Lab 6 report

PB2111711 陈昕琪

实验目的与内容

> 1. 尝试使用学到的知识，搭建出一个计算机系统中的重要组成部分：ALU（Arithmetic Logic Unit，算术逻辑单元），并通过对运算的不同实现方式的对比，来学习如何在设计中权衡时间性能与资源的使用。
> 2. 通过学习加法器和超前进位加法器的知识，举一反三，实现多位运算以及其他运算单元。
> 3. 进一步熟悉 Verilog 语言并学会综合分析电路以及功能实现。

1. 必做内容：加法器

> **要求：**
>
> 参照实验文档中对 4bits 超前进位加法器和层次扩展的描述，设计并编写一个 32bits 加法器。要求分别基于 4bits 超前进位加法器与 8bits 超前进位加法器实现。

本题说明分为三部分

1. 4 位超前进位加法器 + 层次扩展成 32 位

> **逻辑实现：**
>
> 将4位超前进位加法器层次拓展为32位，只需要将8个超前进位加法器进行串联。 例化时，要注意，每个超前进位加法器的高位进位是下一个超前进位加法器的地位进位(这里用cmid[i]来表示每个加法器的进位位)，即可实现加法器的串联。

代码如下：

```
module Adder32_4bits(
    input                   [31 : 0]        a, b,
    input                   [ 0 : 0]        ci,//低位进位
    output                  [31 : 0]        s,//和
    output                  [ 0 : 0]        co//高位进位
);
//需要8个4位超前进位加法器
wire    [6:0] cmid;

Adder_LookAhead4 adder0(
    .a(a[3:0]),
    .b(b[3:0]),
    .ci(ci),
    .s(s[3:0]),
    .co(cmid[0])
```

```
);
Adder_LookAhead4 adder1(
    .a(a[7:4]),
    .b(b[7:4]),
    .ci(cmid[0]),
    .s(s[7:4]),
    .co(cmid[1])
);
Adder_LookAhead4 adder2(
    .a(a[11:8]),
    .b(b[11:8]),
    .ci(cmid[1]),
    .s(s[11:8]),
    .co(cmid[2])
);
Adder_LookAhead4 adder3(
    .a(a[15:12]),
    .b(b[15:12]),
    .ci(cmid[2]),
    .s(s[15:12]),
    .co(cmid[3])
);
Adder_LookAhead4 adder4(
    .a(a[19:16]),
    .b(b[19:16]),
    .ci(cmid[3]),
    .s(s[19:16]),
    .co(cmid[4])
);
Adder_LookAhead4 adder5(
    .a(a[23:20]),
    .b(b[23:20]),
    .ci(cmid[4]),
    .s(s[23:20]),
    .co(cmid[5])
);
Adder_LookAhead4 adder6(
    .a(a[27:24]),
    .b(b[27:24]),
    .ci(cmid[5]),
    .s(s[27:24]),
    .co(cmid[6])
);
Adder_LookAhead4 adder7(
    .a(a[31:28]),
    .b(b[31:28]),
    .ci(cmid[6]),
    .s(s[31:28]),
    .co(co)
);

endmodule
```

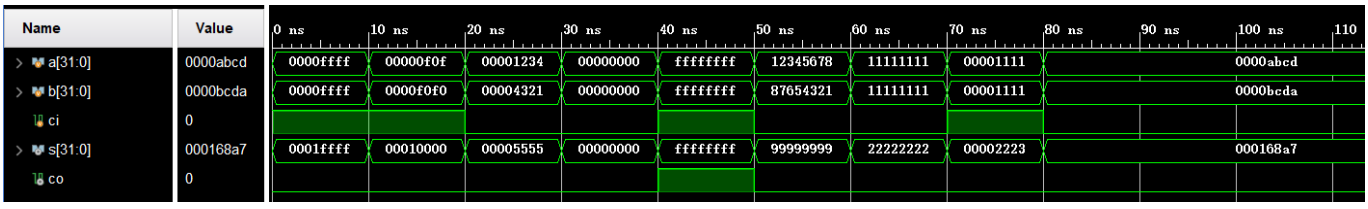
仿真结果与分析

> 根据所写的代码，编写相应的仿真文件验证其正确性。

仿真文件代码如下：

```
module Adder32_4bits_tb();
reg [31:0] a,b;
reg ci;
wire [31:0] s;
wire co;
Adder32_4bits adder(
    .a(a),
    .b(b),
    .ci(ci),
    .s(s),
    .co(co)
);
    initial begin
        a=32'hffff; b=32'hffff; ci=1'b1;
        #10;
        a=32'h0f0f; b=32'hf0f0; ci=1'b1;
        #10;
        a=32'h1234; b=32'h4321; ci=1'b0;
        #10;
        a=32'h0000; b=32'h0000; ci=1'b0;
        #10;
        a=32'hffffffff; b=32'hffffffff; ci=1'b1;
        #10;
        a=32'h12345678; b=32'h87654321; ci=1'b0;
        #10;
        a=32'h11111111; b=32'h11111111; ci=1'b0;
        #10;
        a=32'h1111; b=32'h1111; ci=1'b1;
        #10;
        a=32'habcd; b=32'hbcda; ci=1'b0;
    end
endmodule
```

