Lab4实验报告

1、系统框图

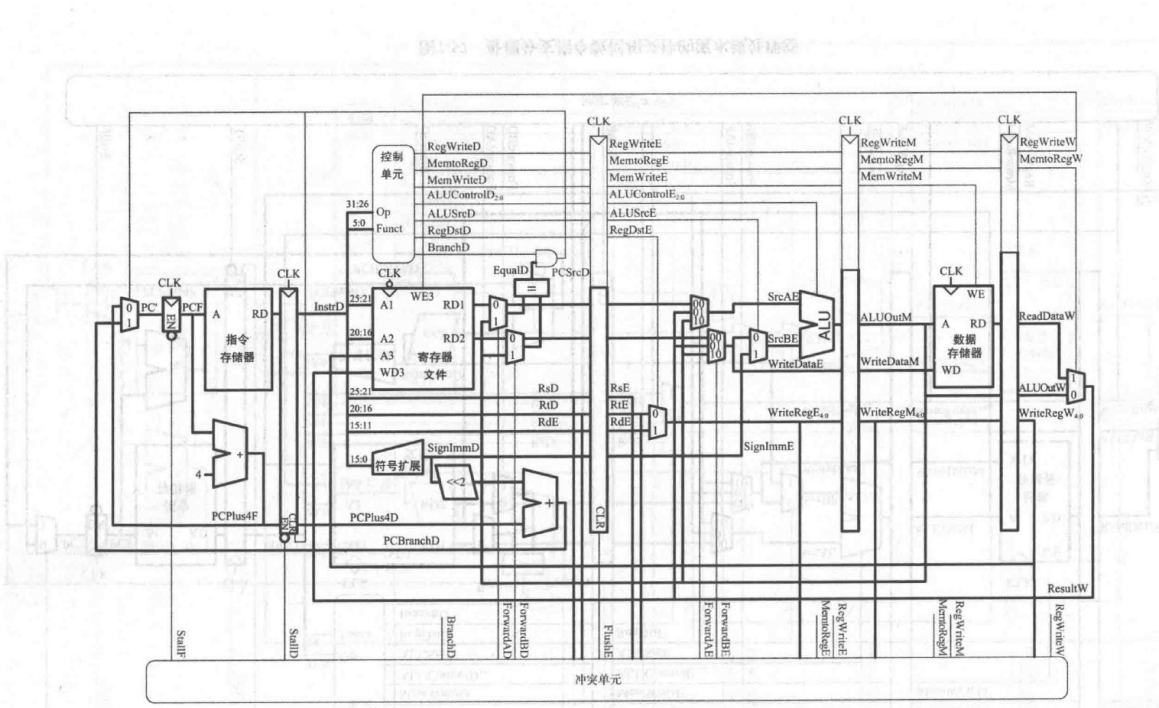


图7-58　处理所有冲突的流水线处理器

2、设计思路

五级流水线cpu的设计思路和多周期cpu的设计十分类似，都是将一条指令分步执行。不同的是流水线cpu采取的是多条指令并行计算的方法，使得指令吞吐率大大提高，cpi变为1；但是流水线cpu的控制信号设计却和单周期cpu十分类似，一条指令执行的过程中控制信号维持不变，只是在不同的阶段使用控制信号中的不同部分，而且之后阶段使用的控制信号不能跨阶段传输，只能够流水线一步一步向后传输。此外各个部件的设计也出现了一点小小的变化，因为比如寄存器的读写需要看作是不同的两个部件，需要进行一点点修改。控制信号的传递也不仅仅是组合电路，需要在每一个阶段用寄存器存储，并将需要的控制信号发送到下一个阶段.最重要的是还需要解决流水线冲突问题，此次的lab我解决了**数据冒险、结构冒险和控制冒险**的问题

1. **带有清零端的寄存器**

```
module flopenrc #(parameter WIDTH = 8)
(input clk, reset,
input en, clear,
input [WIDTH-1:0] d,
output reg [WIDTH-1:0] q);
always @(posedge clk, posedge reset)
if (reset) q <= #1 0;
else if (clear) q <= #1 0;
else if (en) q <= #1 d;
endmodule
```

