中国科学院大学计算机组成原理实验课

实 验 报 告

学号：_____2018K8009909006_____ 姓名：__唐宇菲_____ 专业：数学与应用数学_____

实验序号：__1__ 实验名称：____基本功能部件--寄存器堆和算术逻辑单元

_____

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在~/COD-Lab/reports 目录下。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：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：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：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）。

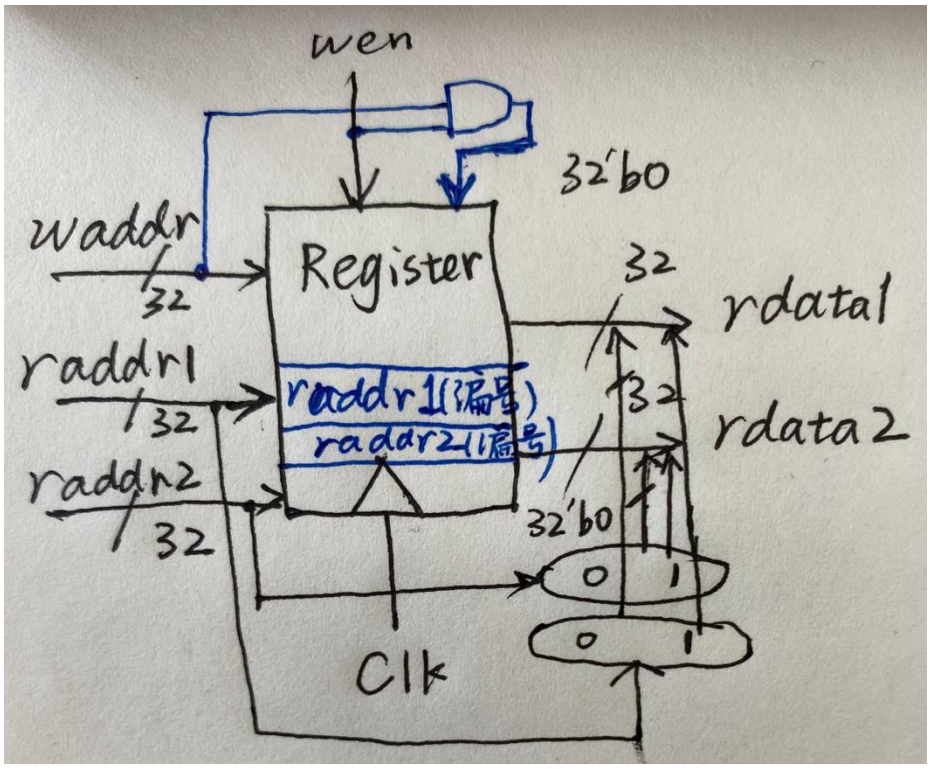
注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

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图、相应信号的仿真波形和信号变化的说明等）[RF 实验]

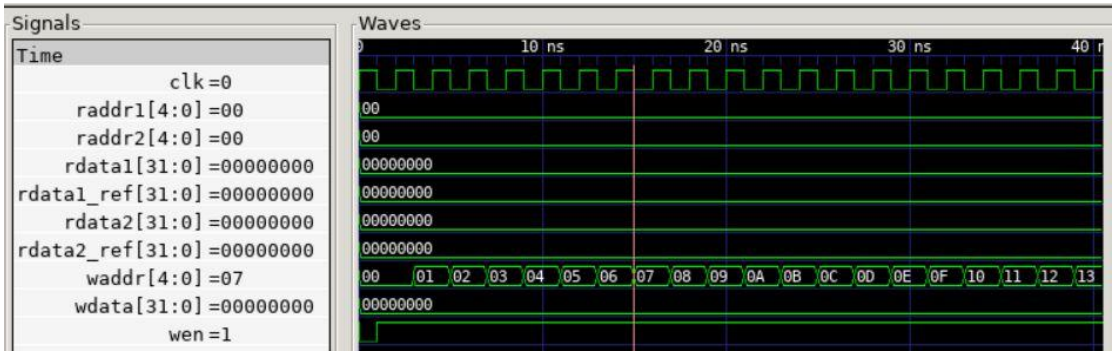
代码：具体的说明在二、中一并解释

```
`timescale 10 ns / 1 ns

`define DATA_WIDTH 32
`define ADDR_WIDTH 5

module reg_file(
    input                       clk,
    input  [`ADDR_WIDTH - 1:0]  waddr,
    input  [`ADDR_WIDTH - 1:0]  raddr1,
    input  [`ADDR_WIDTH - 1:0]  raddr2,
    input                      wen,
    input  [`DATA_WIDTH - 1:0]  wdata,
    output [`DATA_WIDTH - 1:0]  rdata1,
    output [`DATA_WIDTH - 1:0]  rdata2
);

    // TODO: Please add your logic design here
    reg [`DATA_WIDTH - 1 : 0] r[`DATA_WIDTH -1 : 0];
    always@(posedge clk)begin
        if(wen && waddr) // 0地址判断
            r[waddr] <= wdata;
    end
    //写入判断
    assign rdata1 = raddr1 ? r[raddr1] : 32'b0;
    assign rdata2 = raddr2 ? r[raddr2] : 32'b0;
endmodule
```

