计算机组成原理实验 实验报告

实验题目：Lab4 多周期CPU

学生姓名：王章瀚

学生学号：PB18111697

完成日期：2020 年 5 月 28 日

计算机实验教学中心制

2019年09月

1 实验题目

Lab4 多周期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

实验过程主要分为多周期CPU的设计和DBU的设计. 下面分块讲解二者.

4.1 多周期CPU

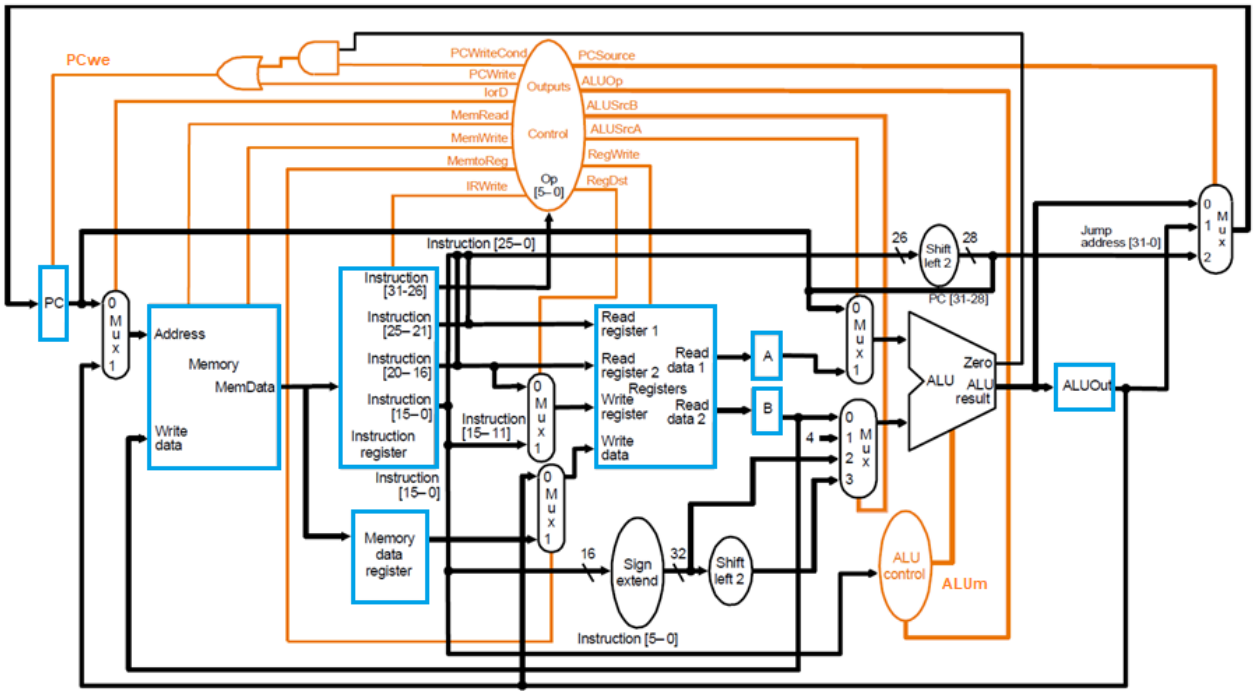
4.1.1 基本过程

多周期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.

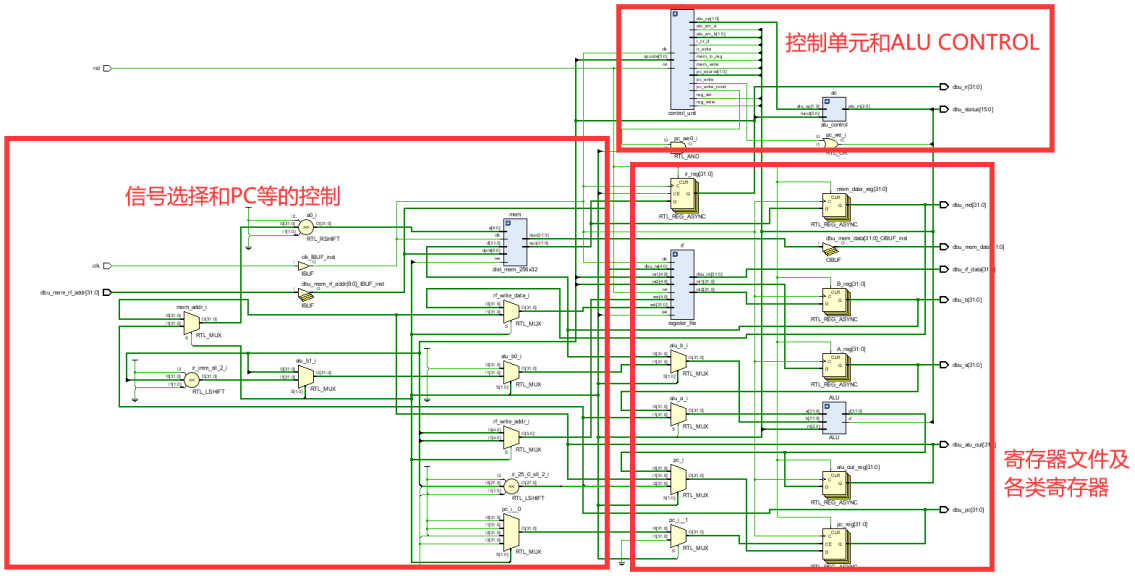
这里需要讲解的部分有: 数据通路, 状态转换, 控制单元及其他代码等.

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.

