计算机体系结构lab2实验报告

PB16111245王立峰

Overview

RISC-V 32I 五级流水线CPU的设计与实现

计算机体系结构lab2实验报告
- 实验目的
- 实验内容
- 实验分析与实现
  - 指令集分析
  - RV32I流水线CPU的Verilog代码实现
  - 配置和使用RISC-V编译工具链
  - 利用RISCV-test测试文件进行仿真和CPU功能验证
- 实验总结

实验目的

初步掌握RISC-V 32I指令集架构，能够基本上通过设计五级流水实现37条整型指令，了解冒险的解决方法。

实验内容

- 配置和使用RISCV编译工具链
- 完成RV32I流水线CPU的Verilog代码
- 利用RISCV-test测试文件进行仿真和CPU功能验证

实验分析与实现

指令集分析

此部分内容已在实验1的设计报告中有了较为详细的阐述，在此不再详细说明。主要需要说明的是如下几点：

1.Load和Store指令

此两种指令的实现较为复杂，主要难点就在于字对齐处理上，需要花费些功夫。

2.Branch和Jump指令

个人觉得这两种应该指令集架构中最为复杂的类型，其中涉及到PC值的变化，控制冒险的处理等一系列的控制。

RV32I流水线CPU的Verilog代码实现

重点是以下几个模块的实现

- **ControlUnit**

  在ControlUnit模块中信号是最多的，需要理清楚各个发送过来的信号指的是哪个指令，对应指令需要用到哪些模块，需要输出的信号有哪些，信号值为多少等信息。

  ```
  //Load指令需要用到的主要信号
  ```

```
    wire OtherWriteReg;
    assign OtherWriteReg=(LUI||AUIPC||JAL||JALR)||(Op==IType_op)||(Op==RType_op);

    always@(*)
    begin
        case ({LB,LH,LW,LBU,LHU,OtherWriteReg})
            6'b100000: RegWriteD <= `LB;
            6'b010000: RegWriteD <= `LH;
            6'b001000: RegWriteD <= `LW;
            6'b000100: RegWriteD <= `LBU;
            6'b000010: RegWriteD <= `LHU;
            6'b000001: RegWriteD <= `LW;
            default: RegWriteD <= `NOREGWRITE;
        endcase
    end

    assign MemToRegD=(LB || LH || LW || LBU || LHU);

//Store指令需要用到的主要信号
    always@(*)
    begin
        case ({SB,SH,SW})
            3'b100: MemWriteD <= 4'b0001;
            3'b010: MemWriteD <= 4'b0011;
            3'b001: MemWriteD <= 4'b1111;
            default: MemWriteD <= 4'b0000;
        endcase
    end

    assign LoadNpcD=(JAL||JALR);

    assign RegReadD[1]=JALR||(Op==br_op)||(Op==load_op)||(Op==store_op)||
(Op==IType_op)||(Op==RType_op);
    assign RegReadD[0]=(Op==br_op)||(Op==store_op)||(Op==RType_op);

//控制Branch的主要信号
    always@(*)
    begin
        case ({BEQ,BNE,BLT,BLTU,BGE,BGEU})
            6'b100000: BranchTypeD <= `BEQ;
            6'b010000: BranchTypeD <= `BNE;
            6'b001000: BranchTypeD <= `BLT;
            6'b000100: BranchTypeD <= `BLTU;
            6'b000010: BranchTypeD <= `BGE;
            6'b000001: BranchTypeD <= `BGEU;
            default: BranchTypeD <= `NOBRANCH;
        endcase
    end

//控制ALU运算的主要信号
    always@(*)
    begin
        if((Op==load_op)||(Op==store_op)||ADD||ADDI||AUIPC||JALR)begin AluContrlD <=
`ADD; end
        else if(SUB)begin AluContrlD <= `SUB; end
        else if(LUI)begin AluContrlD <= `LUI; end
        else if(XOR||XORI)begin AluContrlD <= `XOR; end
        else if(OR||ORI)begin AluContrlD <= `OR;  end
        else if(AND||ANDI)begin AluContrlD <= `AND; end
        else if(SLL||SLLI)begin AluContrlD <= `SLL; end
```

```
        else if(SRL||SRLI)begin AluContrlD <= `SRL; end
        else if(SRA||SRAI)begin AluContrlD <= `SRA; end
        else if(SLT||SLTI)begin AluContrlD <= `SLT; end
        else if(SLTU||SLTIU)begin AluContrlD <= `SLTU; end
        else begin AluContrlD <= 4'dx; end
    end

    assign AluSrc1D = (AUIPC);
    assign AluSrc2D = (SLLI||SRLI||SRAI)? 2'b01 : ((Op==br_op||Op==RType_op)? 2'b00 :
2'b10);

//控制立即数类型的主要信号
    always@(*)
    begin
        if(Op==RType_op)begin ImmType<=`RTYPE; end
        else if(Op==IType_op || Op==load_op || JALR) begin ImmType<=`ITYPE; end
        else if(Op==store_op)begin ImmType<=`STYPE; end
        else if(Op==br_op)begin ImmType<=`BTYPE; end
        else if(LUI||AUIPC)begin ImmType<=`UTYPE; end
        else if(JAL)begin ImmType<=`JTYPE; end
        else begin ImmType<=3'dx; end
    end
```