> 仿真得出的波形图如下：



> 对比分析得知程序正确。

2. 8 位超前进位加法器

> **逻辑实现：**
>
> 根据4位超前进位加法器的原理，并根据已经给出的位的运算方式，用数学归纳法得出C[7]的表达式。
> 同时，根据4位超前进位加法器中s和co的关系，相应地得出8位超前进位加法器中s和co的关系，即可实现8位超前进位加法器。

代码如下：

```
module Add_LookAhead8(
    input                   [ 7 : 0]            a, b,
    input                   [ 0 : 0]            ci,         // 来自低位的进位
    output                  [ 7 : 0]            s,          // 和
    output                  [ 0 : 0]            co          // 向高位的进位
);

wire    [7:0] C;
wire    [7:0] G;
wire    [7:0] P;

assign  G = a & b;
assign  P = a ^ b;

assign  C[0] = G[0] | ( P[0] & ci );
assign  C[1] = G[1] | ( P[1] & G[0] ) | ( P[1] & P[0] & ci );
assign  C[2] = G[2] | ( P[2] & G[1] ) | ( P[2] & P[1] & G[0] ) | ( P[2] & P[1] &
P[0] & ci );
assign  C[3] = G[3] | ( P[3] & G[2] ) | ( P[3] & P[2] & G[1] ) | ( P[3] & P[2] &
P[1] & G[0] ) | ( P[3] & P[2] & P[1] & P[0] & ci );
assign  C[4] = G[4] | ( P[4] & G[3] ) | ( P[4] & P[3] & G[2] ) | ( P[4] & P[3] &
P[2] & G[1] ) | ( P[4] & P[3] & P[2] & P[1] & G[0] ) | ( P[4] & P[3] & P[2] & P[1]
& P[0] & ci );
assign  C[5] = G[5] | ( P[5] & G[4] ) | ( P[5] & P[4] & G[3] ) | ( P[5] & P[4] &
P[3] & G[2] ) | ( P[5] & P[4] & P[3] & P[2] & G[1] ) | ( P[5] & P[4] & P[3] & P[2]
& P[1] & G[0] ) | ( P[5] & P[4] & P[3] & P[2] & P[1] & P[0] & ci );
assign  C[6] = G[6] | ( P[6] & G[5] ) | ( P[6] & P[5] & G[4] ) | ( P[6] & P[5] &
P[4] & G[3] ) | ( P[6] & P[5] & P[4] & P[3] & G[2] ) | ( P[6] & P[5] & P[4] & P[3]
& P[2] & G[1] ) | ( P[6] & P[5] & P[4] & P[3] & P[2] & P[1] & G[0] ) | ( P[6] &
P[5] & P[4] & P[3] & P[2] & P[1] & P[0] & ci );
assign  C[7] = G[7] | ( P[7] & G[6] ) | ( P[7] & P[6] & G[5] ) | ( P[7] & P[6] &
P[5] & G[4] ) | ( P[7] & P[6] & P[5] & P[4] & G[3] ) | ( P[7] & P[6] & P[5] & P[4]
& P[3] & G[2] ) | ( P[7] & P[6] & P[5] & P[4] & P[3] & P[2] & G[1] ) | ( P[7] &
P[6] & P[5] & P[4] & P[3] & P[2] & P[1] & G[0] ) | ( P[7] & P[6] & P[5] & P[4] &
P[3] & P[2] & P[1] & P[0] & ci );

// TODO: 确定 s 和 co 的产生逻辑
assign  s[0] = P[0] ^ ci;
assign  s[1] = P[1] ^ C[0];
assign  s[2] = P[2] ^ C[1];
assign  s[3] = P[3] ^ C[2];
assign  s[4] = P[4] ^ C[3];
assign  s[5] = P[5] ^ C[4];
assign  s[6] = P[6] ^ C[5];
```

```
assign  s[7] = P[7] ^ C[6];
assign  co   = C[7];

endmodule
```

