pipeline 处理器的设计与实现

一.实验目的：

了解并设计 MIPS 流水线处理器，处理冲突与冒险问题，提高执行效率。

二.实验内容：

（1）在单周期处理器的基础上设计五级流水线处理器

（2）通过转发和阻塞解决数据和控制冲突

（3）根据流水线特性，优化流水线结构降低 CPI

三.实验原理

在单周期处理器中，同一时刻只有一条指令在处理器中，依次经过下面 5 个阶段：

（1）取指令阶段（IF）：根据 PC 值，从指令存储器取出一条指令，同时，PC 自动递增产生下一条指令的地址；当遇到地址转移指令（J-type）时，则控制器把目标地址送入 PC。

（2）指令译码阶段（ID）：对指令操作得到的指令译码，得到一系列控制信号。

（3）指令执行阶段（EXE）：执行指令操作，转移到结果写回状态。

（4）访存阶段（MEM）：所有需要访问存储器的操作——包括读或写操作。

（5）结果写回阶段（WB）：指令结果或者访存结果写回寄存器中。

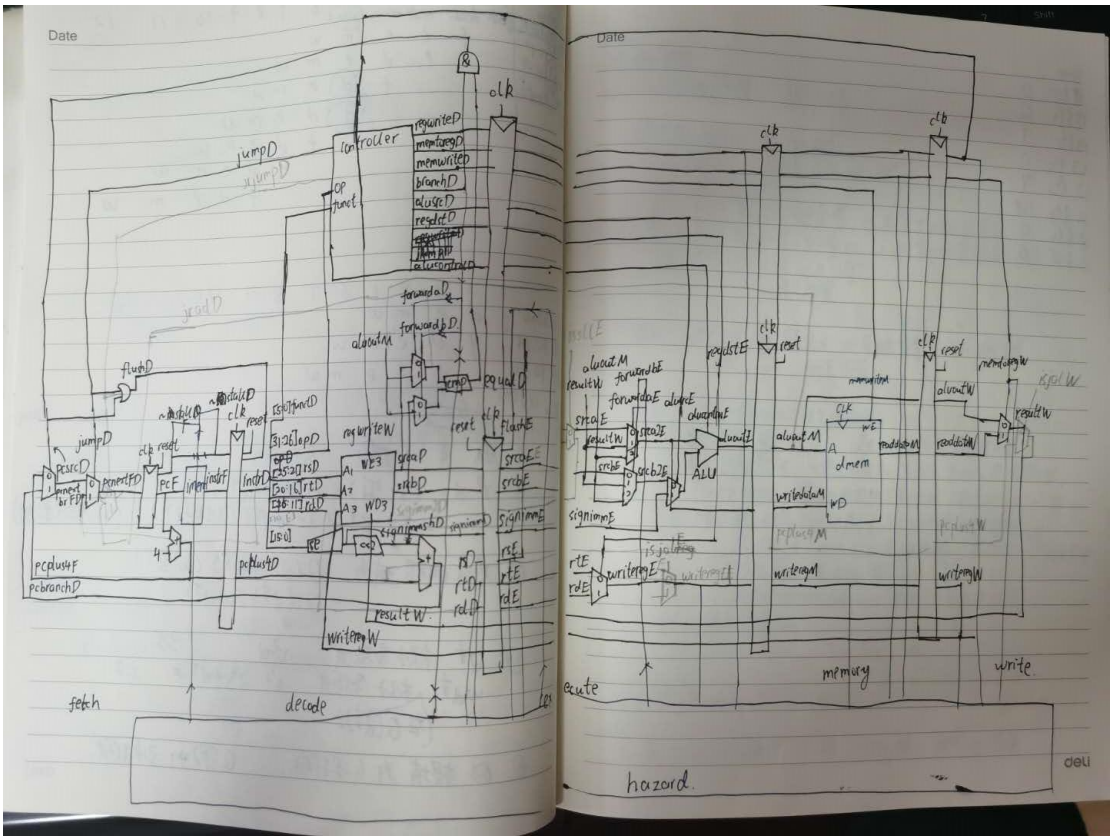
类似的，流水线处理器将处理器分为五个阶段：取指令（fetch），译码（decode），执行（execute），访存（memory），写回（writeback）

