

目 录

第一章 实验目的.....	1
第二章 实验背景.....	2
2.1 数据的表示	2
2.1.1 原码	2
2.1.2 补码	2
2.2 定点加法.....	2
2.2.1 一位全加器	3
2.2.2 RCA.....	3
2.2.3 CLA	3
2.2.4 加法溢出	3
第三章 实验任务与要求	4
3.1 实验要求.....	4
第四章 实验结果.....	5
4.1 32 位串行加法器	5
4.1.1 设计思想	5
4.1.2 一比特全加器	5
4.1.3 32 位串行加法器	5
4.1.4 测试数据与仿真结果	7
4.2 16 位并行加法器	9
4.2.1 设计思想	9
4.2.2 4 位并行加法器	9
4.2.3 16 位并行加法器	12
4.2.4 测试数据与仿真结果	14
4.3 原反码转换	16
第五章 思考与讨论	18

5.1	课后问题.....	18
5.2	讨论	18
附	录	19

图 目 录

图 4.1	上述代码组成的一位全加器	5
图 4.2	32 位串行加法器	7
图 4.3	32 位串行加法器仿真结果	9
图 4.4	4 位并行加法器	12
图 4.5	16 位并行加法器	14
图 4.6	16 位并行加法器仿真结果	16
图 4.7	带有原反码转换的 16 位并行加法器运行结果	17

表 目 录

表 2.1 原码、反码、补码	2
----------------------	---

第一章 实验目的

1. 熟悉 LS-CPU-EXB-002 实验箱和软件平台。
2. 掌握利用该实验箱各项功能开发组成原理和体系结构实验的方法。
3. 理解并掌握加法器的原理和设计。
4. 熟悉并运用 verilog 语言进行电路设计。
5. 为后续设计 CPU 的实验打下基础。

第二章 实验背景

2.1 数据的表示

2.1.1 原码

原码, 是电脑运算的名词, 是指“未经更改”的码。为了便于 ALU 的设计, 又发展出反码、补码等转换过的码。

原码是指一个二进制数左边加上符号位后所得到的码, 且当二进制数大于 0 时, 符号位为 0; 二进制数小于 0 时, 符号位为 1; 二进制数等于 0 时, 符号位可以为 0 或 1(+0/-0)。

2.1.2 补码

补码, 是一种用二进制表示有符号数的方法, 也是一种将数字的正负号变号的方式, 常在计算机科学中使用。补码以有符号比特的二进制数定义。

正数和 0 的补码就是该数字本身再补上最高比特 0。负数的补码则是将其绝对值按位取反再加 1。补码系统的最大优点是可以在加法或减法处理中, 不需因为数字的正负而使用不同的计算方式。只要一种加法电路就可以处理各种有号数加法, 而且减法可以用一个数加上另一个数的补码来表示, 因此只要有加法电路及补码电路即可完成各种有号数加法及减法, 在电路设计上相当方便。

补码系统的 0 就只有一个表示方式, 这和反码系统不同 (在反码系统中, 0 有二种表示方式), 因此在判断数字是否为 0 时, 只要比较一次即可。

表 2.1 原码、反码、补码

数字	原码	反码	补码	数字	原码	反码	补码
0	0000	0000	0000	-1	1001	1110	1111
1	0001	0001	0001	-2	1010	1101	1110
2	0010	0010	0010	-3	1011	1100	1101
3	0011	0011	0011	-4	1100	1011	1100

2.2 定点加法

由于计算机中定点数均以补码的方式表示和存储, 采用补码表示法进行加减运算比原码方便多了, 因为不论是正还是负, 机器总是做加法, 减法运算可变成加法运算。

二进制加法规则为: $0+0=0$, $0+1=1$, $1+0=1$, $1+1=0$ (进位 1)。

2.2.1 一位全加器

由真值表得:

$$\begin{aligned} S &= A \oplus B \oplus C_{in} \\ C_{out} &= (A \wedge B) \vee (C_{in} \wedge (A \oplus B)) \end{aligned} \quad (1)$$

