

**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

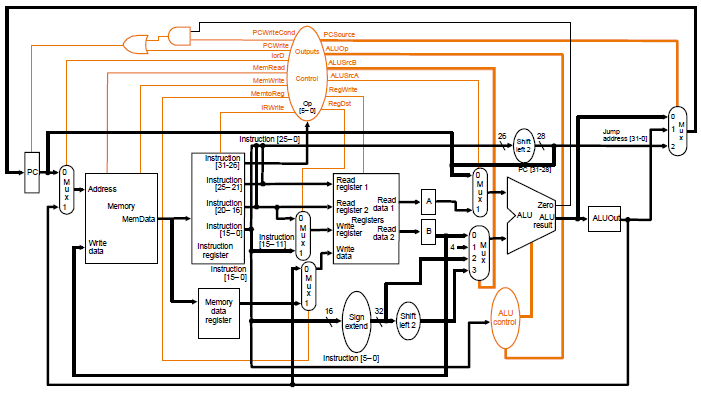
学号： 3120102116

同组学生姓名： 王艺 指导老师： 姜晓红

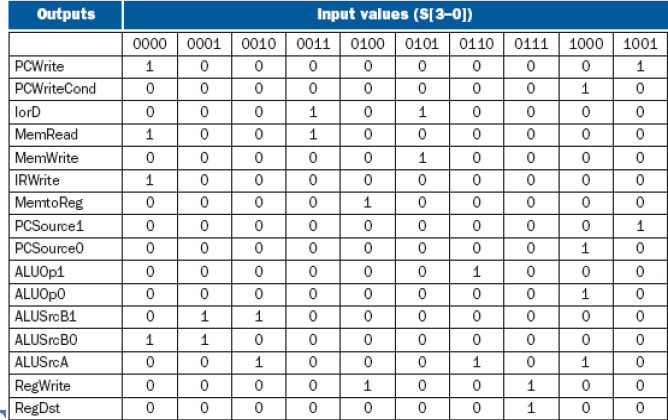
实验地点： 东4-509 实验日期： 2014 年 6月 10 日

1. 实验目的和要求
2. 理解多周期CPU工作原理
3. 熟悉并掌握多周期CPU的数据通路
4. 实现多周期CPU数据通路
5. 实验内容和原理

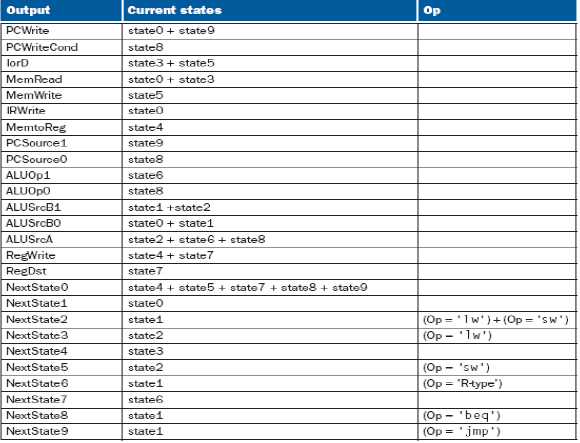
本次多周期CPU的数据通路图如下：



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：



状态与指令的关系如下：



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,ALUsrcA,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,IRWrite,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,  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; |

1. 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

1. 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。

1. 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。