2. **最为重要的部件：冲突检测处理部件conflict**

```
module conflict(input [4:0] rsD, rtD, rsE, rtE,
input [4:0] writeregE,
writeregM, writeregW,
input regwriteE, regwriteM,
regwriteW,
input memtoregE, memtoregM, branchD,
output forwardaD, forwardbD,
output reg [1:0] forwardaE, forwardbE,
output stallF, stallD, flushE);
wire lwstallD, branchstallD;
assign forwardaD = (rsD !=0 & rsD == writeregM &
regwriteM);
assign forwardbD = (rtD !=0 & rtD == writeregM &
regwriteM);
always @(*)
begin
forwardaE = 2'b00; forwardbE = 2'b00;
if (rsE != 0)
if (rsE == writeregM & regwriteM)
forwardaE = 2'b10;
else if (rsE == writeregW & regwriteW)
forwardaE = 2'b01;
if (rtE != 0)
if (rtE == writeregM & regwriteM)
forwardbE = 2'b10;
else if (rtE == writeregW & regwriteW)
forwardbE = 2'b01;
end
assign #1 lwstallD = memtoregE &
(rtE == rsD | rtE == rtD);
assign #1 branchstallD = branchD &
(regwriteE &
(writeregE == rsD | writeregE == rtD) |
memtoregM &
(writeregM == rsD | writeregM == rtD));
assign #1 stallD = lwstallD | branchstallD;
assign #1 stallF = stallD;
assign #1 flushE = stallD;
endmodule
```

详细说明：hazard模块主要处理了四种冲突的情况：**第一种，**上一条指令**（非lw）**的运算结果在这一条需要进行取数计算，解决方式是将alu的运算结果数据转发到下一条指令的alu入口；**第二种，**上上条的指令**（包括lw）**的运算结果需要取数计算，解决方式是将mem阶段结束后读出的数据转发到alu入口；**第三种，**就是loaduse冲突，在只有到mem阶段结束才能够得到的数据在下一条指令的exec阶段开始的时候就需要读入，所以需要阻塞一个周期，清空下一条指令的IF流水线寄存器，相当于插入一条nop指令，然后按照上一种情况进行处理；**第四种，**beq指令进行比较的两个寄存器是上一条或者是上上条指令写入数据的目标寄存器，则将比较过程提前到ID取数译码阶段，然后阻塞一个周期，清空寄存器,同时将计算的结果转发到ID阶段进行比较，将跳转到的pc地址发送到pc写入口等待下一个周期写入

3. **控制信号controller**

```
module controller(input clk, reset,
input [5:0] opD, functD,
input flushE, equalD,
output memtoregE, memtoregM,
```

```
output memtoregW, memwriteM,
output pcsrcD, branchD, alusrcE,
output regdstE, regwriteE,
output regwriteM, regwriteW,
output jumpD,
output [2:0] alucontrolE);
wire [1:0] aluopD;
wire memtoregD, memwriteD, alusrcD,
regdstD, regwriteD;
wire [2:0] alucontrolD;
wire memwriteE;
maindec md(opD, memtoregD, memwriteD, branchD,
alusrcD, regdstD, regwriteD, jumpD,
aluopD);
aludec ad(functD, aluopD, alucontrolD);
assign pcsrcD = branchD & equalD;
// pipeline register
floprc #(8) regE(clk, reset, flushE,
{memtoregD, memwriteD, alusrcD,
regdstD, regwriteD, alucontrolD},
{memtoregE, memwriteE, alusrcE,
regdstE, regwriteE, alucontrolE});
flopr #(3) regM(clk, reset,
{memtoregE, memwriteE, regwriteE},
{memtoregM, memwriteM, regwriteM});
flopr #(2) regW(clk, reset,
{memtoregM, regwriteM},
{memtoregW, regwriteW});
endmodule
```