逻辑电路结构图：



仿真波形截图：



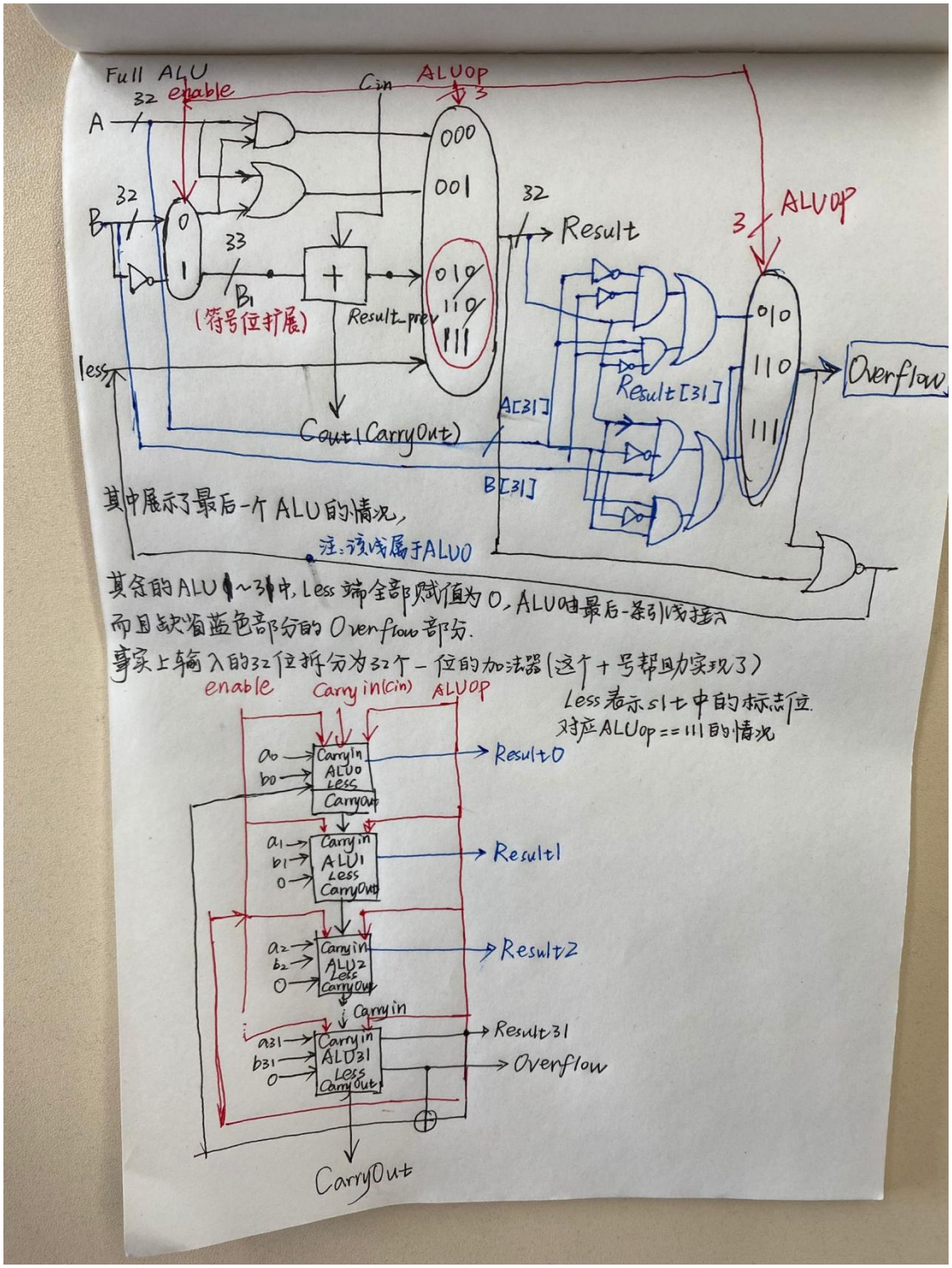
可以看到，当 raddr1 和 raddr2 为 0 的时候，rdata1 和 rdata2 确实输出 32’b0。