仿真结果与分析

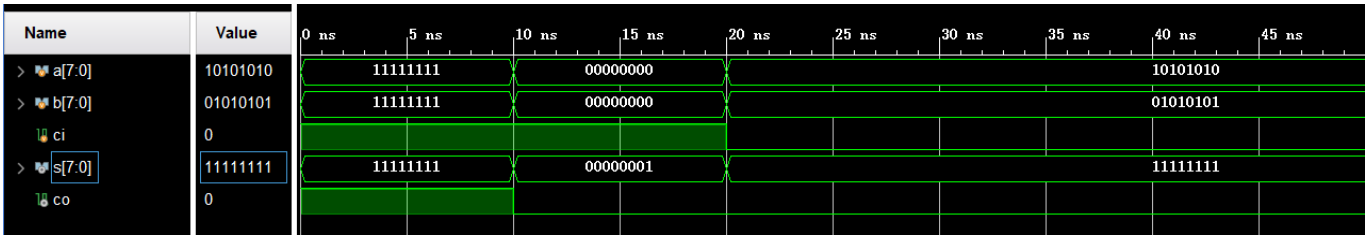
> 根据所写的代码，编写相应的仿真文件验证其正确性。

仿真文件代码如下：

```
module LookAhead8_tb();
reg [7:0] a,b;
reg ci;
wire [7:0] s;
wire co;

Add_LookAhead8 adder(
    .a(a),
    .b(b),
    .ci(ci),
    .s(s),
    .co(co)
);
    initial begin
        a=8'b11111111; b=8'b11111111; ci=1'b1;
        #10;
        a=8'b00000000; b=8'b00000000; ci=1'b1;
        #10;
        a=8'b10101010; b=8'b01010101; ci=1'b0;
        #10;
    end
endmodule
```

> 仿真得出的波形图如下：



> 对比分析得知程序正确。

3. 8 位超前进位加法器 + 层次扩展成 32 位

> **逻辑实现：**

> 类似于4位超前进位加法器层次拓展，将8位超前进位加法器层次拓展为32位，只需要将4个超前进位加法器进行串联。 同样的，例化时，要注意每个超前进位加法器的高位进位是下一个超前进位加法器的地位进位(这里用cmid[i]来表示每个加法器的进位位)，即可实现加法器的串联。

代码如下：

```
module Adder32_8bits(
    input                   [31 : 0]        a, b,
    input                   [ 0 : 0]        ci,//低位进位
    output                  [31 : 0]        s,//和
    output                  [ 0 : 0]        co//高位进位
);
//需要4个8位超前进位加法器
wire    [2:0] cmid;

Add_LookAhead8 adder0(
    .a(a[7:0]),
    .b(b[7:0]),
    .ci(ci),
    .s(s[7:0]),
    .co(cmid[0])
);
Add_LookAhead8 adder1(
    .a(a[15:8]),
    .b(b[15:8]),
    .ci(cmid[0]),
    .s(s[15:8]),
    .co(cmid[1])
);
Add_LookAhead8 adder2(
    .a(a[23:16]),
    .b(b[23:16]),
    .ci(cmid[1]),
    .s(s[23:16]),
    .co(cmid[2])
);
Add_LookAhead8 adder3(
    .a(a[31:24]),
    .b(b[31:24]),
    .ci(cmid[2]),
    .s(s[31:24]),
    .co(co)
);

endmodule
```