流水线处理器希望在同一时刻在 5 个部分分别计算前后 5 条指令，以期提高处理器的利用率。由于每阶段只有逻辑的五分之一，因此时钟频率也可以提高接近 5 倍。

与单周期处理器相比，（在理想情况下）流水线处理器的 CPI 依然为 1，由于时钟周期的缩短，处理器效率（单位时间的指令数）将提高 5 倍，同时每条指令延迟不变。

四.实验内容

1.总电路图



2.流水线逻辑

流水线设计的第一步是把单周期通路在逻辑上分为 5 个部分，中间由五个上升沿同步的触发器连接。根据具体需求，由具体分为：

a.普通触发器（只带有异步重置信号 reset）：flopr

b.带有同步清除信号 clear 的触发器：floprc

c.带有使能信号 en 的触发器：flopenr

d.兼具 en 和 clear 信号的触发器：flopenrc

代码：以 flopenrc 为例

```
module flopenrc #(parameter WIDTH = 8)(
input clk,reset,
input en,clear,
input [WIDTH - 1 : 0] d,
output  logic [WIDTH - 1 : 0] q);
   always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;//pc control
      else if(en & ~clear) q <= d;
      else if(en & clear) q <= 0;

endmodule
```

3.解决数据冲突

数据冲突发生在“写后读”（RAW）情况下，即写某一特定寄存器的指令还没有写回寄存器中，后面的指令已经对这个寄存器进行了读操作。幸运的是大多数情况下可以通过数据转发解决这一问题，而不至于暂停流水线中读后面的指令使得效率降低。

解决方法：

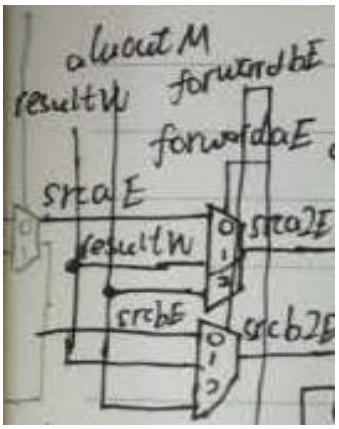
假设指令 A 要更新寄存器 x，指令 B 要读取 x 的值。

（1）当指令 B 在 decode 阶段读取 x 的值时，若 A 已经处在写回阶段，根据同一周期内“先写后读”的原则，B 指令可以得到正确的值，这时不存在数据冲突。

所以我准备在 execute 阶段设计转发的选择器，以供接收来自 memory 和 writeback 阶段的正确数据，来解决数据冲突。

（2）先假设 A 不是 lw 指令（lw 后面单独考虑），当 B 在 execute 阶段时，若 A 在 memory 阶段，aluoutM 保存着 x 正确的值；若 A 在 write 阶段，resultW 保存着 x 的正确值，只需要将这两个信号连接到 execute 阶段，并用一个两位的 forward 信号作为控制信号即可。

电路与代码截图：



```
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;
```

（3）当 A 为 lw 指令时，有一种情况无法用转发的方式解决：B 在 execute 阶段时，A 在 memory 阶段，此时，当转发试图发生时，正确的 x 的值还没有被计算出来，因此，只能将 execute（以及前面 decode 和 fetch）阶段的代码暂停一个周期，当 A 到达 write 阶段后，才能正确转发得到正确的值。

代码：

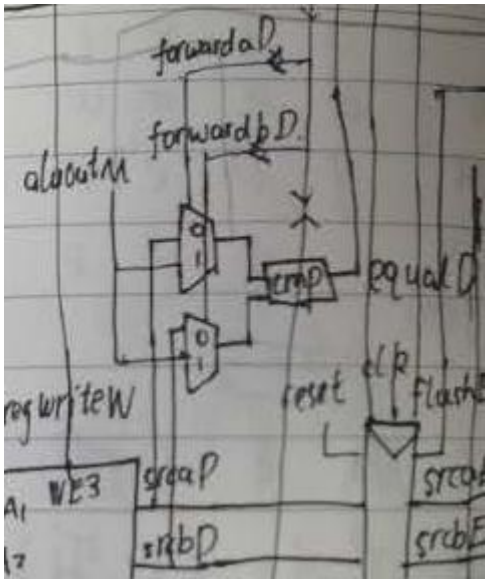
```
assign #1 lwstallD = memtoregE &
(rtE == rsD | rtE == rtD);
```

