上 break 指令，或者更新测试文件。

4. 修改顶层模块，配置并增添分频器和数码管显示模块，使得下板测试的时候可以实时显示，方便评测。

5. 修改约束文件，使之符合我们的 Nexys 4 开发板的引脚，使得下板测试的时候可以实时显示，方便评测。

6. 增添仿真文件，连接 MIPS 流水线 CPU 模块和数码管模块，对测试指令进行一条条连续的测试。

7. 完成以上部署后，进行仿真模拟，该 CPU 采用了读文件的方式读取指令，对应代码为 inst_ram.v 文件，将其读写的文件改为自己的指令即可。

8. 然后进行综合，生成二进制 bit 流，下板调试，根据测试文件的对应结果检验 MIPS 流水线 CPU 是否已经修改成功。

2.2 实验方法

2.2.1 分模块逐个修改

《自己动手写 CPU》中 OpenMIPS 处理器是不断迭代进行的，即首先实现一条简单的 ori 指令，通过该指令建立了基本的流水线结构，然后依次添加实现了逻辑操作指令、移位操作指令、空指令、移动操作指令、算术操作指令、转移指令、加载存储指令、协处理器访问指令、异常相关指令，如此由小到大，由简单到复杂，最终实现的。

而我们在修改 OpenMIPS 基础篇处理器时也可以采用类似的思路，即针对测试文件出错的模块进行改善与处理，由于《自己动手写 CPU》这本书的代码质量较高，模块间耦合性低、模块内内聚性强，因此修改起来还是比较容易的。

比如修改分支预测部分，在《自己动手写 CPU》的代码当中，连续两个分支指令，尤其是一条条件跳转指令，紧接着一条无条件跳转指令会出现一定的问题。例如 bgtz 在计算出是

否应当分支的时候，其目标地址已经被 j 指令的目标地址给覆盖掉，表现为 bgtz 指令无效。因此我们可以修改分支预测部分或者更新测试文件。

又比如 break 指令，对于《自己动手写 CPU》的代码而言，是不存在 break 指令的（查阅了纸质书），代码中执行 break 的逻辑是走 inst_invalid 的路线。而这个操作会将 cause 寄存器赋值为 0x28，而非我们检验所需的 0x24。因此我们可以自行加上 break 指令，或者更新测试文件。

2.2.2 灵活使用 DEBUG 技巧

前后仿真

仿真（Simulation），即使用项目模型将特定于某一具体层次的不确定性转化为它们对目标的影响，该影响是在项目仿真项目整体的层次上表示的。项目仿真利用计算机模型和某一具体层次的风险估计。

前仿真 = 功能仿真 = 行为级仿真 = RTL 级仿真，是指仅对逻辑功能进行测试模拟，以了解其实现的功能是否满足原设计的要求，仿真过程没有加入时序信息，不涉及具体器件的硬件特性，如延时特性；后仿真 = 时序仿真 = 布局布线后仿真 = 门级仿真，也称为布局布线后仿真或时序仿真。是指提取有关的器件延迟、连线延时等时序参数，并在此基础上进行的仿真，它是非常接近真实器件运行情况的仿真。

在计算机系统结构的实验中，若每次测试都先生成二进制 bit 流，再进行下板测试，这将耗费大量的时间，这不利于我们进行调试。利用计算机实现对于系统的仿真研究不仅方便、灵活，而且也是经济的。