2.2.2 RCA

可以使用多个一位全加器来构成 N 位加法器，其中对应低位的全加器将其进位输出信号 C_{out} 连接到高一位的全加器的进入输入端 C_{in} 。这种构成多位加法器的形式被称为“波纹进位加法器”或“脉动进位加法器”（ripple-carry adder），“波纹”形象地描述了进位信号依次向前传递的情形。

波纹进位加法器的电路布局形式较为简单，设计这种电路花费时间较短。然而，波纹进位加法器的进位输出、输入所经过的路径上比其他布局方式具有较多的逻辑门，高位的计算必须等待低位的进位输出信号被计算出来才能开始，因此造成了更大的延迟时间。

2.2.3 CLA

CLA（Carry Look Ahead）超前进位加法器是一种通过预先计算进位信号的加法器，它通过预先计算进位信号来减少延迟时间。CLA 加法器的基本思想是将进位信号的计算从低位向高位推进，使得高位的进位信号不再依赖于低位的进位信号，从而减少了延迟时间。

原理: 令 $G_i = A_i \wedge B_i$, $P_i = A_i \oplus B_i$ ，则公式 (1) 可改写为:

$$\begin{aligned} G_i &= A_i \wedge B_i \\ P_i &= A_i \oplus B_i \\ C_{i+1} &= G_i \vee (P_i \wedge C_i) \end{aligned} \quad (2)$$

2.2.4 加法溢出

加法溢出是指两个正数相加得到负数，或者两个负数相加得到正数的情况。

可以使用 C_s Xor C_p 来判断是否溢出，其中 C_s 为最高位的进位输出， C_p 为最高位的进位输入。即:

$$OV = C_s \oplus C_p \quad (3)$$

第三章 实验任务与要求

1. 阅读 LS-CPU-EXB-002 实验箱相关文档, 熟悉硬件平台, 特别需要掌握利用显示屏观察特定信号的方法。学习软件平台和设计流程。
2. 熟悉计算机中加法器的原理。
3. 自行设计本次实验的方案, 画出结构框图, 详细标出输入输出端口, 本次实验的加法器可以使用全加器自己搭建加法模块, 也可以在 verilog 中直接使用 “+” (系统是自动调用库里加法 IP, 且面积时序更优), 依据教师要求选择一种方法实现。
4. 根据设计的实验方案, 使用 verilog 编写相应代码。
5. 对编写的代码进行仿真, 得到正确的波形图。
6. 将以上设计作为一个单独的模块, 设计一个外围模块去调用该模块, 见图
7. 将编写的代码进行综合布局布线, 并下载到实验箱中的 FPGA 板上进行演示。

3.1 实验要求

1. 做好预习:
 - 1) 了解软硬件平台;
 - 2) 掌握定点加法的工作原理;
 - 3) 确定定点加法的输入输出端口设计;
 - 4) 在课前画好设计框图或实验原理图;
 - 5) 如果对 FPGA 板了解的话, 可确定设计中与 FPGA 板上交互的接口, 画出包含外围模块的整体设计框图, 即补充完善图 2.1。
2. 实验实施:
 - 1) 确认定点加法的设计框图的正确性;
 - 2) 编写 verilog 代码;
 - 3) 对该模块进行仿真, 得出正确的波形, 截图作为实验报告结果一项的材料;
 - 4) 完成调用定点加法模块的外围模块的设计, 并编写代码;
 - 5) 对代码进行综合布局布线下载到实验箱里 FPGA 板上, 进行上板验证。
3. 实验检查:
 - 1) 完成上板验证后, 让指导老师或助教进行检查, 进行现场演示, 可对演示结果进行拍照作为实验报告结果一项的材料。
4. 实验报告的撰写:
 - 1) 实验结束后, 需按照规定的格式完成实验报告的撰写。

