中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K8009915022 姓名：栗祺菁 专业： 计算机科学与技术

实验序号：5.1 实验名称：复杂处理器设计

一、 逻辑电路结构与仿真波形的截图及说明（比如关键RTL代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

此次流水实验主要分为三个部分，首先应让它流起来，即实现数据的一级级缓存，进而空出部件进行下一条指令的相应操作。然后是如何处理跳转指令以及写后读冲突。对于结构相关冲突，此次实验中虽然指令和数据存在同一个memory中，但是它访存的顺序由转接桥进行仲裁，先放行一个，故不需要我们考虑它的访存冲突。对于其他的两种数据相关冲突：读后写和写后写，因为是按序流动的流水线，所以也不需要考虑。

主体设计思路参考的是上一届体系结构的框架，以及《人人可懂的cpu设计》一书。

我设计的是五级流水，总共用IF,ID,EX,MEM,WB五个阶段，将四次握手分布在了IF,EX,MEM三个阶段中，custom_cpu里面主要负责五个板块的实例化，传递数据。每个阶段基本上都有六个控制流水的信号，其中两个信号分别来自其前一个和后一个阶段，例如，对于阶段A、B、C，则阶段B有信号A_to_B_valid，B_ready_go，B_allow_in，B_to_C_valid，B_valid,C_allow_in。

A_to_B_valid为高当上一个阶段流进来的数据是有效的，即不是要被舍去的数据或者已经流过的数据。

B_ready_go为高当B阶段的操作已经完成，并且握手信号已经握手（针对Inst_req，Inst，Mem_Req，MemWrite/MemRead四组信号的握手）。

B_allow_in为高当B当前的数据无效（即需要新的数据流入，这种情况发生在B的数据已经可以流走，而下一级堵住了，C_allow_in为低电平的时候）或者B可以走并且C可以进入（注意第二种情况和第一种情况有差别，因为如果不需要等待的话，B的数据一直是有效的）。

B_to_C_valid为高当此时B中数据有效并且可以走。

B_valid是这些信号中唯一一个reg型变量，即需要用到时序逻辑，复位时置为0，当B_allow_in时，置为A_to_B_valid，这很好理解，当从上一级传入的数据有效时，B中的数据有效，否则无效。

C_allow_in为高表示下一级流水已经处理完数据，并流走，这个时候可以将B的数据传入。

IF阶段：

IF阶段有两组握手信号，Inst_Req_Valid和Inst_Req_Ready、Inst_Ready和Inst_Valid。

只有在IF阶段取到有效的指令的时候才流出去，那么可以想到应该先进行第一组读指令请求握手成功与否的判断。用一个寄存器req_suc存储请求信号有没有握手成功，这个信号一直持续到IF流走，IF流走的条件就是req_suc和Inst_valid都为高。req_suc是时序逻辑，在时钟上升沿并且if_allow_in为高时变化，if_allow_in期间发送了读取指令请求，在此阶段等待，直到握手成功，此时if_valid信号拉高，if_allow_in信号拉低，req_suc不再改变，即相当于存储了握手成功的信号，req_suc维持为1。当前指令请求信号没有握手成功时，if_allow_in肯定是拉高的，当握手成功之后，if_allow_in拉低，表示一个请求已经发出去了，不能再发送请求（由此可知，向外发出的Inst_Req_Valid信号的赋值也由if_allow_in)决定。

Inst_Ready信号在req_suc为高时并且ID阶段可以流入时拉高（需要条件id_allow_in，否则提前完成握手，得到有效指令值，按照if_ready_go的赋值，它就会流出去，但是id阶段现在还不能接受数据），表示准备好接受指令，等待Inst_Valid拉高之后，读入的指令就是有效的，此时流到ID阶段。

经同学提醒，所有输出的Ready信号，均需在复位时拉高（上板测试的要求）。涉及上述信号的代码如下：

```
    //之前已经指令请求信号已经握手成功，并且此时输入的Inst_Valid信号也为高的时候，可以流走
    assign if_ready_go  = Inst_Valid && req_suc;

    //当前if可以接受新数据的话，就发送读取指令请求
    assign Inst_Req_Valid=to_if_valid && if_allow_in ;
    //当前指令请求已经握手成功了，并且id可以接受数据的话，就表明if可以接受指令
    assign Inst_Ready=id_allow_in && req_suc || reset;

    //if_allow_in期间发送了读取指令请求，在此阶段等待，直到握手成功，此时if_valid信号拉高，
if_allow_in信号拉低，req_suc不再改变，即相当于存储了握手成功的信号，req_suc维持为1
    always @(posedge clk)begin
        if(if_allow_in)begin
            req_suc<=Inst_Req_Valid && Inst_Req_Ready;
        end
    end
```