详细说明：与多周期cpu使用有限状态机来表示一条指令在各个阶段的控制信号不同，流水线cpu使用的控制信号在一条指令执行的5个阶段都是相同的，但是又和单周期cpu的组合电路表示控制信号又有一些区别。流水线cpu的控制信号需要一步步向下传递，后面的阶段需要使用到的控制信号都需要存在寄存器中，使用完毕的控制信号可以丢弃

4. **数据通路datapath**

```
module datapath(input clk, reset,
input memtoregE, memtoregM,
memtoregW,
input pcsrcD, branchD,
input alusrcE, regdstE,
input regwriteE, regwriteM,
regwriteW,
input MemWriteM,
input jumpD,
input [2:0] alucontrolE,
output equalD,
output [31:0] pcF,
input [31:0] instrF,
output [31:0] aluoutM, writedataM,
input [31:0] readdataM,
output [5:0] opD, functD,
output flushE);
wire forwardaD, forwardbD,stallD;
wire [1:0] forwardaE, forwardbE;
wire stallF;
```

```
wire [4:0] rsD, rtD, rdD, rsE, rtE, rdE;
wire [4:0] writeregE, writeregM, writeregW;
wire flushD;
wire [31:0] pcnextFD, pcnextbrFD, pcplus4F,
pcbranchD;
wire [31:0] signimmD, signimmE, signimmshD;
wire [31:0] srcaD, srca2D, srcaE, srca2E;
wire [31:0] srcbD, srcb2D, srcbE, srcb2E, srcb3E;
wire [31:0] pcplus4D, instrD;
wire [31:0] aluoutE, aluoutW;
wire [31:0] readdataW, resultW;
// conflict
hazard h(rsD, rtD, rsE, rtE, writeregE, writeregM,
writeregW,regwriteE, regwriteM, regwriteW,
memtoregE, memtoregM, branchD,
forwardaD, forwardbD, forwardaE,
forwardbE,
stallF, stallD, flushE);
// IF
mux2 #(32) pcbrmux(pcplus4F, pcbranchD, pcsrcD,
pcnextbrFD);
mux2 #(32) pcmux(pcnextbrFD,{pcplus4D[31:28],
instrD[25:0], 2'b00},
jumpD, pcnextFD);
// ID
regfile rf(clk, regwriteW, rsD, rtD, writeregW,
resultW, srcaD, srcbD);
flopenr #(32) pcreg(clk, reset, ~stallF,
pcnextFD, pcF);
adder pcadd1(pcF, 32'b100, pcplus4F);
flopenr #(32) r1D(clk, reset, ~stallD, pcplus4F,
pcplus4D);
flopenrc #(32) r2D(clk, reset, ~stallD, flushD, instrF,
instrD);
signext se(instrD[15:0], signimmD);
sl2 immsh(signimmD, signimmshD);
adder pcadd2(pcplus4D, signimmshD, pcbranchD);
mux2 #(32) forwardadmux(srcaD, aluoutM, forwardaD,
srca2D);
mux2 #(32) forwardbdmux(srcbD, aluoutM, forwardbD,
srcb2D);
eqcmp comp(srca2D, srcb2D, equalD);
assign opD = instrD[31:26];
assign functD = instrD[5:0];
assign rsD = instrD[25:21];
assign rtD = instrD[20:16];
assign rdD = instrD[15:11];
assign flushD = pcsrcD | jumpD;
// EXEC
floprc #(32) r1E(clk, reset, flushE, srcaD, srcaE);
floprc #(32) r2E(clk, reset, flushE, srcbD, srcbE);
floprc #(32) r3E(clk, reset, flushE, signimmD, signimmE);
floprc #(5) r4E(clk, reset, flushE, rsD, rsE);
floprc #(5) r5E(clk, reset, flushE, rtD, rtE);
floprc #(5) r6E(clk, reset, flushE, rdD, rdE);
mux3 #(32) forwardaemux(srcaE, resultW, aluoutM,
forwardaE, srca2E);
mux3 #(32) forwardbemux(srcbE, resultW, aluoutM,
```

```
forwardbE, srcb2E);
mux2 #(32) srcbmux(srcb2E, signimmE, alusrcE,
srcb3E);
alu alu(srca2E, srcb3E, alucontrolE, aluoutE);
mux2 #(5) wrmux(rtE, rdE, regdstE, writeregE);
// Mem
flopr #(32) r1M(clk, reset, srcb2E, writedataM);
flopr #(32) r2M(clk, reset, aluoutE, aluoutM);
flopr #(5) r3M(clk, reset, writeregE, writeregM);
// Write
flopr #(32) r1W(clk, reset, aluoutM, aluoutW);
flopr #(32) r2W(clk, reset, readdataM, readdataW);
flopr #(5) r3W(clk, reset, writeregM, writeregW);
mux2 #(32) resmux(aluoutW, readdataW, memtoregW,
resultW);
endmodule
```

