实验报告成绩:	成绩评定日期:

2021 ~ 2022 学年秋季学期

A3705060050《计算机系统》必修课

课程实验报告

班级: 人工智能 1901

组长: 张浩塬 20195289

组员: 刘颖 20195285

组员: 颜季辉 20195277

报告日期: 2021.12.18

目录

代码工作量 60% 20% 20%

组长 张浩塬 score 30

流水线框架

通用寄存器数据相关

跳转与访存和暂停机制

自制乘法器

Pass point 43

组员 刘颖 score 28

访存指令以及框架接线

乘法除法连线以及暂停机制

Pass point 64

组员 颜季辉 score 28

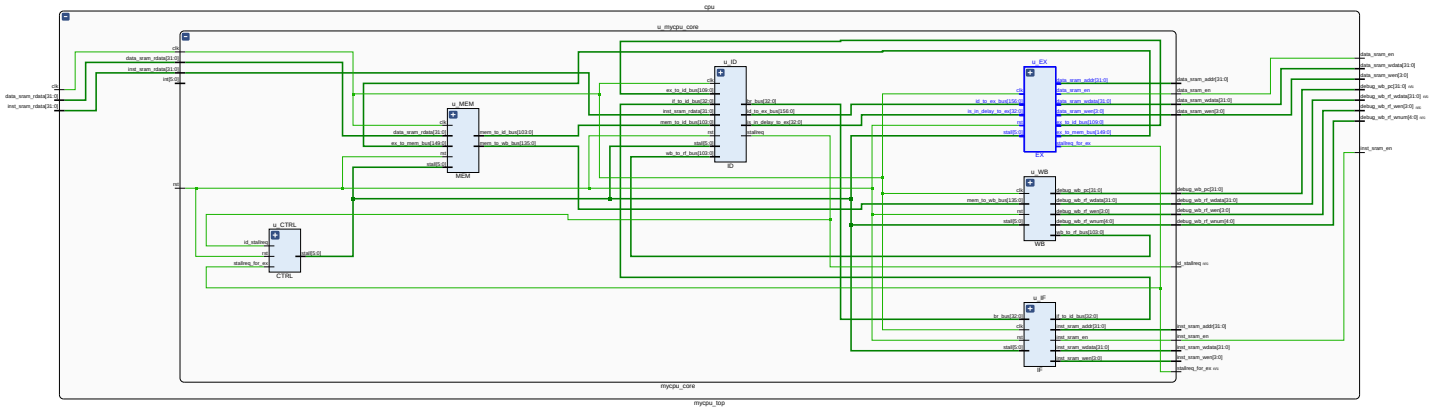
Hilo 特殊寄存器框架

Regfile 复用连线

Pass point 58

共 pass point 64

程序运行环境 Vivado


cpu
u_mycpu_core
u_ID
u_EX
u_MEM
u_WB
u_CTRL
u_IF
ID
EX
MEM
WB
CTRL
IF
mycpu_core
mycpu_top

IF 段

Pc 的值根据 stall[0]判断是否暂停，否则再根据传过来的

br_bus

```
assign next_pc =(stall[0]==`Stop)? pc_reg :br_e ? br_addr
        : pc_reg + 32'h4;
```

如果是跳转，则跳转到 br_addr，否则直接加 4

ID

从 if 段传过来 PC 和 ce

从 wb 段传的 wb_to_rf_bus 用来传入 regfile

将变量传给 regfile 后得到寄存器保存的 rdata 值

以及 hilo 寄存器的相关信息

从 ex 和 mem 数据相关传回来的通用寄存器以及特殊寄存器的信息

```
assign {
    ex_to_id_we,
    ex_to_id_waddr,
    ex_to_id_hi_we,
    ex_to_id_lo_we,
    ex_to_id_hi,
    ex_to_id_lo,
    ex_to_id_op,
    ex_to_id_wdata
}=ex_to_id_bus;

assign {
    mem_to_id_hi_we,
    mem_to_id_lo_we,
    mem_to_id_we,
    mem_to_id_waddr,
    mem_to_id_wdata,
    mem_to_id_hi,
    mem_to_id_lo
}=mem_to_id_bus;
```