关于各类PC：

总共有if_pc,seq_pc,next_pc,next_pc_r四种pc，最终作为PC输出的是next_pc_r，而从if传到id的pc是if_pc。

首先在复位的时候，将if_pc置为32'hfffffffc，每次指令请求信号握手成功时，即PC的值已经被指令存储器接受了，此时更新if_pc为next_pc_r，表示下一条指令真正的PC值。seq_pc=if_pc+4,就是普通情况下的流水，pc值加4，按顺序取下一条指令。next_pc表示考虑了跳转情况的pc值，从id过来的bus中如果表明要跳转则next_pc为从id传过来的跳转地址。这里我实现的和ppt上有差别，我在id阶段判断是否跳转，即对于R[rs1]和R[rs2]的大小的判断没有放在alu中，理论上将其放在alu中也可以，那样的话表示跳转的bus就是从ex传到if阶段，若跳转成功，则比从id传到if多损失一个id阶段。

前面提到从id传到if的bus中包含了跳转地址和跳转使能信号，并且它们（br_en，br_target）是由id阶段的Instruction译码而来的。输入的指令值并不是在整个阶段都都为有效值的，只有在握手成功的时候为真正的Insntruction，之后就变成了32'b0。而if阶段要求next_pc的值保存一段时间，直到第一组握手信号握手成功，将其传给if_pc，所以我们增加一个寄存器next_pc_r存储下一个PC。可以知道，也可以用存储br_en,br_target或者在id阶段对于Instruction进行存储来代替对于next_pc进行存储，其本质都是为了存next_pc，故直接用next_pc_r。

考虑了跳转之后可以知道，next_pc_r的值一定先是seq_pc，再由于br_en拉高，而导致next_pc更新，输出的PC就才是真正的下一条pc值。所以若跳转成功，那么现在传送的PC和读出来的指令是无效的，即应该无效掉一个Id阶段。在if_to_id_valid信号内再与一个!br_en_r信号，即表示没有跳转的话，按普通情况处理，若有跳转，当前传出去的bus数据是无效的。

传向ID的pc是if_pc，因为输出PC之后，if_pc和next_pc_r都会更新，if_pc更新为了next_pc_r,传到ID阶段的bus中的指令对应的PC就是if_pc的值。

注意有一个br_en_r和br_en_t信号是为了处理laod指令后是branch指令，并且发生了写后读冲突的情况。此时由于id阶段的br_en的是组合逻辑，在写后读的数据传到之前，就会放出一个br_bus给if，这个br_bus是由比较错误的寄存器读数得出来的结果，就会造成问题。

下面为if阶段部分核心代码（省略输入输出和声明语句等）：

```
    assign to_if_valid  = ~reset;//复位之后，to_if就是无条件有效的
```

```
    assign if_ready_go  = Inst_Valid && req_suc; //之前已经指令请求信号已经握手成功，并且此时输入的Inst_Valid信号也为高的时候，可以流走
    assign if_allow_in  = !if_valid || if_ready_go && id_allow_in;
    assign if_to_id_valid =  if_valid && if_ready_go && !br_en_r;//如果是跳转指令的话，要无效掉当前传给id的数据

    //if_valid信号，在if_allow_in为高的时间内更新，
    always @(posedge clk) begin
        if (reset) begin
            if_valid <= 1'b0;
        end
        else if (if_allow_in) begin
            if_valid <= to_if_valid && Inst_Req_Ready;
        end
    end

    //当前if可以接受新数据的话，就发送读取指令请求
    assign Inst_Req_Valid=to_if_valid && if_allow_in ;
    //当前指令请求已经握手成功了，并且id可以接受数据的话，就表明if可以接受指令
    assign Inst_Ready=id_allow_in && req_suc || reset;

    //if_allow_in期间发送了读取指令请求，在此阶段等待，直到握手成功，此时if_valid信号拉高，if_allow_in信号拉低，req_suc不再改变，即相当于存储了握手成功的信号，req_suc维持为1
    always @(posedge clk)begin
        if(if_allow_in)begin
            req_suc<=Inst_Req_Valid && Inst_Req_Ready;
        end
    end

    //传给PC的是next_pc_r,所以开始的时候将if_pc设为-4；每次指令存储器接受请求之后，改变if_pc
    always @(posedge clk)begin
        if (reset) begin
            if_pc <= 32'hfffffffc;  //trick: to make nextpc be 0x00000000 during reset
        end
        else if (Inst_Req_Valid && Inst_Req_Ready) begin
            if_pc <= nextpc_r;
        end
    end

    //输出的pc为next_pc_r,因为输出PC之后，if_pc就变为了next_pc_r,所以当前读入的指令对应的是当前if_pc的值，故传给下一个阶段的bus是if_pc和两次握手成功之后的Instruction
    assign PC=nextpc_r;
    assign if_inst=Instruction;
    assign if_to_id_bus = {if_inst,if_pc};

    //next_pc的逻辑，br_en，br_target来自ID阶段
    assign seq_pc       = if_pc + 4;
    assign nextpc       =  (br_en) ? br_target : seq_pc;
```

