南京大学

数电实验十：

CPU 数据通路

课程名称：　数字逻辑与计算机组成实验

姓名：　孙文博

学号：　201830210

班级：　数电一班

邮箱：　201830210@smail.nju.edu.cn

实验时间：　2022.5.20 - 2022.5.25

一、 实验目的

本实验的目标是利在单周期 CPU 的实现之前，先完成 CPU 数据通路中的三个重要部分：寄存器堆、ALU 和数据存储器，并通过功能仿真测试。

二、 实验环境

设计\编译环境：Quartus (Quartus Prime 17.1) Lite Edition

FPGA 芯片：Cyclone II 5CSXFC6D6

三、 实验过程

1. 寄存器堆

寄存器堆是 CPU 中用于存放指令执行过程中临时数据的存储单元。我们将要实现的 RISC-V 的基础版本 CPU RV32I 具有 32 个寄存器。RV32I 采用 Load Store 架构，即所有数据都需要先用 Load 语句从内存中读取到寄存器里才可以进行算术和逻辑操作。因此，RV32I 有 32 个通用寄存器，且每条算术运算可能要同时读取两个源寄存器并写入一个目标寄存器。为支持高速，多端口并行存取的寄存器堆，我们不能直接调用通用的 RAM，而需要用 Verilog 语言单独编写寄存器堆。

RV32I 的 32 个 32bit 通用寄存器为 x0~x31(寄存器地址为 5bit 编码)，其中寄存器 x0 中的内容总是 0，无法改变。其他寄存器的别名和寄存器使用约定参见下表。

表 10-1: RV32I 中通用寄存器的定义与用法

Register	Name	Use	Saver
x0	zero	Constant 0	–
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	–
x4	tp	Thread Pointer	–
x5~x7	t0~t2	Temp	Caller
x8	s0/fp	Saved/Frame pointer	Callee
x9	s1	Saved	Callee
x10~x11	a0~a1	Arguments/Return Value	Caller
x12~x17	a2~a7	Arguments	Caller
x18~x27	s2~s11	Saved	Callee
x28~x31	t3~t6	Temp	Caller

寄存器堆支持同时两个读操作和一个写操作，因此需要有 2 个读地址 Ra 和 Rb，分别对应 RISC-V 汇编中的 rs1 和 rs2。写地址为 Rw，对应 rd，地址均是 5 位。写入数据 busW 为 32 位，写入有效控制为一位高电平有效的 RegWr 信号。

寄存器堆的输出是 2 个 32 位的寄存器数据，分别是 busA 和 busB。寄存器堆有一个控制写入的时钟 WrClk。在时序上我们可以让读取是非同步的，即地址改变立刻输出。写入可以在时钟下降沿写入。注意，寄存器 x0 需要特殊处理，不论何时都是全零。

寄存器堆的实现代码类似实验五，这里不再展示，将会在数据存储器部分直接使用。

2. ALU 实现

ALU 是 CPU 中的核心数据通路部件之一，它主要完成 CPU 中需要进行的算术逻辑运算，我们在前面的实验三中已经实现了一个简单的 ALU。在本实验中只需要对该 ALU 稍加改造即可。针对 RV32I 的运算需求，我们对 ALU 的控制信号进行了重新定义，如下表所示：

表 10-2: 控制信号 ALUctr 的含义

ALUctr[3]	ALUctr[2:0]	ALU 操作
0	000	选择加法器输出，做加法
1	000	选择加法器输出，做减法
×	001	选择移位器输出，左移
0	010	做减法，选择带符号小于置位结果输出，Less 按带符号结果设置
1	010	做减法，选择无符号小于置位结果输出，Less 按无符号结果设置
×	011	选择 ALU 输入 B 的结果直接输出
×	100	选择异或输出
0	101	选择移位器输出，逻辑右移
1	101	选择移位器输出，算术右移
×	110	选择逻辑或输出
×	111	选择逻辑与输出