[ALU 实验]

代码：

```verilog
`define DATA_WIDTH 32
module alu(
    input [`DATA_WIDTH - 1 : 0] A,
    input [`DATA_WIDTH - 1 : 0] B,
    input [             2 : 0] ALUop,
    output  Overflow,
    output  CarryOut,
    output Zero,
    output [`DATA_WIDTH - 1 : 0] Result
);
    //构造一个加减法使能信号enable
    wire cin,cout,enable;
    assign enable = ({ALUop == 3'b010} & 0 ) |
                    ({ALUop == 3'b110 } & 1) |
                    ({ALUop == 3'b111} & 1);
    wire [`DATA_WIDTH - 1 : 0] Result_prev;
    //由于CarryOut是在扩展33位的情况下进行计算，所以需要用33位来存~B
    wire [`DATA_WIDTH : 0] B1;
    assign B1 = enable? ~B:B;
    assign cin = enable? 1:0;
    //计算CarryOut
    assign {cout,Result_prev} = A + B1 + cin;
    assign Result = ( {32{ALUop == 3'b000}} & (A & B)    ) |
                    ( {32{ALUop == 3'b001}} & (A | B)    ) |
                    ( {32{ALUop == 3'b010}} & (Result_prev)    ) |
                    ( {32{ALUop == 3'b110}} &  (Result_prev) ) |
                    ( {32{ALUop == 3'b111}} & (Result_prev[31] ^ Overflow) );
                    //slt是符号位和溢出的异或
    assign CarryOut = ( {ALUop == 3'b010} & cout) |
                      (  {ALUop == 3'b110} & cout);
    //按照正+正=负等条件直接判断
    assign Overflow =   ( {ALUop == 3'b010} & ((~A[31] & ~B[31] & Result_prev[31]) | (A[31] & B[31] & ~Result_prev[31]) )    ) |
                        ( {ALUop == 3'b110} &  (  (~A[31] & B[31] & Result_prev[31])    | (A[31] & ~B[31] & ~Result_prev[31]) ) ) |
                        ( {ALUop == 3'b111} & (  (~A[31] & B[31] & Result_prev[31])    | (A[31] & ~B[31] & ~Result_prev[31])) );
    assign Zero = (Result==32'b0)? 1 : 0;
endmodule
```

