中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 专业 计算机科学与技术

实验项目编号 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 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下 (注意: reports 全部小写)。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` 推送到实验课 SERVE 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

1 寄存器堆设计

1.1 Verilog 代码

```
`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] data_array[`DATA_WIDTH - 1:0];
  always @(posedge clk) begin
    if (wen && waddr != 0) begin
      data_array[waddr] <= wdata;
    end
  end
  assign rdata1 = (raddr1 == {`ADDR_WIDTH{1'b0}} ? {`ADDR_WIDTH{1'b0}} : data_array[raddr1]);
  assign rdata2 = (raddr2 == {`ADDR_WIDTH{1'b0}} ? {`ADDR_WIDTH{1'b0}} : data_array[raddr2]);
```

```
endmodule
```

在这段代码中,我们首先使用一个 ROM 定义了一个寄存器堆。我们规定,当写使能信号 wen 为高电平且写地址不为零时,数据 wdata 将被写入到地址 waddr 所指定的寄存器中。在时钟信号 clk 的上升沿,我们将数据写入到寄存器中。在时钟信号 clk 的下降沿,我们将数据从寄存器中读出。我们使用 raddr1 和 raddr2 两个地址信号来指定要读取的寄存器。最后,我们将读取到的数据 rdata1 和 rdata2 输出到外部。

