Lab 1

一、实验目的

- 温故流水线`CPU`的设计
- 了解并实现`RV32I`指令集
- 理解旁路优化（`Forwarding`）

二、 实验过程

填充数据通路中空缺的代码，熟悉`RV32I`指令集。

控制信号填充（篇幅原因不粘贴所有代码）

理清给定模块的逻辑，首先是通过`opcode`和`funct3, funct7`进行指令译码，区分出所有指令，方便进行指令分类、`ALU、ImmGen`控制信号分配等细节操作。然后分别进行跳转、立即数、`ALU`以及关于寄存器的信号的赋值。以下给出部分填充代码及解释

```
//Branch
assign Branch = (B_valid & cmp_res) | JAL | JALR ;
...
assign cmp_ctrl = {3{BEQ}} & 3'b001 |
                        {3{BNE}} & 3'b010 |
                        {3{BLT}} & 3'b011 |
                        {3{BLTU}} & 3'b100|
                        {3{BGE}} & 3'b101 |
                        {3{BGEU}}& 3'b110 ;
```

跳转指令总共8条分3类，需要注意的点在于虽然信号名为`Branch`，并不是只管`B-type`指令，通过在`CPU`模块中追溯，我们可以发现它实际上直接管理了`PC`是否跳转，也即反映了所有跳转指令。另外，`B-type`指令流程：在控制模块中解析出比较控制信号再输出到比较器中，比较器的结果会再次回到控制模块中进行跳转判断。

```
assign ALUSrc_A = ~ (AUIPC | JALR |JAL);
assign ALUSrc_B = I_valid | L_valid | S_valid | JALR | LUI | AUIPC;
```

以上为`ALU`操作数的选择信号，注意零散的指令，容易遗漏。

```
assign DatatoReg = L_valid;
//因为WB阶段只有ALU,MEM两个数据来源，因此实际上就是load指令的判断信号
assign RegWrite = R_valid | I_valid | JAL | JALR | L_valid | LUI | AUIPC;//写寄存器信号
assign mem_w = S_valid;//写内存信号
assign MIO = L_valid | S_valid;//内存读/写信号
assign rs1use = R_valid | I_valid | S_valid | L_valid  | B_valid | JALR;//指令段rs1使用标记
assign rs2use = R_valid | S_valid | B_valid;//指令段rs2使用标记
//以上信号理清信号含义后仔细填充指令类型即可
```

以上为关于寄存器和内存读写的代码，详见注释。

CPU数据通路

依据给出的原理图可更加直观地理清思路，主要是弄清楚各个信号的作用，填写的部分为涉及`PC`、`ID`阶段`forwarding`，`Ex`阶段`ALU`数据源以及`load-store forwarding`的几个`mux`，注意给出的设计和计组实验课的设计有细节差别，一定要确认各个数据源的位置，与原理图对照好。

```
MUX2T1_32 mux_IF(.I0(PC_4_IF), .I1(jump_PC_ID), .s(Branch_ctrl),
.o(next_PC_IF));
//PC选择，0号为PC+4，1号为跳转地址

MUX4T1_32 mux_forward_A( .I0(rs1_data_reg), .I1(ALUout_EXE),
.I2(ALUout_MEM), .I3(Datain_MEM), .s(forward_ctrl_A), .o(rs1_data_ID));
MUX4T1_32 mux_forward_B (.I0(rs2_data_reg), .I1(ALUout_EXE),
.I2(ALUout_MEM), .I3(Datain_MEM),.s(forward_ctrl_B), .o(rs2_data_ID));
//ID阶段rs1与rs2前递选择，除0号各自译码外其余输入相同

MUX2T1_32 mux_A_EXE( .I0(PC_EXE), .I1(rs1_data_EXE), .s(ALUSrc_A_EXE),
.o(ALUA_EXE));
MUX2T1_32 mux_B_EXE( .I0(rs2_data_EXE), .I1(Imm_EXE), .s(ALUSrc_B_EXE),
.o(ALUB_EXE));
//ALU输入源，A操作数可能是PC(0),rs1(1)
//B操作数可能是rs2(0),imm(1)

MUX2T1_32 mux_forward_EXE( .I0(rs2_data_EXE), .I1(Datain_MEM),
.s(forward_ctrl_ls), .o(Dataout_EXE));
//load-store forwarding
//可能是rs2 or 从mem阶段前递的内存数据
```

