中国科学院大学计算机组成原理实验课

实 验 报 告

学号：_2018K8009922027_ 姓名：_李国峰_ 专业：_计算机科学与技术_

实验序号：__1__ 实验名称：__基本功能部件设计__

注 1：请在实验项目个人本地仓库中创建顶层目录 doc。撰写此 Word 格式实验报告后以 PDF 格式保存在 doc 目录下。文件命名规则：学号-prjN.pdf，其中学号中的字母“K”为大写，“-”为英文连字符，“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：2019K8009929000-prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：学号-prj5-projectname.pdf，例如：2019K8009929000-prj5-dma.pdf。具体要求详见实验项目 5 讲义。 注 2：使用 git add 及 git commit 命令将 doc 目录下的实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 GitLab 远程仓库 master 分支（具体命令详见实验报告）。 注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

（1） reg_file.v 代码说明

本次实验设计的寄存器堆不允许使用 rst 信号进行初始化，因此每个寄存器的数据仅来源数据写端口。满足以下条件时写入数据：捕捉到时钟上升沿且**前一时钟周期**写使能信号 wen 为高电平且写入地址不为 0。因为触发器所采样的信号的值均为前一周期所保持的值。

```
19      // No rst & No init
20      reg [`DATA_WIDTH - 1:0] REG_Files [0:`NUM - 1];
21
22      always@(posedge clk)
23      begin
24          if(wen && waddr)
25              REG_Files[waddr] <= wdata;
26      end
```

值得一提 if 语句的写法。在括号内，waddr 语句等价于 waddr != 5’ b0，亦即只要 waddr 不为 0 语句都为真，因此可以使用简化的写法。

在不允许初始化的前提下，需要解决的一个问题是：怎样保持读 0 号寄存器的结果都为 0？方法如下。

```
28    assign rdata1 = {32{raddr1 != 5'b0}} & REG_Files[raddr1];
29
30    assign rdata2 = {32{raddr2 != 5'b0}} & REG_Files[raddr2];
```

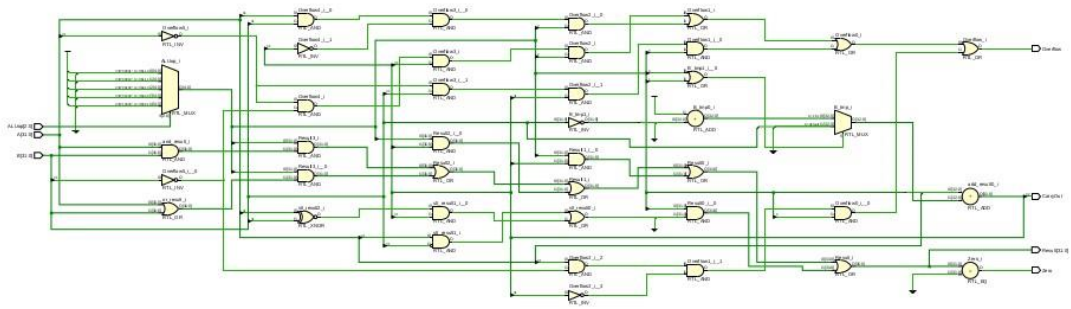
当所读寄存器不是 0 号时，raddr1 != 5’ b0 判断为真，拼接成的数值为 0xffff ffff，读出的值与之做按位与结果不变；否则，raddr1 != 5’ b0 为假，拼接成的数值为 0x0000 0000，按位与之后得到 0x0000 0000，也就相当于 0 号寄存器的值始终为 0x0000 0000。

(2) alu.v 代码说明

使用一组宏定义来提高代码可读性：

```
5  `define AND 3'b000
6  `define OR  3'b001
7  `define ADD 3'b010
8  `define SUB 3'b110
9  `define SLT 3'b111
```

先对实验要求的实现进行说明。为了减少电路面积，加法、减法和比较运算均使用同一套加法器来实现。根据[A]补 - [B]补 = [A]补+~[B]补+1，只需要根据运算种类对 B 进行预处理后再输入加法器。对于比较运算，自然可以根据减法的结果与 0 的关系来进行判断。



图中加法、减法和比较运算电路共享同一套加法器，还有一个加法器是对B进行预处理时用到的。

接下来对CarryOut的生成逻辑进行说明。

```
34    // How to prove this CarryOut is correct ?
35    // The proof will be given in report.
36    assign ext_A = ALUop == `SUB;
37
38    assign A_tmp = {ext_A, A};
39
40    // A - B is needed if ALUop == `SUB or `SLT
41    assign B_tmp = (ALUop == `SUB || ALUop == `SLT) ? {1'b0, ~B} + 33'b1 : {1'b0, B};
42
43    assign {CarryOut, add_result} = A_tmp + B_tmp;
44
45    assign sub_result = add_result;
```

