pipeline50实验报告

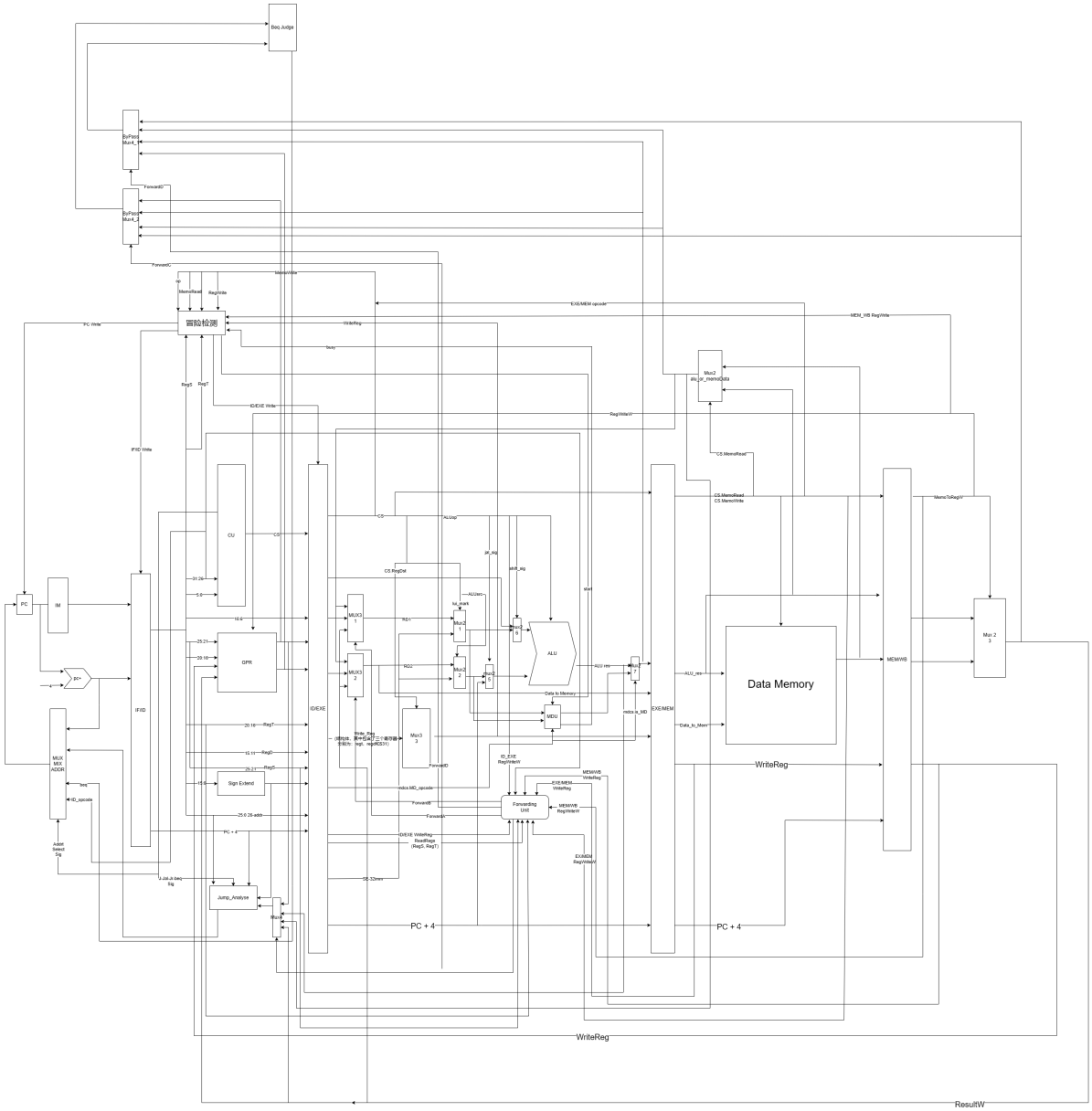
实验目标

实现50条指令的流水线CPU，其中包括乘除法指令，还有处理数据冲突的旁路转发和$stall$功能。

基本思路

从单周期CPU进行改进，加入四个流水线寄存器，冒险检测，旁路转发单元。其余细节在具体架构的设计图中可以看到。更多的关键细节在问题和解决方案的部分有详细说明。

具体架构



这是设计的数据通路图，有极少部分的信号没有加入，乘法器在EXE阶段（图中的MDU)。

问题 & 解决方案

1.冒险检测单元设计与改进