综合下板

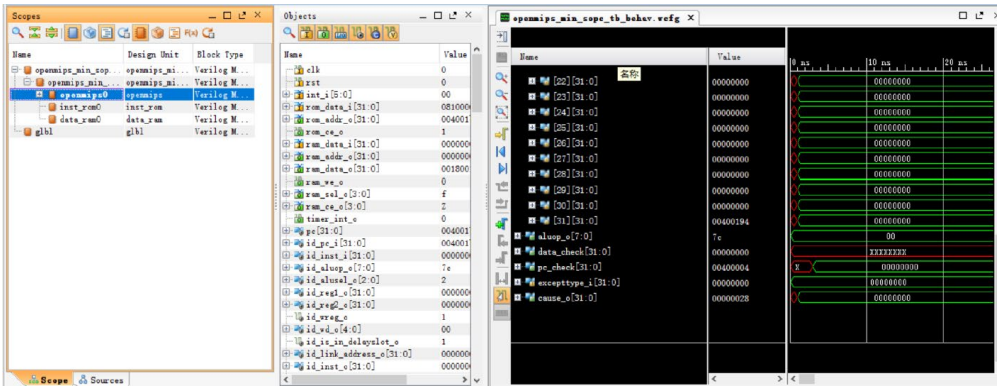
综合是将我们的设计转化为 FPGA 可以读懂的配置文件的第一个步骤，是把 HDL 语言/原理图转换为综合网表的过程。综合的结果是综合网表， Xilinx 自家的综合结果是 NGC 网表。NGC 网表是二进制的文件，不能用文本编辑器打开观察。综合网表中除了包含从 HDL 语言中 infer 出的与门、非门等组合逻辑和寄存器等时序逻辑之外，还包含 FPGA 特有的各种原语(Primitive)比如 LUT，BRAM，DSP48，甚至 PowerPC，PCIe 等硬核模块，以及这些模块的属性和约束信息。

在仿真测试都能通过时，这时为了验证程序的正确性，需要进行下板操作。当下板都能得到理论中的结果时，这时即表明设计无误。

DEBUG 思路

老师分享的测试文件对于出错后的处理比较好，在遇到错误之后会打印错误代码到$5，并且执行死循环。因此可以使用 Vivado 自带的仿真工具、或者使用 ModelSim。

仿真：



将需要查看的寄存器值拖入波形，再执行 Relaunch Simulation 指令就可以查看实时变化的中间变量。可以打印出 reg $5 或者 0x10010000 当中存储的值，通过最终变化的值来判断错误在哪个测试项目，再结合 Mars 等汇编工具，对寄存器组当中的值进行逐指令的排查。

三、程序修改说明

3.1 顶层模块

在《自己动手写 CPU》的 MIPS 流水线基础版 CPU 中，仅包含 MIPS 指令集 CPU 的顶层模块，而没有将 CPU 和数码管连接的模块，因此我们需要重新设计顶层模块，仿照我们上学期在《计算机系统结构》实验中的顶层模块设计，在顶层模块中将 CPU、指令存储器、数据存储器、七段数码显示管模块相连接。注意，顶层模块同时也包含了分频器模块的内容。

代码如下：

```
`include "defines.v"

module openmips_min_sopc(

    input wire clk,
    input wire  rst,
    output [7:0] O_Seg,
    output [7:0] O_Sel

);

  reg clk_s;
  initial clk_s=0;
  integer cnt=0;
  always @(posedge clk)
  begin
  if(cnt < 10000)
    cnt = cnt+1;
  else
     begin
        cnt = 0;
        clk_s = ~clk_s;
```

```
    end
  end


  //连接指令存储器
  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;
  wire[`DataBus] Seg7_In;

  //assign int = {5'b00000, timer_int, gpio_int, uart_int};
  assign int = {5'b00000, timer_int};

    openmips openmips0(
        .clk(clk_s),
        .rst(rst),

        .rom_addr_o(inst_addr),
        .rom_data_i(inst),
        .rom_ce_o(rom_ce),

        .int_i(int),

        .ram_we_o(mem_we_i),
        .ram_addr_o(mem_addr_i),
        .ram_sel_o(mem_sel_i),
        .ram_data_o(mem_data_i),
        .ram_data_i(mem_data_o),
        .ram_ce_o(mem_ce_i),

        .timer_int_o(timer_int)

    );

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

    data_ram data_ram0(
        .clk(clk_s),
        .ce(mem_ce_i),
        .we(mem_we_i),
        .addr(mem_addr_i),
        .sel(mem_sel_i),
```

```
        .data_i(mem_data_i),
        .data_o(mem_data_o),
        .debugreg(Seg7_In)
    );

    Seg7x16 Seg7(clk, rst, 1, Seg7_In, O_Seg, O_Sel);

endmodule
```

3.2 控制器模块

为了配合测试文件的要求，即中断例程起始地址为 0x00400004，需要修改 ctrl.v 文件里的跳转地址， 将中断跳转地址更改为要求的 0x00400004。这样可以保证最后测试输出结果的正确性。

