计算机组成原理实验 实验报告

实验题目：Lab3 单周期CPU

学生姓名：王章瀚

学生学号：PB18111697

完成日期：2020 年 5 月 13 日

计算机实验教学中心制

2019年09月

1 实验题目

Lab3 单周期CPU

2 实验目的

1. 理解计算机硬件的基本组成、结构和工作原理；
2. 掌握数字系统的设计和调试方法；
3. 熟练掌握数据通路和控制器的设计和描述方法。

3 实验平台

Vivado

4 实验过程

实验过程主要分为单周期CPU的设计和DBU的设计. 下面分块讲解二者.

4.1 单周期CPU

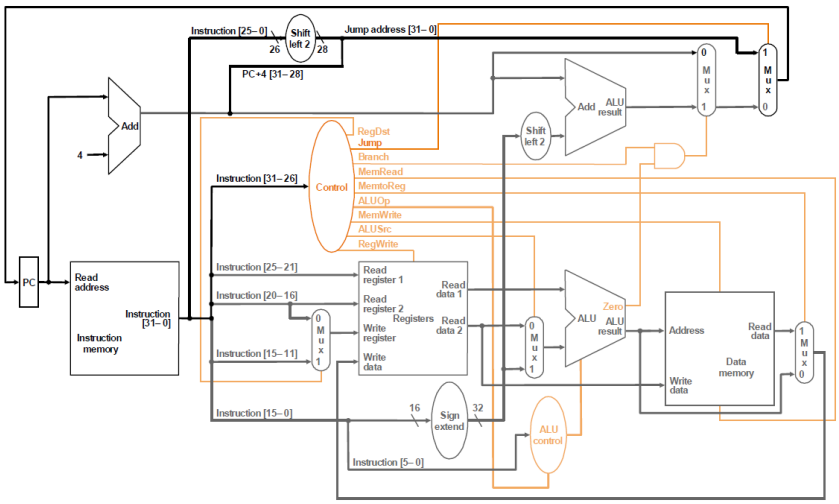
4.1.1 基本过程

单周期CPU的部分中, ALU, 寄存器堆, 数据存储器, 指令存储器等结构单元都已经在前面的实验中完成, 这里可以直接调用, 因此不再赘述这几个元件的相关实现.

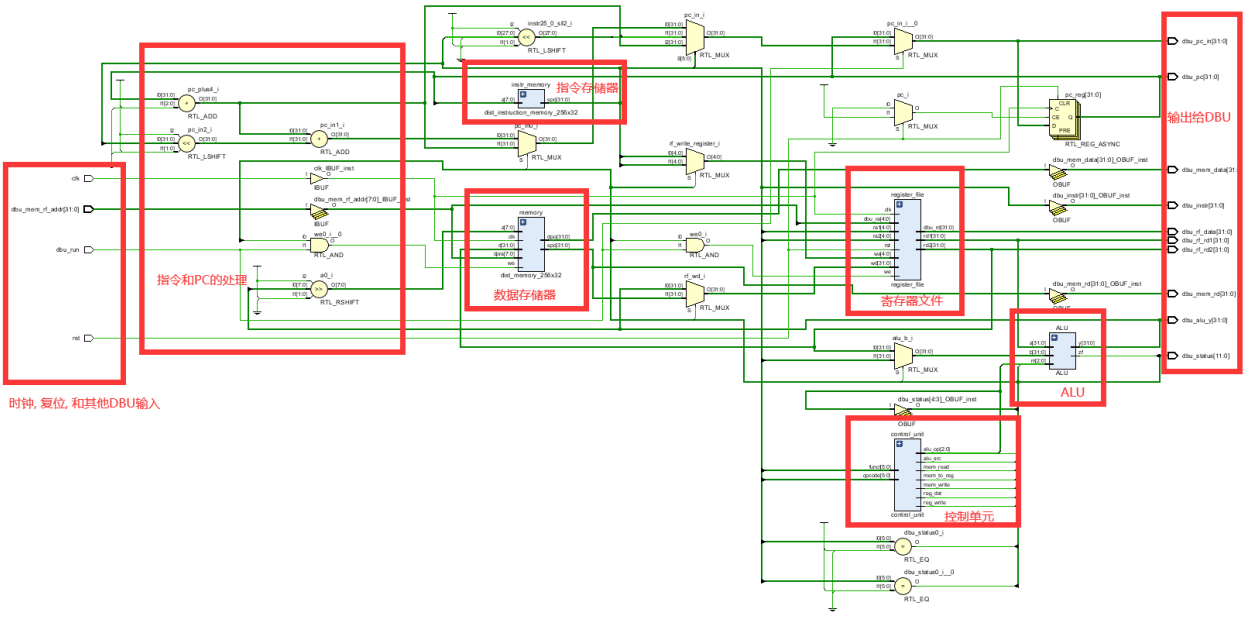
这里需要讲解的部分有: 数据通路, 状态转换, 控制单元及其他代码等.

4.1.2 数据通路

这里数据通路基本上按照老师的来完成的. 下图是老师给定的数据通路.

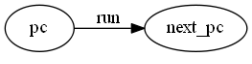


而我对我完成的代码进行RTL分析, 能够得出下面的数据通路. 其中各个模块的功能已经在图中具体标识了.



4.1.3 状态转换

单周期的CPU状态机比较简单, 某种程度上可以认为状态只由PC决定. 因此它的状态转换图也比较简单. 如下.



4.1.4 代码讲解

1. 数据通路代码

   这里只展示CPU内部的数据通路, 传出去给DBU的数据通路就是对应传递即可, 没有什么特别的地方.

