浙江大学

# 本科实验报告

课程名称：计算机组成

姓　　名：胡亮泽

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3120102116

指导教师：姜晓红

2014 年　6　月　17　日

# 浙江大学实验报告

课程名称：Computer Organization 实验类型：综合

实验项目名称：Lab10：微指令

学生姓名：胡亮泽 专业：计算机科学与技术

学号：3120102116

同组学生姓名：王艺 指导老师：姜晓红

实验地点：东 4-509 实验日期：2014 年 6 月 17 日

一、 实验目的和要求

在多周期 CPU 的基础上，用微指令的控制器替换状态控制器，实现微指令控制的多周期 CPU。

二、 实验内容和原理

本次实验和多周期 CPU 的实验代码基本上完全相同，除了用微指令控制器替换了状态控制器。对于其他完全相同的模块，这里不多做赘述，下面将重点介绍微指令控制器的实现原理：

首先是总控制模块：

```
module Micro_control(
                  input wire clk,
                         input wire rst,
                         input wire[5:0] op,
                         output wire[17:0] control,
                         output wire[3:0] current_mpc
   );

wire[3:0] next_mpc;

ctrlmem m1(current_mpc,control);

mpc_change c1(clk,rst,next_mpc,current_mpc);

MicroMux m2(clk,op,current_mpc,control[1:0],next_mpc);


endmodule
```

该模块主要由三个子模块实现，分别是 ctrlmem，在输入当前的微指令地址时输出 control 控制信号；mpc_change，输入下一个微指令地址并在时钟上升沿到来的时候改变当前的微指令地址，达到指令切换的目的；最后一个 MicroMux 模块是根据当前控制信号 control 的最后两位来确定下一时刻的微指令地址的。

关于 ctrlmem 部分的代码：

```
module ctrlmem(
                  input wire[3:0] mpc,
                  output wire[17:0] control
   );

reg[17:0] temp_control;

initial  temp_control <= 0;
assign control = temp_control;

always @(mpc) begin
case(mpc)
0:temp_control<={16'h0851,2'b11};
1:temp_control<={16'h1800,2'b01};
2:temp_control<={16'h3000,2'b10};
3:temp_control<={16'h00C0,2'b11};
4:temp_control<={16'h0300,2'b00};
5:temp_control<={16'h00A0,2'b00};
6:temp_control<={16'hA000,2'b11};
7:temp_control<={16'h0500,2'b00};
8:temp_control<={16'h6006,2'b00};
9:temp_control<={16'h0009,2'b00};
default:temp_control <= 0 ;
endcase
end

endmodule
```

在时钟上升沿到来时，改变 control 的信号，而信号的具体值通过 case 语句存储在该模块中。

关于 mpc_change 部分的代码：

```
module mpc_change(
               input wire clk,
                     input wire rst,
                     input wire[3:0] next_mpc,
                     output wire[3:0] current_mpc
   );
reg[3:0] temp_mpc;
initial temp_mpc = 4;

assign current_mpc= temp_mpc;

always@(posedge clk,posedge rst) begin
if(rst) temp mpc <= 4;
else
temp_mpc <= next_mpc;
end
```

```
endmodule
```

在时钟上升沿到来时，将下一时刻的值赋给 mpc 就可以了。

最后是 MicroMux 的代码：

```
module MicroMux(
              input wire clk,
                   input wire[5:0] op,
                   input wire[3:0] current_mpc,
                   input wire[1:0] select,
                   output wire[3:0] next_mpc
    );
wire[3:0] mpc_plus_1,dispatch1,dispatch2;

mux mux1(4'b0000,dispatch1,dispatch2,mpc_plus_1,next_mpc,select);

mpc_plus_1 plus(current_mpc,mpc_plus_1);

disp1 d1(op,dispatch1);
disp2 d2(op,dispatch2);

endmodule
```

利用 4 选 1 多路选择器选择下一时刻的 mpc 值，分别是返回 0 地址取指令，4 种类型指令的选择，lw 和 sw 指令的选择和当前 mpc 下一组控制信号。通过 disp1 和 disp2 模块输入指令类型 op 来判断 dispatch1 和 dispatch2 的选择。