4.降低控制冒险的代价

当流水线处理器遇到 j 类型（j，jal），jr 指令，以及分支指令（beq，bne）时，由于无法立刻确定下一条指令的地址，且不希望暂停后面的指令，需进行控制冒险（按照某一规定暂时选择某一分支），当后续检测到分支错误时，需要付出代价，如果沿用单周期的阶段划分模式，只有在 3 条指令之后才能得到 equal 信号的值，以确定是否分支预测错误，这样做，预测错误代价为 3 个周期。

处理方法：

可以采用添加一个比较器（速度较快），在 decode 阶段就得出 equal 的值来确定是否预测正确。

电路图： 

5.伴随比较器而来的数据冲突问题

将比较器放在 decode 阶段后会带来新的数据冲突。当相关数据在写回阶段时，根据“先写后读”的原则还是能得到正确的值；当 ALU 结果在 memory 阶段，可通过转发解决数据相关的问题；当 ALU 结果在 execute 阶段或者访存结果（lw）在 memory 阶段，都不得不暂停直到数据准备好。

代码：

```
// forwarding sources to D stage (branch equality)
assign forwardaD = (rsD !=0 & rsD == writeregM &
regwriteM);
assign forwardbD = (rtD !=0 & rtD == writeregM &
regwriteM);

 assign #1 branchstallD = branchD &
 (regwriteE &
 (writeregE == rsD | writeregE == rtD) |
 memtoregM &
 (writeregM == rsD | writeregM == rtD));
```

6.分支预测失败处理

分支预测失效后，会在前面的分支（或跳转）指令进入 decode 阶段时发现，此时已经有错误的 fetch 阶段指令在处理器中，用 flushD 作为 decode 阶段前触

发器的同步清除信号，来消除错误的分支预测，只需插入一个气泡，分支预测错误的代价为 1 个周期。

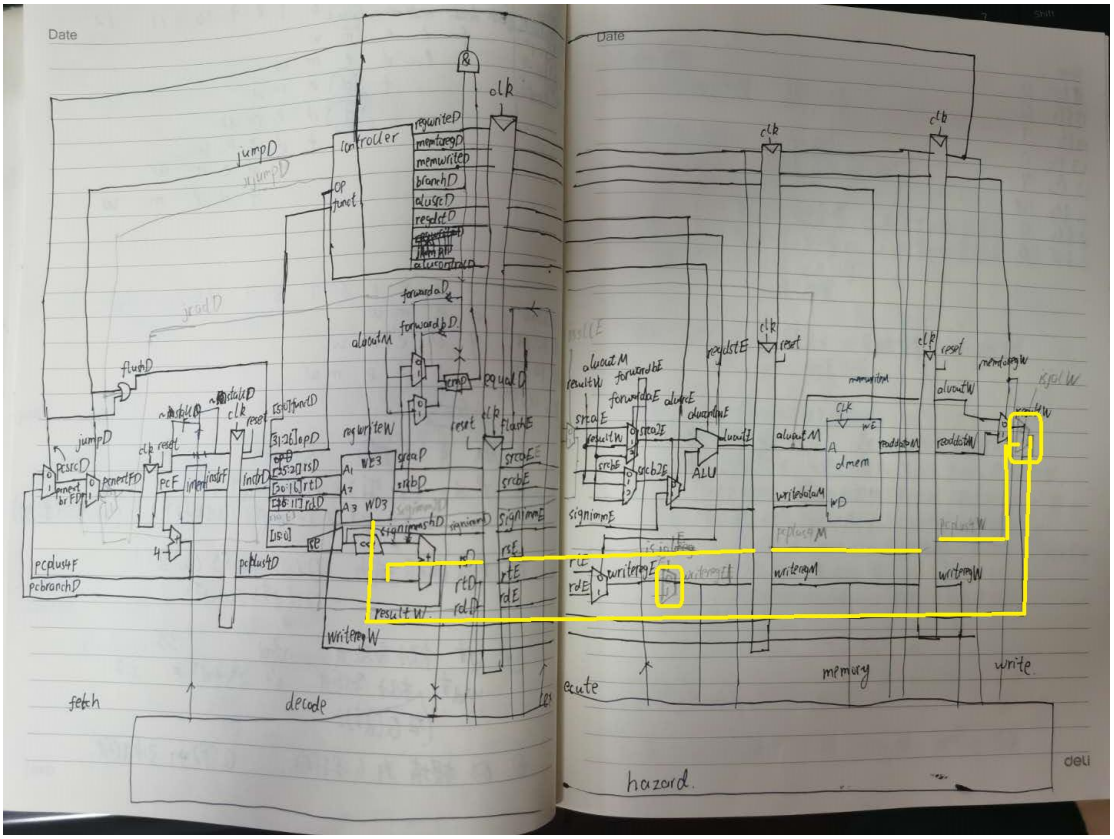
代码：
```
assign flushD = pcsrcD | jumpD | jrjumpD;
```