`HarzardDetectionUnit`模块设计

模块用于探测遇到冲突并给出处理信号。分为`forwarding`和`stall`两部分。

forwarding

数据冲突主要在于`B-type`指令判断以及`ALU`计算时数据的缺失，分别位于`ID,EXE`阶段，同时写寄存器的数据结果绝大部分在`EXE`阶段已经计算出来了，因此我们只需要将结果再传入`ID`阶段替代还未更新的数据就可以实现无`stall`处理`Hazard`。

另外，还有较为特殊的`load-store`类型前递，如果`load`指令的目的寄存器和`store`指令的`rs2`源寄存器相同，可以通过一个`mux`选出前递至`EXE`阶段，就可以无`stall`将两次内存的访问进行下去。

具体设计方面，对于`ID`阶段的两个前递，我们选要判断指令的两个源寄存器段是否有使用并且究竟是否有冲突，寄存器号是否相同，是否有写操作（也即目的寄存器段是否使用），最为重要的是记住对于`RISC-V`架构来说`x0`寄存器是恒零的，无法写入，也就不会存在冲突，不选要进行前递。

`EXE`阶段的大致相同，判断是否是`load-store`型的冲突并排除`x0`，选定信号即可。

具体代码如下：

```
always @* begin
    //forward_A
    if(rs1use_ID & rd_w_EXE & (rs1_ID == rd_EXE) & (rs1_ID != 0) ) begin//ex forward
        forward_ctrl_A = 2'b01;
    end
    else if(rs1use_ID & rd_w_MEM & (rs1_ID == rd_MEM) & (OP_EXE == 5'b00000 ) & (rs1_ID != 0) ) begin//mem load forward
        forward_ctrl_A = 2'b11;
    end
    else if(rs1use_ID & rd_w_MEM & (rs1_ID == rd_MEM) & (rs1_ID != 0) )begin//mem alu forward
        forward_ctrl_A = 2'b10;
    end
    else begin
        forward_ctrl_A = 2'b00;//no forward
    end
    //forward_B
    if(rs2use_ID & rd_w_EXE & (rs2_ID == rd_EXE) & (rs2_ID != 0) ) begin
        forward_ctrl_B = 2'b01;
    end
    else if(rs2use_ID & rd_w_MEM & (rs2_ID == rd_MEM) & (OP_EXE == 5'b00000 ) & (rs2_ID != 0) ) begin
        forward_ctrl_B = 2'b11;
    end
    else if(rs2use_ID & rd_w_MEM & (rs2_ID == rd_MEM) & (rs2_ID != 0) )begin
        forward_ctrl_B = 2'b10;
    end
    else begin
        forward_ctrl_B = 2'b00;
    end
    //forward_ls
    if((OP_EXE == 5'b01000) & (OP_MEM == 5'b00000) & (rs2_EXE == rd_MEM) & (rs2_EXE != 0) )begin //load rd == store rs2
        forward_ctrl_ls = 1;
    end
    else begin
        forward_ctrl_ls = 0;
    end
end
```

stall

当指令执行需要跳转时，由于`PC`更新需要时钟，我们不可避免地需要在`IF`阶段停止并重新取指令，这就是数据冲突。因为`PC`需要更新，也即`PC`使能信号置1；而对于`REG_IF_ID`，我们需要丢弃不需要执行的旧指令，并且因为`ID`及以后的指令并未停止执行，`ID`阶段会出现一个空缺，我们需要插入一个无意义`nop`。