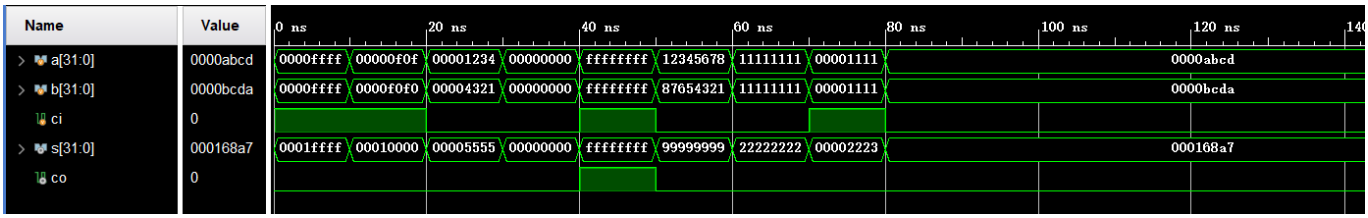
仿真结果与分析

> 根据所写的代码，编写相应的仿真文件验证其正确性。 为了方便比对，这里测试的数据和4位超前进位加法器串联成的测试数据相同，方便比较。

仿真文件代码如下：

```
module Adder32_8bits_tb();
reg [31:0] a,b;
reg ci;
wire [31:0] s;
wire co;
Adder32_8bits adder(
    .a(a),
    .b(b),
    .ci(ci),
    .s(s),
    .co(co)
);
    initial begin
        a=32'hffff; b=32'hffff; ci=1'b1;
        #10;
        a=32'h0f0f; b=32'hf0f0; ci=1'b1;
        #10;
        a=32'h1234; b=32'h4321; ci=1'b0;
        #10;
        a=32'h0000; b=32'h0000; ci=1'b0;
        #10;
        a=32'hffffffff; b=32'hffffffff; ci=1'b1;
        #10;
        a=32'h12345678; b=32'h87654321; ci=1'b0;
        #10;
        a=32'h11111111; b=32'h11111111; ci=1'b0;
        #10;
        a=32'h1111; b=32'h1111; ci=1'b1;
        #10;
        a=32'habcd; b=32'hbcda; ci=1'b0;
    end
endmodule
```

> 仿真得出的波形图如下：



> 对比分析得知程序正确。

2. 必做内容：ALU

> **要求：**

> 参照实验文档中对 ALU 及其部件的描述，设计并编写一个完整的 32 位 ALU，支持表格中列出的全部十二种运算。要求实现：

1. 加法运算
2. 减法运算
3. 有符号比较
4. 无符号比较
5. 其他运算
6. 信号选择

> **逻辑实现：**

1. 加法运算：直接调用第一题中写的32位加法器，并例化相应的模块。(这里使用的是8位超前进位加法器层次拓展构成的32位加法器)。
2. 减法运算：只需要将减数取反加一获得其负数补码，再实现加法运算即可。
3. 有符号比较：首先比较a、b的最高位，判断两者是否同号，如果同号则进行减法运算的时候不会出现溢出情况，所以直接进行减法，判断得出的结果的正负即可。如果两者不同号，则说明一正一负，则负数一定比正数小。
4. 无符号比较：首先还是比较最高位，如果最高位不同则最高位为1的数一定较大。如果最高位相同，则可以假设在32位前增加一个符号位为0，再判断两个33位有符号数字的大小。这里需要用到减法器。由于对b进行取反加一的操作，相当于高位借位位初始值为1，当a-b>=0时，高位借位位保持为1，当a-b<0时，高位借位位为0，因此这里用**u = ~co**得出结果。
5. 信号选择：根据实验教程中对ALU单元的描述，ALU通过独热码信号识别，则判断哪一位为1，并输出相应的结果即可。这里用逻辑或和逻辑与运算即可求出结果。
6. 其他运算则直接通过已有的符号获得。

> **对于符号比较时a=b的情况，根据实验文档中“当 a < b 时结果为 1”的描述，本程序中a=b的情况结果为0**

代码如下：

1. ALU

```
module ALU(
    input                   [31 : 0]        src0, src1,
    input                   [11 : 0]        sel,
    output                  [31 : 0]        res
);
// Write your code here
wire [31:0] adder_out;
wire [31:0] sub_out;
wire [0 :0] stl_out;//有符号小于比较
wire [0 :0] stlu_out;//无符号小于比较
wire [31:0] AND_out;
wire [31:0] OR_out;
wire [31:0] NOR_out;
wire [31:0] XOR_out;
wire [31:0] SLL_out;
wire [31:0] SRL_out;
```

```
wire [31:0] SRA_out;
wire [31:0] SRC1_out;

Adder adder(
    .a(src0),
    .b(src1),
    .ci(1'B0),
    .s(adder_out),
    .co()
);

AddSub sub(
    .a(src0),
    .b(src1),
    .out(sub_out),
    .co()
);

Comp comp(
    .a(src0),
    .b(src1),
    .ul(stlu_out),
    .sl(stl_out)
);

assign AND_out = src0 & src1;
assign OR_out = src0 | src1;
assign NOR_out = ~( src0 | src1 );
assign XOR_out = src0 ^ src1;
assign SLL_out = src0 << src1[4:0];
assign SRL_out = src0 >> src1[4:0];
assign SRA_out = src0 >>> src1[4:0];
assign SRC1_out = src1;
// TODO: 完成 res 信号的选择
assign res = ({32{sel[0]}} & adder_out) |
             ({32{sel[1]}} & sub_out) |
             ({32{sel[2]}} & stl_out) |
             ({32{sel[3]}} & stlu_out) |
             ({32{sel[4]}} & AND_out) |
             ({32{sel[5]}} & OR_out) |
             ({32{sel[6]}} & NOR_out) |
             ({32{sel[7]}} & XOR_out) |
             ({32{sel[8]}} & SLL_out) |
             ({32{sel[9]}} & SRL_out) |
             ({32{sel[10]}} & SRA_out) |
             ({32{sel[11]}} & SRC1_out);
// End of your code
endmodule
```