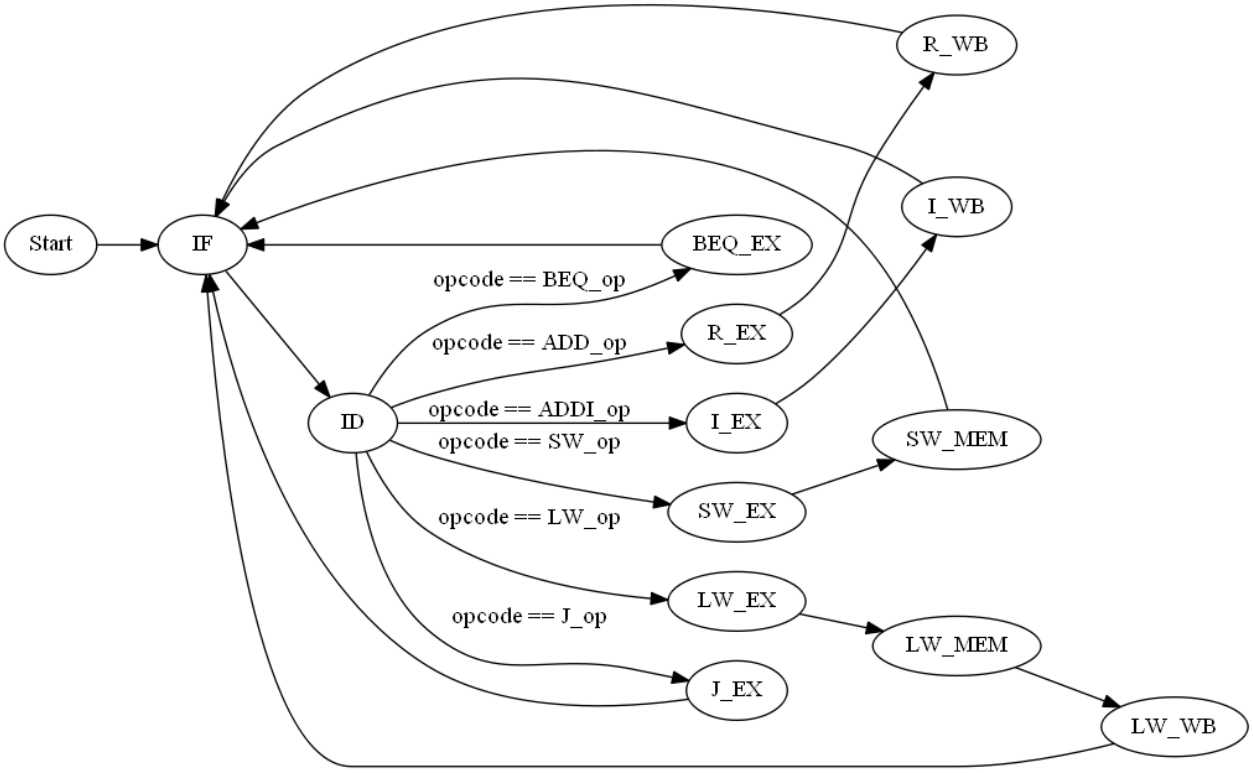


而我对我完成的代码进行RTL分析, 能够得出下面的数据通路. 其中各个模块的功能已经在图中具体标识了.



4.1.3 状态转换

多周期的CPU状态机比较复杂, 用graphviz作图, 得到下图.



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. 并且, 与单周期不同的是, 这里还需要由时钟控制的寄存器的写入. 代码如下:

```
// PC
assign pc_we = pc_write | (pc_write_cond & alu_zero);

// 指令数据存储器
assign mem_addr = i_or_d == 1'b1 ? alu_out : pc;
dist_mem_256x32 mem(.clk(clk), .we(mem_write),
                    .a(mem_addr >> 2), .d(mem_write_data), .spo(mem_read_data),
                    .dpra(dbu_mem_rf_addr), .dpo(dbu_mem_data));
assign ir_imm = {{16{ir[15]}}, ir[15:0]};
assign ir_imm_sll_2 = ir_imm << 2;

// IR - RF
assign rf_write_addr = reg_dst == 1'b1 ? ir[15: 11] : ir[20: 16];
assign rf_write_data = mem_to_reg == 1'b1 ? mem_data : alu_out;
register_file rf(.clk(clk), .rst(rst), .we(reg_write),
                 .ra1(ir[25: 21]), .rd1(rf_rd1),
                 .ra2(ir[20: 16]), .rd2(rf_rd2),
                 .dbu_ra(dbu_mem_rf_addr), .dbu_rd(dbu_rf_data),
                 .wa(rf_write_addr), .wd(rf_write_data));
assign mem_write_data = B;
```

```
21
22 // RF - ALU
23 assign alu_a = alu_src_a == 1'b1 ? A : pc;
24 assign alu_b = alu_src_b == 2'b00 ? B :
25                 alu_src_b == 2'b01 ? {{29{1'b0}}, 3'b100} :
26                 alu_src_b == 2'b10 ? ir_imm : ir_imm_sll_2;
27 ALU ALU(.m(alu_m), .a(alu_a), .b(alu_b),
28         .y(alu_y), .zf(alu_zero));
29
30 // 控制单元
31 alu_control ac(.alu_op(alu_op), .funct(ir[5: 0]), .alu_m(alu_m));
32 control_unit cn(.clk(clk), .rst(rst), .opcode(ir[31: 26]),
33                 .pc_write_cond(pc_write_cond), .pc_write(pc_write),
34                 .i_or_d(i_or_d), .mem_read(mem_read), .mem_write(mem_write),
35                 .mem_to_reg(mem_to_reg), .ir_write(ir_write),
36                 .pc_source(pc_source), .alu_op(alu_op),
37                 .alu_src_a(alu_src_a), .alu_src_b(alu_src_b),
38                 .reg_write(reg_write), .reg_dst(reg_dst));
39
40 // 一些寄存器的写入
41 always @(posedge clk, posedge rst) begin
42     if(rst) begin
43         mem_data <= 32'h0000_0000;
44         ir <= 32'h0000_0000;
45         A <= 32'h0000_0000;
46         B <= 32'h0000_0000;
47         alu_out <= 32'h0000_0000;
48     end
49     else begin
50         // Memory Data Register 的写入
51         mem_data <= mem_read_data;
52
53         // Instruction Register 的写入
54         if(ir_write) ir <= mem_read_data;
55         else ir <= ir;
56
57         // A 和 B 写入
58         A <= rf_rd1;
59         B <= rf_rd2;
60
61         // alu_out 写入
62         alu_out <= alu_y;
63     end
64 end
65
```