- **HarzardUnit**

```
always@(*)
begin
    if(CpuRst)begin
        StallF=1'b0;
        StallD=1'b0;
        StallE=1'b0;
        StallM=1'b0;
        StallW=1'b0;

        FlushF=1'b1;
        FlushD=1'b1;
        FlushE=1'b1;
        FlushM=1'b1;
        FlushW=1'b1;
    end
    else begin
        //需要stall和flush的情况分类
        if(MemToRegE && ((RdE==Rs1D )||(RdE==Rs2D))&& RdE!=0)begin
            StallF=1'b1;
            StallD=1'b1;
            StallE=1'b0;
            StallM=1'b0;
            StallW=1'b0;

            FlushF=1'b0;
            FlushD=1'b0;
            FlushE=1'b0;
            FlushM=1'b0;
            FlushW=1'b0;
        end
        else if(BranchE || JalrE)begin
            StallF=1'b0;
            StallD=1'b0;
            StallE=1'b0;
```

```
                StallM=1'b0;
                StallW=1'b0;

                FlushF=1'b0;
                FlushD=1'b1;
                FlushE=1'b1;
                FlushM=1'b0;
                FlushW=1'b0;
            end
            else if(JalD)begin
                StallF=1'b0;
                StallD=1'b0;
                StallE=1'b0;
                StallM=1'b0;
                StallW=1'b0;

                FlushF=1'b0;
                FlushD=1'b1;
                FlushE=1'b0;
                FlushM=1'b0;
                FlushW=1'b0;
            end
            else begin
                StallF=1'b0;
                StallD=1'b0;
                StallE=1'b0;
                StallM=1'b0;
                StallW=1'b0;

                FlushF=1'b0;
                FlushD=1'b0;
                FlushE=1'b0;
                FlushM=1'b0;
                FlushW=1'b0;
            end
        end
    end

    //以下为用到数据转发信号的控制，个人觉得这里是最复杂的，其中的原理也是研究了很久才明白的，下面是一种参考一个github上的方案
    //Forward Register Source 1
    assign Forward1E[0]=(|RegWriteW)&&(RdW!=0)&&(!((RdM==Rs1E)&&(|RegWriteM)))&&(RdW==Rs1E)&&RegReadE[1];
    assign Forward1E[1]=(|RegWriteM)&&(RdM!=0)&&(RdM==Rs1E)&&RegReadE[1];

    //Forward Register Source 2
    assign Forward2E[0]=(|RegWriteW)&&(RdW!=0)&&(!((RdM==Rs2E)&&(|RegWriteM)))&&(RdW==Rs2E)&&RegReadE[0];
    assign Forward2E[1]=(|RegWriteM)&&(RdM!=0)&&(RdM==Rs2E)&&RegReadE[0];
```