代码如下：

```
`include "defines.v"

module ctrl(

    input wire                  rst,
    input wire[31:0]            excepttype_i,
    input wire[`RegBus]         cp0_epc_i,
    input wire                  stallreq_from_id,

  //来自执行阶段的暂停请求
    input wire                  stallreq_from_ex,

    output reg[`RegBus]         new_pc,
    output reg                  flush,
    output reg[5:0]             stall

);


    always @ (*) begin
        if(rst == `RstEnable) begin
            stall <= 6'b000000;
            flush <= 1'b0;
            new_pc <= `ZeroWord;
        end else if(excepttype_i != `ZeroWord) begin
          flush <= 1'b1;
          stall <= 6'b000000;
            case (excepttype_i)
                32'h00000001:       begin   //interrupt
                    new_pc <= 32'h00000020;
                end
                32'h00000008:       begin   //syscall
                    new_pc <= 32'h00400004;
                end
                32'h0000000a:       begin   //inst_invalid
                    new_pc <= 32'h00400004;
                end
```

```
                32'h0000000d:       begin   //trap
                    new_pc <= 32'h00400004;
                end
                32'h0000000c:       begin   //ov
                    new_pc <= 32'h00400004;
                end
                32'h0000000e:       begin   //eret
                    new_pc <= cp0_epc_i;
                end
                default : begin
                end
            endcase
        end else if(stallreq_from_ex == `Stop) begin
            stall <= 6'b001111;
            flush <= 1'b0;
        end else if(stallreq_from_id == `Stop) begin
            stall <= 6'b000111;
            flush <= 1'b0;
        end else begin
            stall <= 6'b000000;
            flush <= 1'b0;
            new_pc <= `ZeroWord;
        end     //if
    end       //always


endmodule
```

3.3 七段数码管驱动模块

七段数码管是 Nexys4 开发板上用于显示数值的一个部件，我们需要将指定的 0x10010000 内存单元最终结果显示到七段数码管上，以验证 CPU 改造成功。因此需要添加七段数码管的驱动模块。

沿用在 MIPS54 条指令动态流水线项目中的七段数码管显示模块即可。

代码如下：

```
`timescale 1ns / 1ns

module Seg7x16(
    input Clk,
    input Reset,
    input Cs,
    input [31:0] I_Data,
    output [7:0] O_Seg,
    output [7:0] O_Sel
    );

    reg [14:0] Cnt;
    always @ (posedge Clk, posedge Reset)
      if (Reset)
        Cnt <= 0;
      else
        Cnt <= Cnt + 1'B1;
```

```
wire Seg7_Clk = Cnt[14];

 reg [2:0] Seg7_Addr;

 always @ (posedge Seg7_Clk, posedge Reset)
   if(Reset)
      Seg7_Addr <= 0;
    else
      Seg7_Addr <= Seg7_Addr + 1'B1;

 reg [7:0] O_Sel_R;

 always @ (*)
   case(Seg7_Addr)
      7 : O_Sel_R = 8'B01111111;
      6 : O_Sel_R = 8'B10111111;
      5 : O_Sel_R = 8'B11011111;
      4 : O_Sel_R = 8'B11101111;
      3 : O_Sel_R = 8'B11110111;
      2 : O_Sel_R = 8'B11111011;
      1 : O_Sel_R = 8'B11111101;
      0 : O_Sel_R = 8'B11111110;
    endcase

 reg [31:0] I_Data_Store;
 always @ (posedge Clk, posedge Reset)
   if(Reset)
      I_Data_Store <= 0;
    else if(Cs)
      I_Data_Store <= I_Data;

 reg [7:0] Seg_Data_R;
 always @ (*)
   case(Seg7_Addr)
      0 : Seg_Data_R = I_Data_Store[3:0];
      1 : Seg_Data_R = I_Data_Store[7:4];
      2 : Seg_Data_R = I_Data_Store[11:8];
      3 : Seg_Data_R = I_Data_Store[15:12];
      4 : Seg_Data_R = I_Data_Store[19:16];
      5 : Seg_Data_R = I_Data_Store[23:20];
      6 : Seg_Data_R = I_Data_Store[27:24];
      7 : Seg_Data_R = I_Data_Store[31:28];
    endcase

 reg [7:0] O_Seg_R;
 always @ (posedge Clk, posedge Reset)
   if(Reset)
      O_Seg_R <= 8'Hff;
    else
      case(Seg_Data_R)
        4'H0 : O_Seg_R <= 8'HC0;
      4'H1 : O_Seg_R <= 8'HF9;
      4'H2 : O_Seg_R <= 8'HA4;
      4'H3 : O_Seg_R <= 8'HB0;
```

```
            4'H4 : O_Seg_R <= 8'H99;
            4'H5 : O_Seg_R <= 8'H92;
            4'H6 : O_Seg_R <= 8'H82;
            4'H7 : O_Seg_R <= 8'HF8;
            4'H8 : O_Seg_R <= 8'H80;
            4'H9 : O_Seg_R <= 8'H90;
            4'HA : O_Seg_R <= 8'H88;
            4'HB : O_Seg_R <= 8'H83;
            4'HC : O_Seg_R <= 8'HC6;
            4'HD : O_Seg_R <= 8'HA1;
            4'HE : O_Seg_R <= 8'H86;
            4'HF : O_Seg_R <= 8'H8E;
            endcase

      assign O_Sel = O_Sel_R;
      assign O_Seg = O_Seg_R;

endmodule
```

