《计算机组成原理》实验报告

年级、专业、班级	2019 级计算机科学与技术(卓越)02 班	姓名	李燕琴
实验题目	实验二处理器译码实验		
实验时间	2021 年 05 月 20 日	实验地点	DS1410
实验成绩	优秀/良好/中等	实验性质	□验证性 ☑设计性 □综合性

教师评价：

□算法/实验过程正确； □源程序/实验内容提交； □程序结构/实验步骤合理;

□实验结果正确； □语法、语义正确； □报告规范;

其他:

评价教师: 钟将

实验目的

(1)掌握单周期 CPU 控制器的工作原理及其设计方法。

(2)掌握单周期 CPU 各个控制信号的作用和生成过程。

(3)掌握单周期 CPU 执行指令的过程。

(4)掌握取指、译码阶段数据通路执行过程。

报告完成时间: 2021 年 5 月 21 日

1 实验内容

1. PC。D 触发器结构，用于储存 PC(一个周期)。需实现 2 个输入，分别为 *clk, rst*, 分别连接时钟和复位信号；需实现 2 个输出，分别为 *pc, inst_ce*, 分别连接指令存储器的 *addra, ena* 端口。其中 *addra* 位数依据 coe 文件中指令数定义；
2. 加法器。用于计算下一条指令地址，需实现 2 个输入，1 个输出，输入值分别为当前指令地址 *PC*、*32'h4*；
3. Controller。其中包含两部分：
   (a). main_decoder。负责判断指令类型，并生成相应的控制信号。需实现 1 个输入，为指令 inst 的高 6 位 *op*，输出分为 2 部分，控制信号有多个，可作为多个输出，也作为一个多位输出，具体参照参考指导书进行设计；*aluop*，传输至 alu_decoder，使 alu_decoder 配合 *inst* 低 6 位 *funct*，进行 ALU 模块控制信号的译码。
   (b). alu_decoder。负责 ALU 模块控制信号的译码。需实现 2 个输入，1 个输出，输入分别为 *funct, aluop*；输出位 *alucontrol* 信号。
   (c). 除上述两个组件，需设计 controller 文件调用两个 decoder，对应实现 *op*,*funct* 输入信号，并传入调用模块；对应实现控制信号及 *alucontrol*，并连接至调用模块相应端口。
4. 指令存储器。使用 Block Memory Generator IP 构造。(参考指导书)
   注意：Basic 中 Generate address interface with 32 bits 选项不选中；PortA Options 中 Enable Port Type 选择为 Use ENA Pin
5. 时钟分频器。将板载 100Mhz 频率降低为 1hz，连接 PC、指令存储器时钟信号 clk。(参考数字逻辑实验)
   注意：Xilinx Clocking Wizard IP 可分的最低频率为 4.687Mhz，因而只能使用自实现分频模块进行分频

2 实验设计

2.1 控制器 (Controller)

2.1.1 功能描述

输入 32 位指令，通过译码原理，判断指令类型，并输出相应的控制信号，完成指令译码。

2.1.2 接口定义

见表1 Controller 接口定义。

表 1: Controller 接口定义

信号名	方向	位宽	功能描述
instr	Input	32-bit	32 位的指令
zero	Input	1-bit	branch 涉及的两个数是否相等
regwrite	Output	1-bit	是否需要写寄存器堆
regdst	Output	1-bit	写入寄存器堆的地址是 rt 还是 rd,0 为 rt,1 为 rd
alusrc	Output	1-bit	送入 ALU B 端口的值是立即数的 32 位扩展 / 寄存器堆读取的值
branch	Output	1-bit	是否为 branch 指令
memWrite	Output	1-bit	是否需要写数据存储器
memtoReg	Output	1-bit	回写的数据来自于 ALU 计算的结果/存储器读取的数据
jump	Output	1-bit	是否为 jump 指令
pcsrc	Output	1-bit	下一个 PC 值是 PC+4 / 跳转的新地址
alucontrol	Output	3-bit	ALU 控制信号，代表不同的运算类型