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. 和单周期CPU的设计差别不大.

```
1  // PC的更新
2  assign ir_25_0_sll_2 = ir[25: 0] << 2;
3  always @(posedge clk, posedge rst) begin
4      if(rst) begin
5          pc <= 32'h0000_0000;
6      end
7      else if(pc_we) begin
8          case(pc_source)
9              2'b00: pc <= alu_y;
10             2'b01: pc <= alu_out;
```

```
            2'b10: pc <= {pc[31: 28], ir_25_0_sll_2[27: 0]};
            default: pc <= pc;
        endcase
    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, 判断各个状态阶段需要使能哪些信号, 失能哪些信号.

与单周期CPU不同的是, 这里的控制单元需要各种状态的控制, 这里用到的状态如下表.

状态	说明
IF	取指
ID	解码
R_EX	R型指令执行
R_WB	R型指令写回
I_EX	I型指令执行
I_WB	I型指令写回
LW_EX	LW执行
LW_MEM	LW访存
LW_WB	LW写回
SW_EX	LW执行
SW_MEM	LW访存
BEQ_EX	BEQ执行
J_EX	J执行

为避免一次展示过长代码(且代码将作为附件提交), 这里只分别展示一下各个状态阶段需要输出的控制信号以及状态转换的相关代码.

关于控制信号输出的代码:

```
// 输出
{pc_write_cond, pc_write, pc_source,
 i_or_d, mem_read, mem_write, mem_to_reg,
 ir_write, reg_write, reg_dst,
 alu_op, alu_src_a, alu_src_b} = 16'h0000;
case(cur_state)
    IF: begin
        mem_read = 1'b1;
        alu_src_a = 1'b0;
        i_or_d = 1'b0;
        ir_write = 1'b1;
        alu_src_b = 2'b01;
        alu_op = 2'b00;
```

```
14          pc_write = 1'b1;
15          pc_source = 2'b00;
16      end
17      ID: begin
18          alu_src_a = 1'b0;
19          alu_src_b = 2'b11;
20          alu_op = 2'b00;
21      end
22      R_EX: begin
23          alu_src_a = 1'b1;
24          alu_src_b = 2'b00;
25          alu_op = 2'b10;
26      end
27      R_WB: begin
28          reg_dst = 1'b1;
29          reg_write = 1'b1;
30          mem_to_reg = 1'b0;
31      end
32      I_EX, LW_EX, SW_EX: begin
33          alu_src_a = 1'b1;
34          alu_src_b = 2'b10;
35          alu_op = 2'b00;
36      end
37      I_WB: begin
38          reg_dst = 1'b0;
39          reg_write = 1'b1;
40          mem_to_reg = 1'b0;
41      end
42      LW_MEM: begin
43          mem_read = 1'b1;
44          i_or_d = 1'b1;
45      end
46      LW_WB: begin
47          reg_dst = 1'b0;
48          reg_write = 1'b1;
49          mem_to_reg = 1'b1;
50      end
51      SW_MEM: begin
52          mem_write = 1'b1;
53          i_or_d = 1'b1;
54      end
55      BEQ_EX: begin
56          alu_src_a = 1'b1;
57          alu_src_b = 2'b00;
58          alu_op = 2'b01;
59          pc_write_cond = 1'b1;
60          pc_source = 2'b01;
61      end
62      J_EX: begin
63          pc_write = 1'b1;
64          pc_source = 2'b10;
65      end
66      default: begin
67          {pc_write_cond, pc_write, pc_source,
68           i_or_d, mem_read, mem_write, mem_to_reg,
69           ir_write, reg_write, reg_dst,
70           alu_op, alu_src_a, alu_src_b} = 16'h0000;
71      end
72  endcase
73
```

关于状态转移的代码:

