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　10　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：Lab9： 多周期 CPU

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

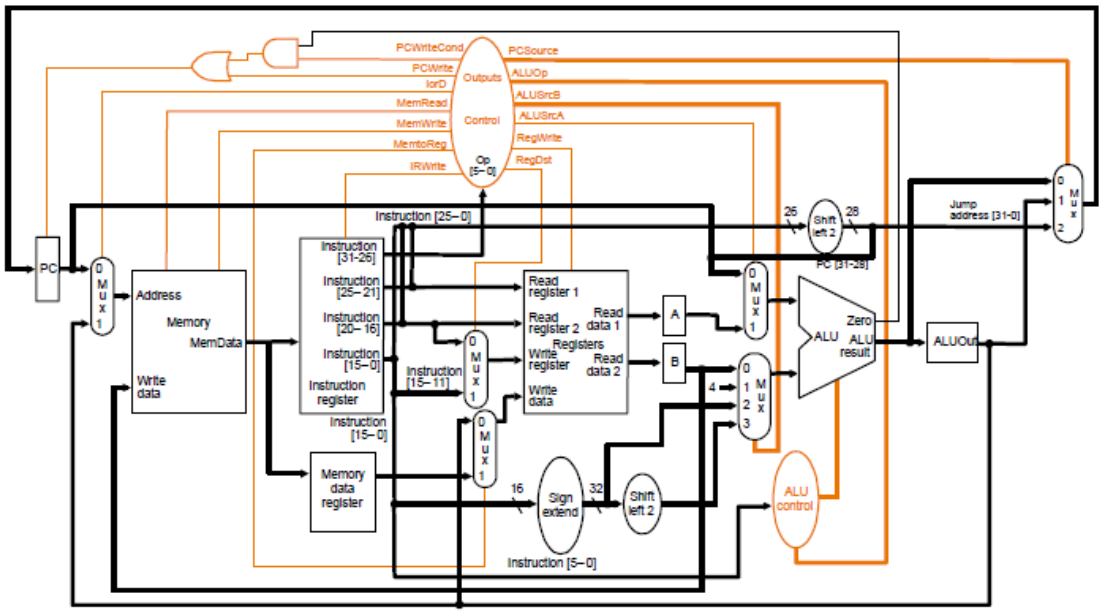
实验地点：东 4-509 实验日期：2014 年 6 月 10 日

一、 实验目的和要求

1. 理解多周期 CPU 工作原理
2. 熟悉并掌握多周期 CPU 的数据通路
3. 实现多周期 CPU 数据通路

二、 实验内容和原理

本次多周期 CPU 的数据通路图如下：



而对于控制模块和状态机的控制信号与状态对应情况如下：

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

状态与指令的关系如下：

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

以上是理论原理，对于实验代码，在单周期 CPU 中的 ALU，ALUC，REGFILE，等模块均能直接使用，接下来重点介绍多周期与单周期 CPU 不同的部分：

首先是关于状态机的两个模块 state_control 和 state_now，前者用于输出下一状态，后者用于在时钟上升沿到来时切换状态，以下是 state_control 模块的代码：

```
module state_control(
                  input wire clk,
                  input wire[5:0] instruction,
                  input wire[3:0] current_state,
                         output reg[3:0] next_state,
                         output reg finish
   );
initial begin
finish <= 0;
end

always @* begin
case(current_state)
4'b0000 : begin
next_state<= 4'b0001;
end
4'b0001 : begin
case(instruction)
6'd35: begin
next state<= 4'b0010;

end
6'd43: begin
next_state<= 4'b0010;

end
6'd0: begin
 next_state<= 4'b0110;

end
6'd4: begin
 next_state<= 4'b1000;

end
6'd2:begin
  next_state<= 4'b1001;

end
endcase
end

4'b0010 : begin
if(instruction==6'd35)
next_state<= 4'b0011;
else if(instruction==6'd43)
next_state<= 4'b0101;
end

4'b0011 : next_state<= 4'b0100;
4'b0100 : begin
next_state<= 4'b0000;

end
4'b0101 :begin
 next_state<= 4'b0000;
```

```
 end
4'b0110 : next_state<= 4'b0111;
4'b0111 :begin
 next_state<= 4'b0000;

 end
4'b1000 :begin
 next_state<= 4'b0000;

 end
4'b1001 : begin
 next_state<= 4'b0000;

 end

endcase
end

endmodule
```