2.1.3 逻辑控制

controller 调用 main_decoder(负责判断指令类型,并生成相应的控制信号)和 alu_decoder(负责 ALU 模块控制信号的译码)两个组合逻辑模块,基于**译码器**原理,按照指导书给定的指令和控制信号对照表,完成指令译码功能。

具体 RTL 原理图见图1。

2.2 存储器 (Block Memory)

2.2.1 类型选择

如图2。

2.2.2 参数设置

如图3。

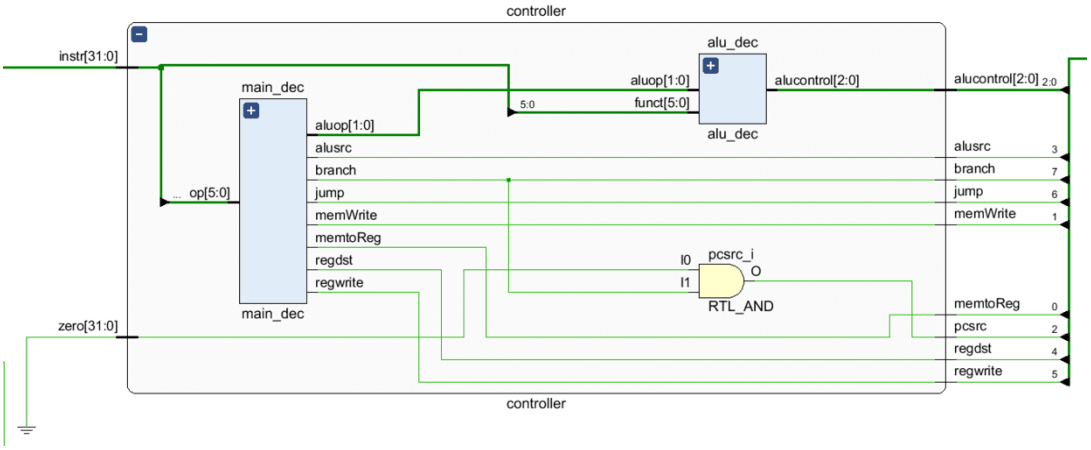

图 1: controller RTL 原理图

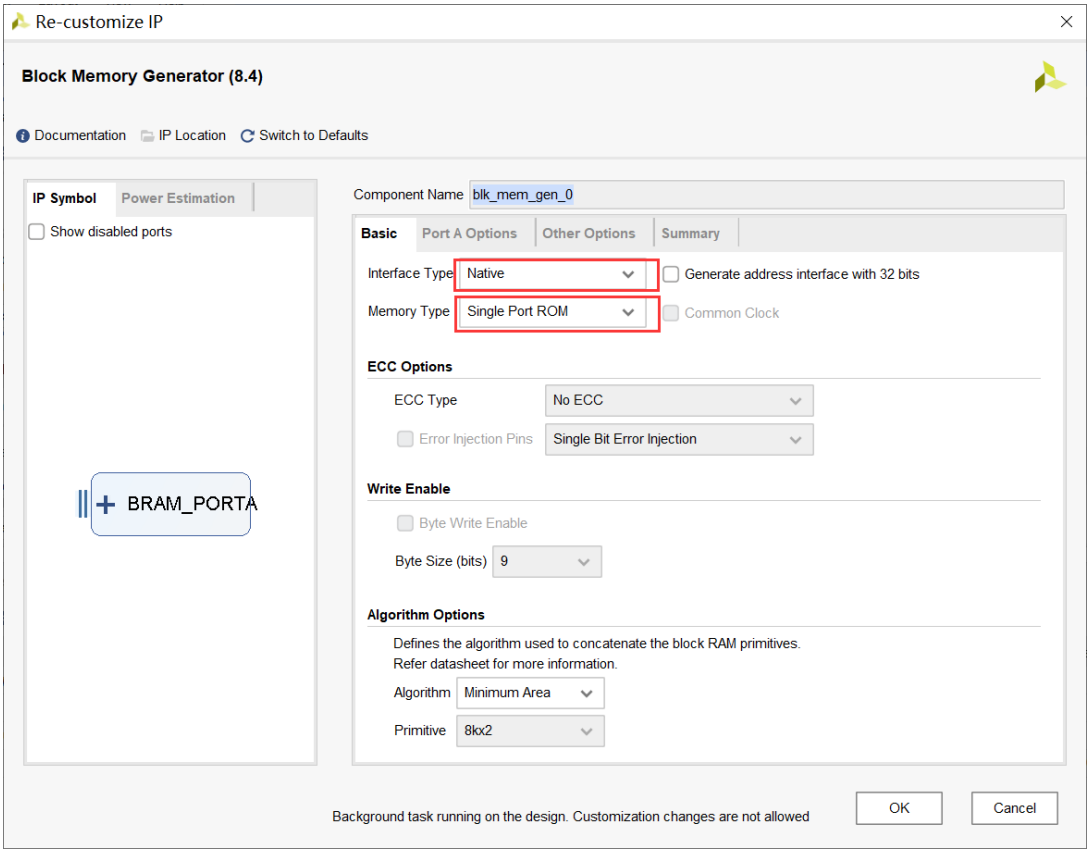