如果其中的 we 为 0，说明不进行回写保存，也就不存在数据相关，因为之后也不会更新。若 we 为 1 而且地址相等，那么就把相应的传回来的值赋给 reg_o

```
assign beq=inst_beq&rs_eq_rt;
assign jr=inst_jr;
assign jal=inst_jal;
assign jalr=inst_jalr;
assign bne=inst_bne&(~rs_eq_rt);
assign beqz=inst_beqz&(~reg_o1);
assign j=inst_j;
assign bgez=~reg_o1[31]&inst_bgez;
assign bgtz=~reg_o1[31]&(reg_o1>32'b0)&inst_bgtz;
assign blez = (reg_o1[31] | reg_o1==32'b0)&inst_blez;
assign bltz = reg_o1[31]&inst_bltz;
assign bgezal =~reg_o1[31]&inst_bgezal;
assign bltzal=reg_o1[31]&inst_bltzal;
```

跳转指令判断是否进行相应的操作，如果为 1 则执行操作

```
};
//暂停机制
 assign stallreq=((ex_to_id_op==`LW|ex_to_id_op==`LB| ex_to_id_op==`LBU|ex_to_id_op==`LH|ex_to_id_op==`LHU)
 &&(ce==1'b1)&&(ex_to_id_we==1'b1)&&(ex_to_id_waddr==rs))?
`Stop :((ex_to_id_op==`LW|ex_to_id_op==`LB| ex_to_id_op==`LBU|ex_to_id_op==`LH|ex_to_id_op==`LHU)
&&(ce==1'b1)&&(ex_to_id_we==1'b1)&&(ex_to_id_waddr==rt))? `Stop: `NoStop;
```

读取指令的读取数在 mem 阶段才能得到，如果下一条指令要用到这个值，而上一条指令此时还在 ex 段,所以要暂停一个周期。判断 ex 阶段的指令是否是读取并且是否地址相等，如果都满足说明要用到读取的值，则暂停。

```
    // end
    else if (stall[1]==`Stop && stall[2]==`NoStop) begin
        if_to_id_bus_r <= `IF_TO_ID_WD'b0;
        id_stop        <= 1'b0;
    end
    else if (stall[1]==`NoStop) begin
        if_to_id_bus_r <= if_to_id_bus;
        id_stop        <=1'b0;
    end
    if (stall[2]==`Stop) begin
        id_stop        <= 1'b1;
    end
end


assign inst =  (~id_stop)?  inst_sram_rdata:inst;
```

由于在 id 段才能发出暂停信号，而此时 if 段 pc 的值已经+4 了

虽然在 id 段的 bus 没变，但是 pc 传到内部输入给 id 段的 inst 已经变了

因为 inst 是根据 if 段 pc 的值。而 if 有两次相同的 pc 值，而第一次是对应上一条指令的，所以要第一次的 inst 取上一次的。那就要判断是否出现了暂停。于是我加了个 id_stop(独创做法，如有雷同，应该是我告诉的...)因为 idstop 通过了级间寄存器，所以跟 if_to_id_bus_r 的时序一样

EX 段

在 ex 段确定要传入内存的 data_sram_wen 和 data_sram_addr 的值

以及存储指令的 data_sram_wdata 的值

```
assign data_sram_addr=data_sram_en? rf_rdata1+{{16{inst[15]}},inst[15:0]}:32'b0;
assign data_sram_wen=inst_sw? 4'b1111:(inst_sb&data_sram_addr[1:0]==2'b00)?4'b0001
:(inst_sb&data_sram_addr[1:0]==2'b01)?4'b0010:(inst_sb&data_sram_addr[1:0]==2'b10)?4'b0100
:(inst_sb&data_sram_addr[1:0]==2'b11)?4'b1000:(inst_sh&data_sram_addr[1:0]==2'b00)?4'b0011
:(inst_sh&data_sram_addr[1:0]==2'b10)?4'b1100:sel_rf_res? 4'b0000:0;

assign data_sram_wdata=inst_sb?{ 4{rf_rdata2[7:0]}}:
```