1.2 逻辑电路图

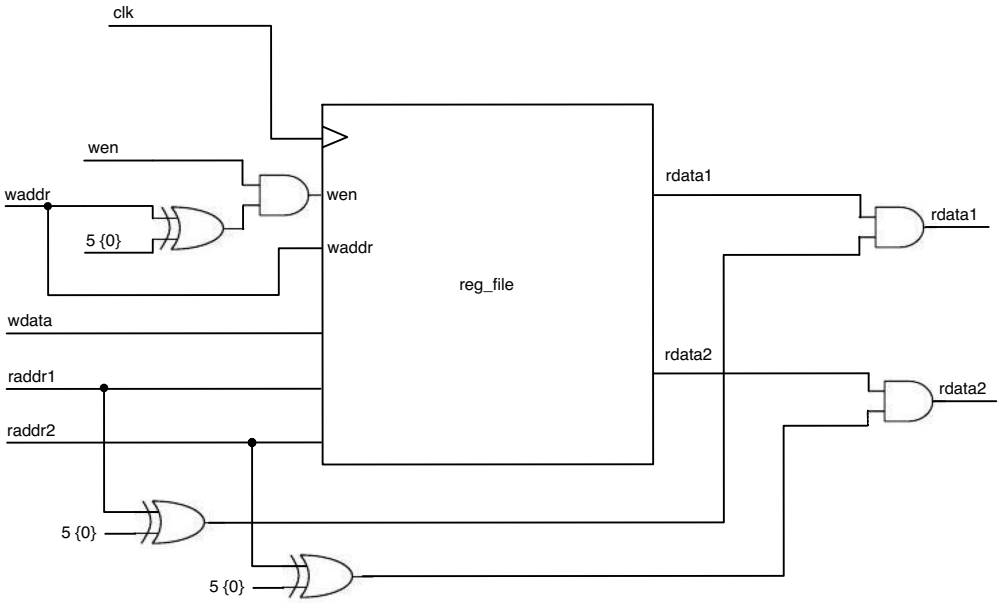

图 1: 寄存器堆逻辑电路图

1.3 仿真波形图

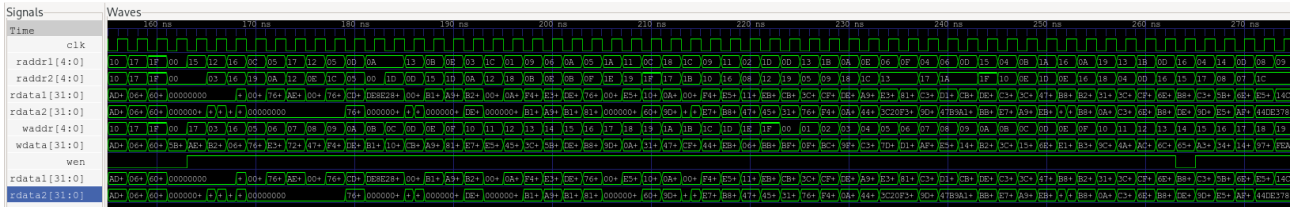

图 2: 寄存器堆仿真波形图

图 2 为我所写的寄存器堆的仿真波形图。这一波形图通过了仿真测试。在本图中,可以看到我的模块与样例模块的行为完全一致。

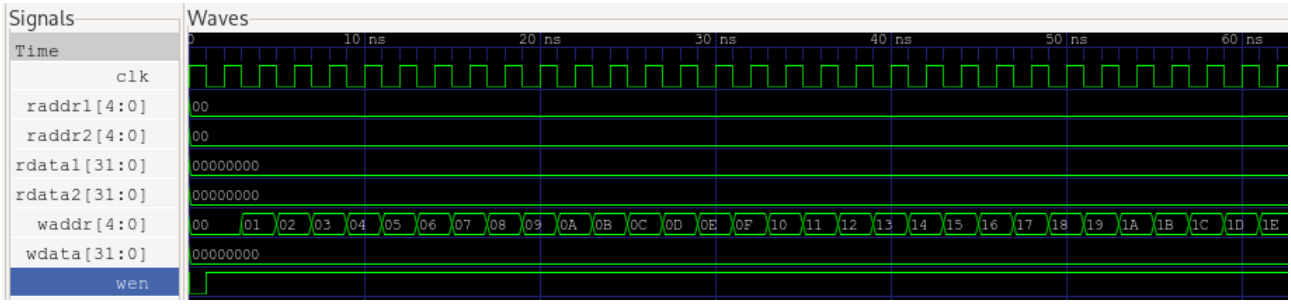

图 3: 寄存器堆仿真初始行为

图 3 为寄存器堆的初始行为。在时钟 clk 的上升沿，外部信号不断将零写入寄存器堆的每一个位置。并且，此时的读地址始终保持为零。

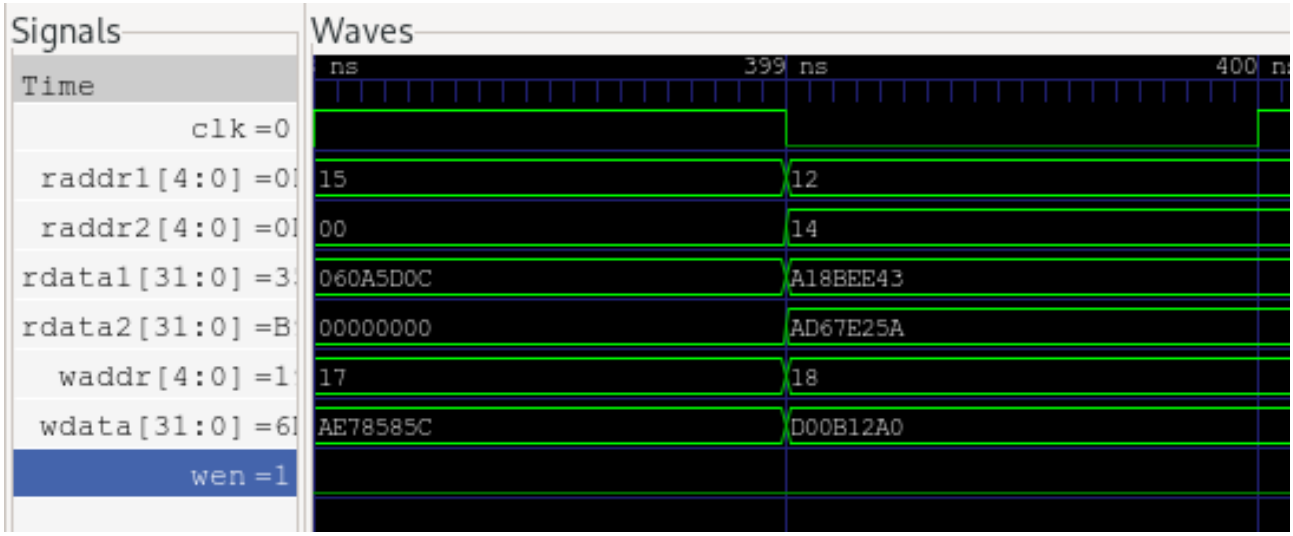

图 4: 寄存器堆读取和写入行为

图 4 展示了寄存器堆的读取行为。可以看出，寄存器堆根据 `raddr1` 和 `raddr2` 所指定的位置进行信息的读取。并且当读地址为零时，读出的数据始终为零。此外，根据读出的数据的变化情况可以看出，寄存器堆能够正常根据 `waddr` 和 `wdata` 两个信号所指定的地址和数据来写入数据。