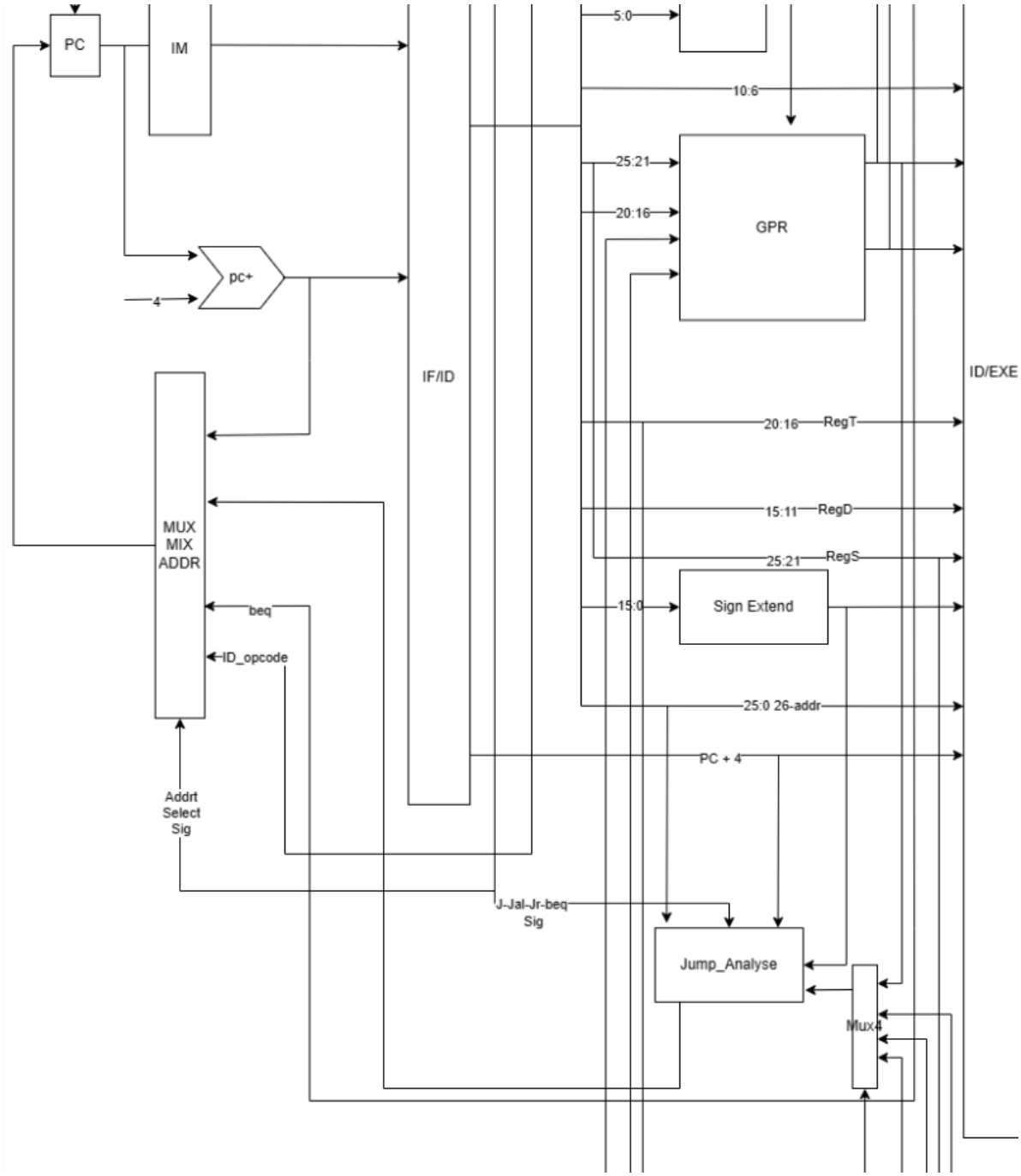
对于beq，jal等跳转指令，跳转地址被安排到了WB阶段才进行写回。所以，在成功跳转的情况下，除了分治延迟槽中的那条指令，跳转指令之后的第二条和第三条指令都不能执行，要进行$stall$。在最初的方案中，我在冒险检测单元中对跳转指令带来的$stall$进行检测，然而，由于需要跟踪指令的执行，以此来决定是否进行$stall$，当多条跳转指令在临近的位置出现时，就会出现十分复杂的冲突情况，大大增加代码的复杂度，并且带来逻辑的高度混乱，无法确定代码出问题的具体位置，也无法进行适当的调试。

