体系结构Lab2实验报告

PB16050567 陈炜

实验目标

本实验的实验目标为：实现一个基于RISCV指令集的流水线CPU，使其完成以下功能：

1、能正确执行RISC-V32位指令集中用户级指令的大部分运算。

2、能实现非对齐的读写。

3、能通过转发正确处理冲突。

实验环境和工具

实验环境：Vivado2017.4

实验工具：Vivado自带编辑器和波形仿真工具

实验内容和过程

阶段一：

阶段一要实现所有的算数运算，主要要完善以下模块：

NPC_Generator:

实现：使用一个always模块完成组合逻辑的NPC_Generator即可

要点：各个不同的跳转信号是有优先级的，BrE和JalrE的优先级应该高于JalD,而BrE和JalrE不会同时出现，所以不用考虑这两个的优先级问题。

IDSeg_Reg:

实现：将clk和A[31:2]传送到Instruction Memory的对应的接口即可。

要点：此处应该传送的是[31:2]，因为Instruction Memory不支持非对齐的访问，所以低两位是没有必要传入的。

ControUnit:

实现：每一个输出信号对应一个always模块，根据传入的Op，Fn3和Fn7的不同在每一个时钟上升沿决定输出信号的值。如下例:

```
//JalrD
    always@(*)
    begin
        if(Op==C_Jalr)
            RJalrD<=1'b1;
        else
            RJalrD<=0;
    end
    //RegWrite
    always@(*)
    begin
        if(Op==C_Load)
        begin
            case(Fn3)
                3'b000: RegWriteD<=`LB;
                3'b001: RegWriteD<=`LH;
                3'b010: RegWriteD<=`LW;
                3'b100: RegWriteD<=`LBU;
                3'b101: RegWriteD<=`LHU;
                default:RegWriteD<=0;
            endcase
        end
        else if(Op==C_Jal||Op==C_Jalr||Op==C_ICom||Op==C_Compute||Op==C_AUIPC||Op==C_LUI)
            RegWriteD<=3'b111;
        else
            RegWriteD<=0;
    end
```

要点：

1、一些输出信号的是wire型的，比如AluSrc2D,JalD等等，wire型变量不能在always模块中出现，为了解决这个问题。我定义了wire型输出变量对应的Reg型变量再将他们相连即可。

2、定义parameter变量使代码更容易阅读和修改：

```
parameter
C_Jal=7'b110_1111,C_Jalr=7'b110_0111,C_Branch=7'b110_0011,C_Load=7'b000_0011,C_Store=7'b010_0011,C
_ICom=7'b001_0011,C_Compute=7'b011_0011,C_LUI=7'b011_0111,C_AUIPC=7'b001_0111;
```

ImmOperand:

实现：根据传入的Type值，对In值进行不同方式的拼接得到Out值即可：

```
always@(*)
    begin
        case(Type)
            `ITYPE:     Out<={ {20{In[31]}}, In[31:20] };
            `STYPE:     Out<={ {20{In[31]}},In[31:25],In[11:7] };
            `BTYPE:    Out<={ {20{In[31]}},In[7],In[30:25],In[11:8],1'b0 };
            `UTYPE:     Out<={ In[31:12],12'b0 };
            `JTYPE:    Out<={ {12{In[31]}},In[19:12],In[20],In[30:21],1'b0 };
            //......
            default:Out<=32'hxxxxxxxx;
        endcase
    end
```

HazzardUnit:

实现：在本阶段，Hazzard模块只需要在CpuReset的时候，置位所有的clear信号。其他时候,Flush和stall信号都为0。两个Alusrc也都为0。

ALU:

实现：根据AluControl的值对Operand1和Operand2做响应的计算即可。

要点：有符号数的计算可以直接调用$signed函数，就不需要复杂的判断了。比如：

```
`SLT:
        begin
            if(($signed(Operand1))<($signed(Operand2)))
                AluOut<=32'h0001;
            else
                AluOut<=0;
        end
```

测试:

需要覆盖所有的计算操作，没有写Hazzard模块，可以在每两条指令之间插入4条Nop指令，保证没有数据冲突。

阶段2:

BranchDecisionMaking:

实现：根据BranchType的值进入不同的判断枝进行判断，并输出最终的值。

```
always@(*)
    begin
        case (BranchTypeE)
            `NOBRANCH:      BranchE<=0;
            `BEQ:
            begin
                if(Operand1==Operand2)
                    BranchE<=1'b1;
                else
                    BranchE<=0;
            end
            `BNE:
```

```
            begin
                if(Operand1!=Operand2)
                    BranchE<=1'b1;
                else
                    BranchE<=0;
            end
            `BLT:
            begin
                if(($signed(Operand1))<($signed(Operand2)))
                    BranchE<=1'b1;
                else
                    BranchE<=0;
            end
            `BGE:
            begin
                if(Operand1==Operand2 ||  ($signed(Operand1))>($signed(Operand2)))
                     BranchE<=1'b1;
                else
                     BranchE<=0;
            end
            `BLTU:
            begin
                if(Operand1<Operand2)
                    BranchE<=1'b1;
                else
                    BranchE<=0;
            end
            `BGEU:
            begin
                if(Operand1>=Operand2)
                    BranchE<=1'b1;
                else
                    BranchE<=0;
            end
        endcase
    end
```

要点：和ALU类似，有符号的运算也可以直接使用$signed函数。

WBSegReg:

实现：要在此处实现非对齐的store：

1、独热码：在ControUnit生成的独热码只有0001，0011和1111三种，仅仅指明了load的种类，具体取哪个位置的数据，需要根据地址的低两位来决定：

```
  assign Input_wea=WE<<A[1:0];
```

2、数据：和独热码相对应的，数据也需要进行左移：

```
 always@(*)
    begin
        case(A[1:0])
        2'b00:  WDin<=WD;
        2'b01:  WDin<=WD<<6'd8;
        2'b10:  WDin<=WD<<6'd16;
        2'b11:  WDin<=WD<<6'd24;
        endcase
    end
```

3、把数据传到DataRam的对应接口：

```
DataRam DataRamInst (
        .clk    (clk),                      //请补?
        .wea    (Input_wea),                      //请补?
        .addra  (A[31:2]),                     //请补?
        .dina   (WDin),                      //请补?
        .douta  ( RD_raw         ),
        .web    ( WE2            ),
        .addrb  ( A2[31:2]       ),
        .dinb   ( WD2            ),
        .doutb  ( RD2            )
    );
```

DataExt:

这个模块主要实现非对齐的load：

实现：

1、独热码：与非对齐store类似的，非对齐的load也需要一个独热码来表明具体load的字节。

```
 reg [3:0] DuRe;       //
    always@(*)
    begin
        case(RegWriteW)
       `LB:DuRe<=4'b0001<<LoadedBytesSelect;
       `LH:DuRe<=4'b0011<<LoadedBytesSelect;
       `LW:DuRe<=4'b1111;
       `LBU:DuRe<=4'b0001<<LoadedBytesSelect;
       `LHU:DuRe<=4'b0011<<LoadedBytesSelect;
       endcase
    end
```

2、根据独热码来获取对OutRaw（输出的中间变量）赋值：

```
reg [31:0] OutRaw;//
    always@(*)
    begin
        if(DuRe[0])
            OutRaw[7:0]=IN[7:0];
         else
            OutRaw[7:0]=0;
    end
```

```
always@(*)
begin
    if(DuRe[1])
        OutRaw[15:8]=IN[15:8];
     else
        OutRaw[15:8]=0;
end
always@(*)
begin
if(DuRe[2])
   OutRaw[23:16]=IN[23:16];
 else
   OutRaw[23:16]=0;
end
always@(*)
begin
 if(DuRe[3])
    OutRaw[31:24]=IN[31:24];
 else
    OutRaw[31:24]=0;
  end
```