1 ALU 设计

1.1 Verilog 代码

```
`timescale 10 ns / 1 ns
`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 is_sub = (ALUop == 3'b110 | ALUop == 3'b111) ? 1'b1 : 1'b0;

wire [32:0] sel_B =
    (ALUop == 3'b010) ? B :
    (ALUop == 3'b110 | ALUop == 3'b111 | ALUop == 3'b011) ? ~B :
    1'b0;

wire [`DATA_WIDTH:0] add_result = A + sel_B + is_sub;
wire [`DATA_WIDTH:0] sub_result = add_result;


wire add_overflow = (A[`DATA_WIDTH-1] == B[`DATA_WIDTH-1]) &&
                (add_result[`DATA_WIDTH-1] ^ A[`DATA_WIDTH-1]);

wire sub_overflow = (A[`DATA_WIDTH-1] != B[`DATA_WIDTH-1]) &&
                (sub_result[`DATA_WIDTH-1] ^ A[`DATA_WIDTH-1]);


wire slt_result;
assign slt_result =
    (A[`DATA_WIDTH-1] & !B[`DATA_WIDTH-1]) ? 1'b1 :
    (!A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1]) ? 1'b0 :
    (sub_overflow ^ sub_result[`DATA_WIDTH-1]);

wire sub_carryout = sub_result[`DATA_WIDTH];

assign Result =
    (ALUop == 3'b000) ? (A & B) :
    (ALUop == 3'b001) ? (A | B) :
    (ALUop == 3'b010) ? (A + B) :
    (ALUop == 3'b100) ? (A ^ B) :
    (ALUop == 3'b101) ? ~(A | B) :
    (ALUop == 3'b110) ? sub_result[`DATA_WIDTH-1:0] :
    (ALUop == 3'b011) ? {{(`DATA_WIDTH-1){1'b0}}, CarryOut} :
    (ALUop == 3'b111) ? {{(`DATA_WIDTH-1){1'b0}}, slt_result} :
    {`DATA_WIDTH{1'bx}};

assign Zero = (Result == 0);

assign CarryOut =
    (ALUop == 3'b010) ? add_result[`DATA_WIDTH] :
    (ALUop == 3'b110 | ALUop == 3'b111) ? sub_carryout :
    1'b0;

assign Overflow =
    (ALUop == 3'b010) ? (A[`DATA_WIDTH-1] == B[`DATA_WIDTH-1] && (Result[`DATA_WIDTH-1] ^
        A[`DATA_WIDTH-1])) :
```

```
    (ALUop == 3'b110 | ALUop == 3'b111) ? (A[`DATA_WIDTH-1] ^ B[`DATA_WIDTH-1]) & (A[`DATA_WIDTH-1]
        ^ Result[`DATA_WIDTH-1]) :
    1'b0;

endmodule
```

针对不同的 ALUop 输入，我的设计方案通过嵌套三目运算符进行选择，最终将所需要的值赋给输出信号，并通过了线上和本地的仿真测试。关于此 ALU 各个信号的生成逻辑，在此实验报告的后面有详细的叙述。