鉴于以上情况，我在第二版的设计中大大精简了冒险检测单元的设计，只用于检测乘除法的$busy$以及lw, lb, lbu, lh, lhu等指令带来的数据冲突情况。这极大降低了逻辑的复杂性，也让运行时的调错过程更加顺利。以下是改进后的冒险检测单元的部分代码：

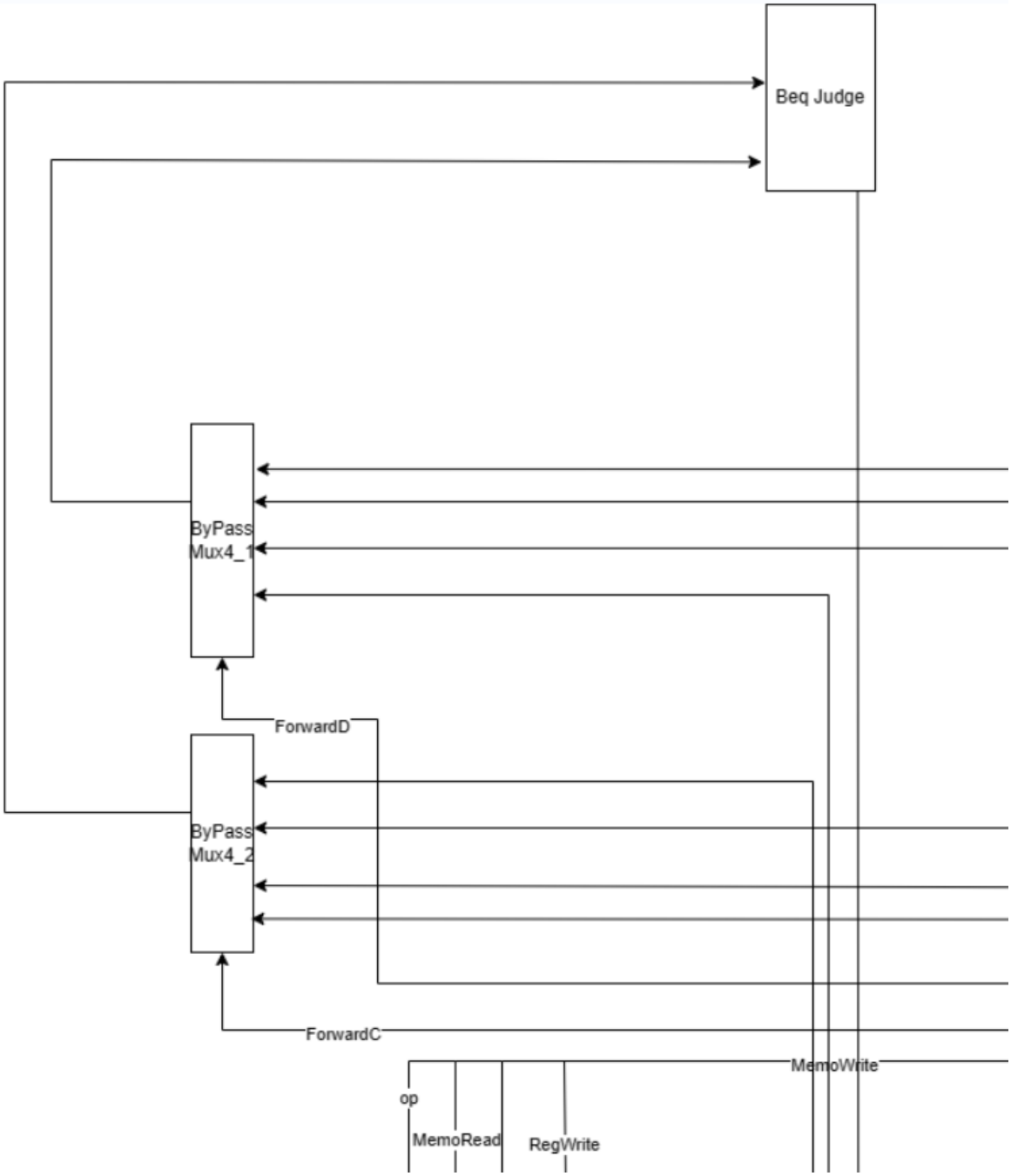
```
    always @(*) begin
        PC_Write <= 3'b111;
        IF_ID_Write <= 3'b111;
        ID_EX_Write <= 3'b111;

        if ((ID_EXE_opcode == 6'b100011 || ID_EXE_opcode == 6'b100000 || ID_EXE_opcode == 6'b100100 || ID_EXE_opcode == 6'b100001 || ID_EXE_opcode == 6'b100101) && (ID_EXE_WriteReg != 0 && (ID_EXE_WriteReg == RegS || ID_EXE_WriteReg == RegT)))
            begin
                    PC_Write <= 3'b000;
                    IF_ID_Write <= 3'b010;
                    ID_EX_Write <= 3'b000;
            end
        if (busy == 1'b1)
            begin
                    PC_Write <= 3'b000;
                    IF_ID_Write <= 3'b010;
                    ID_EX_Write <= 3'b000;
                    start <= 0;
            end
        else
            begin
                start <= 1'b1;
            end
    end
```