对于 sb 和 sh 指令，根据地址后两位的值，确定 wen

除法用的学长的模板

乘法采用自制多周期加法树，并进行了位宽的优化

具体内部操作在后面说明手册上会写

在外部进行乘法的每周期输入

```
else begin
stallreq_for_mul= `NoStop;
mul_op1=`ZeroWord;
mul_op2=`ZeroWord;
mul_start=1'b0;
mul_sign=1'b0;
mul_already_start=1'b0;
case({inst_mult,inst_multu})
2'b10:begin
if(mul_ready==1'b0&~mul_already_start)begin
stallreq_for_mul= `Stop;
mul_op1=rf_rdata1;
mul_op2=rf_rdata2;
mul_start=1'b1;
mul_sign=1'b1;
mul_already_start=1'b1;
end
else if(mul_ready==1'b0&mul_already_start)begin
stallreq_for_mul= `Stop;
mul_op1=`ZeroWord;
mul_op2=`ZeroWord;
mul_start=1'b1;
mul_sign=1'b1;
mul_already_start=1'b1;
end
else if(mul_ready==1'b1)begin
stallreq_for_mul= `NoStop;
mul_op1=`ZeroWord;
mul_op2=`ZeroWord;
mul_start=1'b0;
mul_sign=1'b1;
```

由于加法树操作只有第一个周期需要操作数（第二个周期需要的是相应层的树）所以多设了一个 mul_already_start 代表乘法已经开始一个周期了（跟 mul_start 有区别） 如果已经开始一个周期了，说明已经不需要输入操作数了，置零即可。当 mlu 传出的 mul_ready=1 的时候，说明乘法已经完成，此时把 start 置零，代表乘法关闭。

MEM 段

Input data_sram_rdata

如果之前的 data_sram_en 为 1，data_sram_waddr 为有效地址的情况下，data_sram_rdata 会从内存传回来值，此时要在 mem 段进行数据的处理，方法同存储指令，sel_rf_ld 判断是否是读取指令，如果是，则 memresult 为数据处理后

的值

WB

回写阶段，没啥好说的

Regfile

跟特殊寄存器复用，并在 rf_rdata 复用 hilo 的值

并且在 regfile 中执行相隔三条指令的数据相关。就相当于不需要级间寄存器中保存完值就给 rdata,就相当于进货和出货一起来的，如果要卖出去的货正好跟进的货一致，就在放到仓库之前直接拿出来卖。

在跟 hilo 复用的同时注意区分是用 hilo 的值还是通用寄存器的值，并且注意 addr=0 的位置（因为 Hilo 寄存器要操作的时候 addr 也是为 0 的，此时就要放在 Hilo 的判断之后）

心得体会

组长 张浩塬

跟之前的写程序不一样，以前写程序都是从上往下执行的，而在 CPU 这种硬件层面是并行执行的。让我体验到新的逻辑架构，对流水线的掌握也提高了不少也对各种乘法器有很深的了解，尤其是加法树。时序逻辑以及组合逻辑的概念也清晰起来（所以再也不会用混合逻辑了）

组员 刘颖

通过实验深入了解了 CPU 内部的工作原理，对数据在 CPU 内的变换有了更加深刻的认识，同时在写程序的过程中还学到了 Verilog 语言的使用，收获颇多。最重要的是将课堂上的理论知识运用到实践中，既提高了动手能力，又对知识点有了更深刻的理解。

组员 颜季辉

本次实验首先加深了我对 CPU 结构以及工作原理的整体认知，尤其是对五级流水线结构的认知。在构建 hilo 寄存器框架时，对数据相关有了深刻的认知，并且学习实践了处理数据相关的方法。在查找资料和编写程序时初步学习到了一些 Verlog 的语法与技巧。