1.2 逻辑电路图

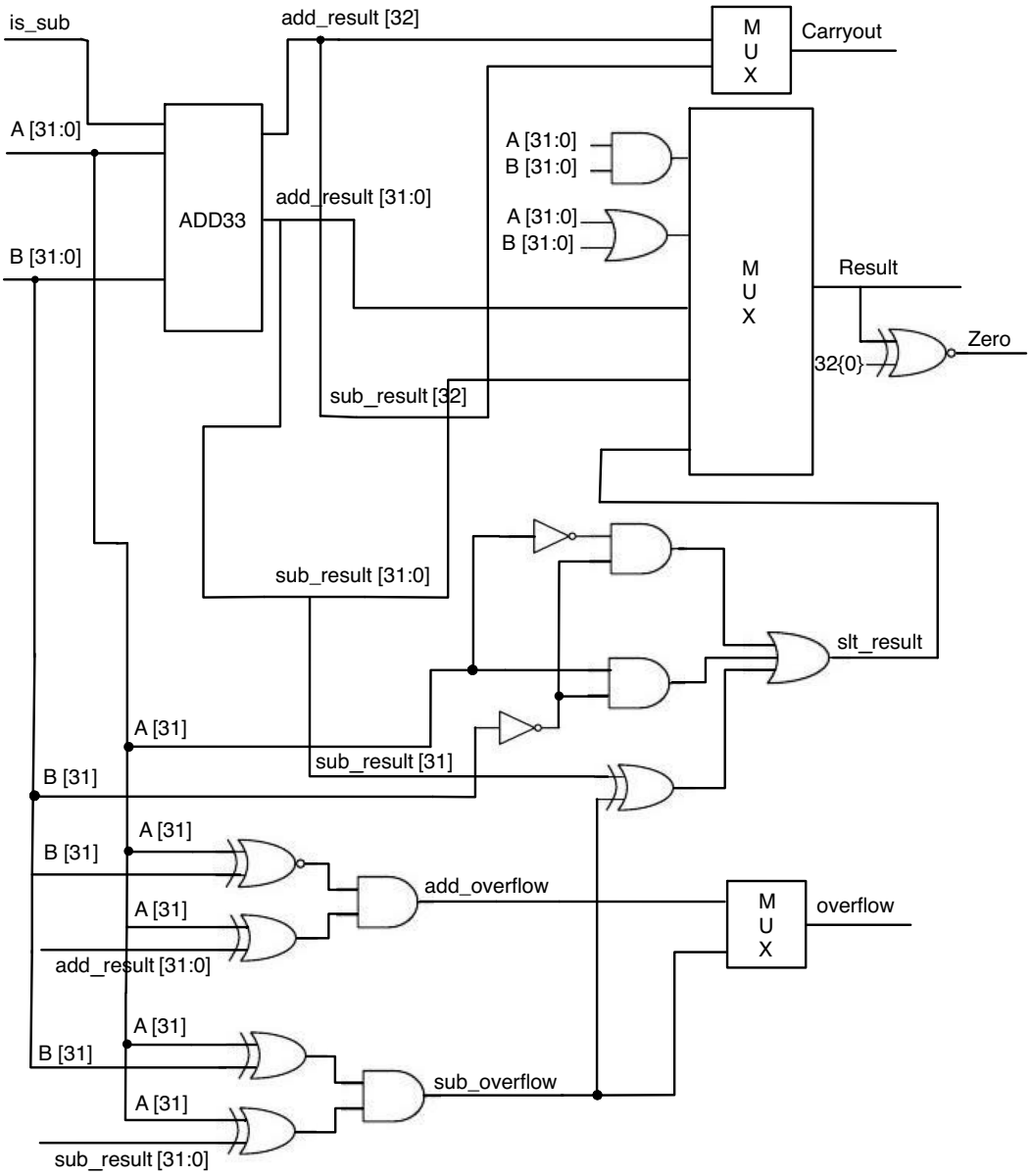

图 5: ALU 逻辑电路图

1.3 仿真波形图

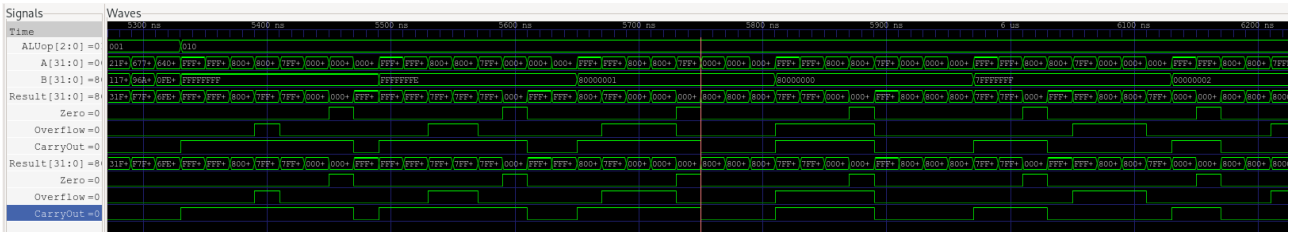

图 6: ALU 仿真波形图

图 6 为我所写的算术逻辑单元的仿真波形图。这一波形图通过了仿真测试。在本图中,可以看到我的模块与样例模块的行为完全一致。

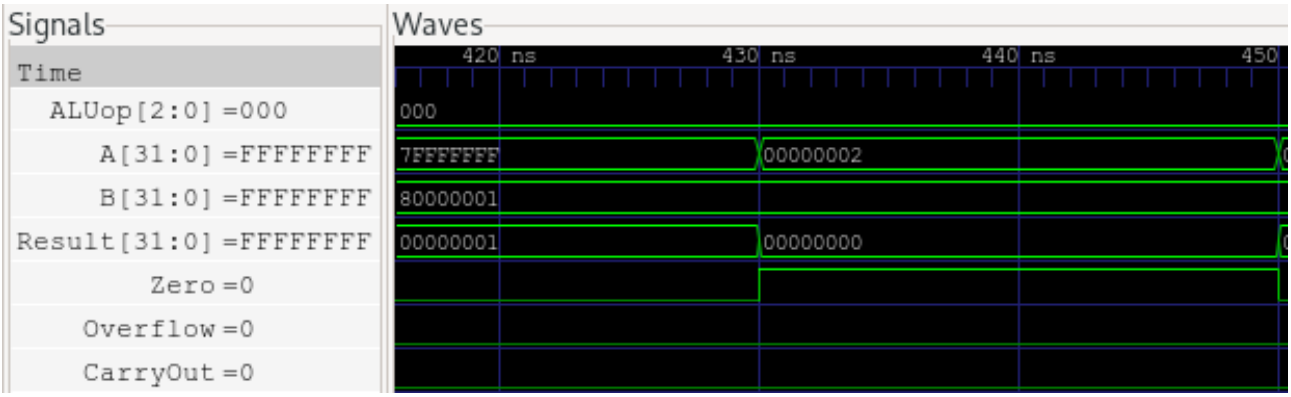

图 7: ALU 与运算行为

图 7 展示了算术逻辑单元的与运算行为。当 7FFFFFFF 和 80000001 进行与运算时,结果为 00000001。