图 2: ROM 基础设置

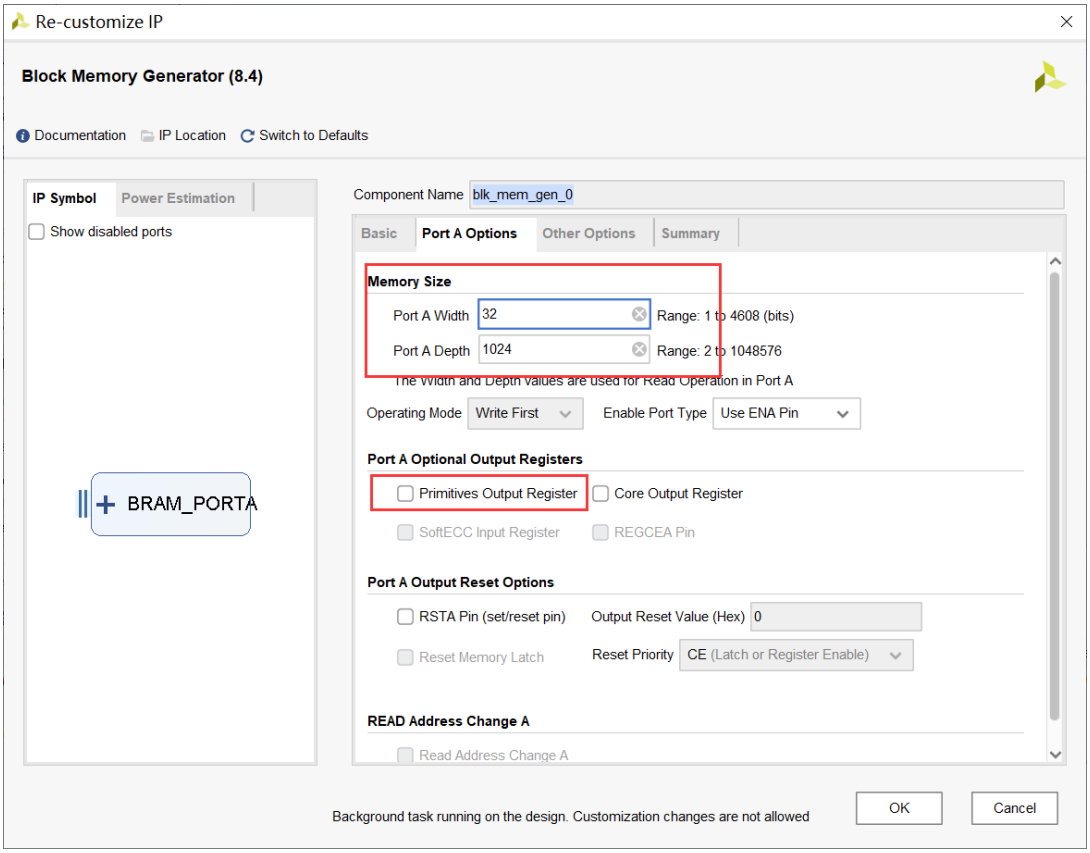

图 3: ROM 端口设置

2.2.3 文件导入

如图4。

3 实验过程记录

3.1 主要工作

实验中，基于单周期 CPU 控制器的工作原理和设计方法，设计了 PC 模块、PC 自增模块、Controller 模块，并通过 Block Memory Generator IP 设计构造了指令存储器 IM，完成了单周期取指、译码阶段的数据通路执行过程；并设计仿真文件，导入本地.ceo 文件，进行行为仿真；设计分频器，以将 N4 板内置 100MHz 时钟降低为 1Hz，定义引脚文件并上板验证控制器译码功能。

3.2 问题记录