   而CPU内部数据通路按照老师给出的数据通路进行连接, 其中值得注意的是, 由于存储器的地址应该为字地址, 传入地址的时候需要进行左移两位的操作. 代码如下:

```
// 指令存储器
dist_instruction_memory_256x32 instr_memory(.a(pc[9:2]), .spo(instr));
// 指令存储器 - 寄存器文件
//// 选择写寄存器
assign rf_write_register = reg_dst == 1'b0 ? instr[20:16] : instr[15:11];
// 寄存器文件
register_file register_file(.clk(clk), .rst(rst),
                            .ra1(instr[25:21]), .rd1(rf_rd1),
                            .ra2(instr[20:16]), .rd2(rf_rd2),
                            .dbu_ra(dbu_mem_rf_addr[4:0]), .dbu_rd(dbu_rf_data),
                            .wa(rf_write_register), .we(reg_write & dbu_run), .wd(rf_wd));
// 寄存器文件 - ALU
//// 带符号扩展后十六位
assign instr_imm = {{16{instr[15]}}, instr[15:0]};
//// 选择alu_src
assign alu_b = (alu_src == 1'b0) ? rf_rd2 : instr_imm;
// ALU
ALU ALU(.m(alu_op), .a(rf_rd1), .b(alu_b), .zf(alu_zero), .y(alu_y));
// ALU - 数据存储器
assign mem_addr = alu_y;
// 数据存储器
dist_memory_256x32 memory(.a(mem_addr >> 2), .d(rf_rd2), .dpra(dbu_mem_rf_addr), .clk(clk), .
    we(mem_write & dbu_run), .spo(mem_rd), .dpo(dbu_mem_data));
assign rf_wd = mem_to_reg == 1'b0 ? alu_y : mem_rd;
// 控制单元
control_unit control_unit(.opcode(opcode), .funct(funct), .reg_dst(reg_dst),
                          .reg_write(reg_write), .mem_read(mem_read),.mem_to_reg(mem_to_reg),
                          .mem_write(mem_write), .alu_op(alu_op), .alu_src(alu_src));

```

2. PC状态更新

   这一部分的代码使得PC状态能够进行状态转移.

```
// PC的更新
wire [31:0] pc_plus4;
wire [27:0] instr25_0_sll2;
assign pc_plus4 = pc + 4;
assign instr25_0_sll2 = instr[25:0] << 2;
always @(*) begin
    if(dbu_run == 1'b0) begin
        pc_in = pc;
    end
    else begin
        // 针对不同跳转指令作不同的PC处理
        case(opcode)
            BEQ_op: pc_in = alu_zero == 1'b1 ? pc + 4 + (instr_imm<<2) : pc + 4;
            J_op: pc_in = {pc_plus4[31:28], instr25_0_sll2[27:0]};
            default: pc_in = pc + 4;
```

```
        endcase
    end
end
always @(posedge clk, posedge rst) begin
    if(rst) begin
        pc <= 32'h0000_0000;
    end
    else begin
        pc = pc_in;
    end
end

```

3. 控制单元

这部分是CPU的控制单元的代码. 它完成了对整个CPU各个地方的使能等信号的控制.

这里主要就是针对每个指令进行解析, 判断各个指令需要使能哪些信号, 失能哪些信号. 代码如下.

```
module control_unit
    #(
    parameter ADD_op = 6'b000000,
    parameter ADD_funct = 6'b100000,
    parameter ADDI_op = 6'b001000,
    parameter LW_op = 6'b100011,
    parameter SW_op = 6'b101011,
    parameter BEQ_op = 6'b000100,
    parameter J_op = 6'b000010,
    parameter ALU_ADD = 3'b000,
    parameter ALU_SUB = 3'b001,
    parameter ALU_AND = 3'b010,
    parameter ALU_OR = 3'b011,
    parameter ALU_XOR = 3'b100
    )
    (
    input [5:0] opcode,
    input [5:0] funct,
    output reg reg_dst, reg_write, mem_read, mem_to_reg, mem_write, alu_src,
    output reg [2:0] alu_op
    );

    // 控制单元
    always @(*) begin
        {reg_dst, reg_write, mem_read, mem_to_reg, mem_write, alu_op, alu_src} = 9'
    h0_0000_0000;
        case(opcode)
            ADD_op: begin
                case(funct)
                    ADD_funct: begin
                        reg_dst = 1'b1; reg_write = 1'b1; alu_op = ALU_ADD;
                    end
                    default: {reg_dst, reg_write, mem_read, mem_to_reg, mem_write, alu_op,
    alu_src} = 9'h0_0000_0000;
                endcase
            end
            ADDI_op: begin
                alu_src = 1'b1; reg_write = 1'b1; alu_op = ALU_ADD;
            end
            LW_op: begin
                alu_src = 1'b1;
```

```
40                  mem_to_reg = 1'b1;
41                  reg_write = 1'b1;
42                  mem_read = 1'b1;
43                  alu_op = ALU_ADD;
44              end
45              SW_op: begin
46                  alu_src = 1'b1;
47                  mem_write = 1'b1;
48                  alu_op = ALU_ADD;
49              end
50              BEQ_op: alu_op = ALU_SUB;
51              default: {reg_dst, reg_write, mem_read, mem_to_reg, mem_write, alu_op, alu_src} =
      9'h0_0000_0000;
52          endcase
53      end
54
55  endmodule
56
```

至此, 单周期CPU的代码讲解部分结束.

4.2 Debug Unit——DBU

为了便于整个CPU的debug, 需要有一个DBU用以查看各个阶段中的各个输出, 寄存器和存储器的内容等, 以此进行便捷的debug工作.