ID阶段：

从上一阶段传来的bus需要用寄存器bus_r存一下，以保持在当前阶段没有流走的时候，bus的内容不变。否则，上一个阶段接受了新的数据，就会时实改变bus的值，这也是流水设计的核心内容之一，即一级级对数据进行缓存。

ID阶段接收IF阶段的PC值和Instruction值作为ID阶段译码的初始数据。因为实例化寄存器是在ID阶段，所以ID阶段还要从WB阶段接收包含，写使能、写地址和写数据的Bus，将数据写到寄存器中去。而ID阶段除了要传给下一个EX阶段的一些数据之外，还需要传给IF阶段一个br_bus，里面含有经过译码得到的跳转使能进和跳转地址，即跳转指令的处理。

以下为以上控制信号和数据bus的部分代码（暂时不考虑写后读的情况，后文会写）。

```
    assign id_allow_in     = !id_valid || id_ready_go && ex_allow_in;
    assign id_to_ex_valid = id_valid && id_ready_go;
    always @(posedge clk) begin
        if (reset) begin
        id_valid <= 1'b0;
        end
        else if (id_allow_in) begin
        id_valid <= if_to_id_valid;
        end
    end

    always @(posedge clk) begin
        if (if_to_id_valid && id_allow_in) begin
        if_to_id_bus_r <= if_to_id_bus;
        end
    end

    assign {id_inst,id_pc} = if_to_id_bus_r;
    assign {rf_we,rf_waddr,rf_wdata} = wb_to_rf_bus;
    assign br_bus       = {br_taken,br_target};
```

按照位置和功能对Instruction进行划分之后，得到opcode,rd,functs,rs1,rs2,shamt,funct7,imm,long_imm的值，含义与实验四汇总一样，不再赘述

PC：

注意，br_bus中跳转的使能信号，除了要求当前的指令是跳转指令，并且能够成功跳转之外，还要求此时id数据是有效的，即id_valid。

```
    //在ID阶段判断是否跳转，则在ID阶段进行大小的比较，less表示有符号比较时R[rs1]<R[rs2]，
lessu表示在无符号比较中R[rs1]<R[rs2]
    assign sub_res=rs1_value-rs2_value;
    assign less=(rs1_value[31]==1 && rs2_value[31]==0) || (((rs1_value[31]==0 &&
rs2_value[31]==0)||(rs1_value[31]==1 && rs2_value[31]==1)) && sub_res[31]==1);

    assign rs1_value_u={1'b0,rs1_value};
    assign rs2_value_u={1'b0,rs2_value};
    assign sub_res_u=rs1_value_u-rs2_value_u;
    assign less_u=sub_res_u[32]==1;

    //控制信号，表示next_PC的来源
    assign PC_jal=(opcode==7'b1101111);//jal
    assign PC_jalr=(opcode==7'b1100111);//jalr
    assign PC_branch=
    (opcode==7'b1100011) &&
    ((funct3==3'b000 && equal)||(funct3==3'b001 && !equal)
    ||(funct3==3'b100 && less)||(funct3==3'b101 && !less)
    ||(funct3==3'b110 && less_u) ||(funct3==3'b111 && !less_u));
    //beq,bne,blt,bge,bltu,bgeu
```

```
    //各类跳转、取数、读数地址的偏移量
    assign store_offset={{20{id_inst[31]}},id_inst[31:25],id_inst[11:7]};

    assign sign_extend_imm={{20{imm[11]}},imm};

    assign branch_offset={{20{id_inst[31]}},id_inst[7],
    id_inst[30:25],id_inst[11:8],1'b0};

    assign jalr_offset=sign_extend_imm+rs1_value;

    assign jal_offset={{12{id_inst[31]}},id_inst[19:12],
        id_inst[20],id_inst[30:21],1'b0};

    //next-pc的四种可能，Jump_address0可以不用，IF阶段考虑了PC+4的情况
    assign Jump_address0=id_pc+4;//else
    assign Jump_address1=id_pc+jal_offset; //jal
    assign Jump_address2={jalr_offset[31:1],1'b0};//jalr
    assign Jump_address3=id_pc+branch_offset;//blt,bltu,bge,bgeu,beq,bne

    //根据控制信号决定跳转地址
    assign br_target=(PC_jal)?Jump_address1:(PC_jalr)?Jump_address2:
    (PC_branch)?Jump_address3:Jump_address0;
    //跳转的使能信号，除了要求当前的指令是跳转指令，并且能够成功跳转之外，还要求此时id数据是有
效的，即id_valid
    assign br_taken=(PC_jal || PC_jalr || PC_branch) && id_valid;
```