3.2.1 问题一：分频器时钟延迟问题

问题描述：自定义分频器，基于计数器原理实现，但是仿真中发现，自定义 clk 会比预期多一个纳秒，即自定义 clk 会延长 1ns，主要代码如下：

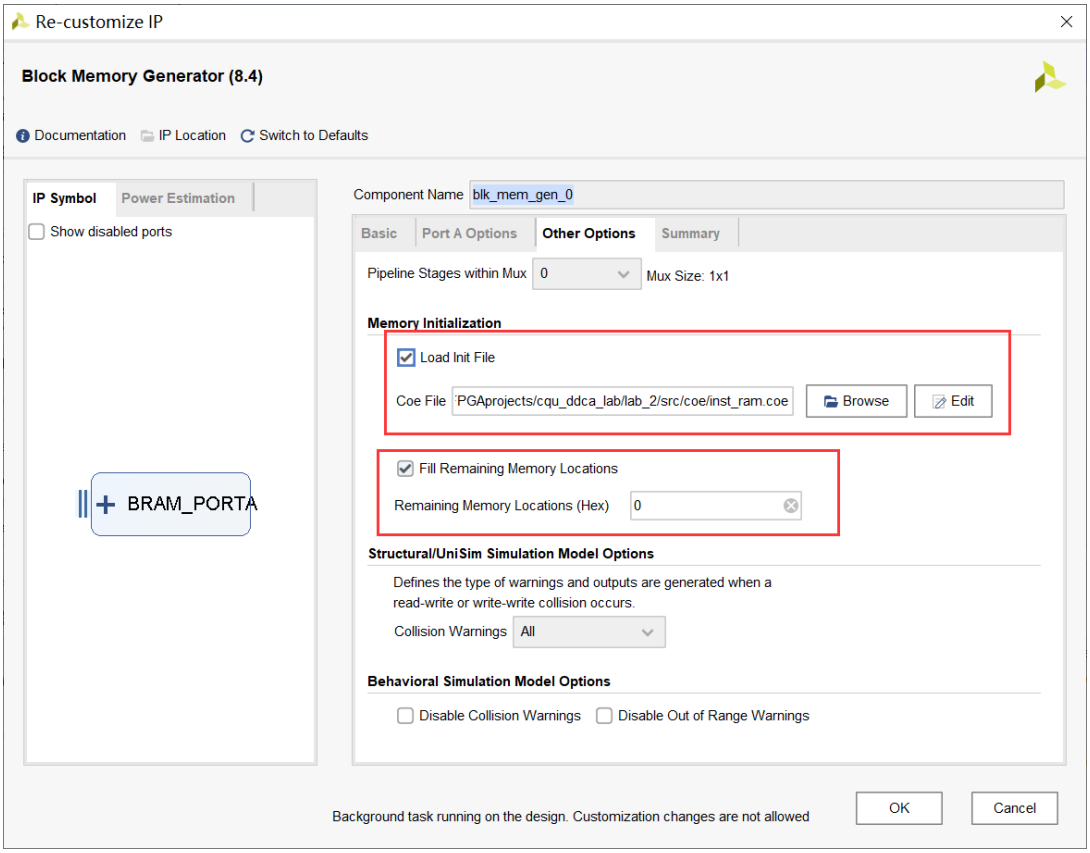

图 4: ROM 其他设置

```
always@(posedge clk) begin  // 方法一
    if(cnt == half_rate) begin
        inlineclk <= ~inlineclk;
        cnt <= 1'b0;
    end
    else begin cnt <= cnt + 1'b1; end
end
```