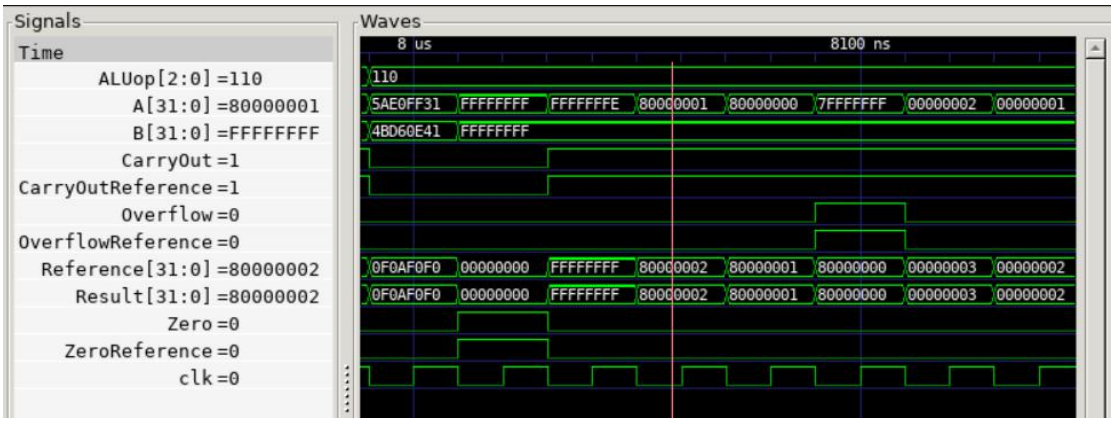
逻辑电路结构图：



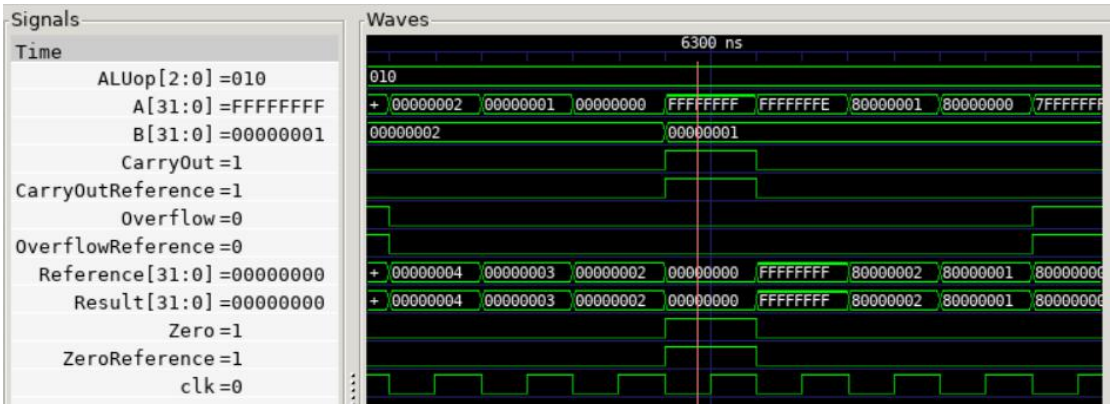
仿真波形截图：

截图说明了，对于 0 地址的处理，并不需要涉及初始化的问题。以加、减法、slt 操作为例：

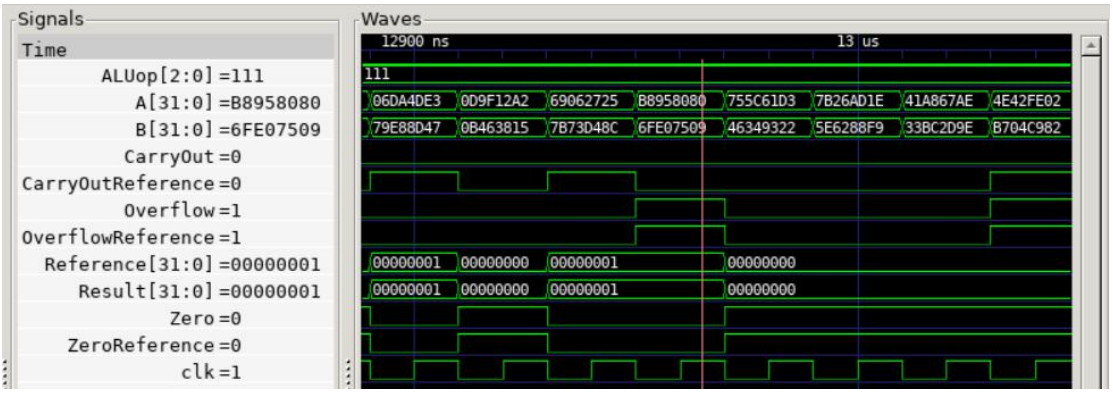
(i)减法



(ii)加法



（iii)slt 操作



主要想解释 CarryOut,波形和我起初理解的不同，具体说明在二、中，并且根据金标准中的 Carryout_reference 得到了正确的理解。

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，逻辑仿真和 FPGA 调试过程中的难点等）

解：RF 实验中关于逻辑与和按位与没搞清楚，调了很久才发现这个错误。

ALU 实验：学会了用 assign 语句的位拼接符来代替 if-else 语句。

遇到的一个问题是对 ALU 实验中的 Carryout 的处理。在未讲解前，认为无符号数的处理需要增加符号位，来区别加法和减法，所以将原来的数据由 32 位扩展成了 33 位，并在减法操作时添加了 1 作为符号位。

但在课上老师讲解，明白补数和补码概念的区别，先有补数的概念，按位取反某位加 1。再有补码的概念，对于有符号数的减法，希望通过加法运算来完成，在数学式的模运算之后，总结出的计算规律是，符号位为 1，其余位，按位取反，