第四章 实验结果

4.1 32 位串行加法器

4.1.1 设计思想

先设计一个简单的一位全加器模块, 用 32 个全加器, 前一个的 `cout` 与下一个的 `cin` 相连, 组成一个 32 位的串行加法器。

4.1.2 一比特全加器

```

1 module bit_adder(
2     input    Bit_1 , Bit_2 ,
3     input    Cin ,
4     output   So ,
5     output   Co);
6
7     wire Xor;
8
9     assign Xor = Bit_1 ^ Bit_2;
10    assign So = Xor ^ Cin;
11    assign Co = Xor & Cin | Bit_1 & Bit_2;
12 endmodule

```

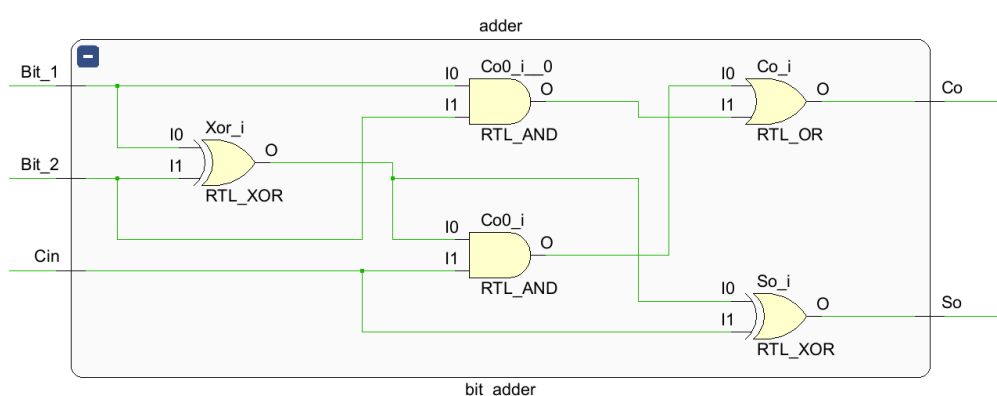


图 4.1 上述代码组成的一位全加器

4.1.3 32 位串行加法器

```
1 module full_adder1 (
2     input [31:0] Num_1, Num_2,
3     input      Cin ,
4     output [31:0] Sum,
5     output      Cout ,
6     output      OV, // overflow
7     output      ZF, // zero flag
8     output      NF, // negative flag
9     output      CF // carry flag
10 );
11
12 genvar i;
13 wire [31:0] Cout_wire;
14
15 generate
16     bit_adder adder(
17         .Bit_1 (Num_1[0]) ,
18         .Bit_2 (Num_2[0]) ,
19         .Cin (Cin) ,
20         .So (Sum[0]) ,
21         .Co (Cout_wire[0])
22     );
23     for (i = 1; i < 32; i = i + 1)
24     begin
25         bit_adder adder(
26             .Bit_1 (Num_1[i]) ,
27             .Bit_2 (Num_2[i]) ,
28             .Cin (Cout_wire[i - 1]) ,
29             .So (Sum[i]) ,
30             .Co (Cout_wire[i])
31         );
32     end
33 endgenerate
34
35 assign Cout = Cout_wire[31];
```

```

36     assign OV = Cout_wire[30] ^ Cout_wire[31];
37     assign ZF = Sum == 0;
38     assign NF = Sum[31];
39     assign CF = Cout_wire[31];
40
41 endmodule