五. 遇到的问题

1. jal 指令

jal 指令的控制逻辑和 j 指令相同，不过要把 PC+4 的值存入寄存器$31 中，按照 5 层流水线逻辑，需要把 pc+4 数据一直传输到 write 阶段再写回寄存器中。同时控制逻辑需要输出 isjalE 和 isjalW 信号来对目的寄存器和写数据进行选择。

电路图（与上述操作相关的电路用黄线标出）：

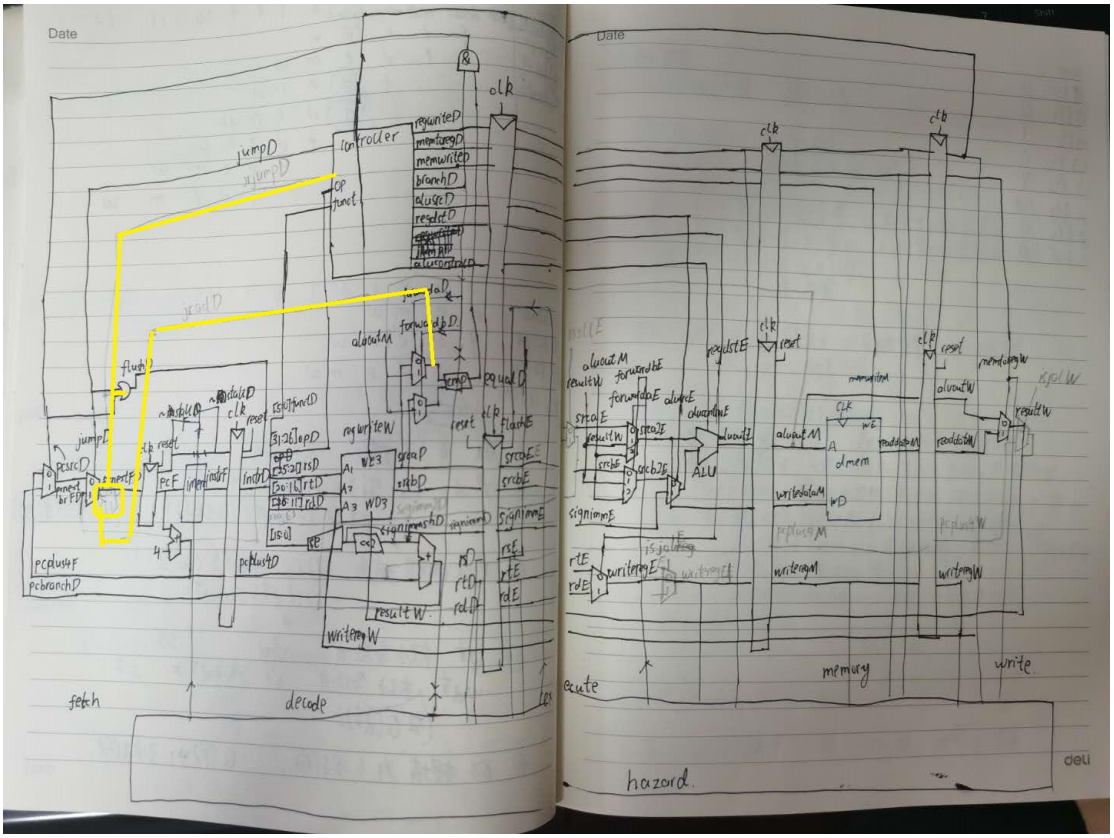


2. jr 指令

jr 指令需要从寄存器中读出目的地址，为了与前面的模式保持一致，希望能在 jr 指令进入 decode 阶段时就确定下一条指令的地址。幸运的是，前面的比较器的一个输入正好是所需的寄存器的值，而且需要处理的数据冲突也已经解决。现在只需要在 pc 确定逻辑中多加一个选择器，用 jrjumpD 信号控制，另一端输