2.跳转指令数据通路的优化

紧跟上一个问题，在第一版的设计中，如果跳转指令成功跳转，其跳转地址的写回也是放到WB阶段，这就造成了两个周期的浪费。经过仔细思考可以发现，所有的跳转指令都可以在ID阶段就完成是否跳转的判断，同时也可以在ID阶段就进行跳转地址的构造等操作，也就是说，在ID阶段完成后其实就可以进行跳转地址的写回，由于这个写回是对PC寄存器的，不是对GPR，所以也没有必要放到WB阶段。经过观察可以看出，如果在ID阶段写回，那么在IF阶段的刚好就是分支延迟槽中的那条指令，而下个周期来临时，这个新的跳转地址的指令就能被取出，相当于“无缝衔接”，不需要$stall$，完美避免了周期的浪费。



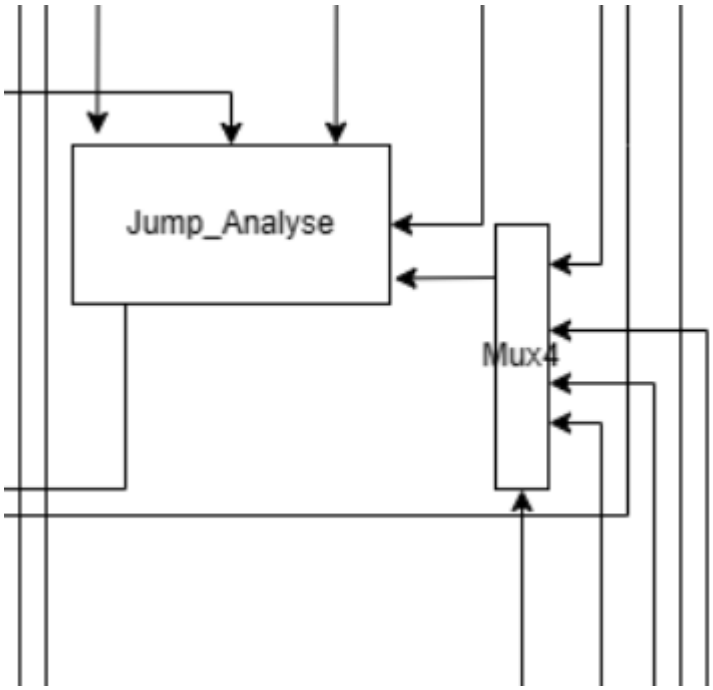
$Jump_Analyse$进行jal、$jalr$和j的跳转地址的拼接，同时接受输入的信号进行选择操作。