```

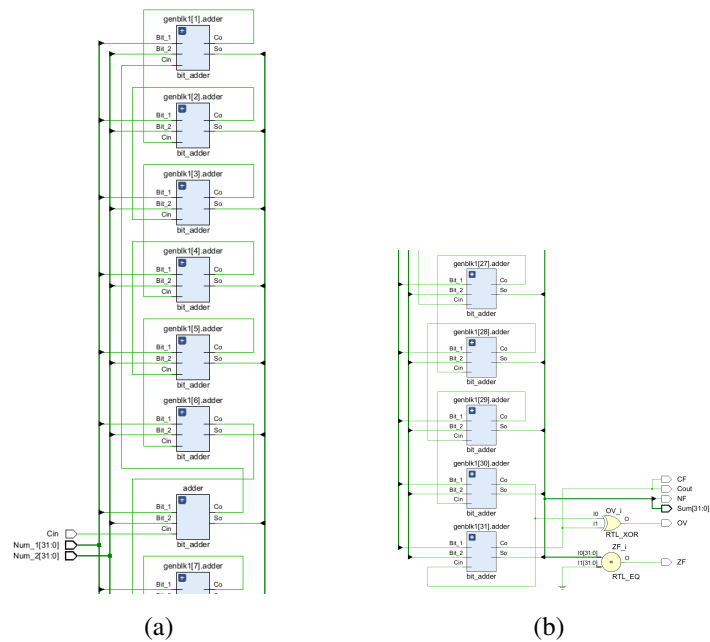


图 4.2 32 位串行加法器

4.1.4 测试数据与仿真结果

```

1 module test ;
2     // input
3     reg [31:0] operand1, operand2;
4     reg        carry_in;
5
6     // output
7     wire [31:0] sum;
8     wire        carry_out;
9     wire        overflow;
10    wire        zero_flag;
11    wire        negative_flag;
12    wire        carry_flag;

```

```
13
14
15     full_adder1 u_adder(
16         .Num_1    (operand1),
17         .Num_2    (operand2),
18         .Cin      (carry_in),
19         .Sum       (sum),
20         .Cout     (carry_out),
21         .OV        (overflow),
22         .ZF        (zero_flag),
23         .NF        (negative_flag),
24         .CF        (carry_flag)
25     );
26
27     initial begin
28         operand1 = 0;
29         operand2 = 0;
30         carry_in = 0;
31         #100;
32
33         operand1 = 1;
34         operand2 = 2;
35         carry_in = 1;
36         #100;
37
38         operand1 = 32'h80000000;
39         operand2 = 32'hfffffff0;
40         carry_in = 0;
41         #100;
42     end
43 endmodule
```

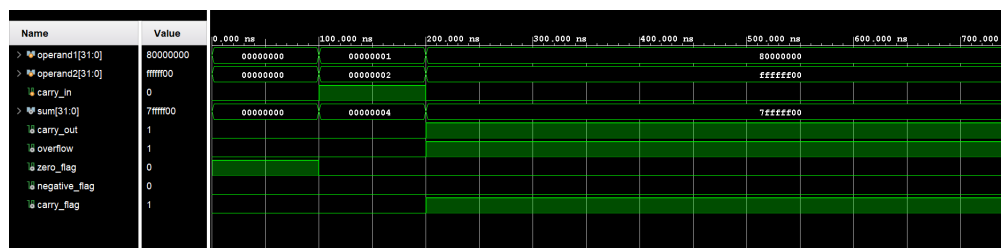


图 4.3 32 位串行加法器仿真结果

4.2 16 位并行加法器

4.2.1 设计思想

和高平老师讨论后我了解到前面的加法器是一个串行的加法器, 速度比较慢, 经过查阅书本, 我认为超前进位加法器是一个更好的选择。

基本的设计思路是: 先使用超前进位加法器设计一个 4 位的并行加法器, 再将 4 个 4 位的并行加法器串联, 组成一个 16 位的并行加法器。

4.2.2 4 位并行加法器

```

1 module one_bit_adder( // 改造的一位全加器
2     input Bit_1 , Bit_2 ,
3     input Cin ,
4     output So ,
5     output G, P
6 );
7
8     assign G = Bit_1 & Bit_2;
9     assign P = Bit_1 ^ Bit_2;
10    assign So = Cin ^ P;
11 endmodule
12
13 module four_bit_CG( // 4位进位生成器
14     input [3:0] p , g ,
15     input cin ,
16     output [4:1] co ,
17     output po , go
18 );
19