```
// 状态机转移
case(cur_state)
    IF: next_state = ID;
    ID: begin
        case(opcode)
            ADD_op: next_state = R_EX;
            ADDI_op: next_state = I_EX;
            LW_op: next_state = LW_EX;
            SW_op: next_state = SW_EX;
            BEQ_op: next_state = BEQ_EX;
            J_op: next_state = J_EX;
            default: next_state = cur_state;
        endcase
    end
    R_EX: next_state = R_WB;
    R_WB: next_state = IF;
    I_EX: next_state = I_WB;
    I_WB: next_state = IF;
    LW_EX: next_state = LW_MEM;
    LW_MEM: next_state = LW_WB;
    LW_WB: next_state = IF;
    SW_EX: next_state = SW_MEM;
    SW_MEM: next_state = IF;
    BEQ_EX: next_state = IF;
    J_EX: next_state = IF;
    default: next_state = cur_state;
endcase

```

至此, 多周期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

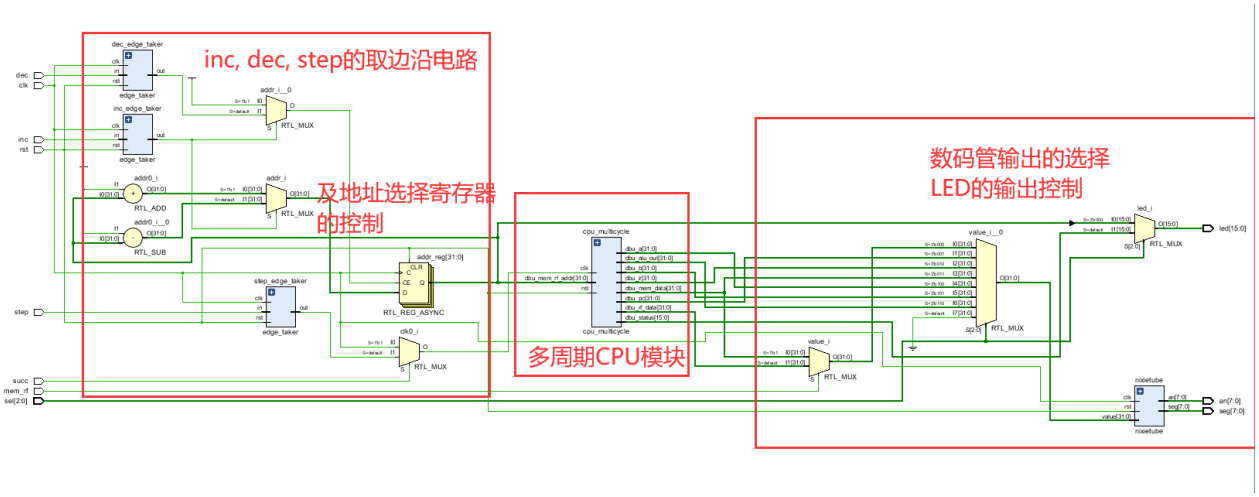
这个DBU单元主要有数码管显示控制, LED显示控制, 开关和电键输入解析等构成. 其中维护了一个地址寄存器, 用以查看寄存器文件和数据存储器的存储信息(这个寄存器内容的修改通过上按键和下按键调节).

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.

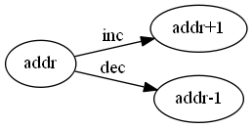
4.2.2 数据通路

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

```
assign dbu_mem_rf_addr = addr;
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(step_edge));
cpu_multicycle cpu_multicycle(.clk(succ == 1'b1 ? clk : step_edge),
                              .rst(rst),
                              .dbu_mem_rf_addr(dbu_mem_rf_addr),
                              .dbu_rf_data(dbu_rf_data),
                              .dbu_mem_data(dbu_mem_data),
                              .dbu_pc(dbu_pc),
                              .dbu_ir(dbu_ir),
                              .dbu_md(dbu_md),
                              .dbu_a(dbu_a),
                              .dbu_b(dbu_b),
                              .dbu_alu_out(dbu_alu_out),
                              .dbu_status(dbu_status));
// LED显示
assign led = sel == 3'b000 ? dbu_mem_rf_addr : dbu_status;

// 数码管
nixietube nixietube(.clk(clk), .rst(rst), .value(value), .an(an), .seg(seg));
```

```
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;
        3'b010: value = dbu_ir;
        3'b011: value = dbu_mem_data;
        3'b100: value = dbu_a;
        3'b101: value = dbu_b;
        3'b110: value = dbu_alu_out;
        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 <= 32'h0000_0000;
    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;
    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
            if(cnt_1000HZ == cnt_target_1000HZ) begin
                cnt_1000HZ <=  0;
                case(an)
                    8'b1111_1110: begin an <= 8'b1111_1101; digit <= value[3:0]; end
                    8'b1111_1101: begin an <= 8'b1111_1011; digit <= value[7:3]; end
                    8'b1111_1011: begin an <= 8'b1111_0111; digit <= value[11:7]; end
                    8'b1111_0111: begin an <= 8'b1110_1111; digit <= value[15:11]; end
                    8'b1110_1111: begin an <= 8'b1101_1111; digit <= value[19:15]; end
                    8'b1101_1111: begin an <= 8'b1011_1111; digit <= value[23:19]; end
                    8'b1011_1111: begin an <= 8'b0111_1111; digit <= value[27:23]; end
                    8'b0111_1111: begin an <= 8'b1111_1110; digit <= value[31:27]; end
                    default: begin an <= 8'b1111_1111; digit <= 4'b0000; end
                endcase
            end
            else begin
                cnt_1000HZ <= cnt_1000HZ + 1;
            end
        end
    end