Register：

在ID阶段不能够直接将此条指令要写的数据写进去，因为有可能是load指令，或者计算类指令，即还没有获得要写的数据，所以我们在ID阶段设置一个位宽为3的RF_wdata_src_id信号，来表示写数据的来源。并将已经知道的写使能信号，写地址信号、写数据来源信号RF_wdata_src_id及部分写数据（lui_wdata，auipc_wdata）向后传递。

下面是RF_wdata_src_id的赋值逻辑及Reg_file实例化。

```
RF_wdata_src_id=3'b000 //写alu_result，ex阶段计算完成
                3'b001 //写PC_id+4，id阶段计算完成
                3'b010 //写shift_result，ex阶段计算完成
                3'b011 //写lui_wdata,id阶段计算完成
                3'b100 //写auipc_wdata,id阶段计算完成
                3'b101 //写modified_read_data，即load指令，经过更改的Read_data，mem阶
段计算完成

//读地址为rs1和rs2,写地址和写数据均由wb阶段传入id的bus提供
reg_file
my_reg_file(.clk(clk),.rst(reset),.waddr(rf_waddr),.raddr1(rs1),.raddr2(rs2),
.wen(rf_we),.wdata(rf_wdata),.rdata1(reg_read_data1),.rdata2(reg_read_data2));
```

ALU和Shift：

同样地，将译码得到的ALUop（此实验中我将load和store指令的地址计算也放在ALU中了），操作数A和操作数B的值传递给EX；将译码得到的Shiftop，操作数A1和操作数B1的值传递给EX。实际上A和A1都是R[rs1]，只用传rs1_value即可。

id_to_ex_bus:

```
assign id_to_ex_bus={
    id_pc,ALUop_id,Shiftop_id,B_id,B1_id,//32+3+2+32+32
    rs1_value,rs2_value,//32+32

RF_write_id,RF_waddr_id,lui_wdata_id,auipc_wdata_id,RF_wdata_src_id,//1+5+32+32+
3
    mem_read_id,mem_write_id,ls_type_id,//1+1+3
    1'b0
    };
//id_pc是为了在wb阶段流出去，进入FIFO进行比对，否则理论上在ID阶段之后，pc值已经没用了（如果将写数据pc+4的值传递下去的话）
//mem_read_id信号和mem_write_id信号表示当前指令是load或者是store，流到EX阶段作为握手信号
//ls_type_id的值为funct3,用于load时对Read_data进行改造、store时计算
Write_data,Write_strb
//最后这个1'b0是为了凑成4的整数倍，方便查看波形
```

EX阶段：

实现Memory请求信号的握手。

EX阶段如果流入的指令是load或store的话，要发出读或写Memory请求（即MemRead或MemWrite），并在Mem_Req_Ready拉高，即握手成功时，进入MEM阶段。若不是load或store指令，因为只有组合逻辑，ex_ready_go一直拉高。以下是部分控制信号。

```
    assign MemWrite=mem_write_ex && ex_valid && !req_suc;//在握手成功之后，拉低MemWrite,防止写入无关数据
    assign MemRead=mem_read_ex && ex_valid && !req_suc;//在握手成功后，拉低MemRead,防止读出无关数据，覆盖了有效数据
    assign ex_ready_go=req_suc || !mem_write_ex && !mem_read_ex;//req_suc表示握手成功，可以流走;如果是别的指令，无条件可以流走
    assign ex_allow_in     = !ex_valid || ex_ready_go && mem_allow_in;
    assign ex_to_mem_valid =  ex_valid && ex_ready_go;

    //req_suc,表示握手成功
    always @(posedge clk)begin
        if(!req_suc || ex_allow_in)begin
            req_suc<=Mem_Req_Ready && (MemWrite || MemRead);
        end
    end
```