该 ALU 的逻辑图如下图所示：

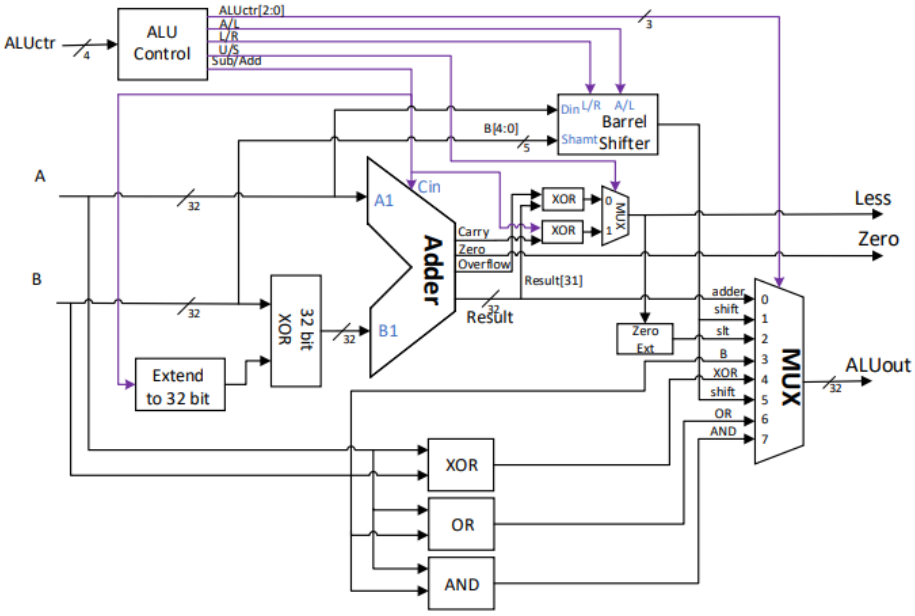

图 10-2: ALU 电路示意图

ALU 对输入数据并行地进行加减法、移位、比较大小、异或等操作。最终 ALUout 输出通过一个八选一选择器选择不同运算部件的结果，选择器的控制端用 ALUctr[2:0] 直接生成。

我们改进实验三完成的 ALU 部分核心代码如下：

```
always @(*) begin
    case(ALUctr)
    4'b0000:
    begin
        aluresult = dataa + datab; ze = (aluresult==0);
    end

    4'b1000:
    begin
        aluresult = dataa - datab; ze = (aluresult==0);
    end

    4'b0001:
    begin
        aluresult = dataa << datab; ze = (aluresult==0);
    end
    4'b1001:
    begin
        aluresult = dataa << datab; ze = (aluresult==0);
    end
```

```
4'b0010:
begin
    if(dataa[31] == 1 && datab[31] == 0) begin
        aluresult = 1; ze = (dataa==datab); le = 1;
    end
    else if(dataa[31] == 0 && datab[31] == 1) begin
        aluresult = 0; ze = (dataa==datab); le = 0;
    end
    else begin
        aluresult = (dataa < datab) ;
        ze = (dataa==datab); le = (aluresult == 1)? 1 : 0;
    end
end

4'b1010:
begin
    aluresult = (dataa < datab) ;
    ze = (dataa==datab); le = (aluresult == 1)? 1 : 0;
end
```

3. 数据存储实现

数据存储器在 CPU 运行中存储全局变量、堆栈等数据，并且需要支持在沿上进行读取和写入操作。RV32I 的字长是 32bit，但是数据存储器不仅要支持 32bit 数据的存取，同时也需要支持按字节（8bit）或半字（16bit）大小的读取。由于单周期 CPU 需要在一个周期内完成一条指令的所有操作，我们需要数据 RAM 有独立的读时钟和写时钟。其中读取操作在系统时钟的上升沿进行（即一个时钟周期的一半时刻），写操作在系统时钟的下降沿进行（即一个时钟周期的结束时刻）。

在本实验中，需要实现一个能够支持 RV32I 中的 lb, lh, lw, lbu, lhu 及 sb, sh, sw 等指令的存储器。该存储器具有独立的读取时钟 rdclk 和写入时钟 wrclk ，均为上升沿有效，读取和写入在上升沿到达后立刻生效。对于不同的访问模式，利用 memop 来控制存储器的操作，具体参见下表：