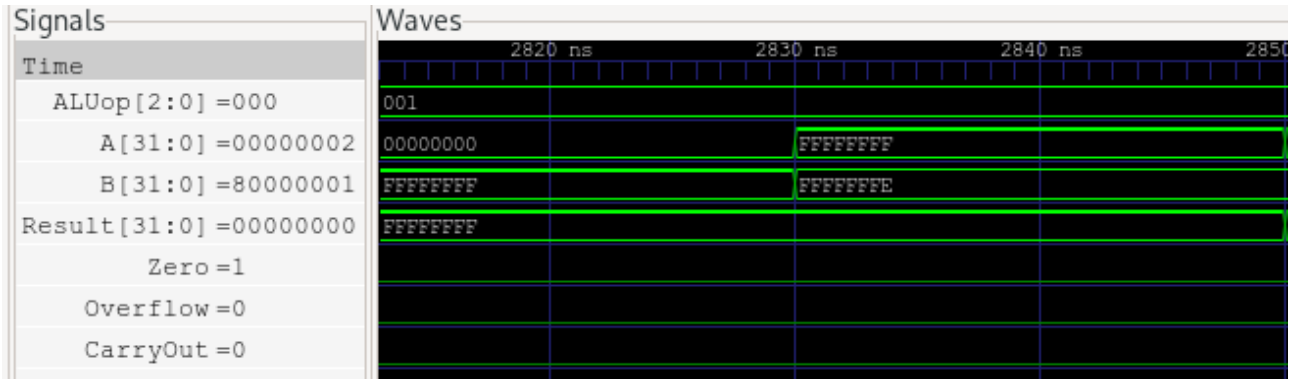

图 8: ALU 或运算行为

图 8 展示了算术逻辑单元的或运算行为。当 00000000 和 FFFFFFFF 进行或运算时,结果为 FFFFFFFF。

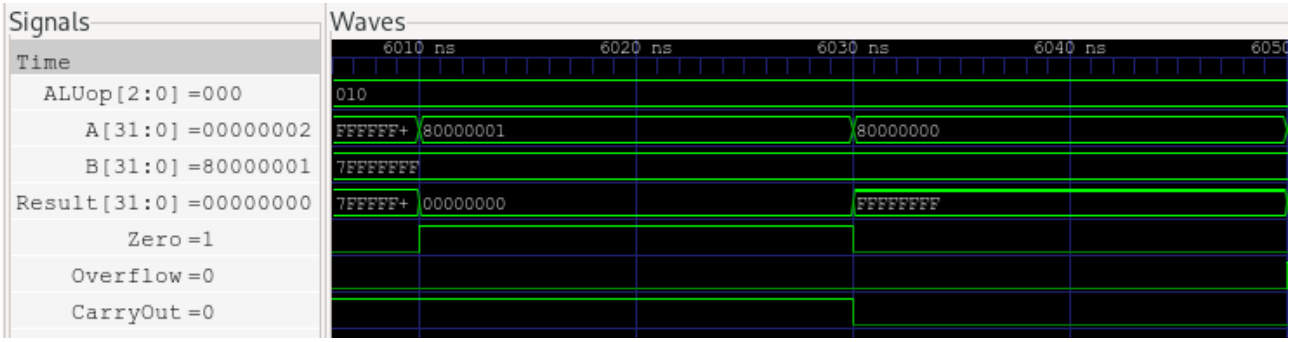

图 9: ALU 加法行为

图 9 展示了算术逻辑单元的加法行为。可以看到,当作为有符号数的 7FFFFFFF 和它的相反数 80000001 相加时,结果为 0,并且溢出信号 Overflow 为 0, 二者作为无符号数相加时进位信号 CarryOut 为 1。

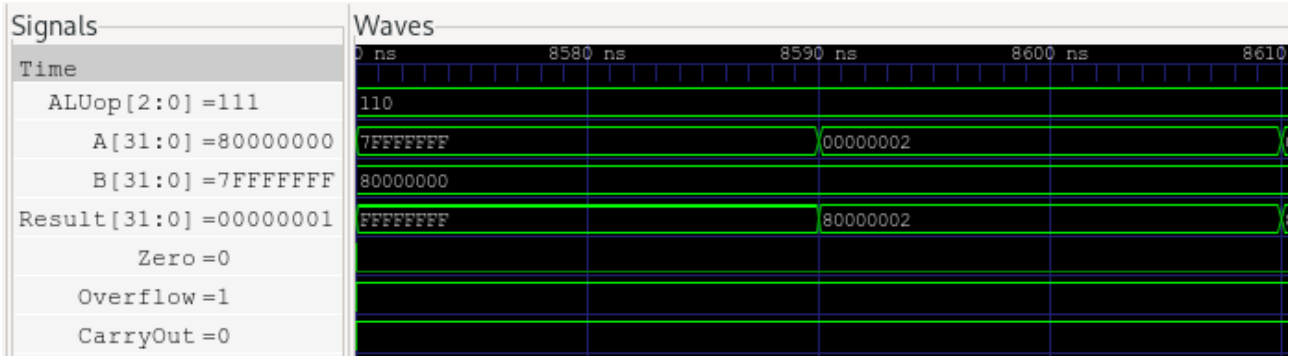