获取id_to_ex_bus_r的各类数据，并实例化ALU,Shifter,逻辑简单，不放代码。

现在已经可以知道store指令的Address、Write_data和Write_strb的值，故在EX阶段计算并进行这些值的输出。Address由此阶段的ALU计算出来，并取低两位为0。Write_data由ID传来的ls_type(funct3)和reg_read_data2决定。Write_strb由ls_type和地址低两位值n决定。

```
     //store:MemWrite
    wire [1:0]n;
    assign Address={alu_result_ex[31:2],2'b00};
    assign n=alu_result_ex[1:0];
    assign Write_data=
        (ls_type_ex==3'b000)?{4{reg_read_data2_ex[7:0]}}:
        (ls_type_ex==3'b001)?{2{reg_read_data2_ex[15:0]}}:
```

```
        (ls_type_ex==3'b010)?reg_read_data2_ex:32'b0;

    assign Write_strb=
        (ls_type_ex==3'b000)?((n==0)?4'b0001:(n==1)?4'b0010:
            (n==2)?4'b0100:(n==3)?4'b1000:4'b0000):
        (ls_type_ex==3'b001)?((n==0)?4'b0011:(n==2)?4'b1100:4'b0000):
        (ls_type_ex==3'b010)?4'b1111:4'b0000;
```

MEM阶段：

实现Read_data信号的握手。

现在已经实现了大多数的指令，在MEM阶段要获得load指令的写寄存器数据，并将一级级传下来的数据利用RF_wdata_src进行整合，获得真正的写数据。

Mem阶段如果是load指令的话要实现一个Read_data的握手信号，其它指令则直接流过即可。mem_read_mem是从ID级的mem_read_id，EX阶段的mem_read_ex，一直流下来的信号。

以下是控制信号的逻辑。

```
    assign Read_data_Ready=mem_read_mem && mem_valid && wb_allow_in || reset ;//除复位时期之外，在该指令是load指令，并且mem_valid为高,即该指令第一次经过此阶段，还未处理的时候，发出读数据请求信号。
    assign mem_ready_go=mem_read_mem?Read_data_Valid:1'b1;//如果是laod指令，则握手成功再走，否则直接流走
    assign mem_allow_in     = !mem_valid || mem_ready_go && wb_allow_in;
    assign mem_to_wb_valid = mem_valid && mem_ready_go;
    always @(posedge clk) begin
        if (reset) begin
            mem_valid <= 1'b0;
        end
        else if (mem_allow_in) begin
            mem_valid <= ex_to_mem_valid;
        end
    end
```

EX阶段完成了与MEM交互请求信号的握手，此时store或者load的地址（alu result)都已经在EX阶段传出去，对于load指令，在MEM阶段由ls_type(funct3)、地址低两位值n、以及握手成功时的Read_data值计算出modified_read_data（所有计算都是组合逻辑，握手成功之前，Read_data也有值，所以中间会有错误的计算值，但是只有在握手成功时，数据才传向WB，组合逻辑的结果是实时变化的，所以流走的是正确的计算值）。至此，所有写回寄存器的可能数据都已经得到，可以由前面传下来的RF_wdata_src、alu_result、pc、shift_result、lui_wdata、auipc_wdata、以及当前算出来的modified_read_data计算出RF_write_data的值，并将此值传入WB阶段。以下为modified_read_data的逻辑。

```
    assign n=alu_result_mem[1:0];
    assign mem_read_data=Read_data;

    //load:Memread
    assign lb_data=(n==0)?{{24{mem_read_data[7]}},mem_read_data[7:0]}:
    (n==1)?{{24{mem_read_data[15]}},mem_read_data[15:8]}:
    (n==2)?{{24{mem_read_data[23]}},mem_read_data[23:16]}:
    (n==3)?{{24{mem_read_data[31]}},mem_read_data[31:24]}:31'b0;

    assign lh_data=(n==0)?{{16{mem_read_data[15]}},mem_read_data[15:0]}:
```

```
    (n==2)?{{16{mem_read_data[31]}},mem_read_data[31:16]}:31'b0;

    assign lbu_data=(n==0)?{24'b0,mem_read_data[7:0]}:
    (n==1)?{24'b0,mem_read_data[15:8]}:
    (n==2)?{24'b0,mem_read_data[23:16]}:
    (n==3)?{24'b0,mem_read_data[31:24]}:31'b0;

    assign lhu_data=(n==0)?{16'b0,mem_read_data[15:0]}:
    (n==2)?{16'b0,mem_read_data[31:16]}:31'b0;

    assign modified_read_data=(ls_type_mem==3'b000)?lb_data:
    (ls_type_mem==3'b001)?lh_data:
    (ls_type_mem==3'b010)?mem_read_data:
    (ls_type_mem==3'b100)?lbu_data:
    (ls_type_mem==3'b101)?lhu_data:32'b0;

    //RF_wdata_src代表写数据的来源，由ID阶段译码的结果得到
    assign RF_write_data_mem=(RF_wdata_src_mem==3'b000)?alu_result_mem:
    (RF_wdata_src_mem==3'b001)?pc_mem+4:
    (RF_wdata_src_mem==3'b010)?shift_result_mem:
    (RF_wdata_src_mem==3'b011)?lui_wdata_mem:
    (RF_wdata_src_mem==3'b100)?auipc_wdata_mem:
    (RF_wdata_src_mem==3'b101)?modified_read_data:31'b0;
```