问题解决:每个时钟周期 cnt 都应该执行 +1 操作,才能保证自定义 clk 一直在预期时间点进行翻转,故当 cnt==half_rate 时,'inlineclk'进行翻转,'cnt'应该重置为 1,即可解决上述问题。即代码更改如下:

```
always@(posedge clk) begin  // 方法一
    if(cnt == half_rate) begin
        inlineclk <= ~inlineclk;
        cnt <= 1'b1;
    end
    else begin cnt <= cnt + 1'b1; end
end
```

3.3 问题二:PC 自增 4 还是 1？

问题描述:当 PC 自增 4 且 IM 以 PC[9:0] 作为地址进行取指时,每次都会跳过 3 条指令,取到第 4 条指令。

问题解决: 为保证 CPU 设计的统一性，这里仍然执行 PC 自增 4 的操作，但是在取指令时，以 PC[11:2] 进行取指即可。

3.4 问题三:行为仿真中的组合环路问题

问题描述:当仿真、综合、实现网表一切正常时,生成比特流时,报了如下错误:

```
[DRC LUTLP-1] Combinatorial Loop Alert: 1 LUT cells form a combinatorial loop.
    This can create a race condition. Timing analysis may not be accurate. The
    preferred resolution is to modify the design to remove combinatorial logic
    loops. If the loop is known and understood, this DRC can be bypassed by
    acknowledging the condition and setting the following XDC constraint on any
    one of the nets in the loop: 'set_property ALLOW_COMBINATORIAL_LOOPS TRUE [
    get_nets <myHier/myNet>]'. One net in the loop is fdivider/cnt[0]. Please
    evaluate your design. The cells in the loop are: fdivider/cnt0_carry_i_6.
```

问题解决:根据错误提示,为组合逻辑环路问题,检查 'fdivider.v'代码,最开始分频器设计的代码如下:

```
always@(clk) begin  // 方法一
    if(cnt == half_rate) begin
        inlineclk = ~inlineclk;
        cnt = 1'b1;
    end
    else begin cnt = cnt + 1'b1; end
end
```

可以看到,上述为组合逻辑,且 inlineclk 和 cnt 均存在组合逻辑环路问题,即不经过任何时序逻辑(寄存器等),而直接将组合逻辑电路的输出信号反馈到其输入节点而形成的环路。此为 FPGA 设计的大忌(详情可见https://zhuanlan.zhihu.com/p/32660579),故需要转为时序逻辑以消除组合逻辑环路冲突问题,修正后正常生成比特流,修正代码如下:

```
always@(posedge clk) begin  // 方法一
    if(cnt == half_rate) begin
        inlineclk <= ~inlineclk;
        cnt <= 1'b1;
    end
    else begin cnt <= cnt + 1'b1;end
end
```