表 10-3: 存储访问指令与 Memop 对应关系

指令		MemOP	操作
lb	rd,imm12(rs1)	000	R[rd] ← SEXT(M_{1B}[R[rs1] + SEXT(imm12)])
lh	rd,imm12(rs1)	001	R[rd] ← SEXT(M_{2B}[R[rs1] + SEXT(imm12)])
lw	rd,imm12(rs1)	010	R[rd] ← M_{4B}[R[rs1] + SEXT(imm12)]
lbu	rd,imm12(rs1)	100	R[rd] ← {24'b0, M_{1B}[R[rs1] + SEXT(imm12)]}
lhu	rd,imm12(rs1)	101	R[rd] ← {16'b0, M_{2B}[R[rs1] + SEXT(imm12)] }
sb	rs2,imm12(rs1)	000	M_{1B}[R[rs1] + SEXT(imm12)] ← R[rs2][7:0]
sh	rs2,imm12(rs1)	001	M_{2B}[R[rs1] + SEXT(imm12)] ← R[rs2][15:0]
sw	rs2,imm12(rs1)	010	M_{4B}[R[rs1] + SEXT(imm12)] ← R[rs2]

对于读取操作，我们可以每次均读取 32bit 的数据，然后根据 MemOP 来判断是需要 8bit，16bit 还是 32bit 的数据，再根据地址

的低 2 位选择合适的数据拼接扩展成读取的结果即可。

对于写入操作，需要对 32bit 中特定的 8bit 或 16bit 数据进行写入，而不能破坏其他比特。这里我们引入了一个变量 sign 用于标记是否处于写入周期，从而使得写入和读取顺序不出错。

具体的实现代码如下（关键部分）：

```
module dmem(
    input  [31:0] addr,
    output reg [31:0] dataout,
    input  [31:0] datain,
    input  rdclk,
    input  wrclk,
    input [2:0] memop,
    input we
    );

//add your code here
reg [7:0] ram [4095:0];
integer sign = 0; //用于先写后读
reg [31:0] cin;
integer i;

always@(posedge wrclk)begin
    cin = datain;
    if(we) begin
        sign = 1;
        case(memop)
        0:begin
            if(cin[7]==0)
                ram[addr] = cin & 32'h000000ff;
            else
                ram[addr] = cin | 32'hffffff00;
        end
```

```
    1:begin
        if(cin[15]==0)begin
            ram[addr] = cin & 32'h000000ff;
            cin = cin >> 8;
            ram[addr+1] = cin & 32'h000000ff;
        end
        else begin
            ram[addr] = cin & 32'h000000ff;
            cin = cin >> 8;
            ram[addr+1] = cin | 32'hffffff00;
        end
    end
```

```
always@(rdclk) begin
    case(memop)
        0:begin
            if(!sign)begin
                if(datain[7]==1)
                    dataout = datain | 32'hffffff00;
                else
                    dataout = datain & 32'h000000ff;
            end
            else begin
                if(ram[addr][7]==1)begin
                    dataout = ram[addr];
                    dataout = dataout | 32'hffffff00;
                end
                else begin
                    dataout = ram[addr];
                    dataout = dataout & 32'h000000ff;
                end
            end
        end
        1:begin
            if(!sign) begin
                if(datain[15]==1)
                    dataout = datain | 32'hffff0000;
                else
                    dataout = datain & 32'h0000ffff;
```

```
        1:begin
            if(!sign) begin
                if(datain[15]==1)
                    dataout = datain | 32'hffff0000;
                else
                    dataout = datain & 32'h0000ffff;
            end
            else begin
                if(ram[addr+1][7]==1)begin
                    dataout = ram[addr+1]*256+ram[addr];
                    dataout = dataout | 32'hffff0000;
                end
                else begin
                    dataout = ram[addr+1]*256+ram[addr];
                    dataout = dataout & 32'h0000ffff;
                end
            end
        end
        2:begin
            if(!sign)
                dataout = datain;
            else
                dataout = ram[addr+3]*16*16*16*16*16*16 +
                    ram[addr+2]*16*16*16*16+ram[addr+1]*16*16+ram[addr];
        end
```

四、 实验结果

1. 思考题

本次实验暂无思考题。

2. 在线测试

完成 CPU 的寄存器堆、ALU 和数据存储器的实现，并顺利通过课程在头歌系统上的两个测试。

五、 总结与反思

本次实验是实验十一的铺垫，这两个实验最终要实现一个完整的 CPU。由于较为抽象且不易验收，因此重点在于头歌的测试部分。本次实验的头歌测试由两部分构成，其中 ALU 部分相对而言简单一些，只需要将实验三中的 ALU 改进即可，而数据通路部分遇到了读写时序不统一的问题，也是 debug 了好久，最终通过引入标志变量 sign 记录读写顺序才顺利通过。