图 10: ALU 减法行为

图 10 展示了算术逻辑单元的减法行为。可以看到,作为有符号数的 7FFFFFFF(最大正数) 和 80000000(最大负数) 相减时,显然超出了补码的表示范围,因此发生了溢出。同时,二者作为无符号数相加时同样会产生进位。

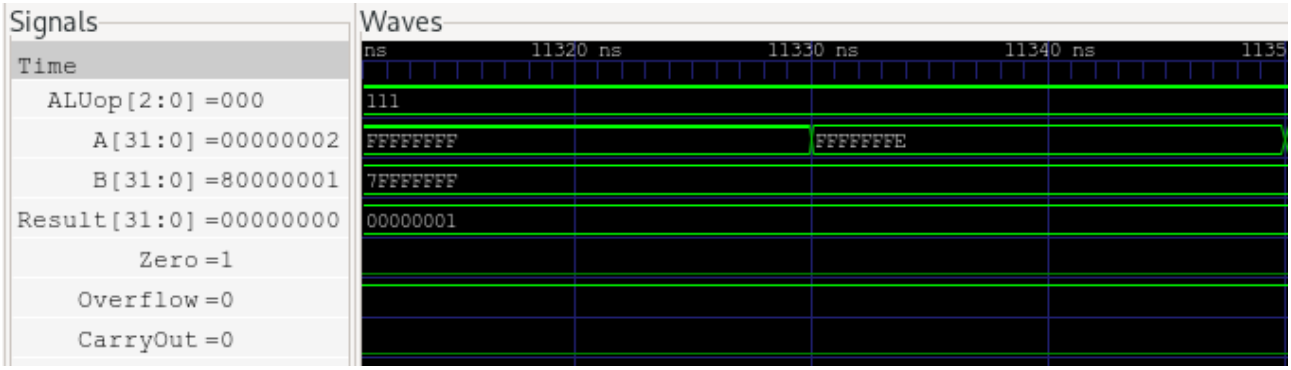

图 11: ALU 比较行为

图 11 展示了算术逻辑单元的比较行为。可以看到,当 FFFFFFFF(-1) 和 7FFFFFFF(最大正数) 相比较时,输出的结果为 1,这符合要求。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