入读出的地址 jradD 即可。当然 flushD 信号的或门也需要加上 jrjumpD 这个输入端。

电路图：



伴随这条数据而来会有新的数据冲突（RAW），转发不能解决所有情况，有时不得不暂停一个时钟周期，即当 jr 指令在 decode 阶段时，用到了 execute 还没有计算出的寄存器值，或者用到在 lw 指令在访存阶段还未读出的值。此时都需要暂停一个周期待寄存器值准备好。为此修改 hazard 逻辑，新加 jrstallD：

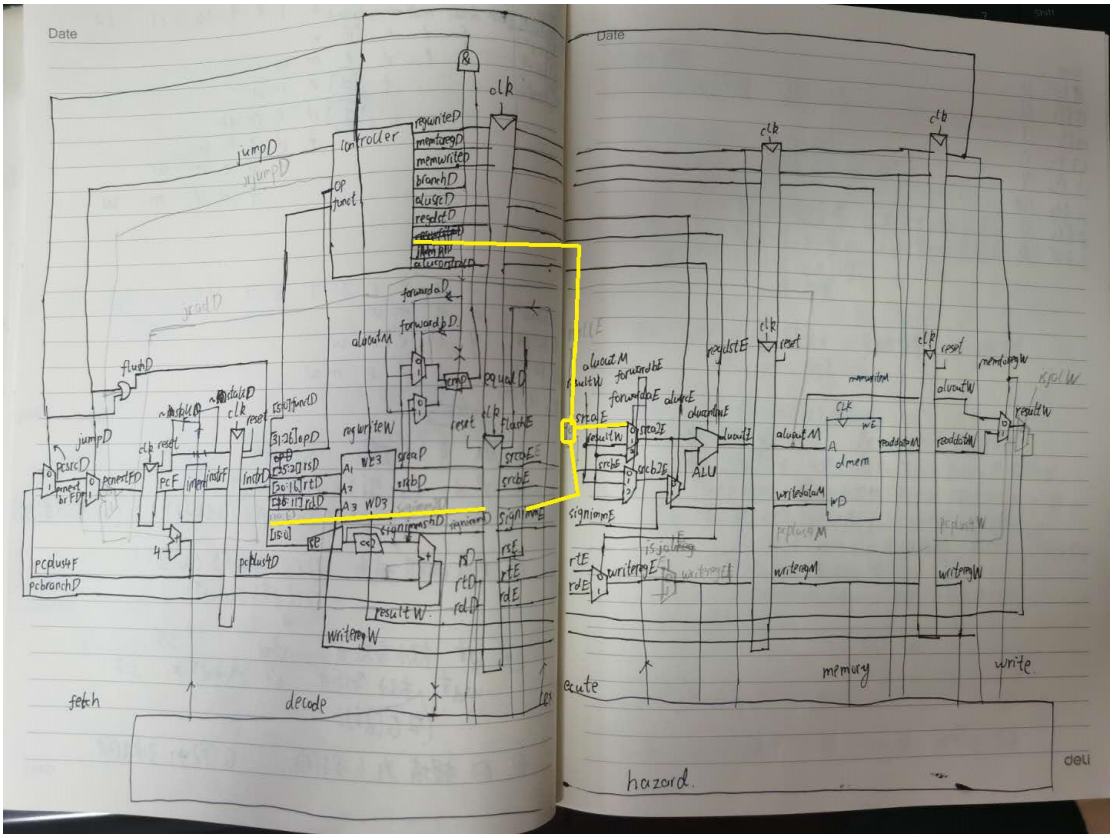
```
assign #1 jrstallD = jrjumpD &
(regwriteE &
(writeregE == rsD ) |
memtoregM &
(writeregM == rsD ));
assign #1 stallD = lwstallD | branchstallD | jrstallD;
```