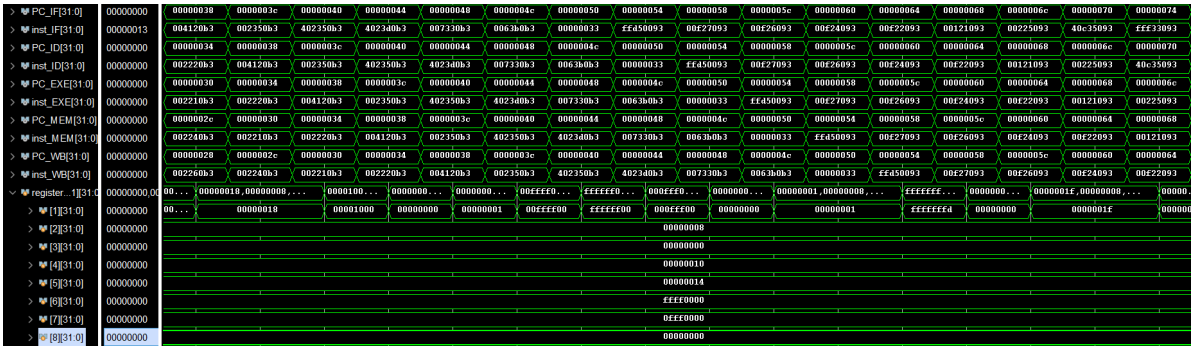
而由于我们并未做从`MEM`到`EXE`阶段的前递，如果`load`指令后的指令依赖它，我们不可避免地需要在`load`指令的`EXE`阶段后暂停一个周期，等它到达`MEM`阶段后向暂停的指令`ID`阶段前递（如无前递将停2个周期）才可以继续执行下去，这也是仅存的数据冲突。当检测单元检测到该冲突时，会重置PC使能为0（不需要取新指令），`REG_ID_IF`需要暂停，不需要更新数据而`REG_ID_EXE`需要插入一个`nop`填补空缺地方。

具体代码如下：（正常情况省略）

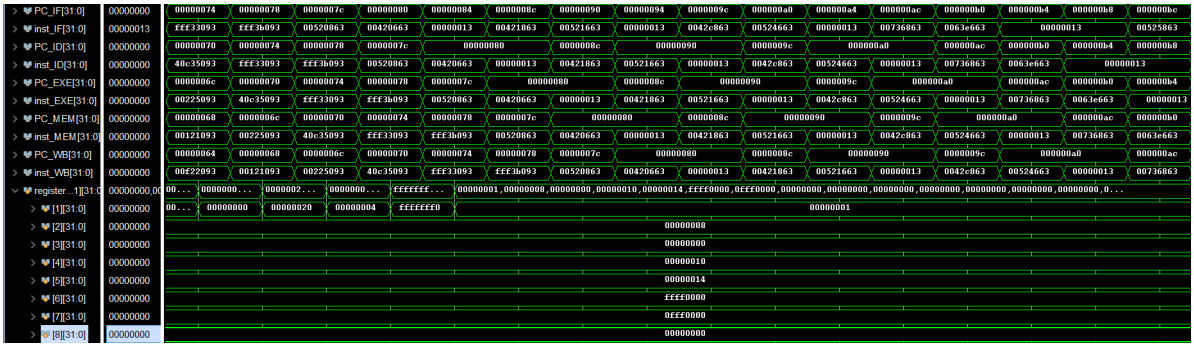
```
always @* begin
    if(Branch_ID)begin
        PC_EN_IF    <= 1;//接纳新的指令
        reg_FD_stall <= 0;//不停顿
        reg_FD_flush <= 1;//丢弃旧指令,ID插入nop,IF进入新指令，旧指令消失
        reg_DE_flush <= 0;//IDEX不用变，跳转指令继续运行
        //不变项
        reg_FD_EN   <= 1;//
        reg_DE_EN   <= 1;//
        reg_EM_EN   <= 1;
        reg_EM_flush <= 0;
        reg_MW_EN   <= 1;
    end
    else if(OP_EXE == 5'b00000 ) begin
        if( (rs1use_ID & rs1_ID == rd_EXE & (rs1_ID != 0) ) |
            (rs2use_ID & rs2_ID == rd_EXE & (rs2_ID != 0) & OP_ID != 5'b01000)
        ) begin
            PC_EN_IF    <= 0;//Ex为load，前面全部阻塞，不取指令
            reg_FD_stall <= 1;//寄存器不变
            reg_FD_flush <= 0;//IFID不用插入nop
            reg_DE_flush <= 1;//IDEX插入nop填补空缺的EX段

            reg_FD_EN   <= 1;//
            reg_DE_EN   <= 1;//
            reg_EM_EN   <= 1;
            reg_EM_flush <= 0;
            reg_MW_EN   <= 1;
        end
```