四、约束文件修改说明

对于约束文件的修改较为容易。根据测试文件的需求，我们需要从 Nexys 4 开发板上得到的选择信号，以及时钟和复位信号，并且将指定的 0x10010000 内存单元最终结果输出到七段数码管上，因此需要驱动数码管的引脚。

具体对应关系为 clk 时钟信号连接 E3 引脚，rst 复位信号连接 N17 按钮引脚，O_Sel, O_Seg 选择信号连接相应的数码管引脚。

根据上述需求，编写约束文件如下：

```
set_property PACKAGE_PIN E3 [get_ports clk]
set_property PACKAGE_PIN N17 [get_ports rst]

set_property PACKAGE_PIN T10 [get_ports {O_Seg[0]}]
set_property PACKAGE_PIN R10 [get_ports {O_Seg[1]}]
set_property PACKAGE_PIN K16 [get_ports {O_Seg[2]}]
set_property PACKAGE_PIN K13 [get_ports {O_Seg[3]}]
set_property PACKAGE_PIN P15 [get_ports {O_Seg[4]}]
set_property PACKAGE_PIN T11 [get_ports {O_Seg[5]}]
set_property PACKAGE_PIN L18 [get_ports {O_Seg[6]}]
set_property PACKAGE_PIN H15 [get_ports {O_Seg[7]}]

set_property PACKAGE_PIN J17 [get_ports {O_Sel[0]}]
set_property PACKAGE_PIN J18 [get_ports {O_Sel[1]}]
set_property PACKAGE_PIN T9 [get_ports {O_Sel[2]}]
set_property PACKAGE_PIN J14 [get_ports {O_Sel[3]}]
set_property PACKAGE_PIN P14 [get_ports {O_Sel[4]}]
set_property PACKAGE_PIN T14 [get_ports {O_Sel[5]}]
set_property PACKAGE_PIN K2 [get_ports {O_Sel[6]}]
set_property PACKAGE_PIN U13 [get_ports {O_Sel[7]}]

set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[7]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Sel[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {O_Seg[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports rst]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

create_clock -period 200.000 -name clk_pin -waveform {0.000 100.000}
[get_ports clk]
set_input_delay -clock [get_clocks *] 1.000 [get_ports rst]
set_output_delay -clock [get_clocks *] 0.000 [get_ports -filter { NAME =~
"*" && DIRECTION == "OUT" }]
```