详细说明：由于加入了数据冒险和结构冒险的解决，在数据通路上的设计也发生了少许改变。**首先是alu的输入需要加一个三选一的复用器**，分别是没有冲突的时候，与上一条指令的目标寄存器冲突，还有和上上条指令的目标寄存器冲突的时候的数据转发；还有就是在译码取数阶段判断是否满足跳转的条件，以及计算跳转的地址，而且在判断是否相等的时候也需要有数据转发，所以在**比较寄存器的取数的时候也加了两个二选一的数据复用器**

5. 其余部件都和多周期cpu以及单周期cpu的部件几乎一致，这里就不多加解释了

3、方案说明

1. **控制信号真值表：**

	regwrite	regdst	aluop	branch	memwrite	memtoreg	jump
add	1	1	add	0	0	0	0
sub	1	1	sub	0	0	0	0
and	1	1	add	0	0	0	0
or	1	1	or	0	0	0	0
slt	1	1	slt	0	0	0	0
addi	1	0	add	0	0	0	0
lw	1	1	add	0	0	1	0
sw	0	x	add	0	1	x	0
beq	0	x	sub	1	0	x	0
j	0	x	xxx	0	0	x	1

2. **结构冒险解决**

   结构冒险的解决方式相对比较简单，只需要保证在一条指令执行的5个时钟周期内，不使用一个部件多次，也不提前或者延后使用某一个部件，就不会造成多条指令执行过程中遇到同一时钟周期使用同一个硬件部件的情况的发生。