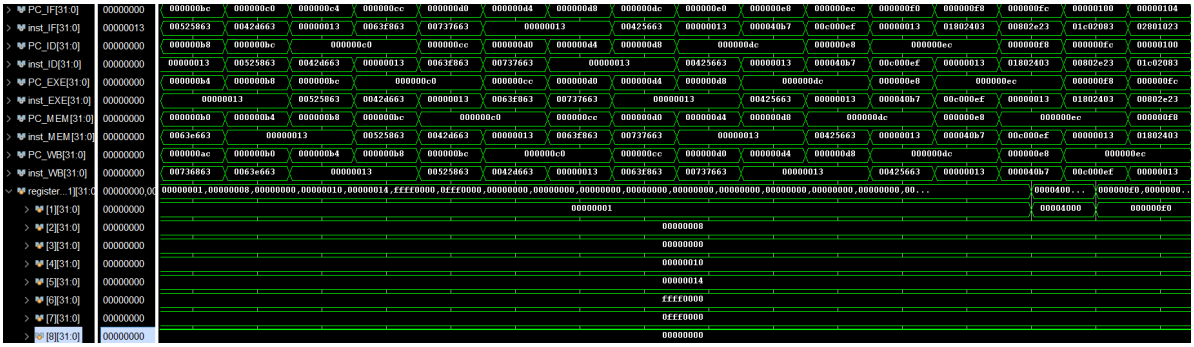
三、实验结果

实验指令与信号众多，仅截取部分仿真 PC 指令与寄存器值，普通的计算指令不再一一解释，有关 `forwarding` 和 `stall` 的部分将在思考题1中给出详细说明。