Beq_Judge单元对beq、$bgtz$、$bgez$、$bltz$、$blez$、bne的判断条件进行处理，传出各个指令的选择信号结果。

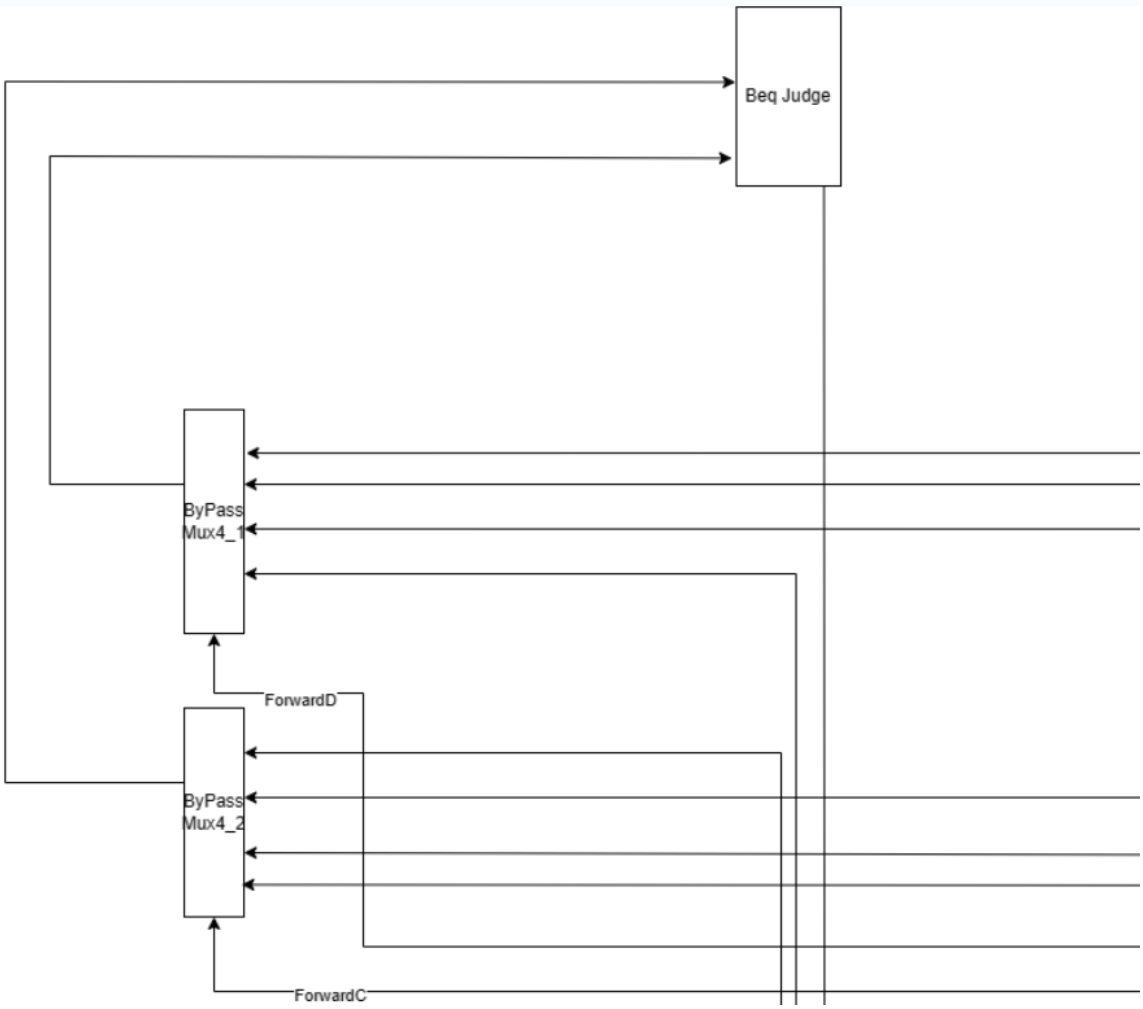
3.关于旁路的理解

对于旁路的透彻理解是：只要用到寄存器的地方，就需要旁路。其中最极端的一种情况为：jal指令的下一条指令是jr指令，并且jr所用的寄存器为31号寄存器。因为jr等指令的跳转地址存在$RegS$中，都需要在ID阶段就确定，而jal对于寄存器的写回要等到WB阶段，当jr执行到ID阶段时，其需要的值还停留在EXE阶段，这时候必须通过旁路将其转发到相应的Mux中。由于这种情况不是在EXE阶段中，比较难想到。