关于实验平台的一点小问题

线上的实验平台似乎不会对提交的代码进行完整的功能测试，也就是说，即便代码不能完成所要求的逻辑功能，平台也不会报错误。因此，完整的功能测试需要进行本地仿真。

寄存器堆设计问题

1. 在本地仿真时出现 `Concatenation operand "'sd0" has indefinite width` 错误

虽然我在进行本地仿真之前已经将代码推送到了远程仓库，并且通过了线上的测试，但是，我在进行本地仿真时仍然遇到了上面的错误。我在网络上查询了相关信息，发现该错误的原因是由于没有对一位宽的 `1'b0` 信号进行位宽指定。这提醒我不同的仿真工具对代码的要求不同。在书写代码的时候需要按照最标准的格式来写，不能随意缺省，否则会造成类似的错误。

ALU 设计问题

1. 在不使用 `if`、`case` 和 `always` 的情况下完成 ALU 的设计

由于距离上次写 Verilog 代码的时间已经比较久远，我对一些 Verilog 语法有所忘记。拿到这个问题后，我的第一个思路是采用 `case` 语句对输入信号不同的情况予以判断和选择。基本形式如下：

```
case (ALUop) begin
   4'b000:
     assign result = a & b;
   4'b001:
     assign result = a | b;
   ...
   default:
     assign result = 0;
endcase
```

但后来，我了解到，`case` 语句只能在 `always` 块下使用。因此，我们需要对代码进行重构。一开始，出于简单考虑，我在保持原始代码结构的情况下将 `case` 语句替换掉，构成了如下样貌的代码。

```
assign Result = ALUop == 3'b000 ? (A & B) : Result;
assign Result = ALUop == 3'b001 ? (A | B) : Result;
...
assign Result = ALUop == 3'b111 ? {{(`DATA_WIDTH - 1) {0}}, temp_result[`DATA_WIDTH-1]} : Result;
```

这段代码正常通过了在线的仿真测试，但是在本地仿真时对某些输入信号出现了未知状态 (X)。我意识到，在组合电路中并行对同一信号进行赋值会产生仿真时出现未知状态的问题。因此，我最后将代码进行了一次较大的重构，采用了嵌套三元运算符的形式完成了最终的设计。

```
assign Result =
   (ALUop == 3'b000) ? (A & B) :
   (ALUop == 3'b001) ? (A | B) :
   (ALUop == 3'b010) ? (A + B) :
   (ALUop == 3'b110) ? sub_result[`DATA_WIDTH-1:0] :
   (ALUop == 3'b111) ? {{(`DATA_WIDTH-1){1'b0}}, slt_result} :
```

```
    {`DATA_WIDTH{1'bx}};
```

以上便是重构后代码的 Result 信号的分配逻辑。该逻辑仅对 Result 信号进行了一次赋值，并且能够顺利实现信号选择的功能。

2. 和 ALU 设计有关的数学思考

carryout 信号的生成

在我的代码当中，进位 (CarryOut) 信号的生成逻辑如下：

```
wire is_sub = (ALUop == 3'b110 | ALUop == 3'b111) ? 1'b1 : 1'b0;

wire [32:0] sel_B =
    (ALUop == 3'b010) ? B :
    (ALUop == 3'b110 | ALUop == 3'b111 | ALUop == 3'b011) ? ~B :
    1'b0;

wire [`DATA_WIDTH:0] add_result = A + sel_B + is_sub;
wire [`DATA_WIDTH:0] sub_result = add_result;

wire sub_carryout = sub_result[`DATA_WIDTH];

assign CarryOut =
    (ALUop == 3'b010) ? add_result[`DATA_WIDTH] :
    (ALUop == 3'b110 | ALUop == 3'b111) ? sub_carryout :
    1'b0;
```