4.2.1 基本过程

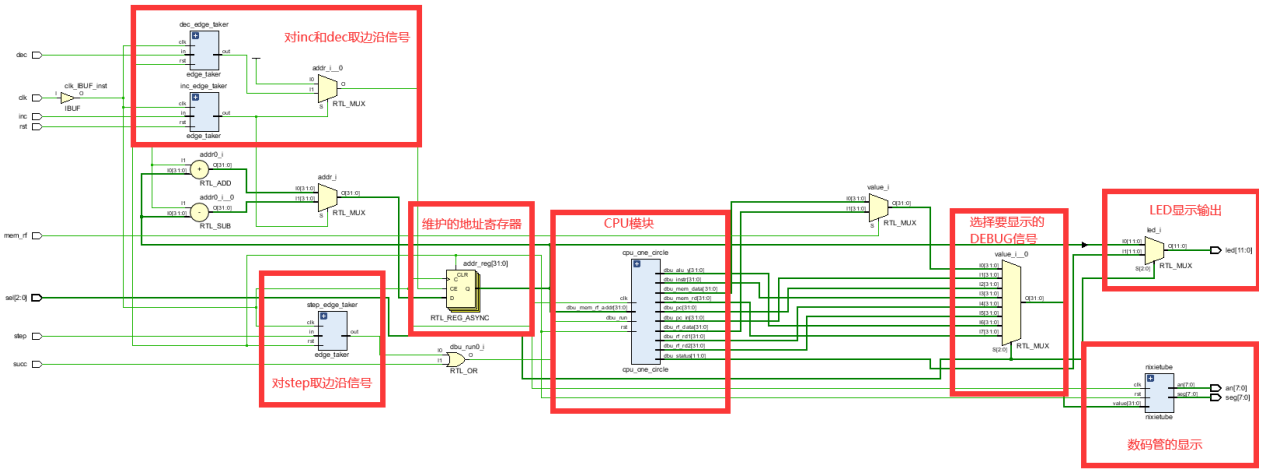
这个DBU单元主要有数码管显示控制, LED显示控制, 开关和电键输入解析等构成. 其中维护了一个地址寄存器, 用以查看寄存器文件和数据存储器的存储信息(这个寄存器内容的修改通过上按键和下按键调节).

由于没有FPGA开发板, 数码管的显示控制无法进行有效调试, 这里暂不讨论, 但为了证明有做这一项, 还是会把代码贴出.

除此之外, 就是一些数据的接线, 以及地址寄存器的内容修改, run信号的生成等. 下面将会讲解.

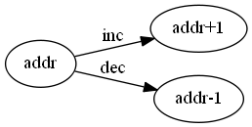
4.2.2 数据通路

DBU这一块的数据通路就由一些CPU模块, 取边沿模块和数码管模块之间的数据通路构成. 为了直观, 下面展示Vivado的RTL分析后的结果.



4.2.3 状态转换

DBU这块主要的状态就是选择地址的转换.



4.2.4 代码讲解

1. DBU数据通路

```
// 取边沿信号(包括inc, dec和step的)
edge_taker #(.N(1)) inc_edge_taker(.clk(clk), .rst(rst), .in(inc), .out(inc_edge));
edge_taker #(.N(1)) dec_edge_taker(.clk(clk), .rst(rst), .in(dec), .out(dec_edge));
edge_taker #(.N(1)) step_edge_taker(.clk(clk), .rst(rst), .in(step), .out(dbu_run));
// CPU模块
cpu_one_circle cpu_one_circle(.clk(clk), .rst(rst), .dbu_run(dbu_run | succ),
                              .dbu_mem_rf_addr(dbu_mem_rf_addr), .dbu_rf_data(dbu_rf_data),
                              .dbu_mem_data(dbu_mem_data), .dbu_pc_in(dbu_pc_in),
                              .dbu_pc(dbu_pc), .dbu_instr(dbu_instr),
                              .dbu_rf_rd1(dbu_rf_rd1), .dbu_rf_rd2(dbu_rf_rd2),
                              .dbu_alu_y(dbu_alu_y), .dbu_mem_rd(dbu_mem_rd),
                              .dbu_status(dbu_status));
// 数码管模块
nixietube nixietube(.clk(clk), .rst(rst), .value(value), .an(an), .seg(seg));
// 传给数码管的要显示的值
always @(*) begin
    case(sel)
        3'b000: begin
            if(mem_rf) value = dbu_mem_data;
            else value = dbu_rf_data;
        end
        3'b001: value = dbu_pc_in;
        3'b010: value = dbu_pc;
        3'b011: value = dbu_instr;
        3'b100: value = dbu_rf_rd1;
        3'b101: value = dbu_rf_rd2;
```

```
        3'b110: value = dbu_alu_y;
        3'b111: value = dbu_mem_rd;
        default: value = 32'h0000_0000;
    endcase
end

```

2. 地址寄存器的increase和decrease

   根据前面数据通路取出的信号边沿, 进行inc和dec操作

```
always @(posedge clk, posedge rst) begin
    if(rst) begin
        addr <= 4'b0000;
    end
    else begin
        if(inc_edge) addr <= addr + 1;
        else if(dec_edge) addr <= addr - 1;
    end
end