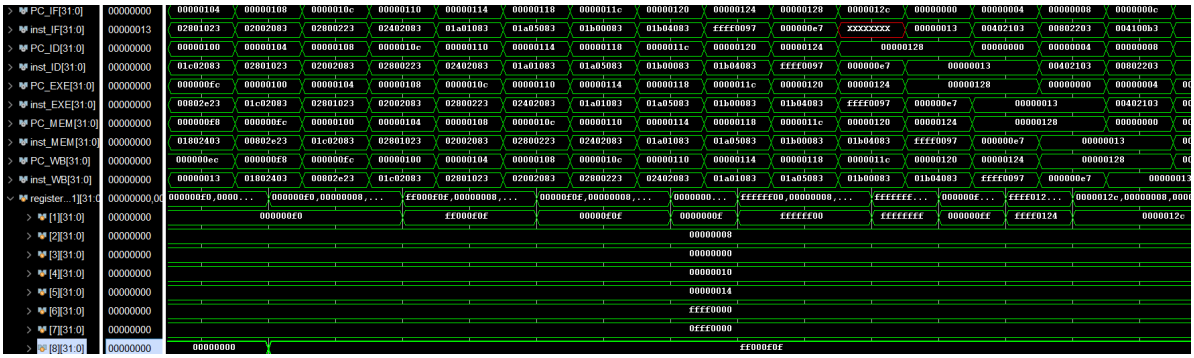


以上主要是R型和I型的计算指令，不一一核对结果。





以上主要是跳转指令，放大图片可检验跳转地址的正确性，并且插入了不属于原代码的`nop`。



检查`load`与`store`指令的正确性，是否按正确的地址和位数操作数据。

四、思考题

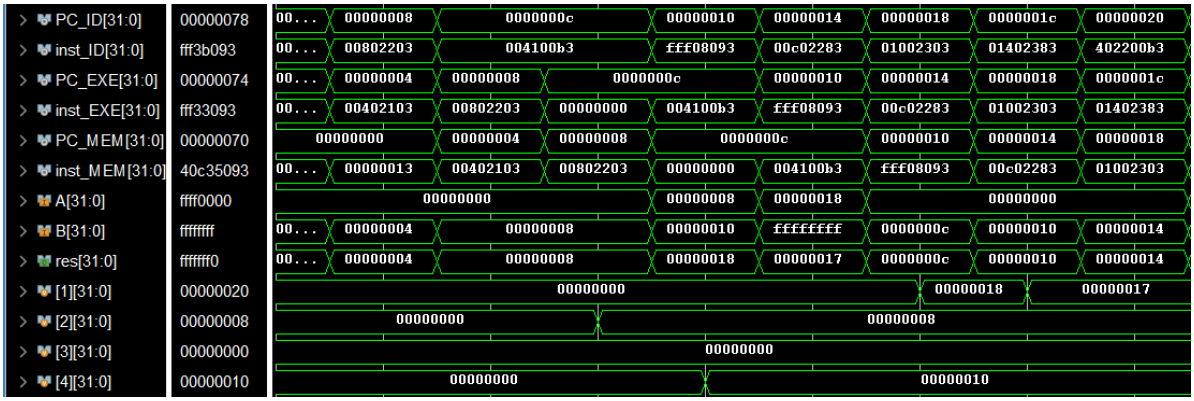
1.

```
1   .global __start
2   __start:
3   addi x0, x0, 0      # 00000013
4   lw x2, 4(x0)        # 00402103
5   lw x4, 8(x0)        # 00802203 stall 1
6   add x1, x2, x4      # 004100B3 forward no stall
7   addi x1, x1, -1     # FFF08093 forwarding
8   lw x5, 12(x0)       # 00C02283
```

如上存在两处`forwarding`以及一处`stall`。

首先是3、4条指令之间，显然存在一处冲突，而且由于3指令是`load`指令，还需要`stall`一个周期，然后待`load`指令执行到`EXE`阶段再前递，而4、5条指令都是纯计算指令，在寄存器`x1`存在冲突，4指令执行完`EXE`阶段就可以前递，不需要`stall`。