```

```
20 assign co[1] = g[0] | p[0] & cin;
21 assign co[2] = g[1] | p[1] & g[0] | p[1] & p[0] & cin;
22 assign co[3] = g[2] | p[2] & g[1] | p[2] & p[1] & g[0] | p[2] & p
    [1] & p[0] & cin;
23 assign co[4] = 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] & cin;
24
25 assign po = p[3] ^ g[3];
26 assign go = g[3];
27
28 endmodule
29
30 module four_bit_LCU_adder( // 4位超前进位加法器
31     input [3:0] input_1, input_2,
32     input cin,
33     output [3:0] sum,
34     output go,
35     output po,
36     output co
37 );
38
39 wire [3:0] p, g;
40 wire [4:1] c;
41 //wire po, go;
42
43 four_bit_CG four_bit_CG_1(
44     .p(p),
45     .g(g),
46     .cin(cin),
47     .co(c),
48     .po(po),
49     .go(go)
50 );
51
52 one_bit_adder bit_adder_1(
```

```
53     .Bit_1 ( input_1 [0] ) ,
54     .Bit_2 ( input_2 [0] ) ,
55     .Cin ( cin ) ,
56     .So ( sum [0] ) ,
57     .P ( p [0] ) ,
58     .G ( g [0] )
59 );
60
61 one_bit_adder bit_adder_2 (
62     .Bit_1 ( input_1 [1] ) ,
63     .Bit_2 ( input_2 [1] ) ,
64     .Cin ( c [1] ) ,
65     .So ( sum [1] ) ,
66     .P ( p [1] ) ,
67     .G ( g [1] )
68 );
69
70 one_bit_adder bit_adder_3 (
71     .Bit_1 ( input_1 [2] ) ,
72     .Bit_2 ( input_2 [2] ) ,
73     .Cin ( c [2] ) ,
74     .So ( sum [2] ) ,
75     .P ( p [2] ) ,
76     .G ( g [2] )
77 );
78
79 one_bit_adder bit_adder_4 (
80     .Bit_1 ( input_1 [3] ) ,
81     .Bit_2 ( input_2 [3] ) ,
82     .Cin ( c [3] ) ,
83     .So ( sum [3] ) ,
84     .P ( p [3] ) ,
85     .G ( g [3] )
86 );
87
```

```

88 assign co = c[4];
89 endmodule