    always @(*)
    begin
        case(digit)
            4'b0000: seg = 8'b1100_0000;
            4'b0001: seg = 8'b1111_1001;
            4'b0010: seg = 8'b1010_0100;
            4'b0011: seg = 8'b1011_0000;
            4'b0100: seg = 8'b1001_1001;
            4'b0101: seg = 8'b1001_0010;
            4'b0110: seg = 8'b1000_0010;
            4'b0111: seg = 8'b1111_1000;
            4'b1000: seg = 8'b1000_0000;
            4'b1001: seg = 8'b1001_0000;
            4'b1010: seg = 8'b1000_1000;
            4'b1011: seg = 8'b1000_0011;
            4'b1100: seg = 8'b1010_0111;
            4'b1101: seg = 8'b1010_0001;
            4'b1110: seg = 8'b1000_0110;
            4'b1111: seg = 8'b1000_1110;
            default: seg = 8'h00;
        endcase
    end
endmodule

```

5 实验结果

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

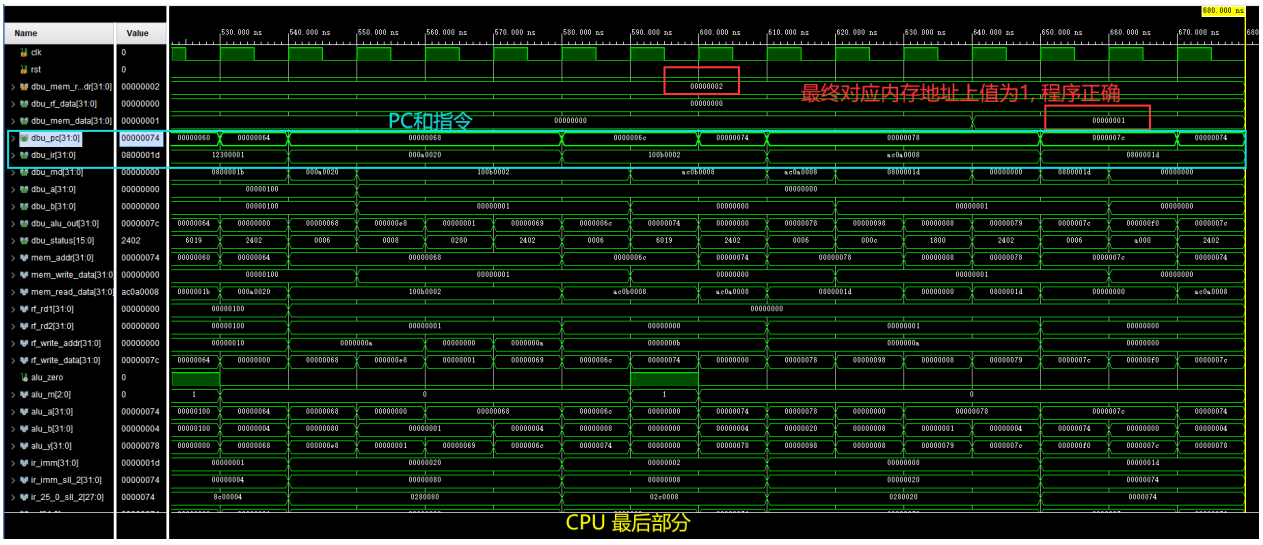
注意, 这里仿真所用的汇编代码为助教提供的代码, 这将极大地方便助教批阅!(见附件)

5.1 多周期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最后一部分的仿真过程, 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.

注意, 这里在除了IF阶段, PC对应下来的指令应当是PC+4后的结果, 这是因为PC在IF做了+4, 而非程序有误.

仿真结果如下图:



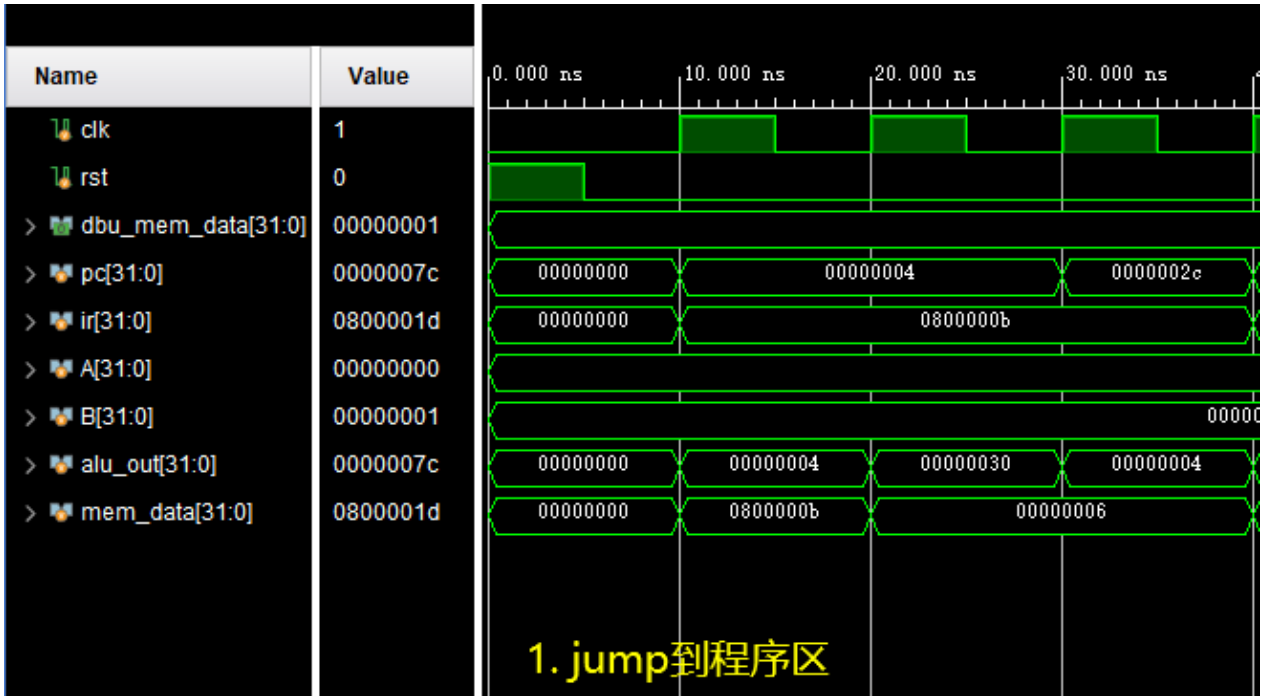
5.2 Debug Unit——DBU

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.

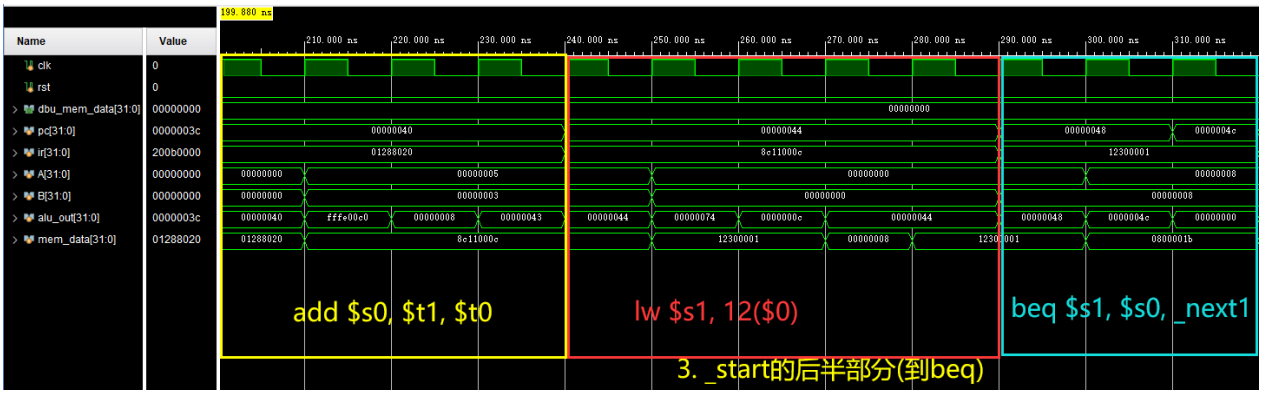
下面就用DBU分步讲解整个代码的运行过程!

5.2.1 第一步的jump(到程序开始的地方)

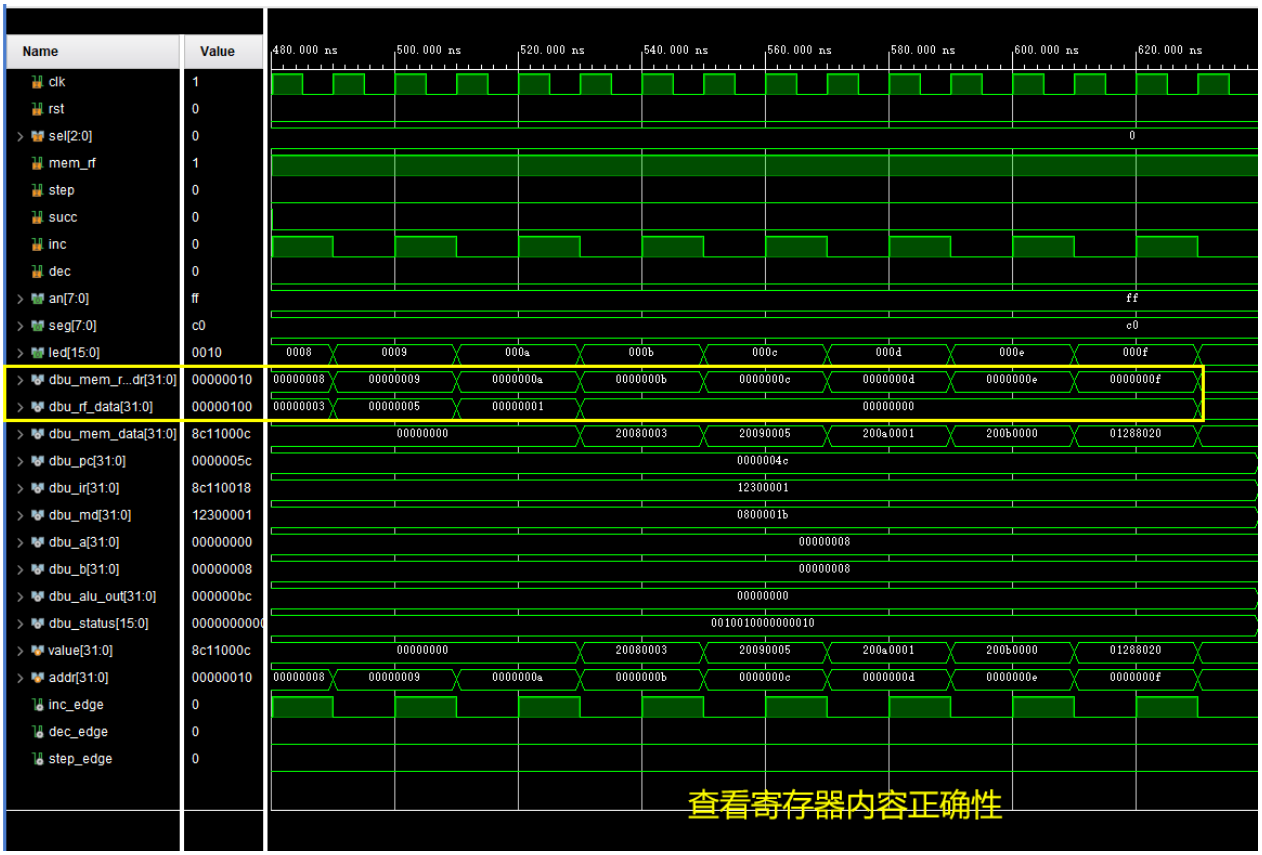


5.2.2 _start部分

_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.



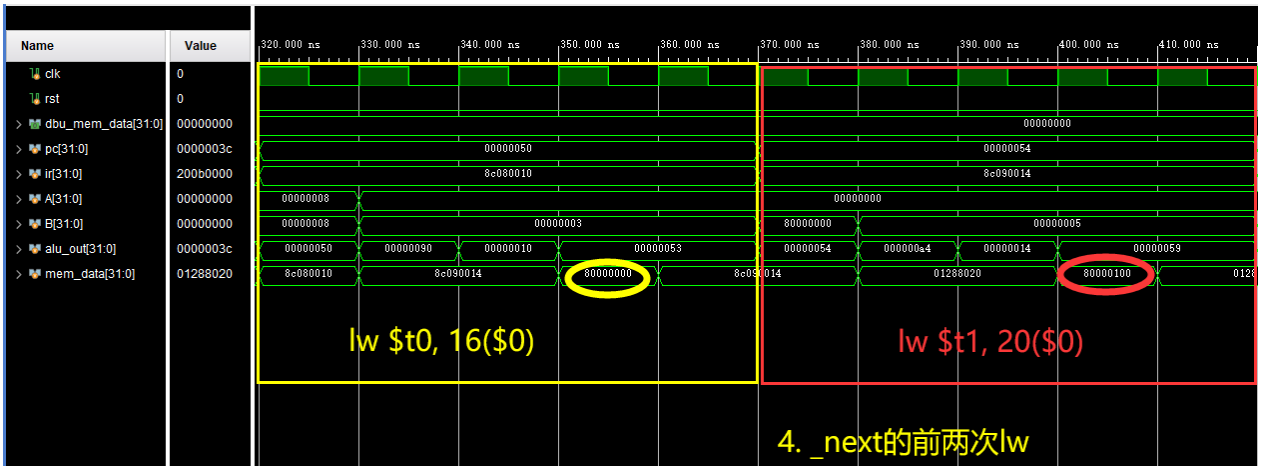
这一步中, 也展示一下寄存器内容修改的正确性. 同时再次检验inc的可用性.

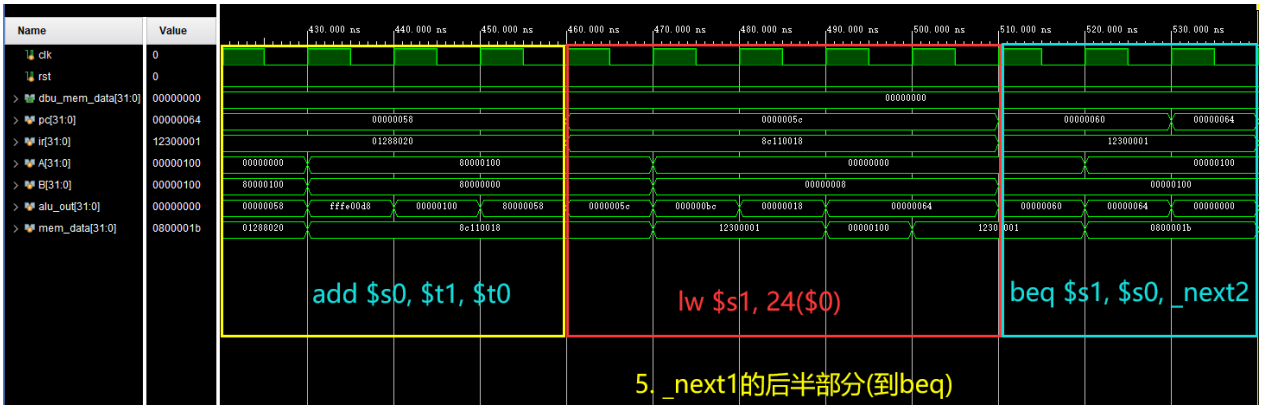