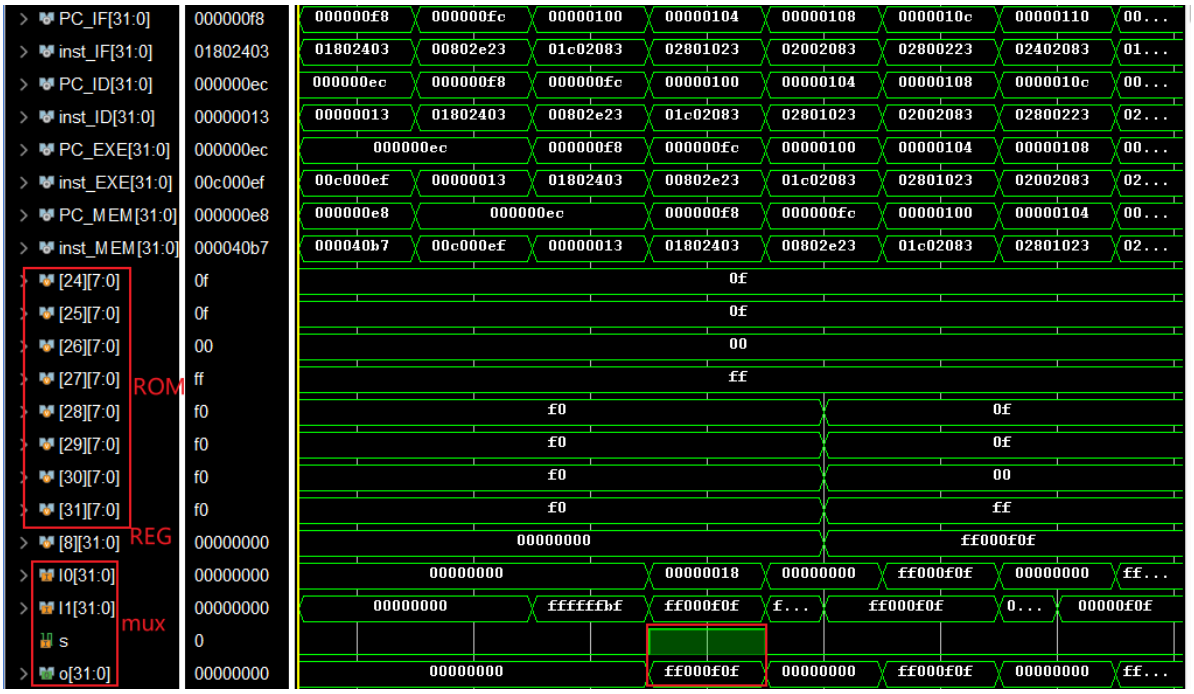
如下见仿真图，可以看出ID段第四条指令004100b3停留了两个周期，而且后一个周期在EXE阶段运行的是nop指令，说明了stall正常。图示信号中A,B,res是ALU的信号，我们可以看出EXE阶段执行的4、5指令确实获得了前递数据并计算出了正确结果。如果无forwarding，这两个冲突都需要两个周期的stall，而现在一个减少到了一个，另一个直接无stall。




```
label6:
lui  x1,4          # 000040B7
jal  x1,12         # 00C000EF
addi x0,x0,0       # 00000013
addi x0,x0,0       # 00000013
lw   x8, 24(x0)    # 01802403
sw   x8, 28(x0)    # 00802E23
```

如上，最下面的两条指令之间存在load-store前递，即EXE阶段的store指令的rs2和MEM阶段的load指令的rd相同，可通过一个mux筛除未写入数据的错误寄存器值，使用新数据存入内存。

如下见仿真示意图，可以看见关于mux的选择信号有一段高位信号，即冲突检测单元检测到了对应冲突，并通过选择信号选出了正确数据。另外，我们也可以看到ROM与REG的数据同时改变，一处由store指令在MEM阶段写入内存，另一处由load指令在WB阶段写入寄存器。



理论上是完全可行的，`forwarding`只是将后面数据传递到前面使用而已，目前仅剩的关于`load`指令的冲突只是缺少从`MEM`到`EXE`的数据路线而已，而实际上我们在`forwarding`部分设计的`load-store`修改一下就可以当作`B`操作数的前递线路，即在`rs`输入源前再添加`mux`用于选择`ID`还是`MEM`来的数据，再在冲突检测单元进行相关的检测和信号产生，即可完成从`MEM`到`EXE`的完全前递，实现只用`forwarding`解决`data hazard`。

但是我们需要注意的是时间问题，前递的数据并不是瞬间即时传输的，也是通过电路传输的，这是需要时间的，下图为如果存在前递，从`MEM`阶段开始的路线，花费的时间大致为两个阶段的时间和，而且还未计入可能存在的从`EXE`再次前递到`ID`阶段的时间，虽然不是完整的时间段，但是肯定是增加时间的。

而流水线`CPU`的时钟周期肯定要考虑到这种因素，不可能让前递的数据传输到一半就过了，即各阶段的实际执行时间都达到了原来的几倍，这种情况下的流水线是否比得过纯粹`stall`甚至单周期`CPU`呢。

现有的路线尽可能只在前递路线上添加`mux`，减少消耗的时间，平衡时钟周期和`stall`数。一味使用`forwarding`减少`stall`数可能并不是好的选择。