```

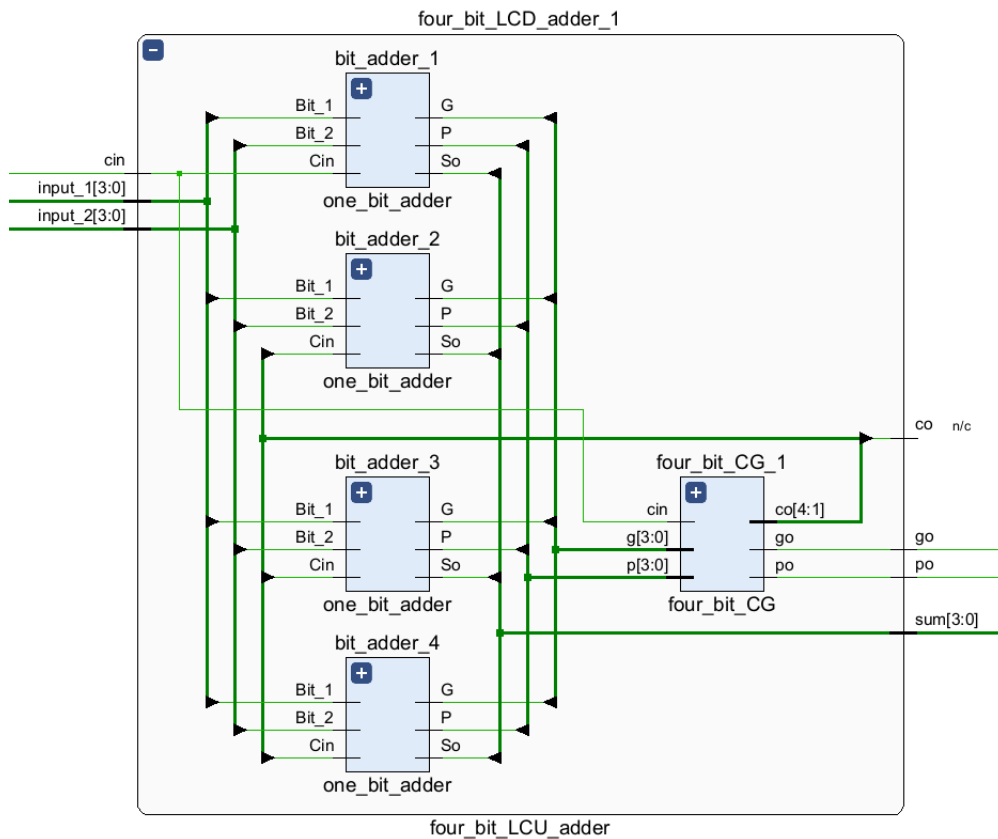


图 4.4 4 位并行加法器

4.2.3 16 位并行加法器

```

1 module sixteen_bit_full_adder(
2     input  [15:0] Num_1, Num_2,
3     input  Cin ,
4     output [15:0] Sum,
5     output go , po ,
6     output Cout ,
7     output OV, // overflow
8     output ZF, // zero flag
9     output NF, // negative flag
10    output CF // carry flag
11 );
12

```



```
13 wire [3:0] G, P, C;
14
15 four_bit_ALU four_bit_ALU_1 (
16     .p(P),
17     .g(G),
18     .cin(Cin),
19     .co(C),
20     .po(po),
21     .go(go)
22 );
23
24 four_bit_LCU_adder four_bit_LCD_adder_1 (
25     .input_1(Num_1[3:0]),
26     .input_2(Num_2[3:0]),
27     .cin(Cin),
28     .sum(Sum[3:0]),
29     .go(G[0]),
30     .po(P[0])
31 );
32
33 four_bit_LCU_adder four_bit_LCD_adder_2 (
34     .input_1(Num_1[7:4]),
35     .input_2(Num_2[7:4]),
36     .cin(G[0]),
37     .sum(Sum[7:4]),
38     .go(G[1]),
39     .po(P[1])
40 );
41
42 four_bit_LCU_adder four_bit_LCD_adder_3 (
43     .input_1(Num_1[11:8]),
44     .input_2(Num_2[11:8]),
45     .cin(G[1]),
46     .sum(Sum[11:8]),
47     .go(G[2]),
```

```

48     .po(P[2])
49 );
50
51 four_bit_LCU_adder four_bit_LCD_adder_4(
52     .input_1(Num_1[15:12]),
53     .input_2(Num_2[15:12]),
54     .cin(G[2]),
55     .sum(Sum[15:12]),
56     .go(G[3]),
57     .po(P[3])
58 );
59
60 assign Cout = C[3];
61 assign OV = C[3] ^ C[2];
62 assign ZF = Sum == 0;
63 assign NF = Sum[15];
64 assign CF = C[3];
65
66 endmodule