```

3. 数码管模块的实现

   这里直接用了上学期模拟与数字电路实验中完成的数码管, 虽然无从调试, 但应大体正确, 若拿到开发板可进行快速调试与修正.

```

module nixietube(
    input clk,
    input rst,
    input [31:0] value,
    output reg [7:0] an,
    output reg [7:0] seg
    );

    // 分频计数器
    integer cnt_target_1000HZ;
    integer cnt_1000HZ;
    reg [3:0] digit;

    initial begin
        cnt_target_1000HZ = 10000;
        cnt_1000HZ = 0;
        an = 8'h00;
        seg = 8'h00;
    end

    always @(posedge clk, posedge rst) begin
        if(rst) begin
            cnt_1000HZ <= cnt_1000HZ + 1;
            an <= 8'b1111_1111;
            seg <= 8'h00;
            digit <= 4'b0000;
        end
        else begin
```

```
30              if(cnt_1000HZ == cnt_target_1000HZ) begin
31                  cnt_1000HZ <=  0;
32                  case(an)
33                      8'b1111_1110: begin an <= 8'b1111_1101; digit = value[3:0]; end
34                      8'b1111_1101: begin an <= 8'b1111_1011; digit = value[7:3]; end
35                      8'b1111_1011: begin an <= 8'b1111_0111; digit = value[11:7]; end
36                      8'b1111_0111: begin an <= 8'b1110_1111; digit = value[15:11]; end
37                      8'b1110_1111: begin an <= 8'b1101_1111; digit = value[19:15]; end
38                      8'b1101_1111: begin an <= 8'b1011_1111; digit = value[23:19]; end
39                      8'b1011_1111: begin an <= 8'b0111_1111; digit = value[27:23]; end
40                      8'b0111_1111: begin an <= 8'b1111_1110; digit = value[31:27]; end
41                      default: begin an <= 8'b1111_1111; digit = 4'b0000; end
42                  endcase
43              end
44              else begin
45                  cnt_1000HZ <= cnt_1000HZ + 1;
46              end
47          end
48      end
49
50      always @(*)
51      begin
52          case(digit)
53              4'b0000: seg = 8'b1100_0000;
54              4'b0001: seg = 8'b1111_1001;
55              4'b0010: seg = 8'b1010_0100;
56              4'b0011: seg = 8'b1011_0000;
57              4'b0100: seg = 8'b1001_1001;
58              4'b0101: seg = 8'b1001_0010;
59              4'b0110: seg = 8'b1000_0010;
60              4'b0111: seg = 8'b1111_1000;
61              4'b1000: seg = 8'b1000_0000;
62              4'b1001: seg = 8'b1001_0000;
63              4'b1010: seg = 8'b1000_1000;
64              4'b1011: seg = 8'b1000_0011;
65              4'b1100: seg = 8'b1010_0111;
66              4'b1101: seg = 8'b1010_0001;
67              4'b1110: seg = 8'b1000_0110;
68              4'b1111: seg = 8'b1000_1110;
69          endcase
70      end
71  endmodule
72
```

5 实验结果

实验结果部分同样分单周期CPU和DBU两块进行讲解. 但由于DBU完全包含CPU, 故这里不会对单周期CPU部分讲解太多. 若有疑问, 在DBU部分应该会有相应描述.

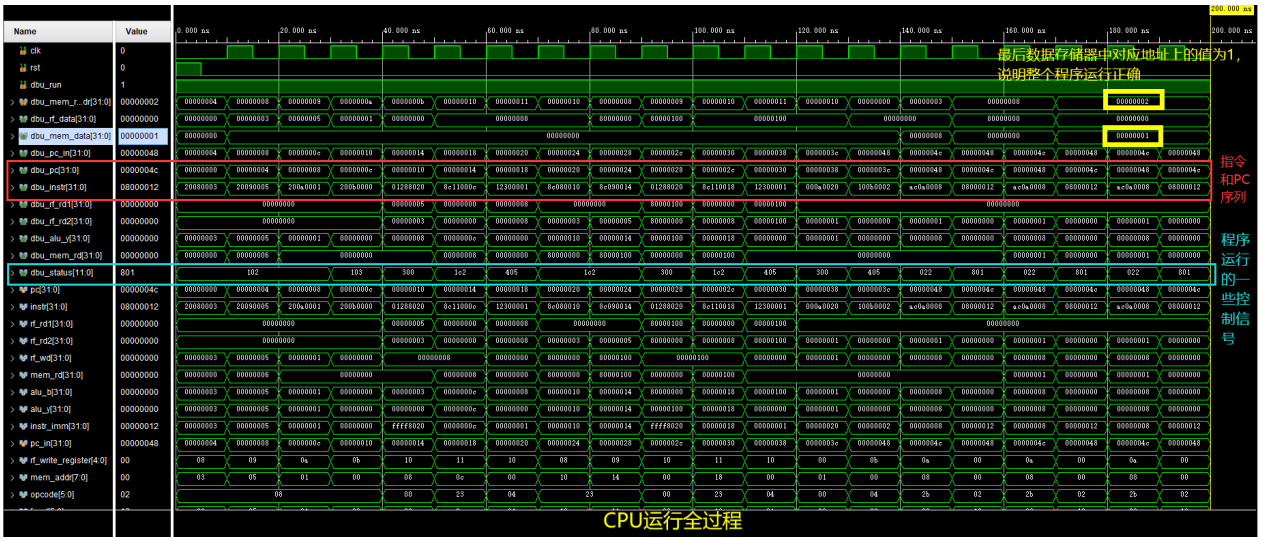
注意, 这里仿真所用的汇编代码为助教提供的代码, 这将极大地方便助教批阅!(见附件)

5.1 单周期CPU

因为后面的DBU仿真结果会按步骤详细讲解程序运行过程, 所以这一部分就展示一下整个仿真过程, 并且标识出PC和指令序列(结合汇编程序的beq跳转条件足以证明CPU工作正常), 和最后的程序正

确运行时, 存储器的标识.