- **ALU**

  此模块主要是一些逻辑、算术、移位运算，并无复杂的结构

```
always@(*)
    begin
        case(AluContrl)
            `SLL: AluOut <= Operand1 << Operand2[4:0];
            `SRL: AluOut <= Operand1 >> Operand2[4:0];
            `SRA: AluOut <= $signed(Operand1) >>> Operand2[4:0];
```

```
            `ADD: AluOut <= $signed(Operand1) + $signed(Operand2);
            `SUB: AluOut <= $signed(Operand1) - $signed(Operand2);
            `XOR: AluOut <= Operand1 ^ Operand2;
            `OR:  AluOut <= Operand1 | Operand2;
            `AND: AluOut <= Operand1 & Operand2;
            `SLT: AluOut <= ($signed(Operand1) < $signed(Operand2)) ? 32'b1:32'b0;
            `SLTU: AluOut<= (Operand1 < Operand2) ? 32'b1:32'b0;
            `LUI: AluOut <= Operand2;
            default: AluOut <= 32'b0;
        endcase
    end
```

- **BranchDecisionMaking**

```
//与ALU类似，也就是针对一些不同的跳转条件进行相应的计算，可以想象成是另一个专门用于跳转控制的ALU
always@(*)
    begin
        case(BranchTypeE)
            `NOBRANCH: BranchE <= 0;
            `BEQ: BranchE <= (Operand1 == Operand2);
            `BNE: BranchE <= (Operand1 != Operand2);
            `BLT: BranchE <= ($signed(Operand1) < $signed(Operand2));
            `BLTU: BranchE <= (Operand1 < Operand2);
            `BGE: BranchE <= ($signed(Operand1) >= $signed(Operand2));
            `BGEU:BranchE <= (Operand1 >= Operand2);
            default: BranchE <= 0;
        endcase
    end
```

- **DataExt**

```
//这个模块里主要是要根据RegWrite信号来确定相应的Load指令，再根据LoadBytesSelect来进行相应的拓展，主要是由于LB、LH、LW对应load的位置不同，而我们需要进行字对齐处理
always@(*)
    begin
        case (RegWriteW)
            `NOREGWRITE: OUT <= 32'b0;
            `LB: begin
                case (LoadedBytesSelect)
                    2'b00: OUT <= {{25{IN[7]}},IN[6:0]};
                    2'b01: OUT <= {{25{IN[15]}},IN[14:8]};
                    2'b10: OUT <= {{25{IN[23]}},IN[22:16]};
                    2'b11: OUT <= {{25{IN[31]}},IN[30:24]};
                    default: OUT <= 32'bx;
                endcase
            end
            `LH: begin
                casex (LoadedBytesSelect)
                    2'b0x: OUT <= {{17{IN[15]}},IN[14:0]};
                    2'b1x: OUT <= {{17{IN[31]}},IN[30:16]};
                    default: OUT <= 32'bx;
                endcase
            end
            `LW: OUT <= IN;
            `LBU: begin
                case (LoadedBytesSelect)
```

```
                    2'b00: OUT <= {24'b0,IN[7:0]};
                    2'b01: OUT <= {24'b0,IN[15:8]};
                    2'b10: OUT <= {24'b0,IN[23:16]};
                    2'b11: OUT <= {24'b0,IN[31:24]};
                    default: OUT <= 32'bx;
                endcase
            end
            `LHU: begin
                casex (LoadedBytesSelect)
                    2'b0x: OUT <= {16'b0,IN[15:0]};
                    2'b1x: OUT <= {16'b0,IN[31:16]};
                    default: OUT <= 32'bx;
                endcase
            end
            default: OUT <= 32'bx;
        endcase
    end
```