## 三、 实验过程和数据记录

```
module tester;

    // Inputs
    reg clk;
    reg mode;
    reg exec;
    reg rst;
    reg [1:0] disp_type;
    reg [4:0] reg_index;

    // Outputs
    wire [7:0] led;
    wire [7:0] segment;
    wire [3:0] anode;
    wire multi_clk;
    wire [31:0] pc;
    wire [31:0] inst_reg;
    wire [31:0] memdata_reg;
    wire [31:0] A;
    wire [31:0] B;
    wire [31:0] ALUout;
    wire [31:0] Rdata1;
    wire [31:0] Rdata2;
    wire [31:0] clock;
    wire [15:0] disp_num;
    wire finish;
    wire [31:0] mem_out;
    wire [31:0] Rdata3;
    wire [31:0] mem_addr;
    wire [4:0] reg_dst;
```

```
    wire [31:0] result_or_data;
    wire [31:0] ALUop1;
    wire [31:0] ALUop2;
    wire [31:0] PC_next;
    wire [31:0] branch_exten;
    wire clk_1s;
    wire [3:0] current_mpc;
    wire [31:0] ALUresult;
    wire [2:0] ALUsignal;
    wire zero;
    wire branch_or_not;
    wire pc_change;
    wire exec_out;
    wire rst_out;
    wire [17:0] control;
    wire PCWrite;
    wire PCWriteCond;
    wire lorD;
    wire MemRead;
    wire MemWrite;
    wire IRWrite;
    wire MemToReg;
    wire ALUsrcA;
    wire RegWrite;
    wire RegDst;
    wire [1:0] PCSource;
    wire [1:0] ALUop;
    wire [1:0] ALUsrcB;
    wire [31:0] register1;
    wire [31:0] register2;
    wire [31:0] register3;
    wire [31:0] register4;
    wire [31:0] register5;
    wire [31:0] register6;

    // Instantiate the Unit Under Test (UUT)
    Micro_CPU uut (
        .clk(clk),
        .mode(mode),
        .exec(exec),
        .rst(rst),
        .disp_type(disp_type),
        .reg_index(reg_index),
        .led(led),
        .segment(segment),
        .anode(anode),
        .multi_clk(multi_clk),
        .pc(pc),
        .inst_reg(inst_reg),
        .memdata_reg(memdata_reg),
        .A(A),
        .B(B),
        .ALUout(ALUout),
        .Rdata1(Rdata1),
        .Rdata2(Rdata2),
        .clock(clock),
        .disp_num(disp_num),
        .finish(finish),
        .mem_out(mem_out),
        .Rdata3(Rdata3),
        .mem_addr(mem_addr),
```

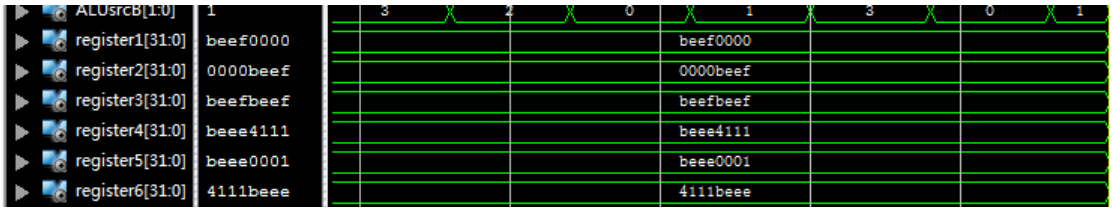
```
        .reg_dst(reg_dst),
        .result_or_data(result_or_data),
        .ALUop1(ALUop1),
        .ALUop2(ALUop2),
        .PC_next(PC_next),
        .branch_exten(branch_exten),
        .clk_1s(clk_1s),
        .current_mpc(current_mpc),
        .ALUresult(ALUresult),
        .ALUsignal(ALUsignal),
        .zero(zero),
        .branch_or_not(branch_or_not),
        .pc_change(pc_change),
        .exec_out(exec_out),
        .rst_out(rst_out),
        .control(control),
        .PCWrite(PCWrite),
        .PCWriteCond(PCWriteCond),
        .lorD(lorD),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .IRWrite(IRWrite),
        .MemToReg(MemToReg),
        .ALUsrcA(ALUsrcA),
        .RegWrite(RegWrite),
        .RegDst(RegDst),
        .PCSource(PCSource),
        .ALUop(ALUop),
        .ALUsrcB(ALUsrcB),
        .register1(register1),
        .register2(register2),
        .register3(register3),
        .register4(register4),
        .register5(register5),
        .register6(register6)
    );
   initial forever begin
    #2;
    clk=~clk;
    end
    initial forever begin
    #20;
    exec=~exec;
    end
    initial begin
        // Initialize Inputs
        clk = 0;
        mode = 0;
        exec = 0;
        rst = 1;
        disp_type = 0;
        reg_index = 0;
      #2;
        rst=0;
        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here

    end
```

```
endmodule
```