3. **数据冒险解决**

主要是使用数据转发的方式解决，除非是遇到了load-use数据冲突在不得已的情况下才使用阻塞一个周期的方式解决

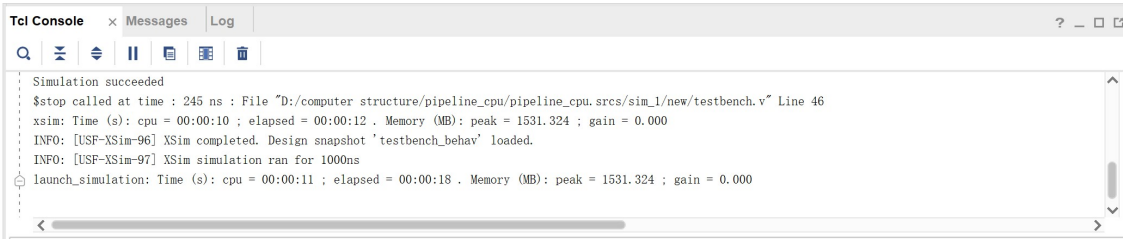
4. **控制冒险解决**

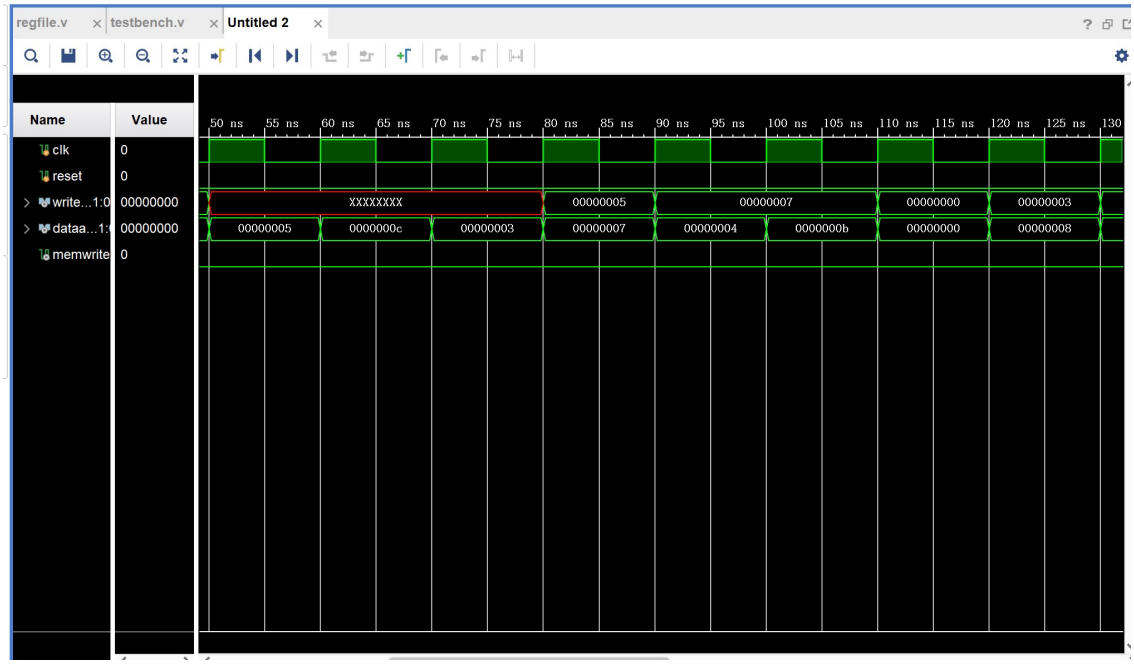
   主要是针对于beq，bne，j等等这类跳转指令而言的，采用将比较操作提前到ID译码取数阶段完成的方式，使得本来需要阻塞三个周期的操作减少到只有一个周期