```

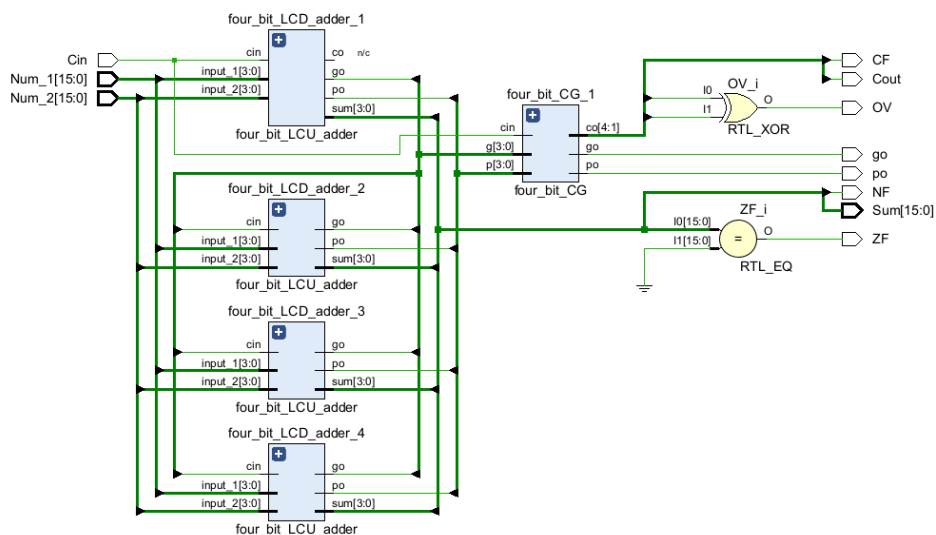


图 4.5 16 位并行加法器

4.2.4 测试数据与仿真结果

```
1 module test ;
2     // input
3     reg [15:0] operand1 , operand2 ;
4     reg        carry_in ;
5
6     // output
7     wire [15:0] sum ;
8     wire        carry_out ;
9     wire        g ;
10    wire        p ;
11    wire        overflow ;
12    wire        zero_flag ;
13    wire        negative_flag ;
14    wire        carry_flag ;
15
16    sixteen_bit_full_adder sixteen_bit_full_adder_1 (
17        .Num_1(operand1) ,
18        .Num_2(operand2) ,
19        .Cin(carry_in) ,
20        .Sum(sum) ,
21        .Cout(carry_out) ,
22        .po(p) ,
23        .go(g) ,
24        .OV(overflow) ,
25        .ZF(zero_flag) ,
26        .NF(negative_flag) ,
27        .CF(carry_flag)
28    ) ;
29
30    initial begin
31        operand1 = 0 ;
32        operand2 = 0 ;
33        carry_in = 0 ;
34        #100 ;
35
```

```

36     operand1 = 1;
37     operand2 = 2;
38     carry_in = 1;
39     #100;
40
41     operand1 = -1;
42     operand2 = -2;
43     carry_in = 0;
44     #100;
45
46     operand1 = 1000_0000_0000_0000;
47     operand2 = 1000_0000_0000_0000;
48     carry_in = 0;
49     end
50 endmodule

```

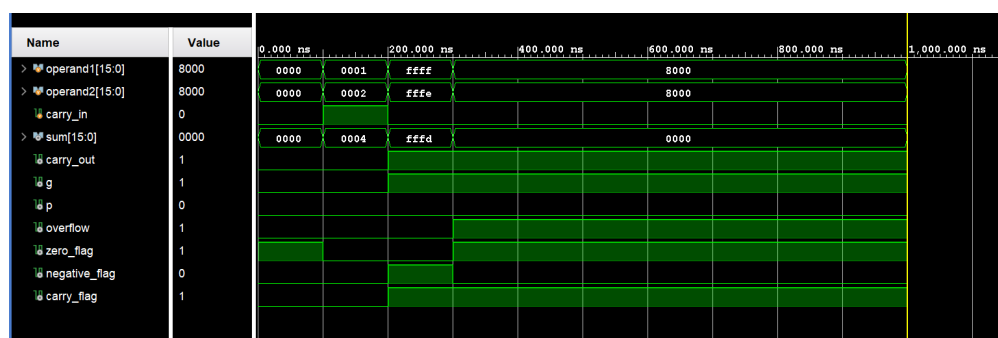


图 4.6 16 位并行加法器仿真结果

4.3 原反码转换

添加如下代码, 实现原反码转换:

```

1 module complement(
2     input  [15:0] Num,
3     output reg [15:0] Comp
4 );
5
6 always @* begin
7     //Comp = Num[15] ? ~Num + 1 : Num;