3. r-type 中的移位指令

sll, srl, sra 三条指令总体逻辑是 r-type 的结构，不过指令格式中包含一个目的寄存器，一个源寄存器和一个立即数移位量 shamt。需要在 decode 阶段把 shamt（指令的第 10 到第 6 位）通过符号扩展并连接到 execute 阶段的 srca 选

择中。这里选择 srca 是由于按照 r-type 的格式移位指令的源寄存器对应于 srcb。然后用控制逻辑输出的 isslIE 信号控制。

电路图：



4. flush 和 stall 的逻辑

0x28 : addi $sp, $sp, 4	23bd0004
0x2c : bne $sp, $t3, for	157dfff9

本来 bubble_sort 运行到这条指令时我一直报错，后来发现这个问题：当出现这样的分支指令时，需要将 decode 和 fetch 阶段暂停一个周期以等待$sp 准备好，因此 stallD 为 1；若在加 4 之前的$sp 与$t3 也不相等，会在暂停的周期中 flushD 也同时为 1。为了防止错误的清除不该清楚的指令（这里 decode 阶段的 bne 指令），应该使得 flush 信号只在使能信号~stall 有效的情况下才起作用，因此 flopenrc 的代码应该为：

```
module flopenrc #(parameter WIDTH = 8)(
input clk,reset,
input en,clear,
input [WIDTH - 1 : 0] d,
output  logic [WIDTH - 1 : 0] q);
   always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;//pc control
      else if(en & ~clear) q <= d;
      else if(en & clear) q <= 0;

endmodule
```

5. I-type 指令的扩充

主要涉及 andi,ori,slti 指令，在 maindec 与 aludec 之间添加 ialuop 传递 i-type 的操作码。代码：

maindec：

```
  always_comb
   case(op)
      6'b001000: ialuop <= 2'b00; //ADDI
      6'b001100: ialuop <= 2'b01; //ANDI
      6'b001101: ialuop <= 2'b10; //ORI
      6'b001010: ialuop <= 2'b11; //SLTI
    endcase
```

aludec：

```
 2'b11: case(ialuopD)//I-type
    2'b00: alucontrol <= 3'b010;//+
    2'b01: alucontrol <= 3'b000;//&
    2'b10: alucontrol <= 3'b001;//|
    2'b11: alucontrol <= 3'b111;//slt
       endcase
```

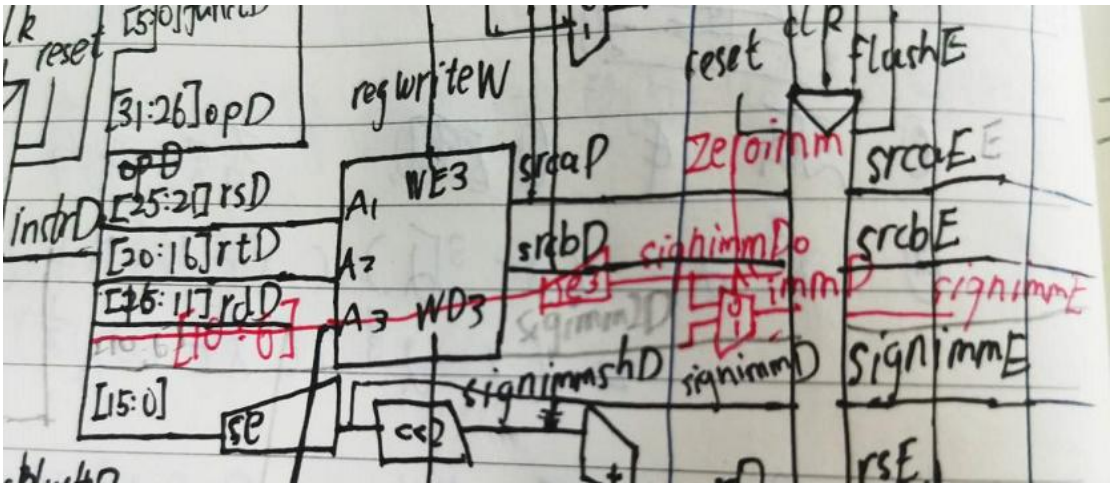
由于 andi,ori,xori 三条指令的立即数应该进行零扩展，而目前所用只有一个 signimmD 是符号扩展，所以要新加一路零扩展立即数，与符号扩展立即数二选一，传递到 execute 阶段，控制信号为 zeroimm。