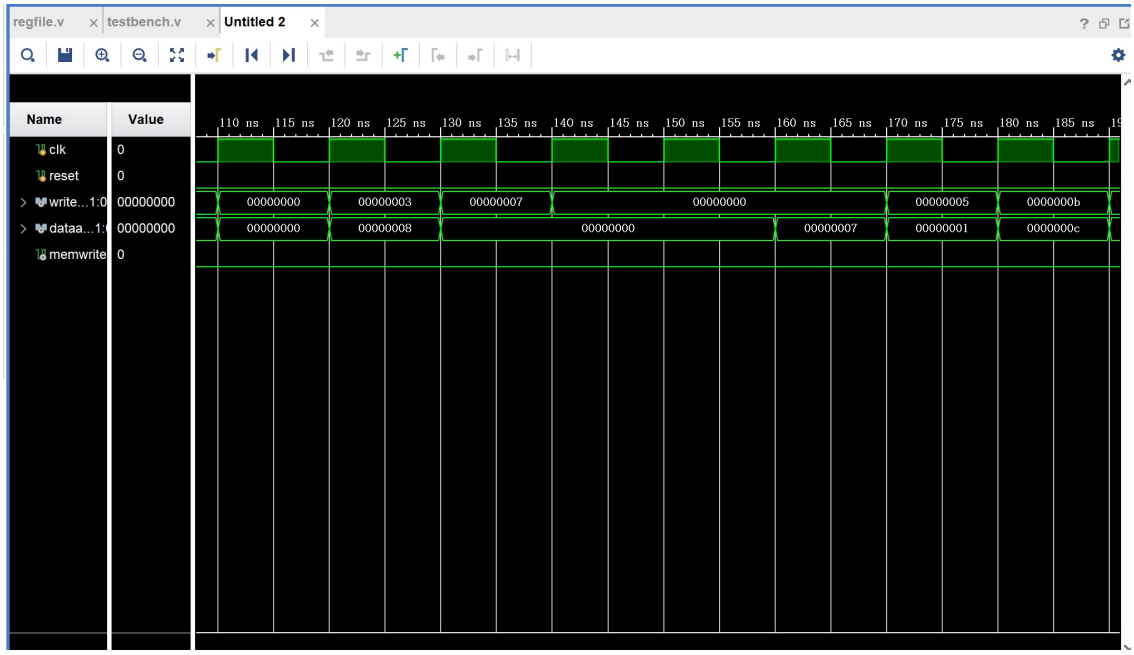
5. **仿真测试代码**：主要使用了书上的测试代码

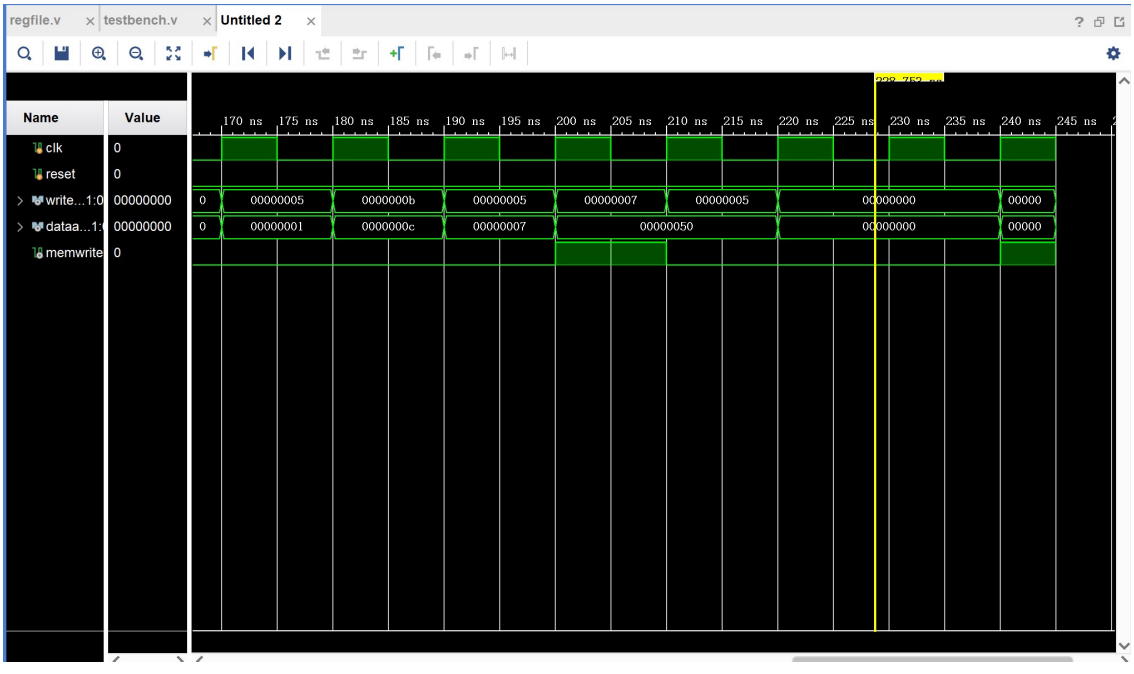
```
module testbench();
reg clk;
reg reset;
wire [31:0] writedata, dataadr;
wire memwrite;
top dut(clk, reset, writedata, dataadr, memwrite);
initial
begin
reset <= 1; # 22; reset <= 0;
end
always
begin
clk <= 1; # 5; clk <= 0; # 5;
end
always@(negedge clk)
begin
if(memwrite) begin
if(dataadr === 84 & writedata === 7) begin
$display("Simulation succeeded");
$stop;
end else if (dataadr !== 80) begin
$display("Simulation failed");
$stop;
end
end
end
endmodule
```