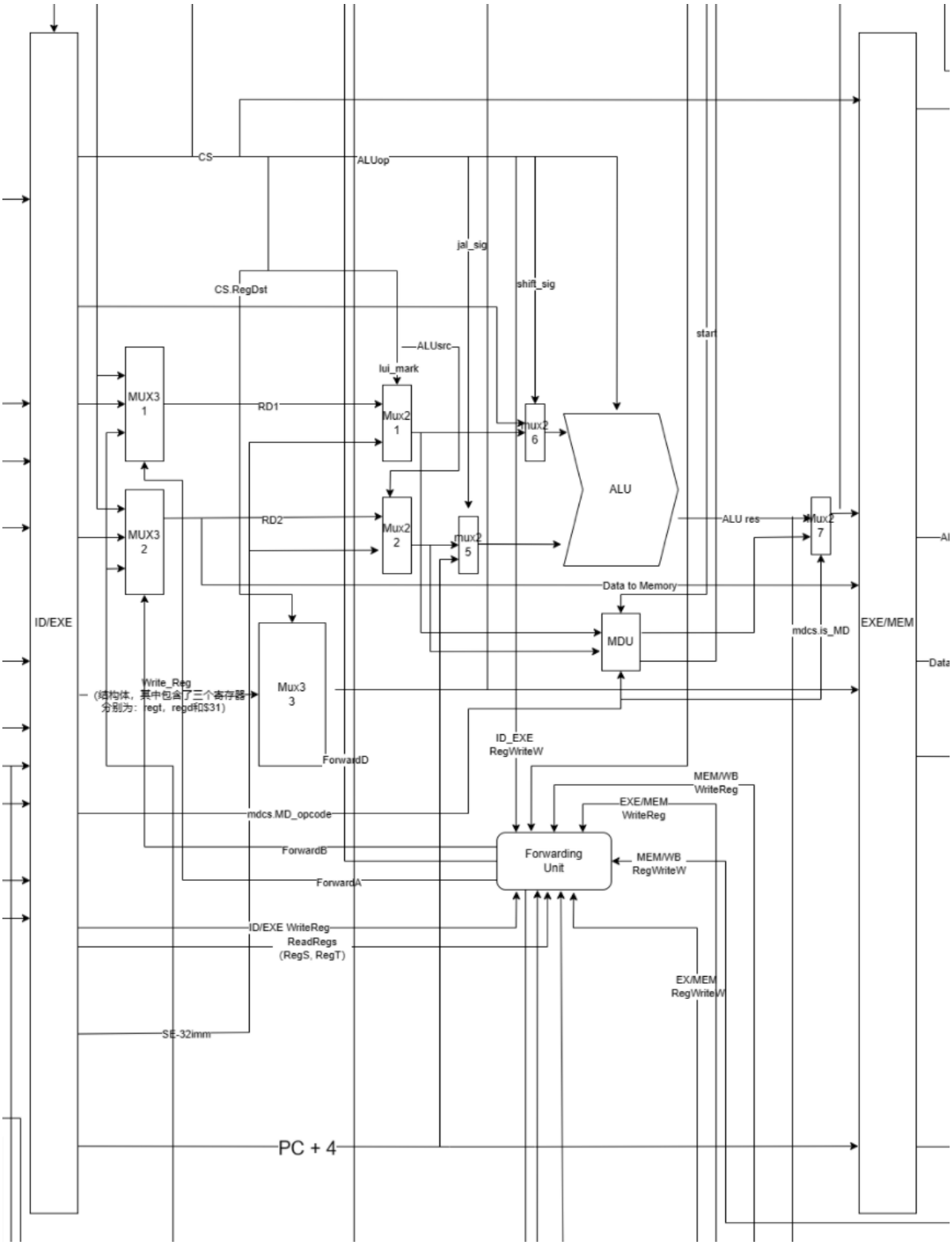


这是ID阶段地址选择的旁路，$Mux4$是选择器，接收$Farwarding\ Unit$的选择信号和ID，EXE，MEM和WB阶段本该写回寄存器的值。

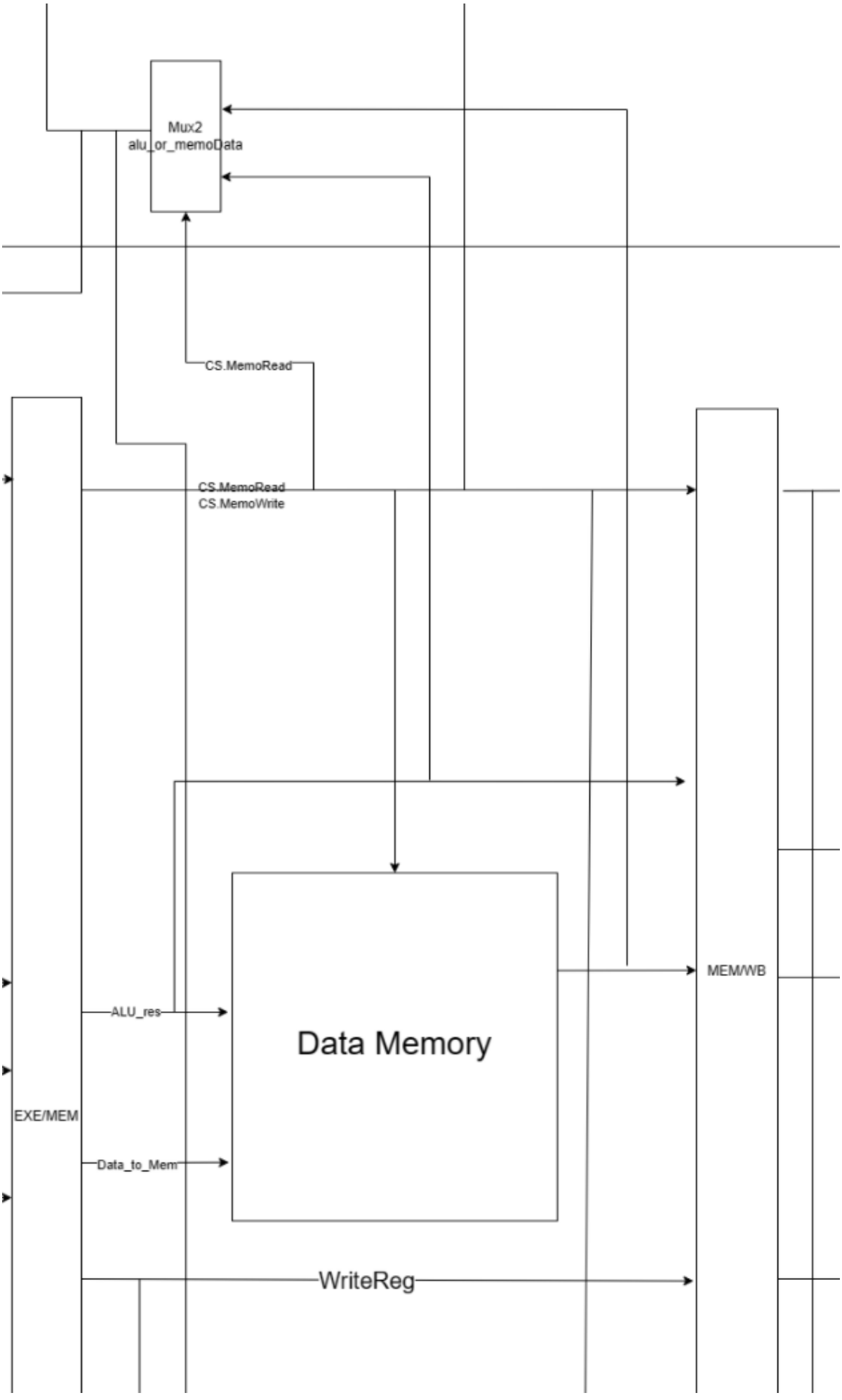
与此相近的另一种情况是，同样是在ID阶段中，Beq_Judge模块也要对寄存器的值进行读取，所以两个输入都需要用到旁路转发，如下图所示：



同样，两个$ByPass\ Mux$接收ID，EXE，MEM和WB四个阶段的值，同时通过旁路转发单元的信号（图中为$ForwardC$和$ForwardD$）进行选择。



最后也是最重要的旁路是ALU相关的两个旁路，如图所示，流水寄存器右侧的两个$MUX3$寄存器用于选择流水寄存器的输出、MEM阶段和WB阶段对寄存器的写回的值。


Mux2
alu_or_memoData
CS.MemoRead
CS.MemoRead
CS.MemoWrite
MEM/WB
ALU_res
Data Memory
EXE/MEM
Data_to_Mem
WriteReg

如图所示，这里有一个细节是，MEM阶段用于写回的值可能是ALU计算结果，也可能是从$DataMemory$中读取的数据，所以需要根据当前指令的$MemoRead$信号做一个选择来决定传递给旁路的数据为哪一个。

4.地址访问越界

在随机测试的过程中，有些地址的范围超过了2048，采用截取$[12:2]$的方法来解决，之前是用的$[31:2]$，只去掉了最后的两位，这是之前设计的一个缺陷。

```
@000031f8: $12 <= 00000002
PC_clock
pc output addr = 00003240
tmp_data1 = d0, tmp_data2 = 00000000,  addr = 0000161e
@00003230: *00005878 <= 000000d0
ins_addr = 00003240, ins_code = 03ea8022
```