五、仿真分析

5.1 仿真模块

使用如下文件进行仿真：

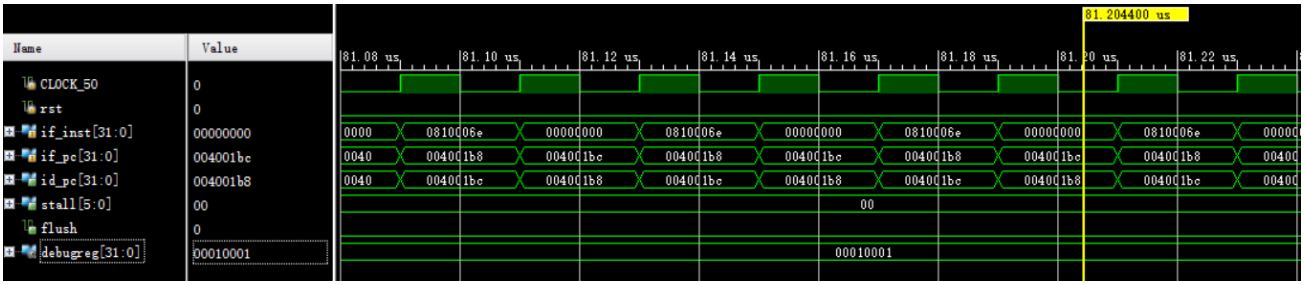
```
`include "defines.v"
`timescale 1ns / 1ps

module openmips_min_sopc_tb();
  reg     CLOCK_50;
  reg     rst;

  initial begin
    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
  end

  initial begin
    rst = `RstEnable;
    #195 rst= `RstDisable;
  end

  openmips_min_sopc openmips_min_sopc0(
    .clk(CLOCK_50),
    .rst(rst)
  );
```

```
endmodule
```

5.2 仿真分析

5.2.1 仿真分析一

仿真分析一展现了流水线能够正常流动，我们可以看到如下的仿真波形图：



可以看到 cpu_5_valid 信号一直保持有效状态，IF 段取指正常，流水线各段的 PC 值也表明了流水线能够正常流动。

5.2.2 仿真分析二

仿真分析二对所使用的 debug 寄存器的值进行分析。如下采用的测试程序由 11 个测试函数构成，测试内容覆盖了 89 条 MIPS 指令、cp0 以及时钟中断，具体的对应大端 CPU 的测试程序请参考附录文件 mips_89_mars_board_switch_student_b.s。

【要求】

1. CPU 采用大端模式。

2. 中断例程起始地址为 0x00400004。

3. 数码管显示地址为 0x10010000 的 DMEM 单元。

【测试结果说明】

1. 正确结果：数码管低半字显示 0x0001，高半字随着时钟中断而计数。

2. 错误结果：数码管显示-1～-10 的错误码，或者数码管低半字显示 0x0001 但高半字不计数。

【测试函数说明】

****

测试函数 1：（上学期 54 条 MIPS 指令流水线标准测试程序）

测试指令：

addi,addiu,andi,ori,slti,sltiu,lui,xori,and,beq,bne,j,jal,jr,lw,sw,mul,sll,sub

错误码：-1

****

测试函数 2：

测试指令： mfhi,mflo,div,divu

错误码：-2

****

测试函数 3：

测试指令：

sb,lb,lbu,sh,lh,lhu,mthi,mtlo,mfhi,mflo,sltu,sllv,srav,srlv,sra,srl,subu,add,addu,slt,clz,clo