测试几乎包括了多路选择器实现过程中所有的控制信号。

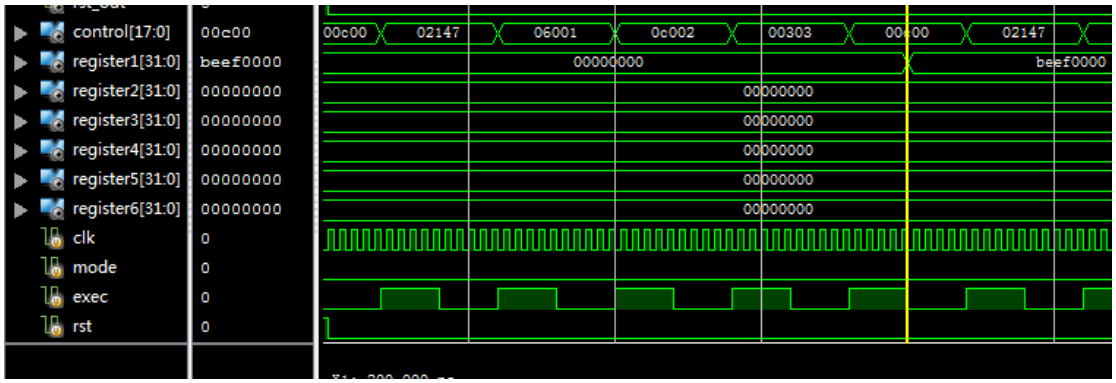
下面是仿真结果：

剔除一些无关的仿真信号后，

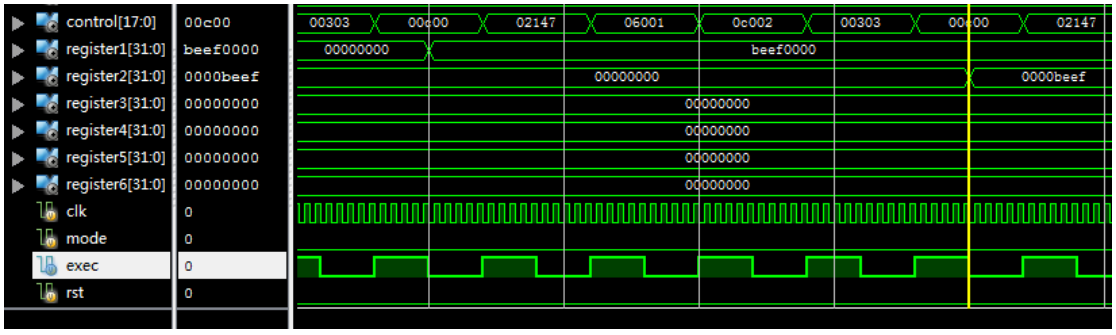



在最后，我们可以发现 6 个目标寄存器中存储了和预期结果完全一样的值。

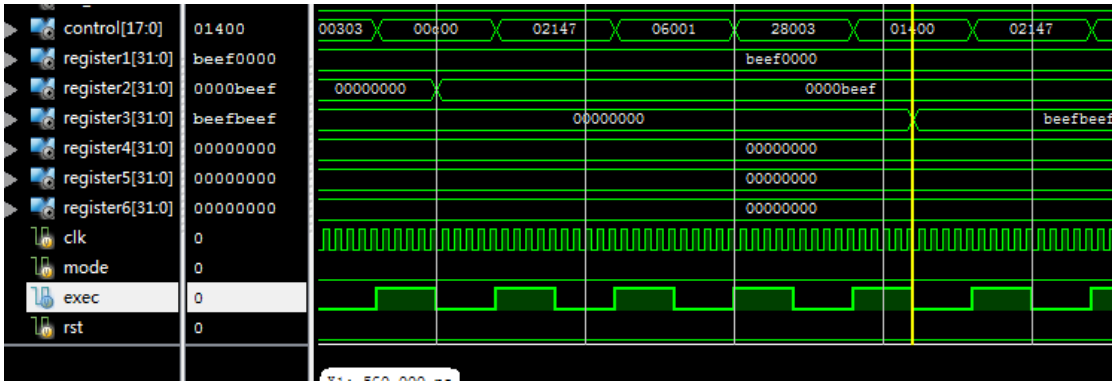
下面是具体实现过程的一些截图：



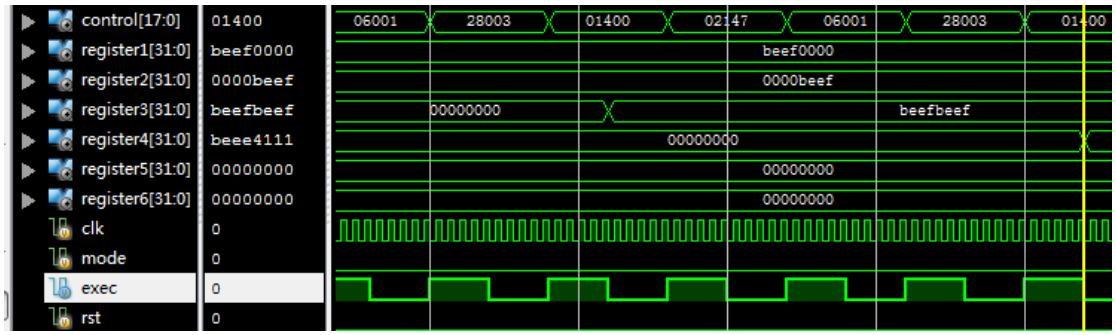

以上是第一条指令的仿真结果，控制信号完全符合预期，并在第五个时钟周期结束时成功写入数据。