```

```

8      if (Num[15] == 1) begin
9          Comp[14:0] = ~Num[14:0];
10         Comp[15] = 1;
11         Comp = Comp + 1;
12     end
13     else begin
14         Comp = Num;
15     end
16 end
17
18 endmodule

```

在 16 位加法器前后实例化该模块, 实现原反码转换, 结果如下:

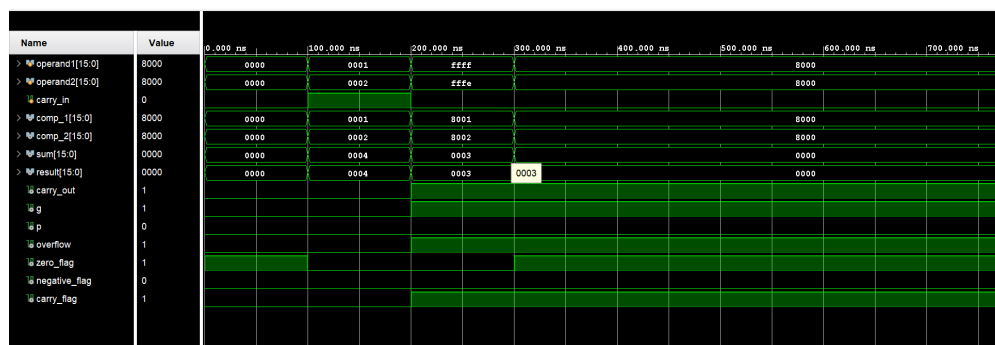


图 4.7 带有原反码转换的 16 位并行加法器运行结果

第五章 思考与讨论

5.1 课后问题

1. 答: 不论是有符号还是无符号数, 都会自动转化为补码进行运算. 如图4.6所示, 在第三次运算中, -1 转换为 ffff, -2 转换为 fffe, cin 为 0, 结果 fffd 为 -3, 与预期结果一致.
2. IP 库使用补码进行运算
3. 4 种标志位分别为: OV(溢出), ZF(零标志), NF(负数标志), CF(进位标志), 溢出通过 Cs Xor Cp 判断, 进位通过最高位的进位输出信号判断, 零标志通过判断结果是否为 0 判断, 负数标志通过最高位判断. 具体实现如代码所示.
4. 如图4.7所示, 第二次运算由于输入为正数, 原码与补码相同. 第三次输入的-1 被自动转换为补码 ffff 后又被转换了一次变为原码 8001, operand -2 同理, 最终被转化为 8002, 进入加法器进行运算后得到了 0003, 发生溢出且答案错误. 因此这里不应该手动进行原反码转换.

5.2 讨论

自己实现的 32 位串行加法器和 16 位并行加法器, 而不是简单的调用 IP 库, 我对加法器的原理有了更深的理解, 课本上学到的原理看似简单, 但在具体实现的时候总是会遇到各种问题, 通过查阅各种资料并与老师讨论, 我很开心我最终能够成功.

在这一过程中我对 verilog 这门语言也有了一些简单的了解, 对 verilog 所谓的并行有了自己的一点理解. 在我看来, verilog 与我之前学的任何一种语言都不一样, 相反, 我觉得他和 tikz 有一些相似之处. 这样说是因为我认为 verilog 所谓的并行编程实际上是在用编程语言绘制电路图, 就像使用 tikz 绘图一样, 然后运行这一电路图, 由于在运行中, 所有并行的电路会被同时接入, 产生了并行的效果, 不知道这样的理解是否正确.

当然我认为我的本次实验中也有很多不足之处, 比如我对 verilog 的理解还不够深入, 代码的风格也不够规范, 代码的复用性也不够好, 有很多地方可以改进. 也由于知识的欠缺, 我并不能准确的计算出加法器的延时.

附 录