- **WBSegReg**

```
//此模块主要是针对Store的字对齐处理，是比较复杂的地方， 以下是一种字对齐处理的解决方案
wire [31:0] RD_raw;
wire [3:0] WE_test;
assign WE_test= (|WE)? ((WE==4'b0001)? (WE<<A[1:0]):((WE==4'b0011)? ((A[1]==1'b0)?
4'b0011:4'b1100):4'b1111)):WE;
wire [31:0]WD_test;
assign WD_test= (|WE)? ((WE==4'b0001)? ({WD[7:0],WD[7:0],WD[7:0],WD[7:0]}) :
((WE==4'b0011)? ({WD[15:0],WD[15:0]}) : WD)  ):WD;

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

- **NPC_Generator**

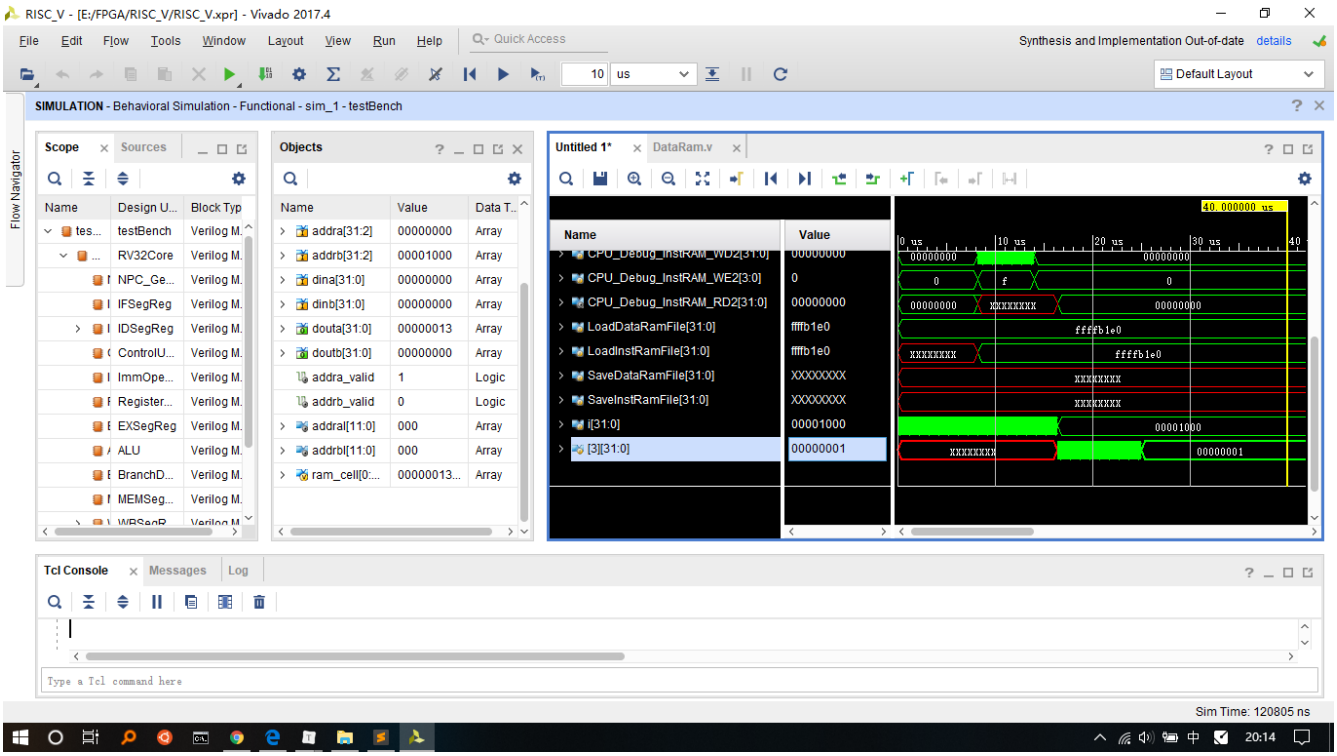
```
always@(*)
    begin
        if(JalrE)begin PC_In<=JalrTarget; end
        else if(BranchE)begin PC_In<=BranchTarget; end
        else if(JalD)begin PC_In<=JalTarget; end
        else begin PC_In<=PCF+4; end
    end
```

配置和使用RISC-V编译工具链

根据实验材料中提供的.S文件和makefile文件使用make指令即可生三组测试样例，不过我在win10系统下make失败了，只有在linux才马克成功了。

利用RISCV-test测试文件进行仿真和CPU功能验证

利用助教提供的testBench.v文件，修改测试文件参数，添加到Vivado项目中即可启动仿真，通过仿真我们可以发现RegisterFile中的3号寄存器的值会随着执行指令的增加而增加，而当执行结束后该寄存器值会回到1，通过改点我们可以判断出仿真结果正确。



实验总结

通过本次实验，对RISC-V指令集架构有了较为深入的了解，并且更加熟悉了五级流水线构造原理，及一些冒险情况处理方案。本次实验内容丰富有意义，紧跟时代潮流，但颇具难度。在此要感谢一些同学和助教的帮助，没有他们的指导和帮助，这次实验的完成将遥遥无期。希望后续课程能提供一些前人的参考性教程给我们这些对体系结构掌握不是很熟练的学生。