错误码：-3

****

测试函数 4：

测试指令： jalr

错误码：-4

****

测试函数 5：

测试指令：mult,multu,madd,maddu,msub,msubu

错误码：-5

****

测试函数 6：

测试指令： movn,movz

错误码：-6

****

测试函数 7：

测试指令： bgez,bgtz,blez,bltz,bltzal,bgezal

错误码：-7

****

测试函数 8：

测试指令： lwl,lwr,swl,swr

错误码：-8

****

测试函数 9：

测试指令： ll,sc

错误码：-9

****

测试函数 10：

测试指令： cp0 相关指令，包括自陷指令、syscall、break、ret、mfc0、mtc0

错误码：-10

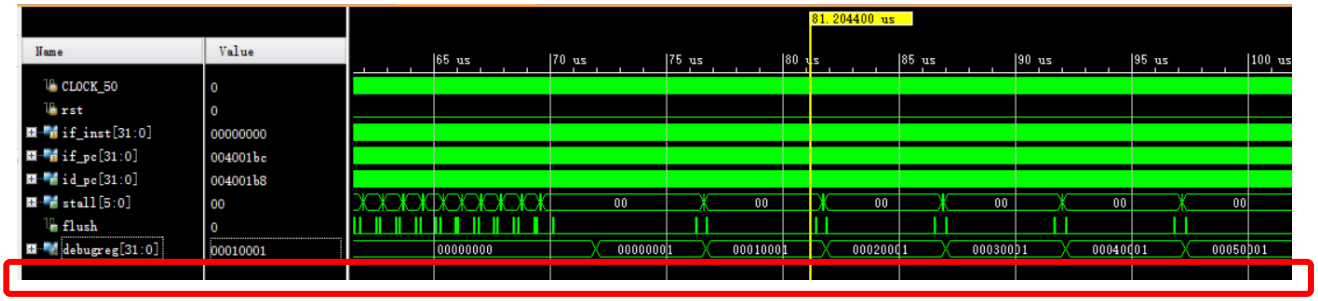
****

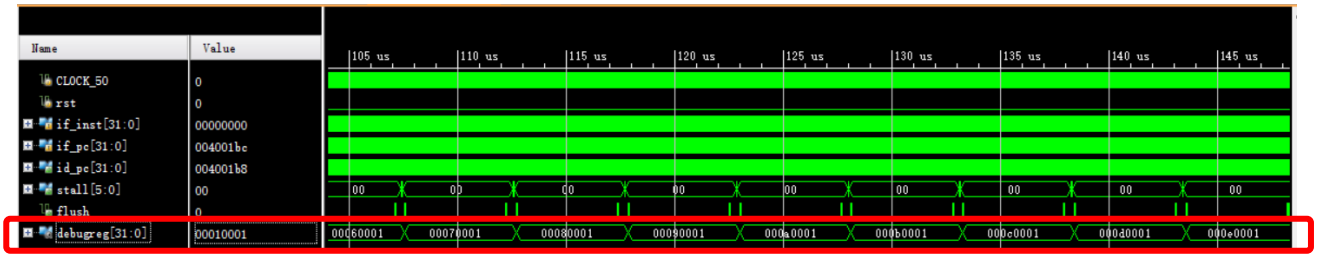
测试函数 11：

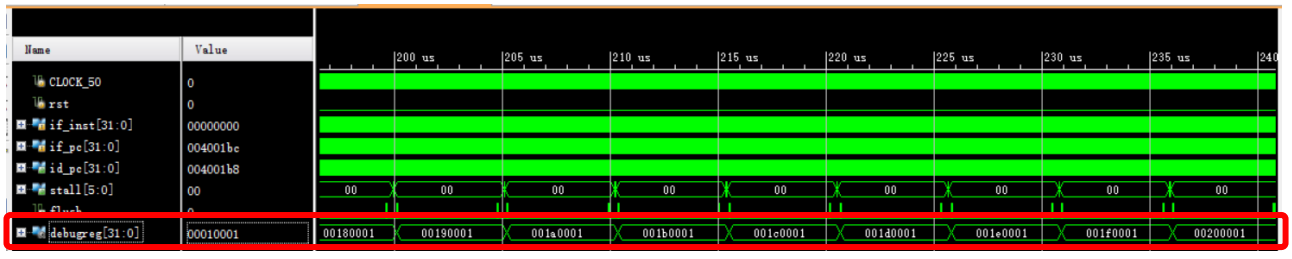
测试功能：时钟中断

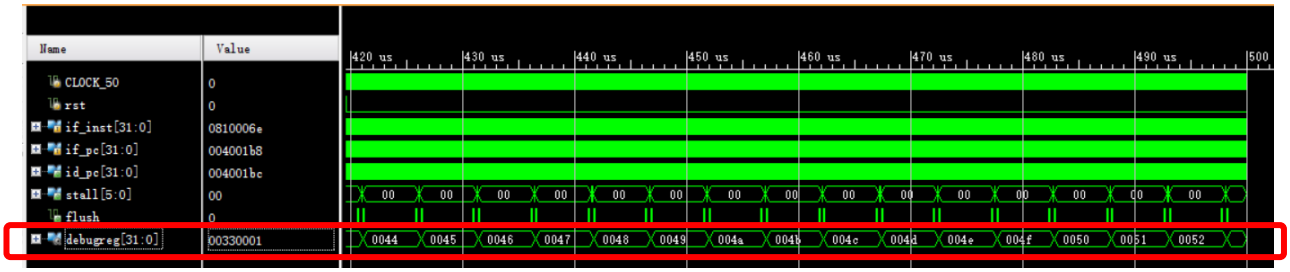
****

实际的仿真分析波形图如下图所示：









这里的 debug 寄存器的赋值部分如下图所示：

```
assign debugreg = {data_mem3[17'b0_0100_0000_0000_0000],
                   data_mem2[17'b0_0100_0000_0000_0000],
                   data_mem1[17'b0_0100_0000_0000_0000],
                   data_mem0[17'b0_0100_0000_0000_0000]};
```