自此，从MEM阶段传向下一阶段的数据就只需要pc,RF_write,RF_waddr,RF_write_data，其中RF_write表示写使能，是从ID阶段流下来的。

WB阶段：

WB阶段需要实现inst_retire_valid的赋值，和inst_retired的赋值，后者只需要按照顺序排列pc，RF_write，RF_waddr，RF_write_data信号即可，都已经在MEM阶段完成了计算。前者应是wb_valid && !inst_retired_fifo_full，即fifo没有满，并且当前wb的数据有效。wb_ready_go信号的赋值如下，注意需要考虑BHV_SIM undef的情况，即`else语句。其它的逻辑比较简单，不再赘述。

```
    `ifdef BHV_SIM
        assign wb_ready_go = !inst_retired_fifo_full;
    `else
        assign wb_ready_go = 1'b1;
    `endif
```

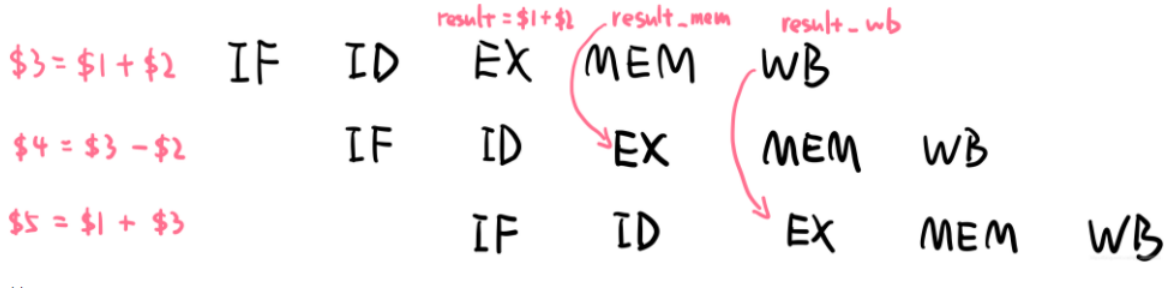
写后读冲突处理：

现在已经实现了所有指令的逻辑，并且顺利地流起来了，但是还有一个重要的问题没有考虑，就是数据冲突之写后读。注意到前面并没有给出id_ready_go的赋值，解决了写后读之后将给出。

容易想到的一点是，如果发生了写后读的情况，就直接堵塞住整个流水线，直到完成了写数据操作，这样读出来的就是有效值。但是这样流水线的吞吐率或效率降低了很多。

我处理写后读的方式是数据前递，参考的是下图的思想。他这里在EX阶段决定真实的rs1_value和rs2_value，我是在ID阶段决定读寄存器应该得到的真实值，因为我的Branch指令的跳转成功与否是由ID阶段决定的，即在ID阶段就得得到到真实的R[rs1]和R[rs2]。理论上和以下思路没有什么差别，只是需要向前传三个通路，从EX到ID，MEM到ID，WB到ID。比较当前指令的源寄存器和以上三条指令的目标寄存器是否相同，并按照优先级选择bus中的数据。注意到，实际上并不是所有的指令都需要用到R[rs1]和R[rs2]，有些指令只需要用到一个，有的一个也不需要用到，比如jal指令。所以可以更加细地分情况考虑，这样可以减少不必要的等待，我这里就没有再细化考虑了。