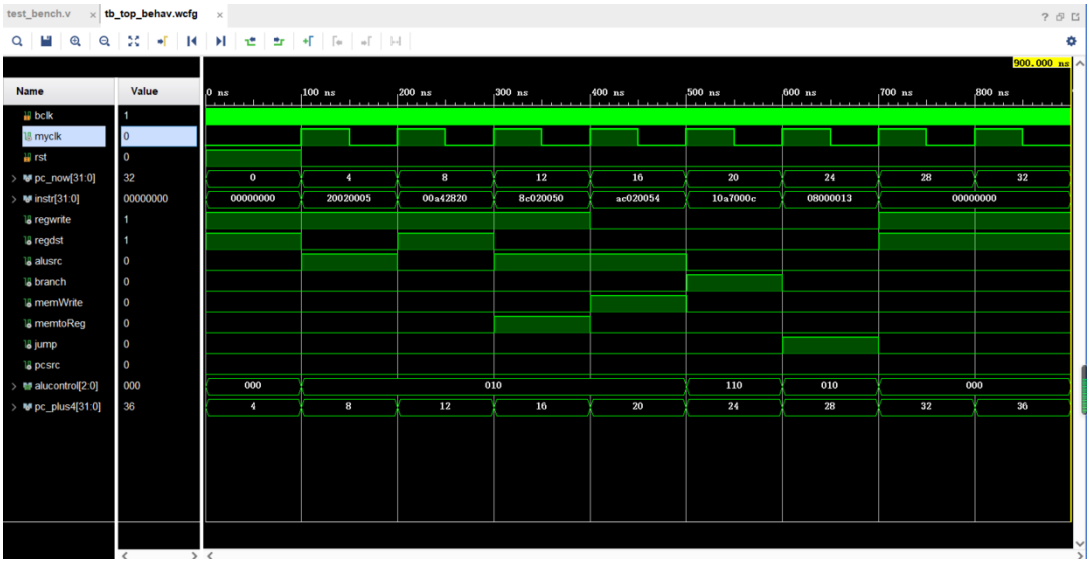

图 5: 实验结果仿真

```
Vivado Simulator 2019.1
Time resolution is 1 ps
 Block Memory Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator module test_bench.top.instr_rom.inst.native_mem_module.blk_mem_gen_v8_4_3_inst is using a behavioral model for simulation which will not precisely model memory co
instruction: 32'h00000000, memtoReg: 0, memWrite: 0, pcsrc: 0, alusrc: 0, regdst: 1, regwrite: 1, jump: 0, branch: 0, alucontrol: 000
instruction: 32'h20020005, memtoReg: 0, memWrite: 0, pcsrc: 0, alusrc: 1, regdst: 0, regwrite: 1, jump: 0, branch: 0, alucontrol: 010
instruction: 32'h00a42820, memtoReg: 0, memWrite: 0, pcsrc: 0, alusrc: 0, regdst: 1, regwrite: 1, jump: 0, branch: 0, alucontrol: 010
instruction: 32'h8c020050, memtoReg: 1, memWrite: 0, pcsrc: 0, alusrc: 1, regdst: 0, regwrite: 1, jump: 0, branch: 0, alucontrol: 010
instruction: 32'hac020054, memtoReg: 0, memWrite: 1, pcsrc: 0, alusrc: 1, regdst: 0, regwrite: 0, jump: 0, branch: 0, alucontrol: 010
instruction: 32'h10a7000c, memtoReg: 0, memWrite: 0, pcsrc: 0, alusrc: 0, regdst: 0, regwrite: 0, jump: 0, branch: 1, alucontrol: 110
instruction: 32'h08000013, memtoReg: 0, memWrite: 0, pcsrc: 0, alusrc: 0, regdst: 0, regwrite: 0, jump: 1, branch: 0, alucontrol: 010
instruction: 32'h00000000, memtoReg: 0, memWrite: 0, pcsrc: 0, alusrc: 0, regdst: 1, regwrite: 1, jump: 0, branch: 0, alucontrol: 000
$stop called at time : 900 ns : File "D:/FPGAprojects/cqu_ddca_lab/lab_2/src/sim/test_bench.v" Line 17
relaunch_sim: Time (s): cpu = 00:00:04 ; elapsed = 00:00:28 . Memory (MB): peak = 2432.773 ; gain = 0.000
```