如图，$0x0000161e$明显超出了$2K$的范围，必须将其截取。但是在输出写入地址时要输出原地址

5.乘除法阻塞的优化

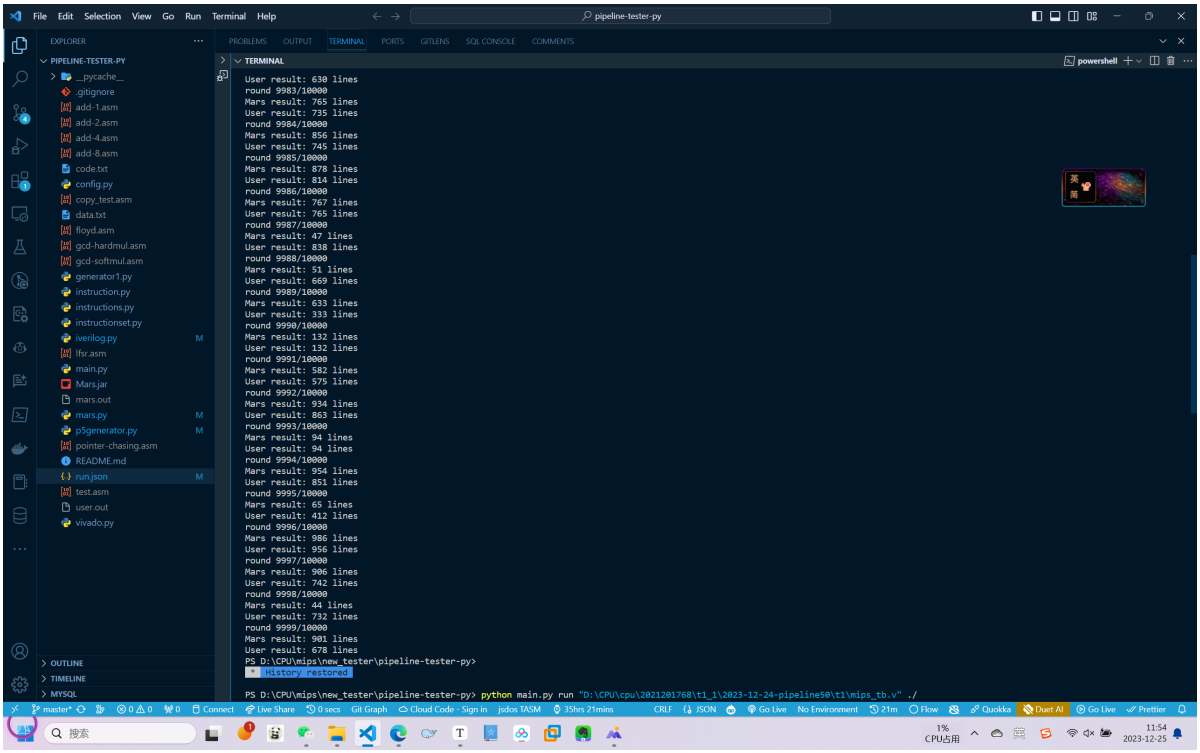
如果对于每个乘法指令都阻塞5个周期，每个除法指令都阻塞10个周期，是对周期的极大浪费。只有在乘除法器是$busy$状态且当前要执行的指令是乘除法相关的指令时，才进行$stall$，否则可以直接执行。这里体现了ALU和乘除法器分离的优点。

对于乘除法指令，我设置了一个信号结构体，其中有一个乘除法指令的标志位is_MD，如果这个标志位为1，则说明是和乘除法器相关的指令，只有在ID阶段的指令的is_MD信号为1且EXE阶段的乘除法器在当前周期还处于$busy$状态时，才需要设置$stall$。

具体的实现只需要添加一个小小的"&"操作：

```
AdventureDetect advdet(
    .ID_EXE_opcode(ID_EXE_out_CS.opcode),
    .ID_EXE_WriteReg(mux3_3_out),
    .RegS(RegS),
    .RegT(RegT),
    .busy(busy & CU_out_mdcs.is_MD),
    .PC_Write(PC_Write_ad_out),
    .IF_ID_Write(IF_ID_Write_ad_out),
    .ID_EX_Write(ID_EXE_Write_ad_out),
    .start(start)
);
```

测试结果



指定测试程序测试