参考资料

自己动手做 CPU-雷思磊

乘法器说明

以加法树算法为基础，纯时序逻辑，6 层树加上输入输出周期时延，共八周期。

```
assign mul_sel2={mul_op2[31]&mul_sign}? ~mul_op2+1 : mul_op2;
always @(posedge clk)
begin
    if(!resetn | !mul_start_i )begin
      state <=0;
      mul_ready<=0;
    end
    else
    begin
        case(state)
        3'b000:begin
        state<=1;
        mul_ready<=0;
        end
        3'b001:begin
        state<=2;
        mul_ready<=0;
        end
        3'b010:begin
        state<=3;
        mul_ready<=0;
        end
        3'b011:begin
        state<=4;
        mul_ready<=0;
        end
        3'b100:begin
        state<=5;
        mul_ready<=0;
        end
        3'b101:begin
```

每次时钟周期的 state

```
always @(posedge clk)begin
if(mul_start_i&state==0)begin

for (loop=0;loop<32;loop=loop+1)
        begin
        tree1[loop] <= mul_sel2[loop]? mul_sel1:0;
        end
 end
 else begin
       for (loop=0;loop<32;loop=loop+1)
        begin
        tree1[loop] <= 0;
        end
 end
end
 always @(posedge clk)begin
 if(state==1)begin
     tree2_1<= tree1[0]+{tree1[1],1'b0} ;
     tree2_2<= tree1[2]+{tree1[3],1'b0} ;
     tree2_3<= tree1[4]+{tree1[5],1'b0} ;
     tree2_4<= tree1[6]+{tree1[7],1'b0} ;
     tree2_5<= tree1[8]+{tree1[9],1'b0} ;
     tree2_6<= tree1[10]+{tree1[11],1'b0} ;
     tree2_7<= tree1[12]+{tree1[13],1'b0} ;
     tree2_8<= tree1[14]+{tree1[15],1'b0} ;
 always @(posedge clk)begin
if(state==4)begin
       tree5_1<= tree4_1+{tree4_2,8'b0} ;
       tree5_2<= tree4_3+{tree4_4,8'b0} ;
      end
      else begin
       tree5_1<=0 ;
       tree5_2<=0 ;
 end
 end
 always @(posedge clk)begin
if(state==5)begin
     tree6_out<=tree5_1+{tree5_2,16'b0};
     end
     else begin
      tree6_out <=0;
 end
 end
  always @(posedge clk)begin
if(state==6)begin
     result<=(mul_sign&(mul_op1[31]^mul_op2[31]))? ~tree6_out+1 : tree6_out;
     end
     else begin
     result <=0;
 end
 end
```

具体运算过程，最后判断结果。

此外，对于位宽进行了优化。

第一层树获取的是操作数 1 的每一个有效位的值，而第二层树把第一层相邻两个位相加，由于不相关性，所以 16 次加法是并行的。对于这次加法来说，位数的差值为 1，但也要考虑到进位的情况，所以多分配两位

```
reg [31:0] tree1 [31:0];
reg [33:0] tree2_1;
reg [33:0] tree2_2;
reg [33:0] tree2_3;
```

同理，在第三层树相加的时候，把前面看作整体，相差 2 位，即 2^1 位同时也要考虑到进位的情况，所以在原有的基础上多分配三位。

```
reg [33:0] tree2_16;
reg [36:0] tree3_1;
reg [36:0] tree3_2;
reg [36:0] tree3_3;
reg [36:0] tree3_4;
```

于是，第 4 层多分配 4+1 位 第五层多分配 8+1 位

```
reg [41:0] tree4_4;
reg [50:0] tree5_1;
reg [50:0] tree5_2;
```

此时 第五层的位数为 51，第六层按照这个加法是 16+1，但是最高位是 64，也就对应 63 索引，所以最后一层是 63

```
reg [63:0] tree6_out;
```

之所以会出现这个问题是进位分配过多。但前面的进位不能不考虑，所以可以从后往前推。对于 51 位的第五层，第六层树相加需要将其中一个移 16 位

```
tree6_out<=tree5_1+{tree5_2,16'b0};
end
```

51+16 为 67，而结果位数为 64，也就是说，最高三位的值必是 0

所以对 51 进行优化，减三位变 48

```
reg [47:0] tree5_1;
reg [47:0] tree5_2;
```

对于加法树来说，只需要在第一周期把操作数传入即可，其他周期置零就行，因为后面周期的输入是上一层的树的输出。