于是我们可以创建这样的转发机制：在每条指令的EX级开始时都进行一个检测，看本条指令的源寄存器与上两条指令的目标寄存器是否相同（由下面流水线的图也可以看出，一条指令的数据会影响到接下来的两条数据），如果相同，则把处于MEM / WB级的上一条 / 上两条的指令的数据转发过来，再在本条指令的 ALU 前做一个MUX，决定最终进入 ALU 运算的是原始的rs1 / rs2的数据还是转发过来的新数据。



先给出三个阶段传给id的写后读相关bus的逻辑。

每一个写后读bus中都有四个元素：addr_valid,data_valid,RF_waddr,RF_write_data。

后两个比较好理解，就是当前指令要写的地址和数据，写数据对于ex阶段，就是alu_result或者shift_result；对于mem阶段就是RF_write_data_mem，wb阶段就是RF_write_data_wb。

第一个元素addr_valid的值均为A_valid && RF_write，其中A为ex或mem或wb。表示的含义是这条指令确实有效，并且该指令包含有写操作，只有这样才需要继续比较地址。

第二个元素data_valid对于不同的阶段有不同的值，表示此时bus传入id的写的数据是有效的。ex阶段的data_valid是~mem_read_ex，这是因为对于load指令，它确实要写寄存器，但是ex阶段算出来的是读Memory的Address，而不是要写的数据，真正要写的数据是在mem阶段得到的。所以如果是mem_read，即load指令的话，ex传到id的写后读bus的数据（alu_result或shift_result）是无效的。mem阶段的data_valid是mem_ready_go，回忆一下，mem_ready_go的逻辑是mem_ready_go=mem_read_mem?Read_data_Valid:1'b1;即如果是laod指令，则握手成功再走，否则直接流走。在mem_ready_go为低的时候，因为还没有握手成功，所谓的RF_write_data其实是无效的。对于WB阶段，没有握手信号，也没有特殊情况，data_valid直接为1。

```
    //ex_to_id写后读bus
    assign write_valid=ex_valid && RF_write_ex;

    assign ex_read_after_write_bus=
    {write_valid,~mem_read_ex,RF_waddr_ex,useful_result_ex};
    assign useful_result_ex=(ls_type_ex==3'b001 || ls_type_ex==3'b101)?
shift_result_ex:alu_result_ex;

    //mem_to_id写后读bus
    assign write_valid=mem_valid && RF_write_mem;
    assign mem_read_after_write_bus=
{write_valid,mem_ready_go,RF_waddr_mem,RF_write_data_mem};


    //wb_to_id写后读bus
    assign write_valid=wb_valid && RF_write_wb;
    assign wb_read_after_write_bus=
{write_valid,1'b1,RF_waddr_wb,RF_write_data_wb};
```

下面看ID阶段的处理写后读的逻辑。

前面已经说过了，先看bus中的第一个元素addr_valid，实际上叫这个名字并不合理，它代表的不是地址有效，而是是不是真的要写，但是鉴于它为高的时候，才需要比较地址，并且bus中的地址此时一定真的是写地址，姑且叫这个名字。按照EX>MEM>WB的优先级，判断R[rs1]和R[rs2]的真值来自哪个bus,若都不是，说明没有发生写后读冲突，直接从Register里面读数就可以。