3、符号扩展：

根据独热码和Load的类型来进行拼接和符号扩展：

```
always@(*)
     begin
       if(RegWriteW==`LB||RegWriteW==`LH)     //�������ż
         begin
            case(DuRe)
            4'b0001:    OUT<={  {24{ OutRaw[7] }},OutRaw[7:0] };
            4'b0010:    OUT<={ {24 {OutRaw[15]} } , OutRaw[15:8] };
            4'b0100:   OUT<={ {24{OutRaw[23]}},OutRaw[23:16] };
            4'b1000:    OUT<={ {24{OutRaw[31]}},OutRaw[31:24] };
            4'b0011:   OUT<={ {16{OutRaw[15]}},OutRaw[15:0] };
            4'b0110:    OUT<={ {16{OutRaw[23]}},OutRaw[23:8] };
             4'b1100:    OUT<={ {16{OutRaw[31]}},OutRaw[31:16] };
            default:    OUT<=OutRaw;    //1111
            endcase
         end    //if
       else if(RegWriteW==`LBU||RegWriteW==`LHU)     //�������ż
                 begin
                    case(DuRe)
                    4'b0001:    OUT<={  24'b0,OutRaw[7:0] };
                    4'b0010:    OUT<={ 24'b0,OutRaw[15:8] };
                    4'b0100:   OUT<={ 24'b0,OutRaw[23:16] };
                    4'b1000:    OUT<={ 24'b0,OutRaw[31:24] };
                    4'b0011:   OUT<={ 16'b0,OutRaw[15:0] };
                    4'b0110:    OUT<={ 16'b0,OutRaw[23:8] };
                     4'b1100:    OUT<={ 16'b0,OutRaw[31:16] };
                    default:    OUT<=OutRaw;    //1111
                    endcase
                 end    // else if
        else
```

```
            OUT<=OutRaw;
      end    //always
endmodule
```

测试：

需要覆盖所有的计算操作，没有写Hazzard模块，可以在每两条指令之间插入4条Nop指令，保证没有数据冲突。

阶段3：

HazzardUnit：

处理写后读问题，思路如下：

1、数据来自上一条指令的EX段(ALU模块)：设置Forward为01.

2、数据来自上上一条指令的MEM段：设置Forward为11.

3、数据来自上一条指令的MEM段：将IF,ID,EX段寄存器stall，MEM段寄存器clear。在下一个周期，会变成情况2。

处理跳转问题：

在跳转执行后，flush掉ID和EX段寄存器即可。

stall和forward代码如下:

```
//stall
  always@(*)
  begin
   if( ((RegReadE[1]&&RdM==Rs1E)||(RegReadE[0]&&RdM==Rs2E))&&(|MemToRegM)&&RegWriteM)
     begin
             StallF<=1'b1;
             StallD<=1'b1;
             StallE<=1'b1;
             StallM<=0;
             StallW<=0;
     end
     else
     begin
               StallF<=0;
               StallD<=0;
               StallE<=0;
               StallM<=0;
               StallW<=0;
     end
  end

   //Forward Register Source 1
  always@(*)
  begin
       if((RegReadE[1]&&RdM==Rs1E&&RdM!=0)&&(MemToRegM==0)&&RegWriteM)        //
           Forward1E=2'b10;    //MEM��forward
       else if(RegReadE[1]&&RdW==Rs1E&&RdW!=0&&RegWriteW)
           Forward1E=2'b01;
```

```
        else
            Forward1E=0;
    end
     //Forward Register Source 2
     always@(*)
     begin
         if((RegReadE[0]&&RdM==Rs2E&&RdM!=0)&&(MemToRegM==0)&&RegWriteM)
                Forward2E=2'b10;    //MEM��forward
            else if(RegReadE[0]&&RdW==Rs2E&&RdW!=0&&RegWriteW)
                Forward2E=2'b01;
             else
                 Forward2E=0;
     end
```

实验总结：

本次实验代码+调试大概花了10多个小时的时间，两者的关系大概是三七开。

一、主要问题：

1、不知道sign函数的存在，为了判断有符号数的大小写了很多代码，不易读还很容易错。

2、没弄明白run simulation到底做了什么工作，run simulation只跑10us，应该在run simulation以后紧接着run all才对，因为这个浪费了时间。、

3、对Hazzard模块的MemToReg信号不应该是MemToRegE而应该是MemToRegM。

二、实验收获：

1、温习了一遍流水线的知识，理解更进了一步。

2、了解了verilog也有函数，可以为以后的verilog代码编写带来方便。

3、又体悟到一些verilog代码调试的心得。

实验改进意见：

我觉得助教准备的实验很不错，让我们抓住了重点而不用做那些顶层的连线的工作。不过我觉得Harzard模块中的MemToReg信号输入MemToRegM会比较好处理一点。