图 6: 控制台输出结果

4 实验结果及分析

4.1 仿真图

实验结果仿真如图5。仿真结果符合预期。其中可以看到，取到的指令 instr 是用上一个时钟周期的 pc 作为取指地址得到的。

4.2 控制台输出

控制台输出如图6。输出符合预期。

4.3 上板验证

上板验证结果符合预期。如图7

A Controller 代码

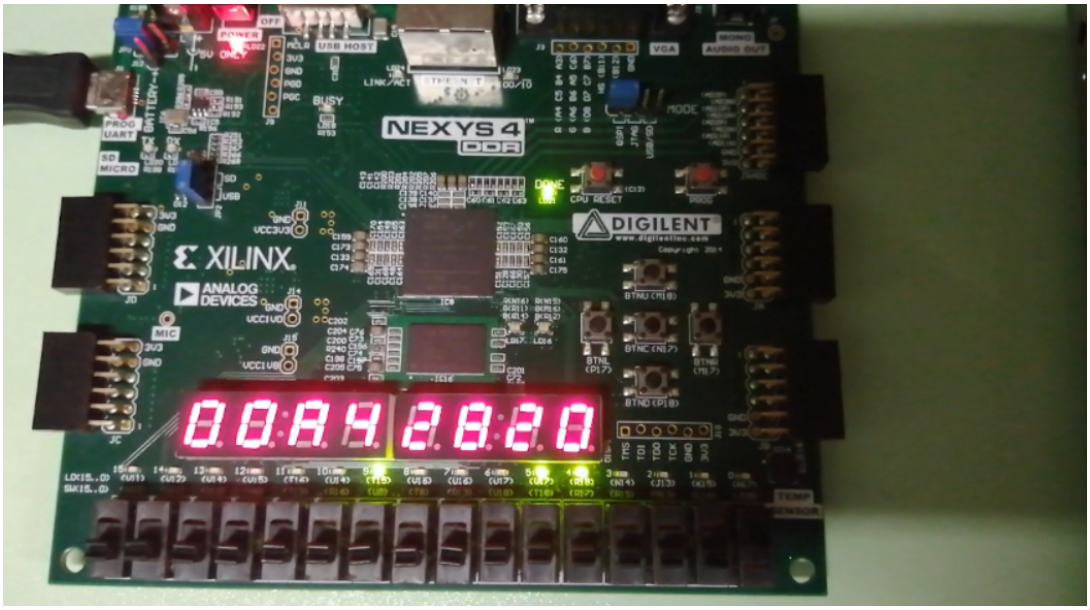

图 7: 上板验证结果

```
‘timescale 1ns/1ps
module controller (
    input wire [31:0]instr,zero,
    output wire regwrite,regdst,alusrc,branch,memWrite,memtoReg,jump,pcsrc,
    output wire [2:0]alucontrol
);
    wire [1:0]aluop;
    assign pcsrc = zero & branch;
    main_dec main_dec(
        .op(instr[31:26]),
        .regwrite(regwrite),
        .regdst(regdst),
        .alusrc(alusrc),
        .branch(branch),
        .memWrite(memWrite),
        .memtoReg(memtoReg),
        .jump(jump),
        .aluop(aluop)
    );

    alu_dec alu_dec(
        .aluop(aluop),
        .funct(instr[5:0]),
        .alucontrol(alucontrol)
    );

endmodule

module main_dec (
    input wire[5:0]op,
```

```
    output wire regwrite,regdst,alusrc,branch,memWrite,memtoReg,jump,
    output wire [1:0]aluop
);
    wire [8:0]temp;
    assign  temp =  (op==6'b000000) ? 9'b110000100: //最后一位是jump
                    (op==6'b100011) ? 9'b101001000:
                    (op==6'b101011) ? 9'b001010000:
                    (op==6'b000100) ? 9'b000100010:
                    (op==6'b001000) ? 9'b101000000:
                    (op==6'b000010) ? 9'b000000001:
                    9'b000000000; // 注意：这里的X信号，全部视为0
    assign {regwrite,regdst,alusrc,branch,memWrite,memtoReg,aluop,jump} = temp;
endmodule

module alu_dec (
    input wire[1:0]aluop,
    input wire[5:0]funct,
    output wire[2:0]alucontrol
);
    assign alucontrol = (aluop==2'b00) ? 3'b010: // add
                        (aluop==2'b01) ? 3'b110: // sub
                        (aluop==2'b10) ?
                            (funct==6'b100000) ? 3'b010:
                            (funct==6'b100010) ? 3'b110:
                            (funct==6'b100100) ? 3'b000:
                            (funct==6'b100101) ? 3'b001:
                            (funct==6'b101010) ? 3'b111:
                            3'b000:
                        3'b000; // default
endmodule
```