注意若采用数据前递的值要与上一个|rs1或|rs2信号（为高时，表示读地址不为0）。这是因为，若读地址为0，则根据我们写的Register，读出来的数应该为全零。若写地址为0，则写不进去。若不加上这个条件，在ex/mem/wb阶段如果要在地址0写一个不为0的数（alu_result……），之后又去读地址0的值，那么就发生了写后读冲突，并且采用了数据前递的值——一个不为0的数据。这是不希望看到的，毕竟没有写进去，自然也不应该读出来。

```
    //将bus中的各数据取出来，涉及写后读的各类判断都是组合逻辑，所以不需要像A_to_B_bus那样专门设置一个bus_r，直接取数就可以。
    //raw=read after write
    assign {ex_raw_addr_valid,ex_raw_data_valid,ex_raw_addr,ex_raw_data}=ex_read_after_write_bus;
    assign {mem_raw_addr_valid,mem_raw_data_valid,mem_raw_addr,mem_raw_data}=mem_read_after_write_bus;
    assign {wb_raw_addr_valid,wb_raw_data_valid,wb_raw_addr,wb_raw_data}=wb_read_after_write_bus;

    //判断R[rs1]和R[rs2]来自哪里，注意要与上一个|rs1或|rs2,原因在前面说了。下面是按照优先级EX>MEM>WB来判断的数据来源的
    assign rs1_from_ex=(|rs1) & ex_raw_addr_valid & (ex_raw_addr==rs1);
    assign rs2_from_ex=(|rs2) & ex_raw_addr_valid & (ex_raw_addr==rs2);
    assign rs1_from_mem=(|rs1) & ~rs1_from_ex & mem_raw_addr_valid & (mem_raw_addr==rs1);
    assign rs2_from_mem=(|rs2) & ~rs2_from_ex & mem_raw_addr_valid & (mem_raw_addr==rs2);
    assign rs1_from_wb=(|rs1) & ~rs1_from_ex & ~rs1_from_mem & wb_raw_addr_valid & (wb_raw_addr==rs1);
    assign rs2_from_wb=(|rs2) & ~rs2_from_ex & ~rs2_from_mem & wb_raw_addr_valid & (wb_raw_addr==rs2);
    assign rs1_from_rf=~(rs1_from_ex | rs1_from_mem | rs1_from_wb);
    assign rs2_from_rf=~(rs2_from_ex | rs2_from_mem | rs2_from_wb);
```

```
    //将bus中的数据或者rf读出来的值赋值给rs1_value(或rs2_value)
    assign rs1_value=({32{rs1_from_ex}} & ex_raw_data)|
            ({32{rs1_from_mem}} & mem_raw_data)|
            ({32{rs1_from_wb}} & wb_raw_data)|
            ({32{rs1_from_rf}} & reg_read_data1);

    assign rs2_value=({32{rs2_from_ex}} & ex_raw_data)|
            ({32{rs2_from_mem}} & mem_raw_data)|
            ({32{rs2_from_wb}} & wb_raw_data)|
            ({32{rs2_from_rf}} & reg_read_data2);
```

下面介绍id_ready_go的逻辑，这里要用到第二个元素A_raw_data_valid信号，其中A是ex,mem或者wb。前面说过，只有data_valid为高时bus中的数据才能够当做真正的rs1_value或rs2_value，故在它为高时，才让id流下去。我们考虑不能流下去的情况：若rs1或rs2至少有一个来自ex、mem、wb这三个bus之一，并且其对应的data_valid为低，则堵塞。

```
assign id_ready_go     =
!(opcode==7'b1100011 && (!rs1_from_rf || !rs2_from_rf) //branch
||!ex_raw_data_valid && (rs1_from_ex || rs2_from_ex)
||!mem_raw_data_valid && (rs1_from_mem || rs2_from_mem)
||!wb_raw_data_valid && (rs1_from_wb || rs2_from_wb)
);
```

至此完成了对于写后读的判断。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

主要需要考虑的就是怎么让流水线流起来，以及流起来之后如何处理跳转指令和写后读冲突。中间最大的问题就是怎么调节各阶段间握手信号的时序，经常随便改改就又堵住了，流不出来指令。开始就把思路想好了，有了参考书也觉得不会很难，写起来才发现问题越来越多。最后更新框架没有成功，手动更改了test_cpu.v中的内容。很多点都在前面说过了，不细说了，写太多了。

三、 对讲义中思考题（如有）的理解和回答

思考题大概是分支预测？粗略看了看提供的资料《分支预测的过去现在和将来》，类似于综述，讲得不是很具体。这个实验我没有实现分支预测，以后再系统地了解这方面内容吧。

四、 在课后，你花费了大约35小时完成此次实验。

不记得做了多少小时了，三周多。

五、 对于此次实验的心得、感受和建议

这个实验做了三周多，写这个实验报告写了一整天了，八千多字，不想写了。很多细节问题，如果还要展开说的话，觉得可以突破一万五千字。做实验的过程中，无数次和同学吐槽，我好想放弃，好想中途而废。但是又觉得已经花了这么多时间了，实在是舍不得。写代码就是这样，如果最后没有写成功，不管花了多少时间，也相当于没有写（当然，可能debug能力还有耐心得到了增长)。记得刚开始写两三天的时候，我就和楼持恒说，我觉得我写完一半了，结果写到后面，越写越发现要考虑的问题还有很多。调的过程更加是这样，从第一条指令就开始有问题，一条一条指令地调，都不记得花费了多少个日日夜夜。行为仿真通过后，又担心会像别的同学一样上板会有问题，幸好最后上板也顺利通过了。写的过程、调的过程都太艰难了，希望老师能多给点辛苦分。

最后感谢王嵩岳助教和楼持恒同学对我的帮助，在我找不到bug的时候和不知道怎么写的时候，给我提供正确的思考方向，没有他们的帮助和鼓励，我觉得我完成不了这个实验。