2. 减法部分(加法部分在上一题已经展示过)

```
module AddSub (
    input                   [ 31 : 0]        a, b,
    output                  [ 31 : 0]        out,
    output                  [ 0 : 0]        co
);

Adder add(
    .a(a),
    .b(~b),
    .ci(1'b1),
    .s(out),
    .co(co)
);
endmodule
```

3. Comp模块

```
module Comp(
    input [31:0] a,b,
    output ul,sl//ul,无符号比较;sl,有符号比较
    );
reg temp;
reg u;//sl1表示a,b同号，sl2表示a，b异号
wire [31:0] s;
wire co;

AddSub sub1(
    .a(a),
    .b(b),
    .out(s),
    .co(co)
);

//无符号比较
always @(*) begin
    if(a[31] == 1 && b[31] == 0) begin//a>b
        u = 0;
    end
    else if(a[31] == 0 && b[31] == 1) begin
        u = 1;
    end
    else if(a[31]==b[31]) begin
        u = ~co;//进位为1，表示a>b,u=0
    end
end
assign ul = u;
//有符号比较
always @(*) begin
    if(a[31] == b[31]) begin//ab同号
        temp = s[31];
```

```
    end else begin
        if(a[31] == 1 && b[31] == 0) begin//ab异号
            temp = 1;//a为负，b为正
        end else temp = 0;
    end
end
assign sl = temp;
endmodule
```

仿真结果与分析

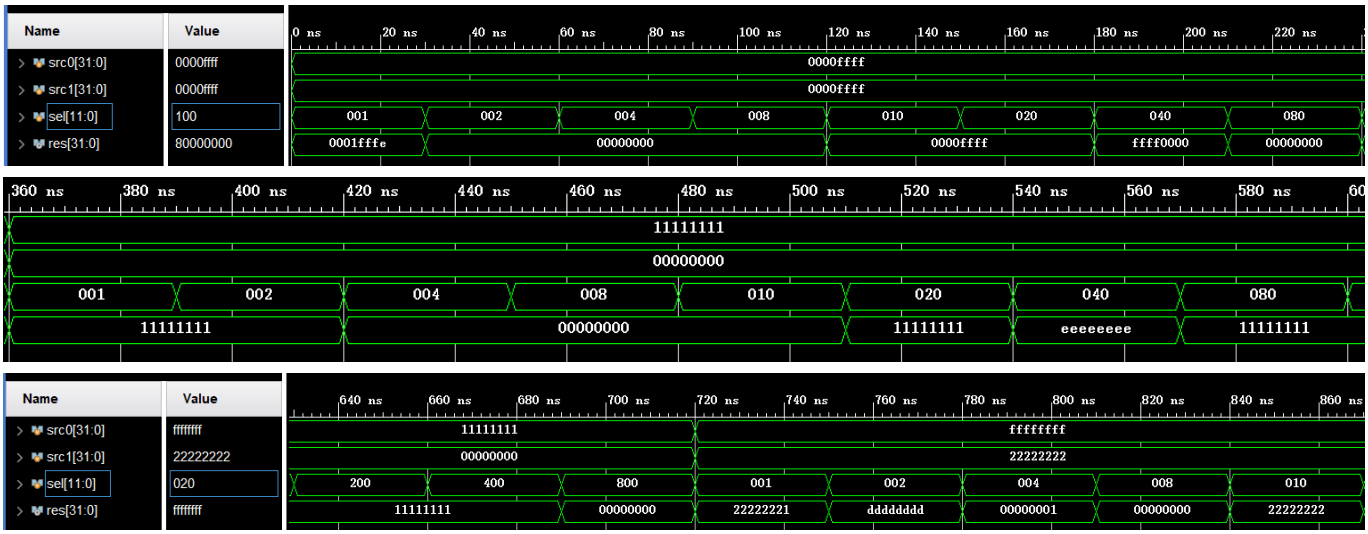
> 根据所写的代码，编写相应的仿真文件验证其正确性。

仿真文件代码如下：

```
module ALU_tb();
reg [31:0] src0,src1;
reg [11:0] sel;
wire [31:0] res;
ALU alu(
    .src0(src0),
    .src1(src1),
    .sel(sel),
    .res(res)
);
initial begin
    src0=32'hffff; src1=32'hffff; sel=12'h001;
    repeat(12) begin
        #30 sel = sel << 1;
    end
    src0=32'h11111111; src1=32'h0; sel=12'h001;
    repeat(12) begin
        #30 sel = sel << 1;
    end
    src0=32'hffffffff; src1=32'h22222222; sel=12'h001;
    repeat(12) begin
        #30 sel = sel << 1;
    end
    src0=32'hf1111111; src1=32'h22222222; sel=12'h001;
    repeat(12) begin
        #30 sel = sel << 1;
    end
end

endmodule
```