通过多个 case 语句的嵌套，根据状态机的状态表，根据输入的 current_state 信号和指令获得下一时刻 next_state 的值并输出。

下面是 state_now 模块：

```
module state_now(
           input wire finish,
           input wire multi_clk,
           input wire[3:0] next_state,
               output wire[3:0] current state
   );

reg [3:0] temp_state;

initial temp_state <= 0;

assign current_state = temp_state;

always @(posedge multi_clk)
temp_state = next_state;

endmodule
```

在 always 模块中，

```
always @(posedge multi_clk)
temp_state = next_state;
```

每当时钟上升沿到来时，将下一状态 next_state 赋值给 temp_state，而 temp_state 通过 assign 语句和 current_state 永远相连。由此，状态机的工作就完成了。

下面是控制器 control 模块的代码：

```
module control(
             input wire[3:0] current_state,
                  output wire PCWrite,
                              PCWriteCond,
                                  lorD,
                                  MemRead,
                                  MemWrite,
                                  IRWrite,
                                  MemToReg,
                                  ALUsrcA,
                                  RegWrite,
                                  RegDst,
                  output wire[1:0] PCSource,
                                  ALUop,
                                       ALUsrcB
    );
assign PCWrite = ((current_state==0)|(current_state==9));
assign PCWriteCond = (current_state==8);
assign lorD = ((current_state==3)|(current_state==5));
assign MemRead = ((current_state==0)|(current_state==3));
assign MemWrite = (current_state==5);
assign IRWrite = (current_state==0);
assign MemToReg = (current_state==4);
assign                            ALUsrcA                                  =
((current state==2)|(current state==6)|(current state==8));
assign RegWrite = ((current_state==4)|(current_state==7));
assign RegDst =(current_state==7);

assign PCSource[1] = (current_state==9);
assign PCSource[0] = (current_state==8);
assign ALUop[1] = (current_state==6);
assign ALUop[0] = (current_state==8);
assign ALUsrcB[1] = ((current_state==1)|(current_state==2));
assign ALUsrcB[0] = ((current_state==0)|(current_state==1));


endmodule
```

利用 assign 语句将各个控制信号利用逻辑关系，在相应要置为 1 的所有状态下将其置为 1，如：

```
assign PCWrite = ((current_state==0)|(current_state==9));
```

中，PCWrite 在状态 0 和状态 9 的时候置为 1.current_state 表示当前状态

由于本次实验具有自动运行功能，所以添加了一个 1 秒的时钟，具体原理与 1 毫秒的时钟相同，不做赘述。

控制 PC 变化的 PC 模块与单周期 CPU 的模块基本相同，只多出了一个 PC_change 信号来控制 PC 的值是否要发生变化，具体如下：

```
module pc(      input wire pc_change,
               input wire clk,
                    input wire rst,
                    input wire[31:0] i_pc,
                    output wire[31:0] o_pc
    );
reg[31:0] t_pc;

initial t_pc = 0;
assign o_pc = rst?0:t_pc;

always @(posedge clk,posedge rst)
if(rst) t_pc <= 0;
else if(pc_change)
t_pc <= i_pc;


endmodule
```