5.2.3 _next1部分

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.



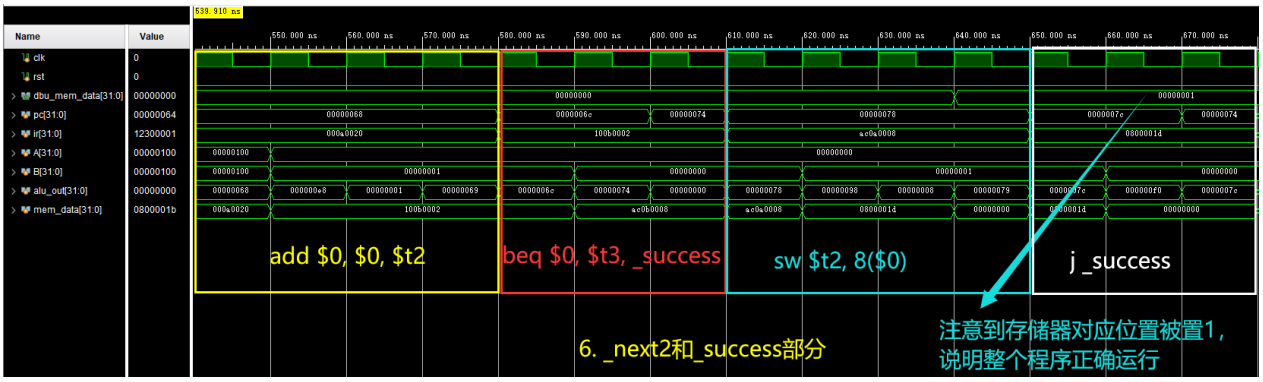


5.2.4 _next2和_success部分

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.

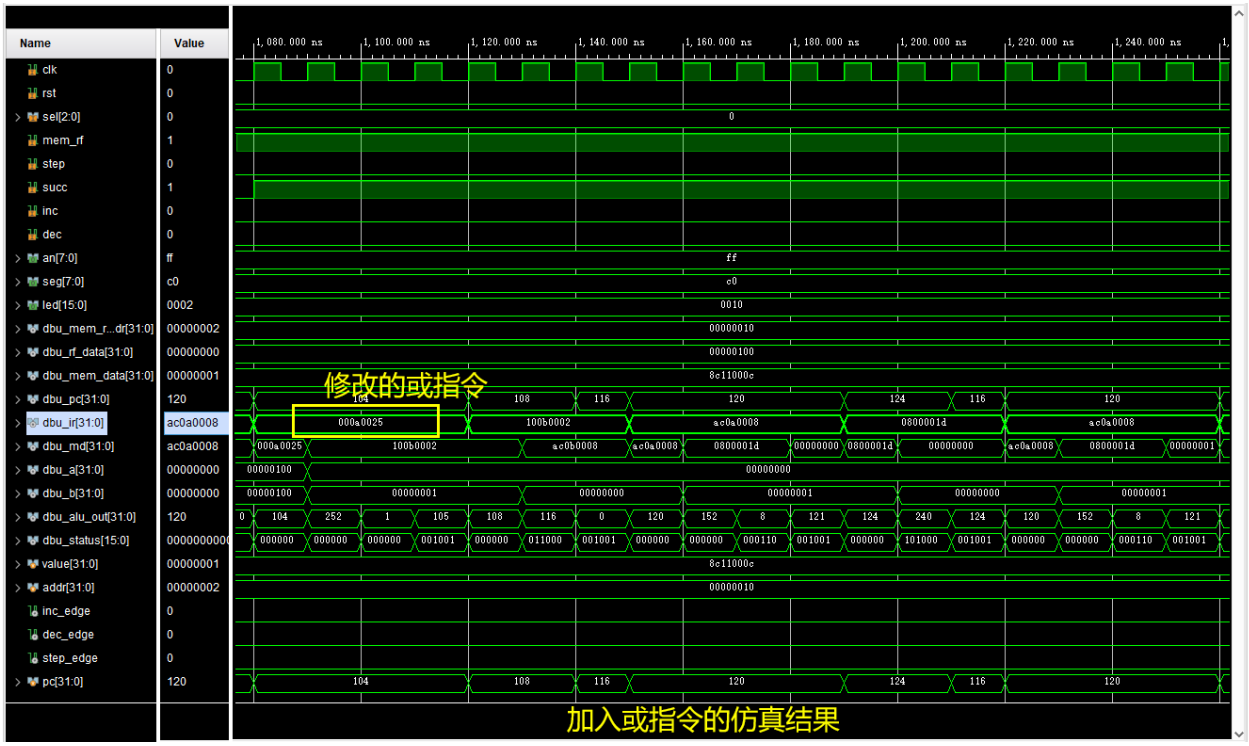
下面展示了寄存器$0永远为0, 因此能够正确地进行beq. 而在_success阶段, 数据存储器地址0x08(字地址为0x2)的数值被置为1. 仿真结果如下:



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，扩展对其他MIPS指令的支持，并进行功能仿真和下载测试。 实际上, 有了前面对R类, I类, LW, SW, BEQ, J的设计, 要增加一条指令的支持十分简单, 只不过是工程量的问题. 因此为表示我可以完成, 这里加一条**或指令**. 并将测试代码中_next2的 "or $0, $0, $t2" 改为 "add $0, $0, $t2".



并且从beq的结果也能看出, 这里的执行是正确的.

7 心得体会

经过上一次单周期CPU的设计与实现, 已经对CPU的实现有一定理解与基础, 因此这次实现多周期CPU也避免了很多坑.

这次实验的收获主要是明白了多周期CPU设计原理, 并且对多周期CPU的了解更进一步. 我想这对后面流水线CPU的设计将有很大帮助.

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