末位加 1。整个代码的写作过程，一开始属于先厘清各个部分，所以布局了 6 个加法器，分别处理 result，Overflow 和 Carryout 对应的部分。这样写的好处是可以清晰地看清各个部分需要的内容，和所要表达的部分。
从 12 行看起，代码是 12-34 行：
首先根据课上老师教授的加法器的写法，可以用表达式 assign{cout,result}=A+B+cin;的形式来描述，所以先设置了两个加法器，分别处理加法和减法的情况。这里的 CarryOut 直接用一行 assign 语句表达后，起初没有仔细思考原因，看到运行通过就理所当然以为上述 assign 语句可实现进借位要求，所以在后续拆分时出现了一些疑问。15-16 行处理了 CarryOut 情况，在 28-29 行也得到了对应的输出。

```
1  `define DATA_WIDTH 32
2
3  module alu(
4      input [`DATA_WIDTH - 1 : 0] A,
5      input [`DATA_WIDTH - 1 : 0] B,
6      input [              2 : 0] ALUop,
7      output  Overflow,
8      output  CarryOut,
9      output Zero,
10     output [`DATA_WIDTH - 1 : 0] Result
11 );
12     wire cout_adder,cout_sub;
13     wire [`DATA_WIDTH - 1 : 0] Result_adder,Result_sub;
14     //直接计算符号位的进位，或者无符号数的Carryout
15     assign {cout_adder,Result_adder} = A + B;
16     assign {cout_sub,Result_sub} = A + ~B + 1;
17
18     wire [`DATA_WIDTH - 1 : 0] set_adder,set_sub;
19     assign set_adder = A + B;
20     assign set_sub = A + ~B + 1;
21
22     assign Result = ( {32{ALUop == 3'b000}} & (A & B)    ) |
23                     ( {32{ALUop == 3'b001}} & (A | B)    ) |
24                     ( {32{ALUop == 3'b010}} & (A + B)    ) |
25                     ( {32{ALUop == 3'b110}} &  (A + ~B + 1) ) |
26                     ( {32{ALUop == 3'b111}} & (set_sub[31] ^ Overflow) );
27
28     assign CarryOut = ( {ALUop == 3'b010} & cout_adder) |
29                       (  {ALUop == 3'b110} & cout_sub);
30     assign Overflow =   ( {ALUop == 3'b010} & ((~A[31] & ~B[31] & set_adder[31]) | (A[31] & B[31] & ~set_adder[31]) )    ) |
31                         ( {ALUop == 3'b110} & (  (~A[31] & B[31] & set_sub[31])    | (A[31] & ~B[31] & ~set_sub[31]) ) ) |
32                         ( {ALUop == 3'b111} & (  (~A[31] & B[31] & set_sub[31])    | (A[31] & ~B[31] & ~set_sub[31])) );
33     assign Zero = (Result==32'b0)? 1 : 0;
34 endmodule
```

18-20 行其实是一个复用的累赘，可以省略。但因为一开始在写代码的时候，从 Result 的赋值开始写起，所以导致后续无法复用(24 行和 25 行），因此多用了一些逻辑门。27 行的判断，是由于 slt 判断根据 A-B 的结果是正或者负，需要用减法结果的符号位和溢出标志进行异或。
30-33 行处理 Overflow,用符号位的进位异或最高有效位的进位总是结果出错。（因为我对于符号位的进位：处理是将两个 32 位的运算结果，用第 33 位记录，最后根据第 33 位的 0 或 1 来判断，这应该也是 assign {cout,result}=A+B+cin;由于左边是 1 位＋32 位，所以自动将右侧扩展为 33 位。）对于最高有效位的进位，需要将 A 和 B 截断为 31 位，用 32 位的数去存，并取最高位的加法结果。但是对于减法的处理，如果将 B 取反加 1 截断成 31 位，那是否需要先用一个 33 位的数来存 B 取反加 1 的结果？这些分析比较复杂，最后的代码还是采用了朴实的：正+正=负，负+负=正，正-负=负，负-正=正，符号位判断。代码的可读性还不错，但是中间采用了 6 个加法器，过于累赘。改写成 2 个加法器版本：
具体来说改动的地方不多，只是把原来的 set_adder 和 set_sub 去掉，用 Result_adder 和 Result_sub 完全可以替代。把原来 assign Result 语句中的 A+B