仅在 PC_change 值为 1 时，当前 PC 才会变为下一个 PC。

最后是 top 顶层模块的连线：

```
module mul_CPU(
          input wire clk,
             input wire mode,   //自动运行或手动运行
             input wire exec,
          input wire rst,
          input wire[1:0] disp_type,
             input wire[4:0] reg_index,
             output wire[7:0] led,
             output wire[7:0] segment,
             output wire[3:0] anode


    );
wire multi_clk;   //控制运行的时钟，由 mode 选择
wire[31:0] pc;
reg[31:0] inst_reg;  //存储指令的寄存器
reg[31:0] memdata_reg; //存储内存中输出数据的寄存器
wire[31:0] A,B;
reg[31:0] ALUout;   //存储 ALU 计算结果的寄存器
reg[31:0] Rdata1,Rdata2; //存储寄存器中读取的两个数据的寄存器


reg[31:0] clock;
reg[15:0] disp_num;
wire finish;

wire[31:0] mem_out;  //寄存器中读出的数据
wire[31:0] Rdata3;   //用于验收

wire[31:0] mem_addr; //内存地址
wire[4:0] reg_dst;   //选择后的寄存器写地址
wire[31:0] result_or_data; //选择后的寄存器写数据来源
wire[31:0] ALUop1,ALUop2;  //两个 ALU 操作数
wire[31:0] PC_next;    //下一个 PC

wire[31:0] branch_exten;
```

```
wire clk_1s; //1s 时钟信号
wire[3:0] next_state,current_state; //当前的状态和下一个状态
wire[31:0] ALUresult; //ALU 计算结果
wire[2:0] ALUsignal;  //ALU 控制信号
wire zero;            //0 信号
wire branch_or_not;   //用于判断是否 branch
wire pc_change;       //用于判断 PC 值是否要变化
wire exec_out,rst_out; //去抖

wire
PCWrite,PCWriteCond,lorD,MemRead,MemWrite,IRWrite,MemToReg,ALUsrc
A,RegWrite,RegDst; //控制信号
wire[1:0] PCSource,ALUop,ALUsrcB;

initial begin

clock = 0;
end

pbdebounce p1(clk,exec,exec_out);
pbdebounce p2(clk,rst,rst_out);

assign led[1:0] = (inst_reg[31:26]==0)?0:
                ((inst_reg[31:26]==35)||(inst_reg[31:26]==43))?1:
                      (inst_reg[31:26]==4)?2:
                      (inst_reg[31:26]==2)?3:
assign led[4:2] = current_state;
assign led[5] = (rst_out);
assign led[6] = (mode);
assign led[7] = (~mode);

data
m1(.addr(mem_addr[8:0]),.clk(clk),.din(Rdata2),.dout(mem_out),.we
(MemWrite));


timer_1s t1(clk,clk_1s);

pc pp1(pc_change,multi_clk,rst_out,PC_next,pc);

state_now s1(finish,multi_clk,next_state,current_state);
state_control
s2(multi_clk,inst_reg[31:26],current_state,next_state,finish);

control
c1(current_state,PCWrite,PCWriteCond,lorD,MemRead,MemWrite,IRWrit
e,MemToReg,ALUsrcA,RegWrite,RegDst,PCSource,ALUop,ALUsrcB);

regfile
r1(multi_clk,rst_out,RegWrite,inst_reg[25:21],inst_reg[20:16],reg
_index,reg_dst,result_or_data,A,B,Rdata3);

aluc a1(ALUop,inst_reg[5:0],ALUsignal);
alu  a2(ALUop1,ALUop2,ALUsignal,zero,ALUresult);

mux mu0(exec_out,clk_1s,multi_clk,mode);
mux_32 mu1(pc,ALUout,0,0,mem_addr,{1'b0,lorD});
mux_5
mu2(inst_reg[20:16],inst_reg[15:11],0,0,reg_dst,{1'b0,RegDst});
mux_32 mu3(ALUout,memdata_reg,0,0,result_or_data,{1'b0,MemToReg});
```

```
mux_32 mu4(pc,Rdata1,0,0,ALUop1,{1'b0,ALUsrcA});
assign branch_exten = {{16{inst_reg[15]}},inst_reg[15:0]};
mux_32 m6(Rdata2,1,branch_exten,branch_exten,ALUop2,ALUsrcB);
mux_32 m7(ALUresult,ALUout,inst_reg[25:0],0,PC_next,PCSource);

and and1(branch_or_not,zero,PCWriteCond);
or  or1(pc_change,branch_or_not,PCWrite);


always @* begin
if(IRWrite) inst_reg <= mem_out;
end

always @(negedge multi_clk) begin
ALUout <= ALUresult;
Rdata1 <= A;
Rdata2 <= B;
end

display d1(clk,disp_num,anode,segment);

always @* begin
case(disp_type)
2'b00:
disp_num<= Rdata3[15:0];
2'b01:
disp_num<= Rdata3[31:16];
2'b10:
disp_num<= pc;
2'b11:
disp_num<= clcok;
endcase
end

always @(negedge multi_clk)
memdata_reg <= mem_out;




always@(posedge multi_clk,posedge rst_out) begin
if(rst_out) clock=0;
else if(~finish) clock = clock + 1;
end

endmodule
```