```
# 本文档存储器以字编址
# 本文档存储器以字编址
# 本文档存储器以字编址
# 初始PC = 0x00000000


    j _start    # 0 0800000b

.data
    .word 0,6,0,8,0x80000000,0x80000100,0x100,5,0,0,0   #编译成机器码时，编译器会在前面多加个0，所以后面lw指令地址会多加4

_start:
        addi $t0,$0,3           #t0=3   44                              20080003
        addi $t1,$0,5           #t1=5   48                              20090005
        addi $t2,$0,1           #t2=1   52                              200a0001
        addi $t3,$0,0           #t3=0   56                              200b0000

        add  $s0,$t1,$t0        #s0=t1+t0=8  测试add指令    60          01288020
        lw   $s1,12($0)         #                            64         8c11000c
        beq  $s1,$s0,_next1     #正确跳到_next              68          12300001

        j _fail                                             #       0800001b

_next1:
        lw $t0, 16($0)          #t0 = 0x80000000    76                  8c080010
        lw $t1, 20($0)          #t1 = 0x80000100    80                  8c090014

        add  $s0,$t1,$t0        #s0 = 0x00000100 = 256  84              01288020
        lw $s1, 24($0)          #                        88             8c110018
        beq  $s1,$s0,_next2     #正确跳到_success       92              12300001

        j _fail                                                 #   0800001b

_next2:
        add $0, $0, $t2         #$0应该一直为0          100             000a0020
        beq $0,$t3,_success     #                        104            100b0002


_fail:
        sw   $t3,8($0) #失败通过看存储器地址0x08里值，若为0则测试不通过，最初地址0x08里值为0 108    ac0b0008
        j    _fail                                              #   0800001b

_success:
        sw   $t2,8($0)    #全部测试通过，存储器地址0x08里值为1  116        ac0a0008
        j   _success                                            #   0800001d

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