controller：

```
assign zeroimm = (op == 6'b001101) | (op == 6'b001100);//zeroimm
```

Opcode	名称	描述	操作
001100(12)	andi rt, rs, imm	立即数与	[rt] = [rs] & ZeroImm
001101(13)	ori rt, rs, imm	立即数或	[rt] = [rs] | ZeroImm
001110(14)	xori rt, rs, imm	立即数异或	[rt] = [rs] ^ ZeroImm

电路图（红色部分）：



五.实验结果

运行截图：

最终 CPI 约为 1.73

```
========== Test: en & clear ==========
WARNING: file D:/pipelineMIPS/benchtest/en & clear/en & clear.data could not be opened
========== In init ==========
13 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] en & clear

========== Test: i-type ==========
WARNING: file D:/pipelineMIPS/benchtest/i-type/i-type.data could not be opened
========== In init ==========
18 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] i-type

========== Test: mutual recursion ==========
WARNING: file D:/pipelineMIPS/benchtest/mutual recursion/mutual recursion.data could not be opened
========== In init ==========
55 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] mutual recursion

========== Test: testjr ==========
WARNING: file D:/pipelineMIPS/benchtest/testjr/testjr.data could not be opened
========== In init ==========
64 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] testjr
```

```
========== Test: ad hoc ==========
WARNING: file D:/pipelineMIPS/benchtest/ad hoc/ad hoc.data could not be opened
========== In init ==========
22 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] ad hoc

========== Test: factorial ==========
WARNING: file D:/pipelineMIPS/benchtest/factorial/factorial.data could not be opened
========== In init ==========
28 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] factorial

========== Test: bubble sort ==========
========== In init ==========
21 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] bubble sort

========== Test: gcd ==========
WARNING: file D:/pipelineMIPS/benchtest/gcd/gcd.data could not be opened
========== In init ==========
18 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] gcd

========== Test: quick multiply ==========
WARNING: file D:/pipelineMIPS/benchtest/quick multiply/quick multiply.data could not be opened
========== In init ==========
17 instructions in total
========== In runtime checker ==========
```

```
successfully pass memory judge
[OK] quick multiply

========== Test: bisection ==========
========== In init ==========
21 instructions in total
========== In runtime checker ==========
successfully pass runtime checker
========== In memory judge ==========
successfully pass memory judge
[OK] bisection

[Done]

CPI = 1.733869
```

六.感想与体会

通过本次实验，在单周期处理器的基础上，我学习了五级流水线的设计方法，尝试并解决了流水线设计过程中的控制冒险与数据冲突的问题。从自己绘制数据通路，构建初步模型，到处理数据冲突，调试样例，虽然遇到了很多困难，但在解决问题的过程中确实感觉到收获良多。

七.参考文献

1. 数字设计和计算机体系结构 原书第 2 版 ,戴维·莫尼·哈里斯 231-238

页：流水线处理器

2. https://blog.csdn.net/wbwwf8685/article/details/53762908

CSDN 博客：32 个 MIPS 通用寄存器

3. https://blog.csdn.net/qq_41848006/article/details/82256626

CSDN 博客：MIPS 指令集架构