Verilog-Lab2 实验报告

彭浩然 **PB19051055**

1 各阶段工作

1.1 阶段1和阶段2

因为对于阶段2有现成的测试样例，而手写不包含数据相关的测试样例相对麻烦，我直接合并了阶段1和阶段2。

在这个阶段中，我补全了以下模块中`TODO`的部分：`ALU`，`BranchDisision`，`ControllerDecoder`，`DataExtend`，`Hazard`，`ImmExtend`，`NPCGenerator`。同时，为了在不加入CSR指令前CPU也能正常工作，我将`CSR_EX`和`CSR_Regfile`中的输出均硬编码为0，防止仿真时出现不确定状态。

在补全`ControllerDecoder`前，我先对整个代码结构做了比较大的调整。为了使代码紧凑、逻辑清晰，我在`always`块的开头将所有的控制信号分为几组并预设为0：

```
always @ (*) begin
    // ALU相关
    ALU_func = 0; // [3:0]
    imm_type = 0; // [2:0]
    // 寄存器堆相关
    op1_src = 0;
    op2_src = 0;
    // 写回相关
    reg_write_en = 0;
    load_npc = 0;
    wb_select = 0;
    // 跳转指令相关
    br_type = 0; // [2:0]
    jal = 0;
    jalr = 0;
    // load指令相关
    load_type = 0; // [2:0]
    cache_write_en = 0; // [3:0]
    // CSR指令相关
    CSR_write_en = 0;
    CSR_zimm_or_reg = 0;
    ...
```

这样，每一条指令的解码只需要对相关的组中的控制信号进行赋值即可，可以忽略无关的控制信号（例如这个阶段的所有指令都忽略了CSR指令相关的分组），无需在每一个条件分支里重复每一个控制信号，使代码更简洁。另外，RISC-V的指令解码分为opcode、funct3、funct7总共三层，所以整个解码过程最多用三层的`case`即可。形式上如同下面的例子：

```
case (opcode)
    ...
    `I_ARI: begin
        ALU_func = 0;
        imm_type = `ITYPE;

```

```
        op1_src = 0;
        op2_src = 1;

        reg_write_en = 1;

        case (funct3)
            `I_ADDI:    ALU_func = `ADD;
            ...
            `I_SR: begin
                case (funct7)
                    `I_SRAI:    ALU_func = `SRA;
                    `I_SRLI:    ALU_func = `SRL;
                endcase
            end
        endcase
    end
    ...
endcase
```

另外，我将表示JALR指令的常量`J_JALR`重命名为`I_JALR`，因为JALR指令属于I类指令（而非想当然的J类），其立即数应该按照I类指令处理，按照原来的写法，容易引起误解。

在修改`DataExtend`前，我也对代码结构做了修改。原本的结构是通过两层`case`来实现的（外层是`load_type`，内层按照`addr`的后两位找到非字对齐的数据），而我通过在最外层先引入下面的语句，将数据先对齐，从而摆脱了会让代码量膨胀4倍的内层`case`：

```
assign shifted_data = data >> (8 * addr);
```

此后只需要在`shifted_data`上处理已经经过对齐的数据即可：

```
case (load_type)
    `NOREGWRITE:dealt_data = data;
    `LW:        dealt_data = shifted_data;
    `LH:        dealt_data = {{16{shifted_data[15]}}, shifted_data[15:0]};
    `LHU:       dealt_data = {16'b0, shifted_data[15:0]};
    `LB:        dealt_data = {{24{shifted_data[7]}}, shifted_data[7:0]};
    `LBU:       dealt_data = {24'b0, shifted_data[7:0]};
    default:    dealt_data = 0;
endcase
```

最后的行数甚至少于修改前的代码。

其他的部件变化不多，按部就班完成即可，不再赘述。

1.2 阶段3

在这个阶段中，我补全了`CSR_EX`和`CSR_Regfile`中的`TODO`部分。根据代码中呈现出来的数据通路，我推测这个框架对CSR的设计是将其读取、计算和写回全部放在EX段完成，因此，对于CSR的寄存部分，需要注意在上升沿写入数据（而非像寄存器堆一样的下降沿），否则计算完毕新写入的CSR值会直接被传递给EX/MEM级间寄存器，使得最后新值被写入通用寄存器堆，而实际上应该写回的是旧值。

另外，我注意到在ALU相关常量中预留了一个`NAND`常量，猜测与CSR指令要求的ALU功能有关，因此最开始实现ALU的时候我也直接实现了NAND运算。但是最后将`CSRRC`和`CSRRCI`指令的`ALU_func`设为`NAND`时却无法通过测试。仔细思考发现，位清零运算所需的表达式是`NOT(mask) AND csr`，而`NAND`（即“与非”）通常指的是`NOT(op1 AND op2)`，两者是有区别的。因此，为了避免这个命名造成的困扰，我将与CSR的位清零相关的这个ALU运算重命名为`CLR`，并定义为`NOT(op1) AND op2`。

2 总结

2.1 遇到的问题

在测试时，我曾经遇到3号寄存器的值在少数几个测试样例上被写入2，后又继续写回正常值的问题。通过阅读汇编代码，我发现下面一段与之相关：

```
000121f8 <test_229>:
   121f8:   00100093            li  ra,1
   ...
   1221c:   0e500193            li  gp,229
   12220:   13d09463            bne ra,t4,12348 <fail>

00012224 <jal_test_2>:
   12224:   00200193            li  gp,2
   12228:   00000093            li  ra,0
   1222c:   0100026f            jal tp,1223c <jal_target_2>

00012230 <jal_linkaddr_2>:
   12230:   00000013            nop
   12234:   00000013            nop
   12238:   1100006f            j   12348 <fail>

0001223c <jal_target_2>:
   1223c:   00000317            auipc   t1,0x0
   12240:   ff430313            addi    t1,t1,-12 # 12230 <jal_linkaddr_2>
   12244:   10431263            bne t1,tp,12348 <fail>

00012248 <test_232>:
   12248:   00100093            li  ra,1
   ...
   1226c:   0e800193            li  gp,232
   12270:   0dd09c63            bne ra,t4,12348 <fail>
```

可以看到，在229和232测试样例之间，为了测试JAL指令，确实有一段将`gp`写入了2，而后恢复正常。通过阅读汇编码，我意识到这虽然与文档描述不完全一致，但是是正常现象，也就没有在这上面多浪费时间了。

2.2 实验收获

通过这次试验，我增强了阅读理解他人的硬件代码并在其基础上进行开发的能力。与个人开发不同，这需要通过阅读代码和注释来揣测原作者的意图，并敏锐地识别出一些可能引起误会的点。这些情况让这次实验的收获并不少于个人开发。

2.3 时长统计

代码与调试：9小时
实验报告：1小时

3 改进意见

报告里提到的容易让人产生误解的部分都可以视为改进意见。