仿真结果如下图:



5.2 Debug Unit——DBU

这一部分的实验结果比较复杂, 按照助教对汇编代码分的几个部分来描述. 对于DBU_LED的输出(仿真中变量为dbu_status), 不再过分陈述, 因为只是进行数据传递, 如果这有错, 那么CPU将无法正常运行.

而寄存器文件和数据存储器的内容正确性, 也是程序运行到最后的_success的必要条件, 因此仿真结果中会有显示, 但并不会做过多说明.

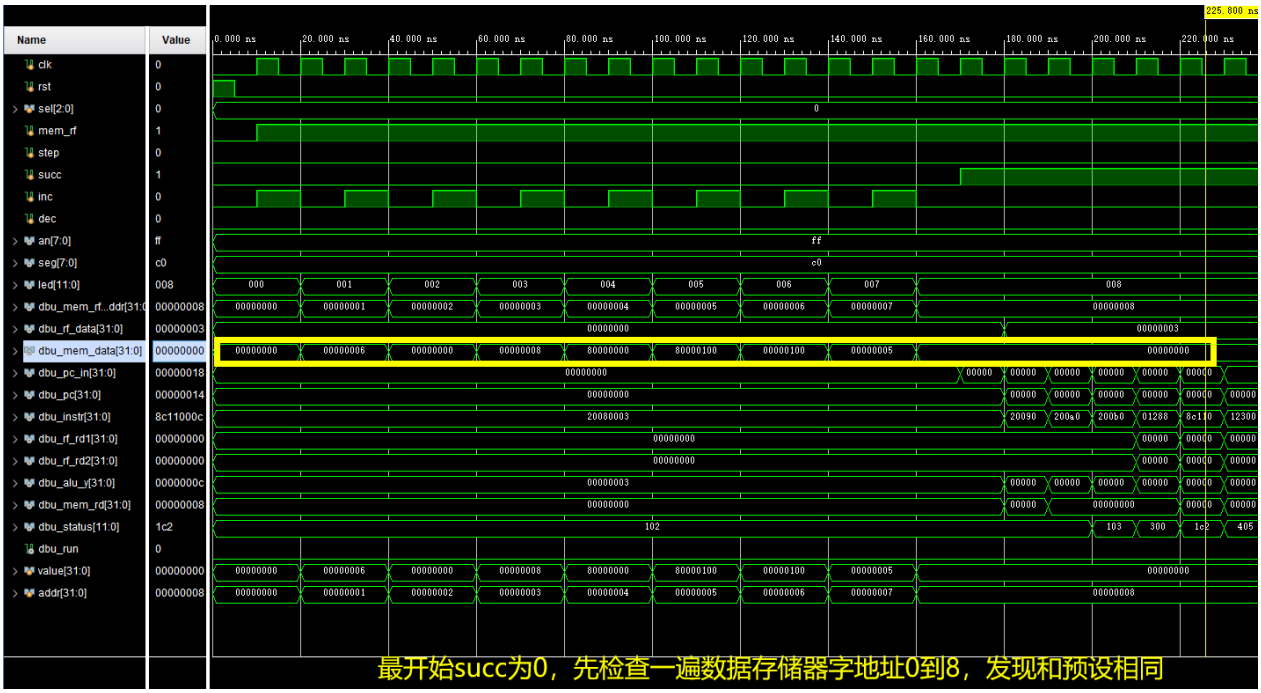
下面就用DBU分步讲解整个代码的运行过程!

5.2.1 检查数据寄存器中的数据

这一步用到了inc来检查数据寄存器中的数据, 这有两个好处:

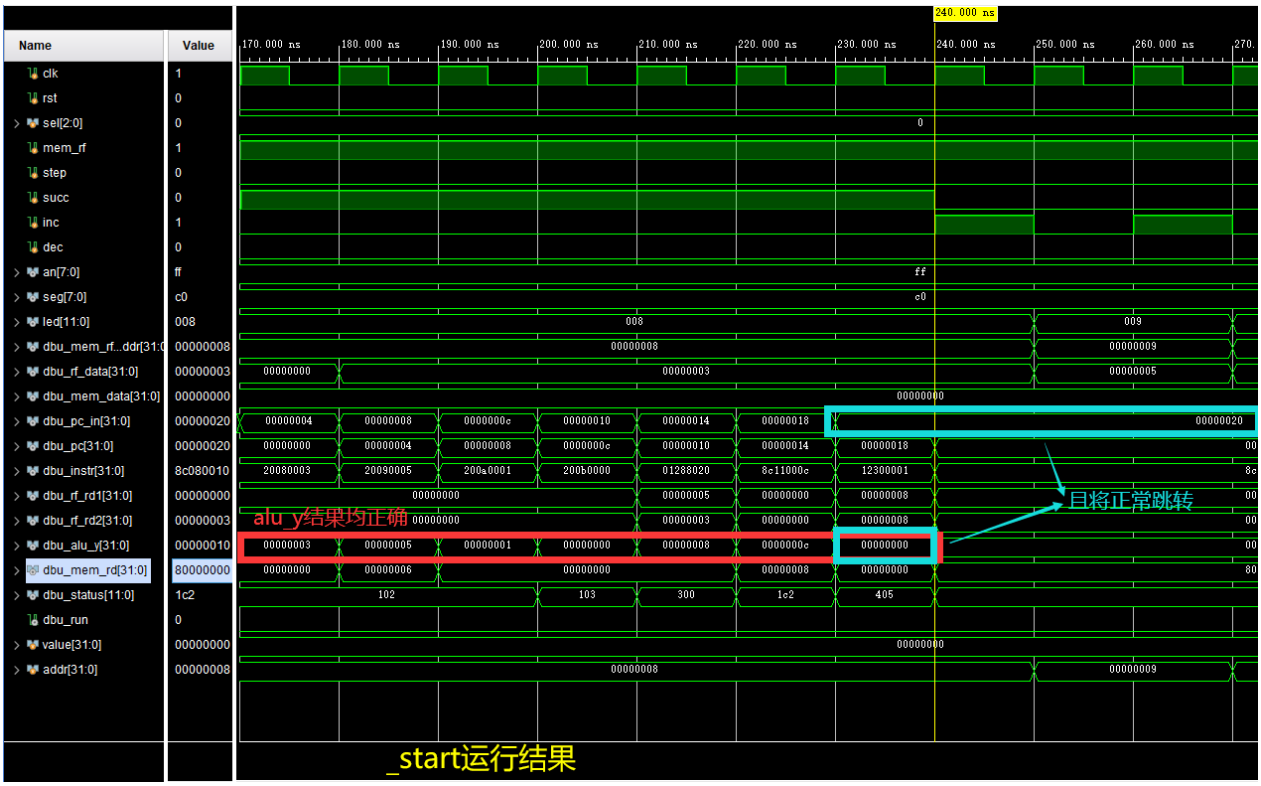
1. 可以检查inc等部分的正确性
2. 可以检查数据存储器的工作是否正常

仿真结果如下,

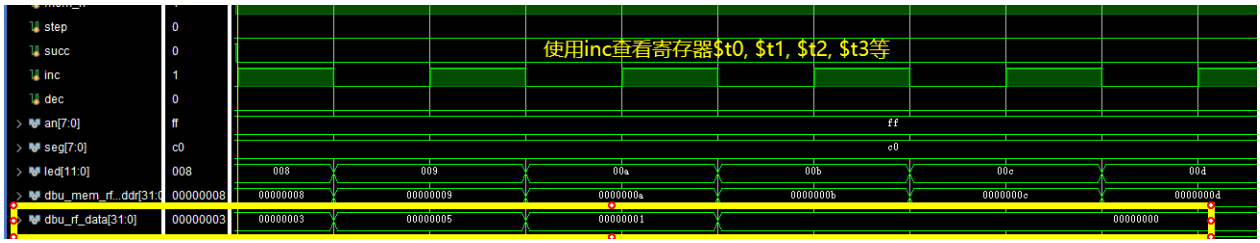


5.2.2 _start部分

_start部分, 对几个寄存器做了addi等操作, 并做了一次lw, 而后beq. 如果能正常beq, 也可以说明代码运行正常. 仿真结果如下. 可以看到alu_y的结果均正确, 并且pc_in也表示将会进行程序正确运行时的跳转.



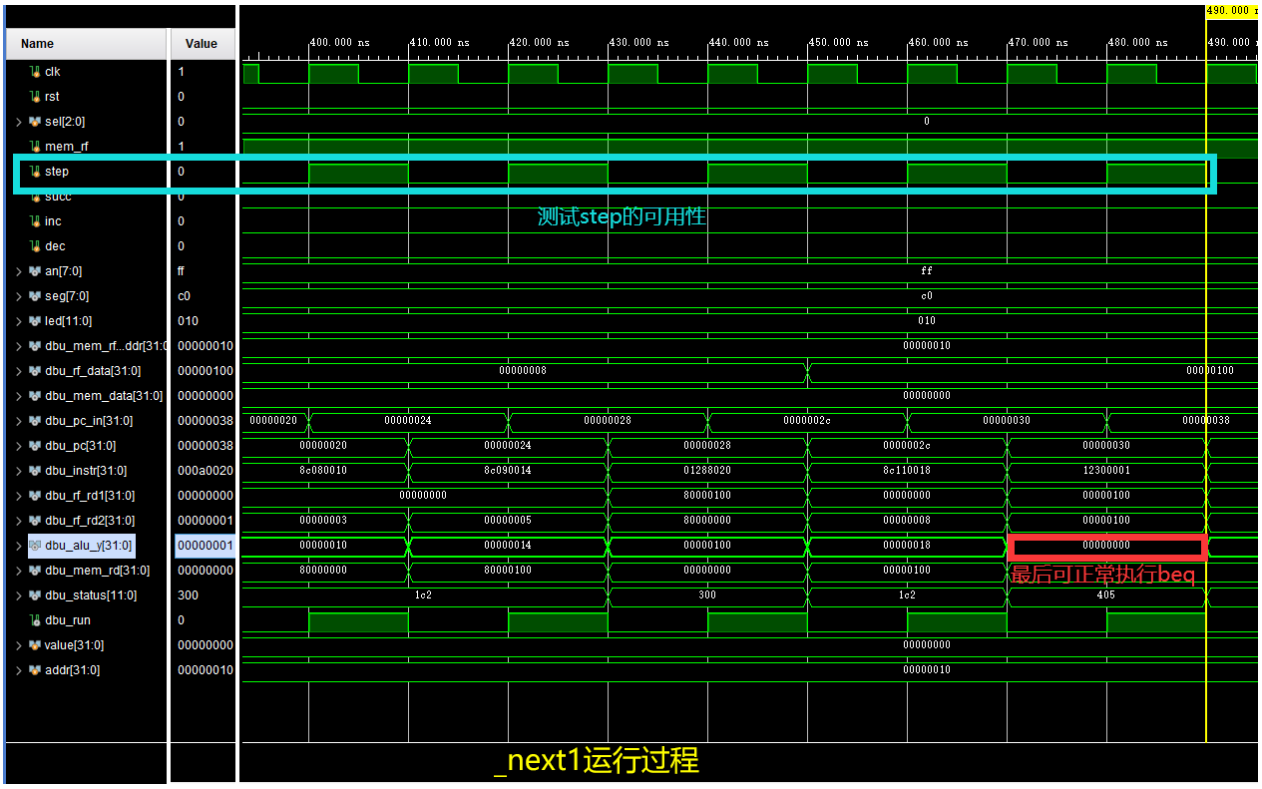
这一步中, 也展示一下寄存器内容修改的正确性. 同时再次检验inc的可用性.