相较于单周期 CPU，多周期 CPU 由于要许多个周期才能完成一条指令，并且控制信号增多，变量也相应的增加了不少，各个变量和寄存器的含义以及作用已经在代码中标注。

根据 mode 的变化，在手动的 exec 信号和 clk_1s 信号间选出代表周期的时钟信号，用 multi_clk 表示：

```
mux mu0(exec_out,clk_1s,multi_clk,mode);
```

并在各个模块中传递该信号，完成模式的切换

在指令变化的操作中：

```
always @* begin
if(IRWrite) inst_reg <= mem_out;
end
```

当 IRWrite 信号出现高电平时，马上改变指令寄存器的内容

将其余模块全部按照数据通路连接好之后，就可以正常工作了。

本次实验中的内存的 IP 核是指令和数据公用的，COE 文件内容为：

```
MEMORY_INITIALIZATION_RADIX = 16;
MEMORY_INITIALIZATION_VECTOR =
8c010014,
8c020015,
00221820,
00222022,
00642824,
00853027,
ac060016,
08000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
beef0000,
0000beef,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000,
00000000;
```

UCF 文件内容为：

```
NET"clk" LOC="T9";

NET"exec" LOC="M13";

NET"mode" LOC="K13";

NET "anode[0]" LOC="D14";
NET "anode[1]" LOC="G14";
NET "anode[2]" LOC="F14";
NET "anode[3]" LOC="E13";

NET "segment[0]" LOC="E14";
NET "segment[1]" LOC="G13";
NET "segment[2]" LOC="N15";
NET "segment[3]" LOC="P15";
```

```
NET "segment[4]" LOC="R16";
NET "segment[5]" LOC="F13";
NET "segment[6]" LOC="N16";
NET "segment[7]" LOC="P16";



NET"rst" LOC="M14";
NET"led[0]" LOC="K12";
NET"led[1]" LOC="P14";
NET"led[2]" LOC="L12";
NET"led[3]" LOC="N14";
NET"led[4]" LOC="P13";
NET"led[5]" LOC="N12";
NET"led[6]" LOC="P12";
NET"led[7]" LOC="P11";


NET"disp_type[0]" LOC="J13";
NET"disp_type[1]" LOC="K14";

NET"reg_index[0]" LOC="F14";
NET"reg_index[1]" LOC="G13";
NET"reg_index[2]" LOC="H14";
NET"reg_index[3]" LOC="H13";
NET"reg_index[4]" LOC="J14";

NET "rst" CLOCK_DEDICATED_ROUTE = FALSE;
NET "exec" CLOCK_DEDICATED_ROUTE = FALSE;
```

三、 实验过程和数据记录

将 mode 开关拨到自动挡位，将显示的状态切换到 clock，发现每隔 1s 后 clock 的值会加 1，并在 clock 的值分别为 4,9,13,17,21,25 时，寄存器 1,2,3,4,5,6 的值分别变为 BEEF0000，0000BEEF，BEEFBEEF，BEEE4111，BEEE0001，4111BEEE

四、 实验结果

本次实验的过程中还是出现了一些问题的。比如某些写操作的触发是在时钟的上升沿进行的，导致数据还没读取，写入错误。比如某些寄存器的赋值是在任意情况下进行的，但是这种赋值并不是在任何时刻都可以进行的。过多的使用*会出现错误，目前原因不明。

不过，最后的运行结果还是完全准确的，整个数据通路能够很好的完成指定的工作。

五、 讨论与心得

本次实验让我在单周期 CPU 的基础上了解到了另外一种实现 CPU 的方式，由于有了单周期 CPU 实现的经验作为基础，这次实现起来虽然工作量也不小，但是已经没有上次那么困难了。主要难点还是对于新知识的理论掌握不

够准确，以至于在实现过程中因为一些小问题导致整个 CPU 的工作出现崩溃性的问题，需要慢慢仿真并调试。本次实验也为下一次的实现奠定了一定的经验性基础，也加深了我理论方面知识的熟练程度。