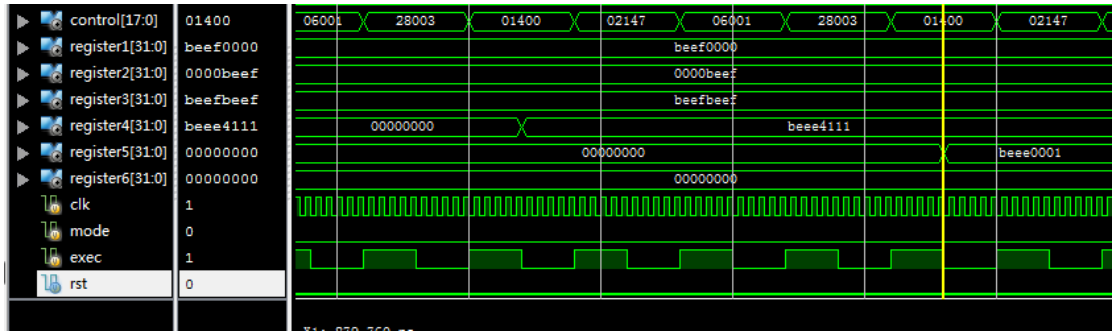

第二条指令，



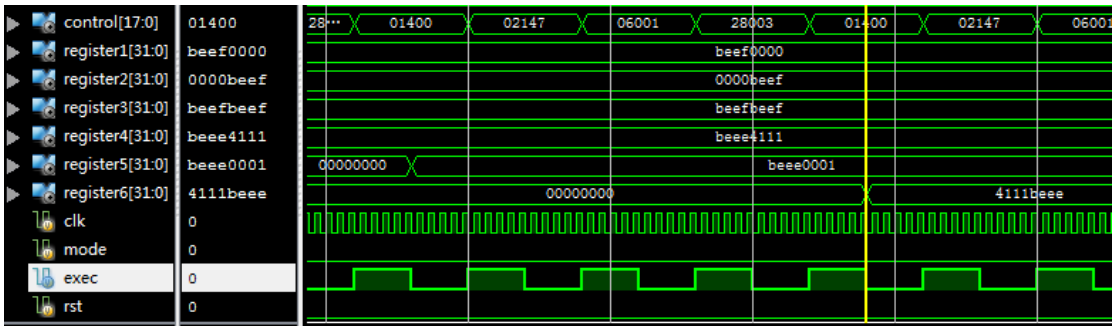
第三条指令




第四条




第五条




第六条

四、 实验结果

由于本次实验只是在上次多周期 CPU 的实验基础上替换掉控制器模块，而微指令控制模块并不复杂。因此，除了在仿真阶段遇到一点问题外，仿真成功后实验过程还是很顺利的，没有出现很大的错误。

五、 讨论与心得

微指令实现给 CPU 的设计提供了很多的思路。毕竟控制信号和指令类型的数量在实际应用中都是很庞大的，因此，在实际的设计中，微指令的意义要远远高出利用状态控制器设计出的多周期 CPU。相信本次实验将会为将来的学习和实践奠定深厚的基础。