波形图中展示了所使用的 debug 寄存器值的变化，与程序执行结果是一致的。数码管低半字显示 0x0001，高半字随着时钟中断而计数。

六、下板验证

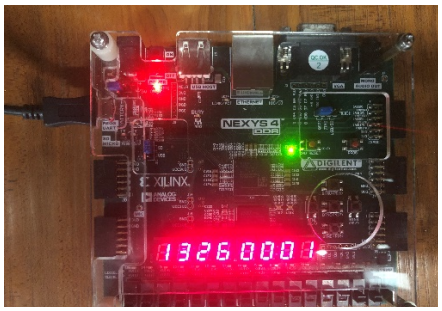
6.1 下板测试过程

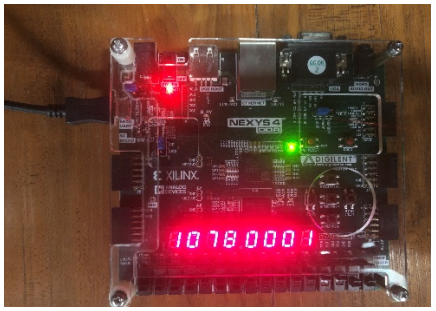
- 编写 16 进制的七段数码管驱动程序；
- 对时钟信号进行分频，保证能够在板子上看到清晰的结果；
- 将七段数码管驱动程序与 89 条指令动态流水线 CPU 在顶层模块做连接；
- 进行综合，编写管脚约束文件；
- 生成二进制 bit 流，下板。

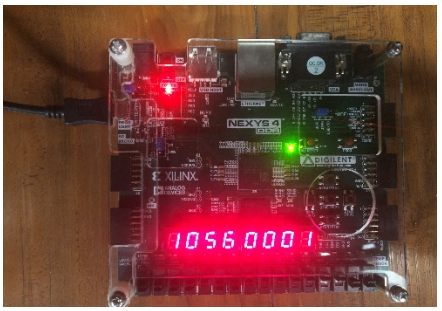
通过正确的文件约束，使板子上依次显示某一寄存器中的值，从而验证程序的正确性。下板进行测试，依次展示七段数码管选择的 debug 寄存器的数值。