这种算法乍一看不符合逻辑，但是可以证明这样做是正确的。证明如下。

设有两个32位无符号数A和B，它们的补码分别是[A]补和[B]补，做加法运算时,对[A]补和[B]补都在最高位前添0,若产生进位则第33位为1，这是显而易见的。

重点关注做减法。我们知道做减法需要对[B]补按位取反再加1。我们不妨先假设[A]补=[B]补。此时对[B]补按位取反再加1，再加上[A]补，结果必然是2^32，亦即0x1 0000 0000。根据补码定义，如果B > A，必有[A]补大于[B]补，此时将不会产生进位，第33位为0，但实际上已经产生了借位。反之，如果A>B，则[A]补小于[B]补，做加法后会产生进位，第33位为1，但实际上没有借位。

综上，我们可以发现，加法的进位情况与第33位的值保持一致，而减法的借位情况则与第33位的值相反。因此，我们对[A]补和[B]补做扩展的理由如下：如果做加法，则只需要给[A]补和[B]补高位添一位0，做加法后最高位自然表达了进位的情况；如果做减法，则为[A]补添0，为[B]补添1，这样一来，当发生了借位时，原来的33位为0，加上1后为1，正确表达了借位情况;如果没有发生借位,原来的33位为1,加上1后为0，正确表达了没有借位的情况。证毕。

接下来说明比较逻辑。

```
49    // If A < 0 and B >= 0, A < B.
50    // If A * B >= 0, A < B if and only if A - B < 0.
51    assign slt_result[0] = (A[31] & ~B[31]) ||
52                           (A[31] ~^ B[31]) && add_result[31];
```

不能直接使用“<”符号，因此利用符号位进行比较。如果 A<0 且 B>0，那么必有 A<B。

如果 A 和 B 同号，那么只需要利用减法运算的结果与 0 做比较即可。

对 Overflow 的生成逻辑进行说明。

```
54    // If A > 0 and B > 0 but A + B < 0
55    // If A < 0 and B < 0 but A + B > 0
56    // If A > 0 and B < 0 but A - B < 0
57    // If A < 0 and B > 0 but A - B > 0
58    assign Overflow = !A[31] && !B[31] &&  add_result[31] && ALUop == `ADD ||
59                       A[31] &&  B[31] && !add_result[31] && ALUop == `ADD ||
60                      !A[31] &&  B[31] &&  sub_result[31] && ALUop == `SUB ||
61                       A[31] && !B[31] && !sub_result[31] && ALUop == `SUB  ;
```

在验收的时候老师提到 Overflow 和 slt_result 有一定联系，但是我下来思考了之后，认为这两者并没有直接联系，理由如下：对于同一组数 A 和 B，其运算溢出的情况是固定的，但是只需要交换顺序，比较运算的结果就会发生改变。

我按照老师的建议和 PPT 给出的判断方法进行修改，尝试将所有的运算都转为加法运算后再进行判断，以减少电路开销。但是在修改和测试的过程中，我发现利用预处理后的数据和结果进行判断容易导致一些特殊的数据无法得到正确的结果。例如[A]补=0xffff ffff，[B]补=0x1000 0000，做减法不会溢出，但是~[B]补+1 还是 0x1000 0000，如果单纯利用加法的溢出判断逻辑进行判断，会误判发生了溢出。所以我想到的较理想的做法还是根据运算种类对两个源操作数和结果的符号位进行比较。

当然，判断溢出还可以利用双符号位的方法，但是为了和判断进位、借位的逻辑相协调，不采用这种方式。

最后，ALU 只有一个输出端口，怎样才能获得需要的结果？显而易见，我们无法利用 ALUop 来选择进行哪种运算，但我们可以把所有运算种类的结果都求出，再利用 ALUop 选出需要的结果。这里需要一个多路选择器，更进一步说，这个多路选择器理想情况下应该是不含优先级的，对所有运算的结果并行处理。

```
63    assign Result = {32{ALUop == `AND}} & and_result |
64                    {32{ALUop == `OR }} & or_result  |
65                    {32{ALUop == `ADD}} & add_result |
66                    {32{ALUop == `SUB}} & sub_result |
67                    {32{ALUop == `SLT}} & slt_result ;
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，仿真、云平台调试过程中的难点等）

由于本实验第一版代码完成的时间是在开学前，因此没有保留遇到的 bug 等相关实验记录。新遇到的 bug 几乎都是在测试过程中发现由于笔误或修改不全面导致的，因此不做记录。