> 仿真得出的波形图如下：



> 对比分析得知程序正确。 其中有符号比较部分测试了**src0=32'hffffffff; src1=32'h22222222**这组数据，判断出有符号比较时**src0<src1**，无符号比较时**src1>src0**，由此判断结果正确。

3. 选择性必做内容：移位器

> **要求：**
>
> 自己去实现移位操作。在这里，只实现逻辑右移 SRL 和算术右移 SRA。
>
> **逻辑实现：**
>
> 这里实现移位器的三种方法

1. 枚举所有的右操作数，并通过对应的位拼接来实现；注意到右操作数的范围是 0~31，所以需要枚举 32 种情况。这里用case语句即可实现。
2. 按照右操作数 src1[4:0] 二进制的各位数字，对 src0 连续地进行 16、8、4、2、1 移位。这里用if-else语句即可实现。
3. 根据加法器的串联思路，将两种方法结合。这里用到的方法是，对src1[2:0]使用第一种方法，对于出现的8种情况，用case语句判断需要移多少位。之后用第二种方法分别判断是src1[3]和src1[4]，并相应地进行移位操作。这种方法经过了3次位拼接和多选器的延迟，使用了10个位拼接单元，得到在时间和空间上都占优的方法。

代码分为4部分：Shifter，way1, way2, way3

1.Shifter:

代码如下：

```
module Shifter(
    input                   [31 : 0]        src0,
    input                   [ 4 : 0]        src1,
    output                  [31 : 0]        res1_1,       //第一种方法，逻辑右移
    output                  [31 : 0]        res1_2,       //第二种方法，逻辑右移
    output                  [31 : 0]        res2_1,       //第一种方法，算数右移
```

```
    output                  [31 : 0]        res2_2,       //第二种方法，算数右移
    output                  [31 : 0]        res3_1,       //第一种方法，逻辑右移
    output                  [31 : 0]        res3_2        //第二种方法，逻辑右移
);
// Write your code here
way1 w1(
    .src0(src0),
    .src1(src1),
    .res1_1(res1_1),
    .res2_1(res1_2)
);

way2 w2(
    .src0(src0),
    .src1(src1),
    .res1_2(res2_1),
    .res2_2(res2_2)
);

way3 w3(
    .src0(src0),
    .src1(src1),
    .res1_3(res3_1),
    .res2_3(res3_2)
);
// End of your code
endmodule
```

2.way1:

代码如下(32种情况过长，这段代码只列举几种情况):

```
module way1(
    input                   [31 : 0]        src0,
    input                   [ 4 : 0]        src1,
    output  reg             [31 : 0]        res1_1,       //逻辑右移
    output  reg             [31 : 0]        res2_1        //算数右移
);

always @(*) begin
    case(src1)
        5'b00000 :begin
            res1_1 = {src0[31:0]};
            res2_1 = {src0[31:0]};
        end
        5'b00001 :begin
            res1_1 = {1'b0,src0[31:1]};
            res2_1 = {src0[31],src0[31:1]};
        end
        //……
        5'b11110 :begin
```

```
            res1_1 = {30'b0,src0[31:30]};
            res2_1 = {{30{src0[31]}},src0[31:30]};
        end
        5'b11111 :begin
            res1_1 = {31'b0,src0[31]};
            res2_1 = {32{src0[31]}};
        end
        default: ;
endcase
end
endmodule
```

3.way2:

代码如下:

```
module way2(
    input                   [31 : 0]        src0,
    input                   [ 4 : 0]        src1,
    output                  [31 : 0]        res1_2,       //逻辑右移
    output                  [31 : 0]        res2_2        //算数右移
);

reg [31:0] temp1_1,temp1_2,temp1_3,temp1_4,temp1_5;
reg [31:0] temp2_1,temp2_2,temp2_3,temp2_4,temp2_5;

always @(*) begin
    if(src1[4] == 1) begin
        temp1_1 = {16'b0,src0[31:16]};
        temp2_1 = {{16{src0[31]}},src0[31:16]};
    end else begin
        temp1_1 = src0[31:0];
        temp2_1 = src0[31:0];
    end

    if(src1[3] == 1) begin
        temp1_2 = {8'b0,temp1_1[31:8]};
        temp2_2 = {{8{temp2_1[31]}},temp2_1[31:8]};
    end else begin
        temp1_2 = temp1_1[31:0];
        temp2_2 = temp2_1[31:0];
    end

    if(src1[2] == 1) begin
        temp1_3 = {4'b0,temp1_2[31:4]};
        temp2_3 = {{4{temp2_2[31]}},temp2_2[31:4]};
    end else begin
        temp1_3 = temp1_2[31:0];
        temp2_3 = temp2_2[31:0];
    end
```

```
    if(src1[1] == 1) begin
        temp1_4 = {2'b0,temp1_3[31:2]};
        temp2_4 = {{2{temp2_3[31]}},temp2_3[31:2]};
    end else begin
        temp1_4 = temp1_3[31:0];
        temp2_4 = temp2_3[31:0];
    end

    if(src1[0] == 1) begin
        temp1_5 = {1'b0,temp1_4[31:1]};
        temp2_5 = {temp2_4[31],temp2_4[31:1]};
    end else begin
        temp1_5 = temp1_4[31:0];
        temp2_5 = temp2_4[31:0];
    end
end

assign res1_2 = temp1_5;
assign res2_2 = temp2_5;

endmodule
```

4.way3:

代码如下:

```
module way3(
    input                   [31 : 0]        src0,
    input                   [ 4 : 0]        src1,
    output                  [31 : 0]        res1_3,       //逻辑右移
    output                  [31 : 0]        res2_3        //算数右移
);

reg[31:0] temp1;
reg[31:0] temp2;

always @(*) begin
    case(src1[2:0])
        3'b000 :begin
            temp1 = {src0[31:0]};
            temp2 = {src0[31:0]};
        end
        3'b001 :begin
            temp1 = {1'b0,src0[31:1]};
            temp2 = {src0[31],src0[31:1]};
        end
        3'b010 :begin
            temp1 = {2'b0,src0[31:2]};
            temp2 = {{2{src0[31]}},src0[31:2]};
        end
        3'b011 :begin
```

```
            temp1 = {3'b0,src0[31:3]};
            temp2 = {{3{src0[31]}},src0[31:3]};
        end
        3'b100 :begin
            temp1 = {4'b0,src0[31:4]};
            temp2 = {{4{src0[31]}},src0[31:4]};
        end
        3'b101 :begin
            temp1 = {5'b0,src0[31:5]};
            temp2 = {{5{src0[31]}},src0[31:5]};
        end
        3'b110 :begin
            temp1 = {6'b0,src0[31:6]};
            temp2 = {{6{src0[31]}},src0[31:6]};
        end
        3'b111 :begin
            temp1 = {7'b0,src0[31:7]};
            temp2 = {{7{src0[31]}},src0[31:7]};
        end
    endcase
end

reg [31:0] temp1_1,temp1_2;
reg [31:0] temp2_1,temp2_2;

always @(*) begin
    if(src1[3] == 1) begin
        temp1_1 = {8'b0,temp1[31:8]};
        temp2_1 = {{8{temp2[31]}},temp2[31:8]};
    end else begin
        temp1_1 = temp1[31:0];
        temp2_1 = temp2[31:0];
    end

    if(src1[4] == 1) begin
        temp1_2 = {16'b0,temp1_1[31:16]};
        temp2_2 = {{16{temp2_1[31]}},temp2_1[31:16]};
    end else begin
        temp1_2 = temp1_1[31:0];
        temp2_2 = temp2_1[31:0];
    end
end

assign res1_3 = temp1_2;
assign res2_3 = temp2_2;

endmodule
```