6.2 结果检验展示

下面截取部分图片证明 89 条流水线的正确性。可以发现，数码管低半字显示 0x0001，高半字随着时钟中断而计数，符合测试正确结果预期。







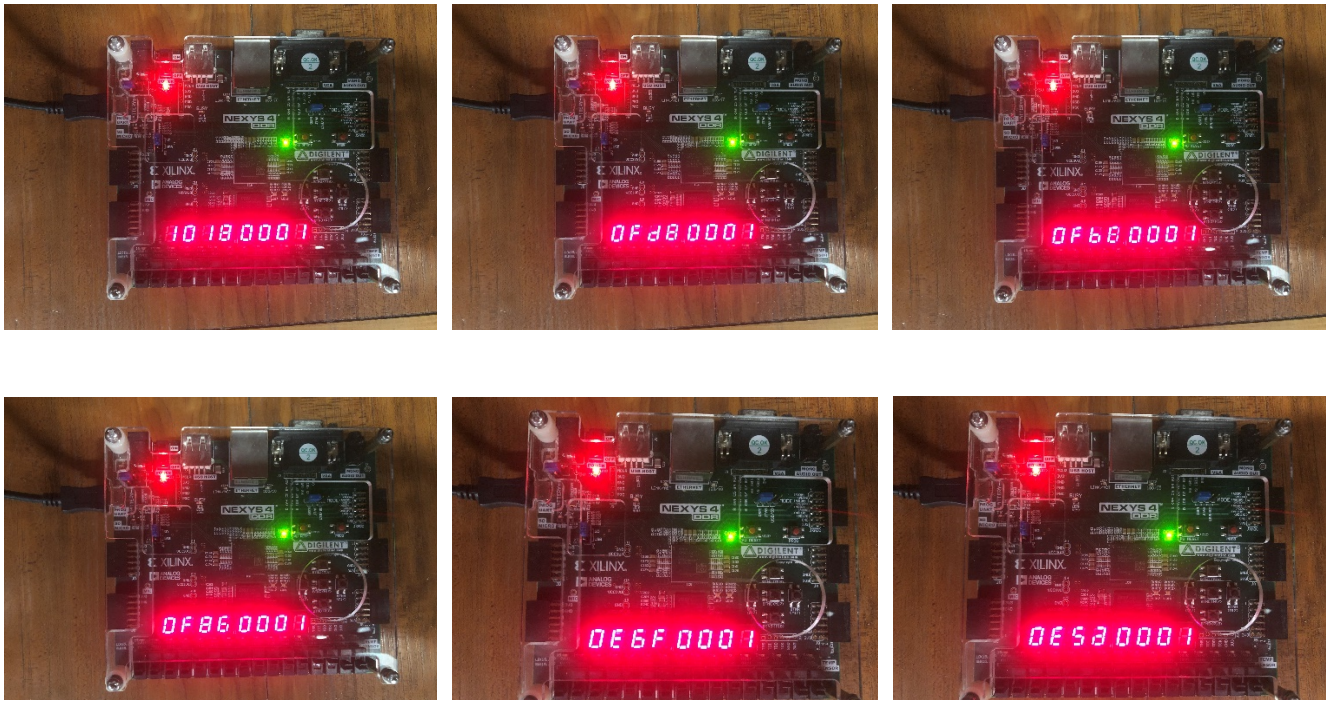


图 6-1 下板图片展示

七、实验体会

在本次的 MIPS 指令集流水线 CPU 实验中，我对 CPU 的运行有了更全面的认识。本次实验内容为更改现有 CPU 使之能够在我们的 Nexys 4 开发板上正常运行。实验本身需要的代码量并不是很大，但与自己动手写 CPU 不一样的是——改造现有 CPU 需要阅读别人的程序，完全搞明白别人的代码功能与代码逻辑以及各模块之间的连接关系，某种意义上来说这是更加困难的一件事情。因为这需要对 MIPS 指令集流水线 CPU 的原理有一个更好的理解。

经过实践，我认为可以慢慢理清思路，针对测试数据对各个模块逐个击破，在修改单个模块的同时，也要考虑这一模块对全局的影响，比如控制信号等是否需要修改。比如修改分支预测部分，在《自己动手写 CPU》的代码当中，连续两个分支指令，尤其是一条条件跳转指令，紧接着一条无条件跳转指令会出现一定的问题。例如 bgtz 在计算出是否应当分支的时候，其目标地址已经被 j 指令的目标地址给覆盖掉，表现为 bgtz 指令无效。因此我们可以修改分支预测部分或者更新测试文件。

另外，我们还需要注意一些 debug 上的小技巧。对于硬件程序，波形图仿真验证、生成比特流后直接下板验证都是行之有效的调试手段，合理配套使用能够极大地帮助自己寻找问题所在。在本次实验中，VAVIDO 的仿真波形功能就十分有效，想要具体了解模块内部运行细节，比如 Regfile、Hi、Lo 等寄存器数据的变化，可以在不影响模块功能的情况下，将一些内部信号也实时显示，这样便于直观观察，尽快找出问题所在。

然后，在硬件程序设计方面，也需要注意代码规范，宏定义的使用会方便程序的修改与移植。在《自己动手写 CPU》这本书中，使用了不少宏定义，这使得修改一些重要参数（如指令存储器的起始地址、指令的标号）等都十分方便。

最后的感想就是硬件设计不能一蹴而就，检验耗费的时间、精力都是软件测试不能比的，所以需要更多地耐心与真诚。本次实验也有一些不足与缺陷，因此后续的移植操作系统、编写可视化界面等实验，我会更加认真地去完成。

八、参考文献

[1] 秦国锋，王力生，陆有军，郭玉臣. 计算机系统结构实验指导，清华大学出版社,2019.

[2] 张晨曦，王志英等. 计算机体系结构，高等教育出版社,2014.

[3] 雷思磊. 自己动手写 CPU，电子工业出版社,2014.