在这段代码中，我将 32 位加法扩展为 33 位。在执行加法时，将加数 B 的扩展位初始值设置为 0；在执行减法时，将加数 B 的扩展位初始值设置为 1 。对于加法，这一赋值是十分自然的。当执行加法之后第 33 位为 1 时，自然代表两个加数的加和大于 2^n，即出现进位。对于减法，将 $\overline{\mathrm{B}}$ 的扩展位初始值设置为 1 的目的是，无符号数减法产生进位时，说明够减，此时会将扩展位上的 1 冲掉；当无符号数减法不产生进位时，说明不够减，此时扩展位上的 1 会保留。这样，扩展位上的值就恰好说明了是否出现借位。于是，我们要证明的问题就是：当无符号数减法够减时，一定产生进位。

对于无符号数 B，其补数为 $\sim B+1$，等价于 2^n-B(n 为位数)。因此，运算 $A+\sim B+1$ 可表示为：

$$A+(2^n-B)=A-B+2^n$$

当这个等式成立时，$A>B$ 恰恰说明相加后的结果会超出 2^n 的范围，因此会产生进位。也就证明了上面的问题。

Overflow 信号的生成

在我的设计当中，溢出 (Overflow) 信号的生成逻辑如下：

```
wire add_overflow = (A[`DATA_WIDTH-1] == B[`DATA_WIDTH-1]) &&
                (add_result[`DATA_WIDTH-1] != A[`DATA_WIDTH-1]);

wire sub_overflow = (A[`DATA_WIDTH-1] != B[`DATA_WIDTH-1]) &&
                (sub_result[`DATA_WIDTH-1] != A[`DATA_WIDTH-1]);
```

这两段代码所表达的逻辑是比较容易理解的。我们不妨从数轴的角度来理解这个问题。当两个加数都为正时，它们都位于数轴的右端。因此二者相加很容易超过 n 位补码能表达的最大正数 2^n-1 。并且由于两个加数的补码均以 0 开头，因此溢出后开头的数只可能是 1 。因此两个正数加法的溢出逻辑可以由第一个语句来判断。

当两个加数都为负时，假设 A 和 B 的真值为 a 和 b，那么

$$A + B = 2^n + a + 2^n + b = 2^{n+1} + a + b \mod 2^n$$

当不发生溢出，即 $A + B \geq -2^{n-1}$，有

$$2^{n+1} + a + b \mod 2^n = 2^n + a + b \geq 2^{n-1}$$

由此可知，当不发生溢出时，符号位始终为 1 。同理易推得，当发生溢出时，符号位必定为 0 。最后，从数轴的观点易知，不同符号的两个数相加，不可能发生溢出。因此，第一个语句判断两个数相加是否发生溢出的逻辑是严密的。

最后，对于减法，由于减法的计算方式为 A 加上 B 的补码，因此减法最终转换为了加法。其溢出判断逻辑和加法的溢出判断逻辑是相似的。

`slt_result` 信号的生成

我的代码中，`slt_result` 信号的生成逻辑如下：

```
wire slt_result;
assign slt_result =
   (A[`DATA_WIDTH-1] & !B[`DATA_WIDTH-1]) ? 1'b1 :
   (!A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1]) ? 1'b0 :
   (sub_overflow ^ sub_result[`DATA_WIDTH-1]);
```

前两段逻辑是显然的。第三段判断逻辑可以由前面对减法的描述来解释：当不发生溢出时，减法结果为负时，减法结果的符号位为 1 ，即 `slt_result` 为 1 ；当发生溢出时，减法结果的符号位为 0 ，但此时 `sub_overflow` 的取值为 1 ，即 `slt_result` 为 0 。因此，第三段逻辑的设计是合理的。

三、 对讲义中思考题(如有)的理解和回答

未见讲义中的思考题。

四、 实验所耗时间

在课后，你花费了大约____8____ 小时完成此次实验。

作为本学期计组研讨课的第一个实验，本次实验还是让我有很大收获的。我回忆起了 Verilog 的基本语法以及需要注意的规范。同时，这两个模块的设计也让我更深刻的理解了真实的计算机中寄存器堆和算术逻辑单元这两个模块的功能和工作方式。此外，本实验也让我对计算机的计算方式 (也就是有符号数和无符号数的加减) 以及补码有了更深刻的理解。