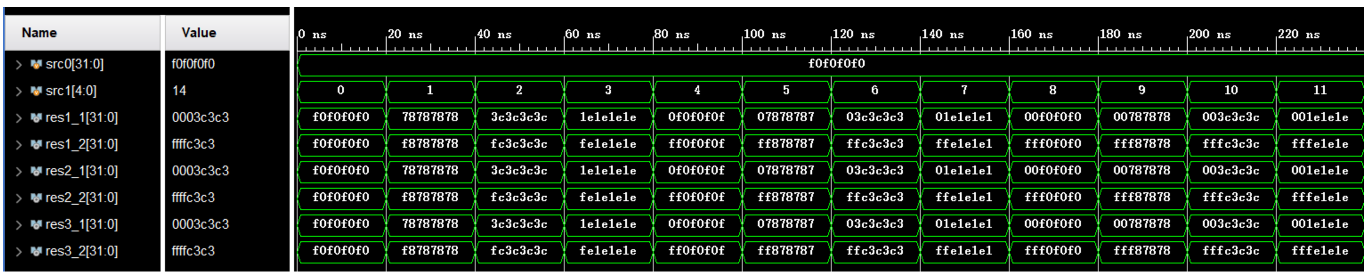
仿真结果与分析

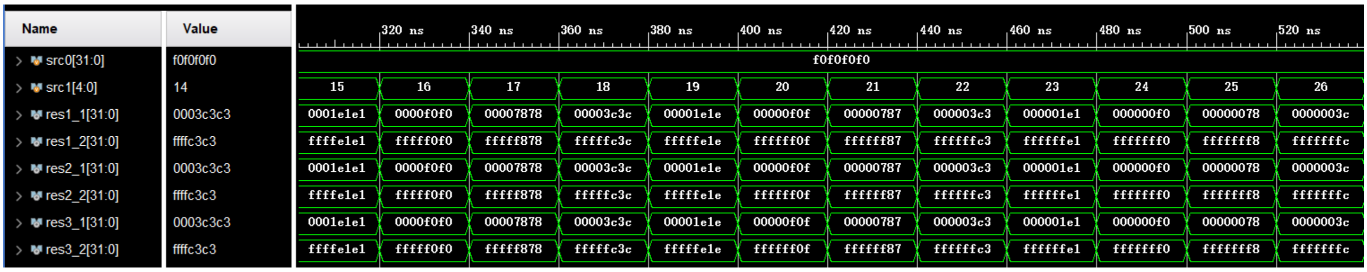
> 根据所写的代码，编写相应的仿真文件验证其正确性。

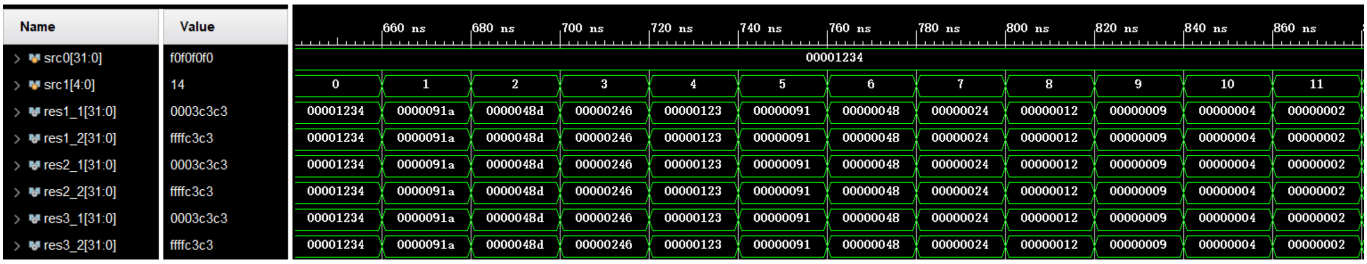
仿真文件代码如下：

```
module Shifter_tb();
reg [31:0] src0;
reg [4:0] src1;
wire [31:0] res1_1,res1_2,res2_1,res2_2,res3_1,res3_2;
Shifter shift(
    .src0(src0),
    .src1(src1),
    .res1_1(res1_1),       //第一种方法，逻辑右移
    .res1_2(res1_2),       //第二种方法，逻辑右移
    .res2_1(res2_1),       //第一种方法，算数右移
    .res2_2(res2_2),       //第二种方法，算数右移
    .res3_1(res3_1),       //第一种方法，逻辑右移
    .res3_2(res3_2)       //第二种方法，逻辑右移
);
initial begin
    src0=32'hf0f0f0f0; src1=5'b0000;
    repeat(32) begin
        #20 src1 = src1 + 1;
    end
    src0=32'h1234; src1=5'b0000;
    repeat(32) begin
        #20 src1 = src1 + 1;
    end
    src0=32'h12345678; src1=5'b0000;
    repeat(32) begin
        #20 src1 = src1 + 1;
    end
    src0=32'h11111111; src1=5'b0000;
    repeat(32) begin
        #20 src1 = src1 + 1;
    end
end
//...
endmodule
```

> 仿真得出的波形图如下：







> 观察比对得知程序正确。

总结

> 1. 本次实验，对于 Verilog 语言有了更深入的了解，同时学会利用已经学到的加法器知识实现其他运算。
> 2. 能够独立编写程序并完成仿真，并可以基本保证程序正确性。学会根据已有的方法实现时间空间上的优化节省。
> 3. 最初进行实验时，在测试ALU比较单元时，没有测试有效数据，在实验检查前发现并将算法改正过来。由此可见数据有效性典型性对于测试算法正确性的贡献很大，在今后的实验中，要大胆测试数据，学会举反例检查程序漏洞。