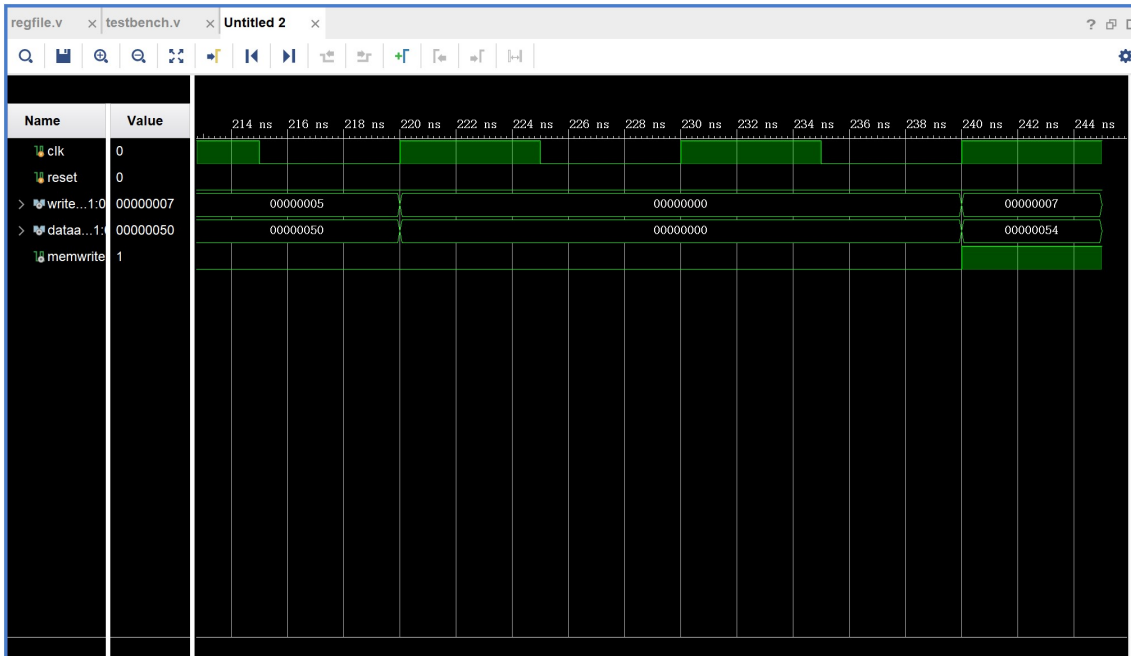
仿真结果：




regfile.v
testbench.v
Untitled 2
Name
Value
clk
0
reset
0
write...1:0
00000000
dataa...1:
00000000
memwrite
0
50 ns 55 ns 60 ns 65 ns 70 ns 75 ns 80 ns 85 ns 90 ns 95 ns 100 ns 105 ns 110 ns 115 ns 120 ns 125 ns 130
XXXXXXXX
00000005
00000007
00000000
00000003
00000005
0000000c
00000003
00000007
00000004
0000000b
00000000
00000008


regfile.v
testbench.v
Untitled 2
Name
Value
clk
0
reset
0
write...1:0
00000000
dataa...1:
00000000
memwrite
0
110 ns 115 ns 120 ns 125 ns 130 ns 135 ns 140 ns 145 ns 150 ns 155 ns 160 ns 165 ns 170 ns 175 ns 180 ns 185 ns
00000000
00000003
00000007
00000000
00000005
0000000b
00000000
00000008
00000000
00000007
00000001
0000000c


regfile.v
testbench.v
Untitled 2
Name
Value
clk
0
reset
0
write...1:0
00000000
dataa...1:
00000000
memwrite
0
170 ns 175 ns 180 ns 185 ns 190 ns 195 ns 200 ns 205 ns 210 ns 215 ns 220 ns 225 ns 230 ns 235 ns 240 ns 245 ns
0
00000005
0000000b
00000005
00000007
00000005
00000000
00000
0
00000001
0000000c
00000007
00000050
00000000
00000



4、问题解决

相比于单周期cpu和多周期cpu的设计，五级流水线cpu的设计遇到了非常大的困难。流水线cpu和之前所有的cpu设计最大的不同点就在于流水线在一个时钟周期内部完成一条指令，但是需要同时执行5条不同指令各自不同的阶段，所以逻辑方面比较难以理解。数据的转发更是重中之重，特别是beq指令和j指令需要解决的结构冒险的问题，需要先将比较和地址计算提前到ID阶段，然后阻塞一个周期，将原本取出来的下一条指令从寄存器中清空，才能够解决跳转的问题。