5.2.3 _next1部分

这一步进行了一些lw操作, 可以检查lw操作的正确处理, 并进行了beq. 同样, 如果能正常beq, 也可以说明代码运行正常.

同时, 为了检查step的可行性, 这一部分暂停使用succ, 改用step不断输入, 以逐条执行指令.

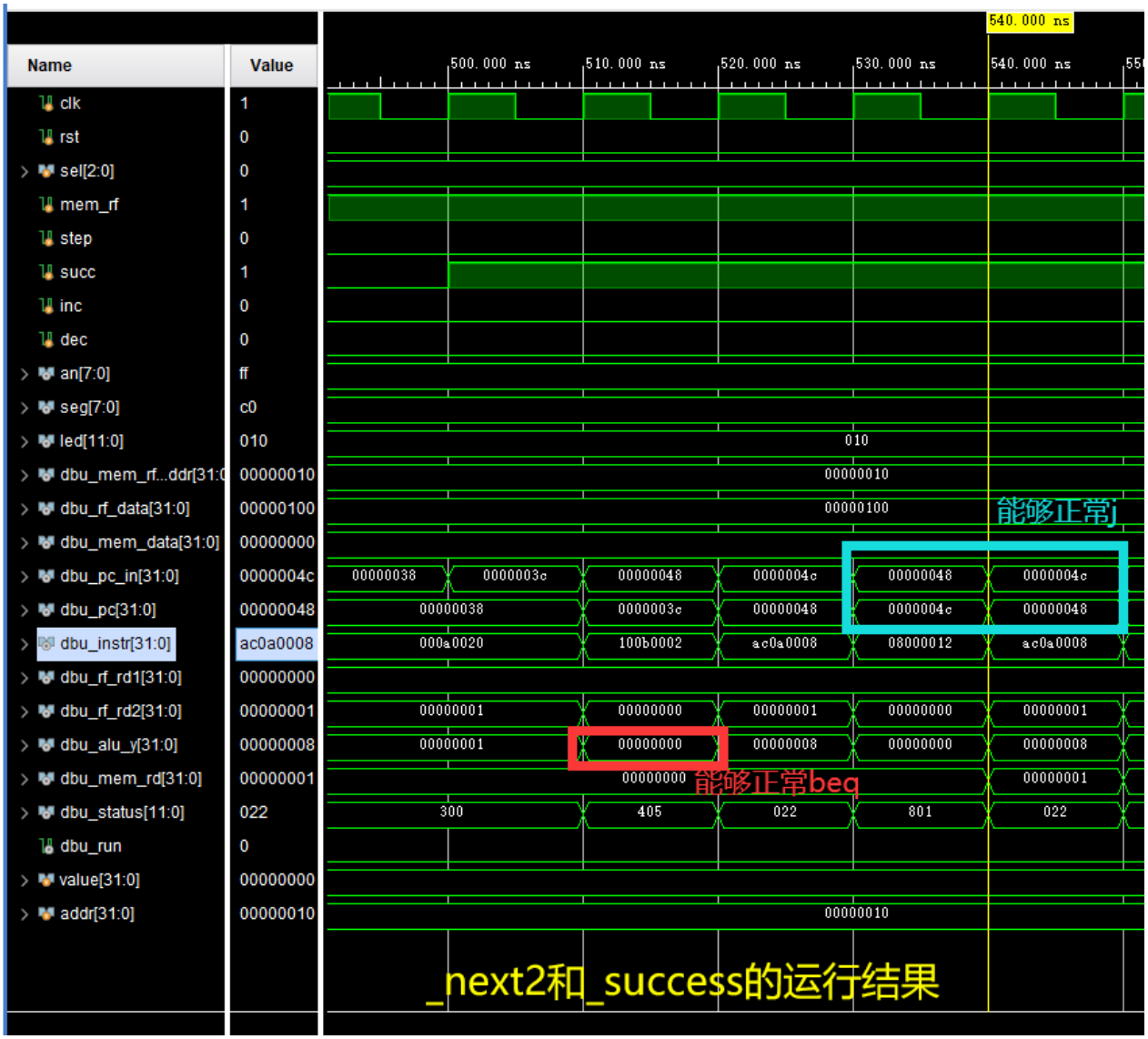


5.2.4 _next2和_success部分

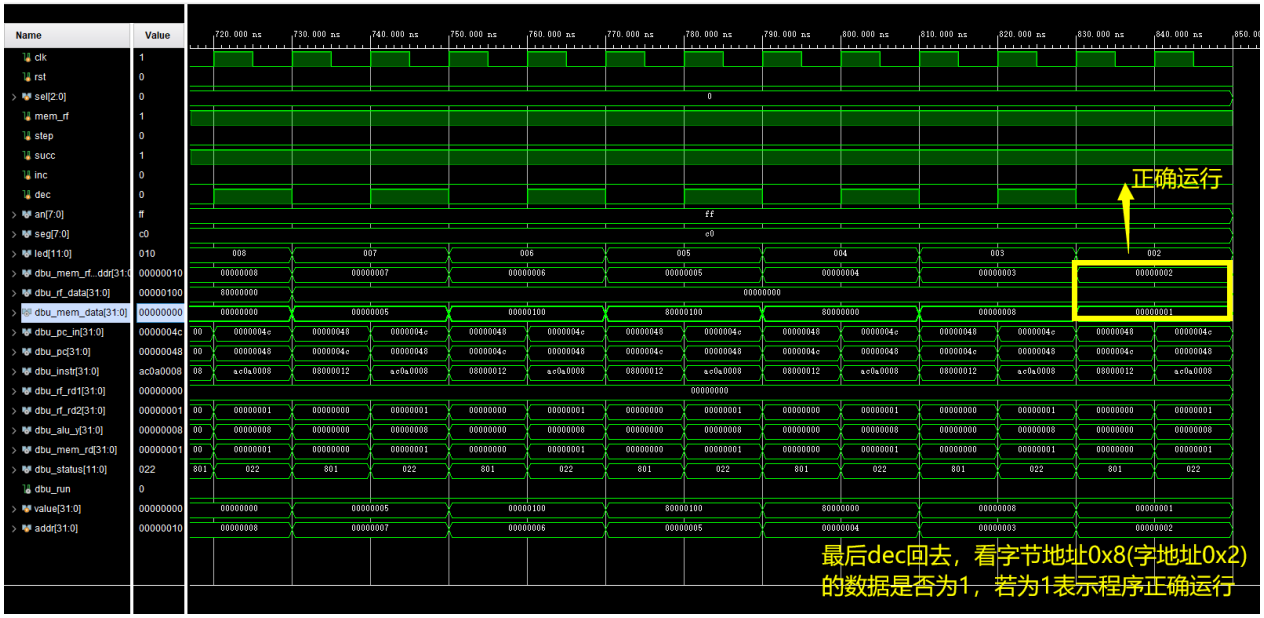
这里主要是检查了寄存器$0永远为0以及_success部分的j指令.

并且最后展示一下数据存储器地址0x08(字地址为0x2)的数值为1, 表示整个程序运行是正确的.

下面展示了寄存器$0永远为0, 因此能够正确地进行beq. 而在_success阶段, 能够正常的jump(j指令), 仿真结果如下:



数据存储器地址0x08(字地址为0x2)的数值为1. 仿真结果如下图,



至此, DBU的仿真过程讲解完毕.

6 思考题

题目: 修改数据通路和控制器, 增加支持如下指令:

$$\boxed{\text{accm: rd} \leftarrow \text{M(rs)} + \text{rt; op} = 000000, \text{funct} = 101000}$$

这里需要修改的数据通路和控制器有如下:

1. 控制单元 $\leftrightarrow$ 各个模块
   控制单元需要解析这条新的指令, 并且相应地去产生数据存储器, 寄存器文件等的存取使能信号.
2. 寄存器堆rs $\leftrightarrow$ 数据存储器读地址 需要将rs作为地址, 读出数据存储器上的M(rs)值.
3. 数据存储器访存结果 $\leftrightarrow$ ALU 将上述M(rs)结果传给ALU, 和rt进行加法运算
4. ALU_out $\leftrightarrow$ 寄存器堆写数据 将上述ALU加法计算结果传给寄存器堆的写数据端口, 在下一个时钟上升沿进行写入.

以上增加的数据通路和控制器足以完成这样一条指令.

7 心得体会

这是我第一次亲手完成一个CPU的verilog代码, 并附有DBU(Debug Unit), 虽然比较复杂, 但由于有了老师精细的讲解和各位助教的帮助, 整个过程比较顺利.

这次实验的收获主要是明白了单周期CPU设计原理, 并且对单周期CPU的了解更进一步. 此外, 还明白了怎么样去设计一个DBU来对自己设计的CPU进行检验.

虽然没能拿到FPGA开发板进行测试, 但仿真成功的结果属实令人开心!

8 意见建议

这次老师和助教们都准备得很充分, 没有什么太多建议.

9 附件

```
# 初始PC = 0x00000000

.data
    .word 0,6,0,8,0x80000000,0x80000100,0x100,5,0