和 A+ ~B + 1 去掉，用 Result_adder 和 Result_sub 替代。这是从 6 个加法器->2 个加法器的版本。

```
`define DATA_WIDTH 32

module alu(
    input [`DATA_WIDTH - 1 : 0] A,
    input [`DATA_WIDTH - 1 : 0] B,
    input [             2 : 0] ALUop,
    output  Overflow,
    output  CarryOut,
    output Zero,
    output [`DATA_WIDTH - 1 : 0] Result
);
    wire cout_adder,cout_sub;
    wire [`DATA_WIDTH - 1 : 0] Result_adder,Result_sub;
    //直接计算符号位的进位，或者无符号数的Carryout
    assign {cout_adder,Result_adder} = A + B;
    assign {cout_sub,Result_sub} = A + ~B + 1;

    assign Result = ( {32{ALUop == 3'b000}} & (A & B)    ) |
                    ( {32{ALUop == 3'b001}} & (A | B)    ) |
                    ( {32{ALUop == 3'b010}} & (Result_adder)    ) |
                    ( {32{ALUop == 3'b110}} &  (Result_sub) ) |
                    ( {32{ALUop == 3'b111}} & (Result_sub[31] ^ Overflow) );

    assign CarryOut = ( {ALUop == 3'b010} & cout_adder) |
                      (  {ALUop == 3'b110} & cout_sub);
    assign Overflow =   ( {ALUop == 3'b010} & ((~A[31] & ~B[31] & Result_adder[31]) | (A[31] & B[31] & ~Result_adder[31]) )    ) |
                        ( {ALUop == 3'b110} & (  (~A[31] & B[31] & Result_sub[31])    | (A[31] & ~B[31] & ~Result_sub[31]) ) ) |
                        ( {ALUop == 3'b111} & (  (~A[31] & B[31] & Result_sub[31])    | (A[31] & ~B[31] & ~Result_sub[31])) );
    assign Zero = (Result==32'b0)? 1 : 0;
endmodule
```

一个加法器的更正，来自于对 CarryOut 的理解，之前根据两个加法器的版本更改了一个错误版：

```
    //构造一个加减法使能信号
    wire cin,cout,enable;
    assign enable = ({ALUop == 3'b010} & 0 ) |
                    ({ALUop == 3'b110 } & 1) |
                    ({ALUop == 3'b111} & 1);
    wire [`DATA_WIDTH - 1 : 0] B1,Result_prev;
    assign B1 = enable? ~B:B;
    assign cin = enable? 1:0;
    assign {cout,Result_prev} = A + B1 + cin;

    assign Result = ( {32{ALUop == 3'b000}} & (A & B)    ) |
                    ( {32{ALUop == 3'b001}} & (A | B)    ) |
                    ( {32{ALUop == 3'b010}} & (Result_prev)    ) |
                    ( {32{ALUop == 3'b110}} &  (Result_prev) ) |
                    ( {32{ALUop == 3'b111}} & (Result_prev[31] ^ Overflow) );

    assign CarryOut = ( {ALUop == 3'b010} & cout) |
                      (  {ALUop == 3'b110} & cout);
    assign Overflow =   ( {ALUop == 3'b010} & ((~A[31] & ~B[31] & Result_prev[31]) |
                        ( {ALUop == 3'b110} & (  (~A[31] & B[31] & Result_prev[31])
                        ( {ALUop == 3'b111} & (  (~A[31] & B[31] & Result_prev[31])
    assign Zero = (Result==32'b0)? 1 : 0;
endmodule
```

一开始认为 21 行和原来两个加法器的 15、16 行是等价的。后来发现报错信息的 CarryOut 和我认为的 CarryOut 有一定差别：

```
24  ================================================
25  ERROR: A = 5ae0ff31, B = 4bd60e41, ALUop = 6, Result = 0f0af0f0, CarryOut = 1, C
    arryOutReference = 0.
26  ================================================
```

关于 CarryOut 为什么是 0 不是 1，感谢群里同学的解答，在运算时自动将 A 和 B 进行 33 位的符号位扩展。所以在最高位引入一个 0 后，按位取反操作得到 1，与最终的结果抵消。所以在将两个加法器合并为一个的时候，需要用一个 33 位宽的数据来处理 B 的按位取反结果。19 行需要进行改动，最后得到正确版本。

三、 对讲义中思考题（如有）的理解和回答

四、 在课后，你花费了大约_____5___小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

感谢群里 2 班-李子恒同学对于我问题的解答，感谢蒋卓伦同学指出我的一些错误之处，感谢助教陈飞羽同学的指点，感谢张科老师的精心讲授和解答。

逻辑与&&和按位与&，我容易弄混，推荐 HDLBits 上练习题 96 边沿捕获寄存器。如果将两个 32 位的数进行按位与操作，则得到一个 32 位的数，这个 32 位的数只要有不为 0 的部分，条件判断为真。

if(~in & d_last)在条件判断时，可以等价地视为：

if(~in[0] & d_last[0] | ~in[1] & d_last[1] | ~in[2] & d_last[2] | ... |)

但该题只需要考虑当前位 in[i]与 d_last[i],与其他 in[j]和 d_last[j]都无关。所以条件判断和多维向量的按位与结合时，容易出错。