                ;
_start:
        addi $t0,$0,3           #t0=3   0  20080003
        addi $t1,$0,5           #t1=5   4  20090005
        addi $t2,$0,1           #t2=1   8  200a0001
        addi $t3,$0,0           #t3=0   12  200b0000

        add  $s0,$t1,$t0        #s0=t1+t0=8  测试add指令    16  01288020
        lw   $s1,12($0)         #                          20  8c11000c
        beq  $s1,$s0,_next1     #正确跳到_next             24  12300001

        j _fail                     # 08000010

_next1:
        lw $t0, 16($0)          #t0 = 0x80000000    32  8c080010
        lw $t1, 20($0)          #t1 = 0x80000100    36  8c090014

        add  $s0,$t1,$t0        #s0 = 0x00000100 = 256  40  01288020
        lw $s1, 24($0)          #                       44  8c110018
        beq  $s1,$s0,_next2     #正确跳到_success       48  12300001

        j _fail                 # 08000010

_next2:
        add $0, $0, $t2         # $0应该为0         56  000a0020
        beq $0,$t3,_success     #                   60  100b0002


_fail:
        sw   $t3,8($0)  # ac0b0008
        j    _fail      # 08000010

_success:
        sw   $t2,8($0)      #全部测试通过，存储器地址0x08里的值为1  ac0a0008
        j   _success        #08000012

                            #判断测试通过的条件是最后存储器地址0x08里的值为1，说明全部通过测试
```