实验三　加法器与 ALU

2022 年春季学期

"Six by nine. Forty-two."

"That's it. That's all there is."

"I always thought something was fundamentally wrong with the universe."

–"The Restaurant at the End of the Universe", Douglas Adams

加法是数字系统中最常执行的运算，加法器是 ALU（算术逻辑部件Arithmetic-Logic Unit）的核心部件。减法可以看作是被减数与取负后的减数进行加法。即用加法器同时实现加法和减法两种运算。乘法也可以利用移位相加的算法来实现。因此，加法器可以说是计算机中最 "繁忙" 的部件了。

本实验的目的是复习加法器的原理，学习用简单 ALU 的设计方式。

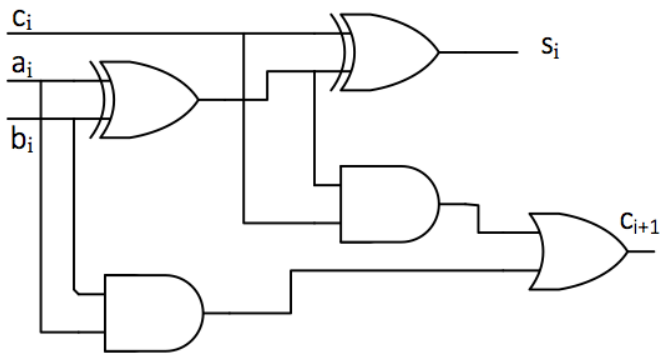
3.1 加法器

多位加法器可以由一位加法器级联而成，图 3-1(a)是一位全加器真值表，输入为 a_i、b_i 和 c_i（两个加数及进位位），输出为 s_i 和 c_{i+1}（结果和下一位进位）；图 3-1(b)是一位加法器电路图，图 3-1(d)是四位行波进位加法器框图。输入为 a（a_0-a_3）、b（b_0-b_3）和 c_{in}，输出为 s（s_0-s_3）和 c_{out}；

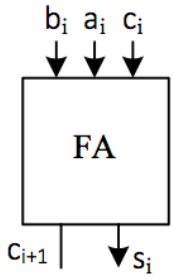
一位全加器的设计相对简单，请读者根据电路图自行思考如果设计一个串行进位加法器电路。串行加法器速度很慢，因为进位必须从最低位传至最高位。要想构建速度较快的加法器，就要利用附加逻辑，提前算出进位信息，这就是先行进位加法器的设计思想，先行进位加法有几种常用的算法，感兴趣的同学可以查找资料阅读。

c_i	a_i	b_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

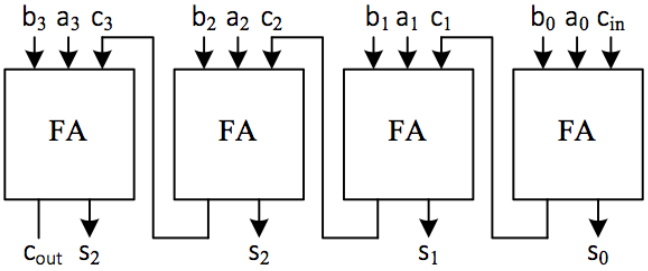
(a) 一位全加器真值表



(b) 一位全加器电路图



(c) 一位全加器框图



(d) 四位串行全加器

图 3-1: 全加器

3.2 八位加法器实现

上述四位串行加法器的设计，如果推广到更多位加法器的设计，程序代码将会变得异常复杂，运行效率也将很低。在 Verilog 语言中，可以使用算术赋值语句和向量来执行这种算术运算。如果我们定义如下向量：

```
input  [n-1:0]  in_x, in_y;
output [n-1:0]  out_s;
```

则算术赋值语句

```
out_s = in_x + in_y;
```

就可以实现 n 位加法器了。

请注意，该代码定义了可以生成 n 位加法器的电路，但是该加法器电路并不包含加法过程中产生的进位输出信号和算术溢出信号。

溢出信号是用于判断运算结果 out_s 是否正确的信号。可以证明，对于有符号的补码运算的算术溢出信号可以用下面的表达式得到：

$$Overflow = (in\_x_{n-1} == in\_y_{n-1})\&\&(out\_s_{n-1}! = in\_x_{n-1}) \tag{3-1}$$

其判断原理是：如果两个参加加法运算的变量符号相同，而运算结果的符号与其不相同，则运算结果不准确，产生溢出。即两个正数相加结果为负数，或者两个负数相加结果为正数，则发生了溢出。一正一负两个数相加是不会产生溢出的。当然，还有其他的判断溢出位的方式，请大家参照相关资料，了解其他判断运算是否溢出的方法。

对于进位信号可以用下面的表达式得到：

```
1 {out_c,out_s} =in_x + in_y;
```

此表达式执行后，out_c 即为进位位，out_s 即为加法运算结果，这里的进位位仅用于表示在加法运算过程中，操作数的最高位是否对外有进位，和 X86 体系中借位 CF 的概念在减法操作中是相反的，即 X86 中的 $CF = out_c \oplus cin$，其中 cin 在减法时置 1。在有符号的加减法中，溢出判断依据为溢出位，进位位不用。而在无符号数的加减法中，溢出判断依据进位位，溢出位不用。

对于加法的进位位是有公认的定义的，但是对于减法的进位方式有两种截然不同的定义方法，具体可以参考https://en.wikipedia.org/wiki/Carry_flag。在 ICS 课程中介绍的 X86 处理器中，减法的进位是以借位方式实现的。

3.3 简单加减法运算器的设计

上述加法器只能实现加法运算，如果要执行减法运算，必须再设计一个减法运算器。其实，在实际的运算器中，如果参加运算的操作数都是补码的话，可以用加法器同时实现加法和减法运算。

图 3-2是一个 32 位加减法器的参考逻辑图，输入信号有：两个 32 位的参与运算的数据的补码操作数 A 和操作数 B，一个控制做加法还是做减法的加/减控制端 Sub/Add，为 1 时进行减法运算。输出信号有：一个 32 位的结果 Result、一位进位位，一位溢出位和一位判断结果是否为零的输出位。

在图中已经加入了对减数进行求补操作，将减法变成加法的过程。即：如 A －B，可以先求出（－B）的值，然后再和 A 相加，即 A －B = A +（－B）。

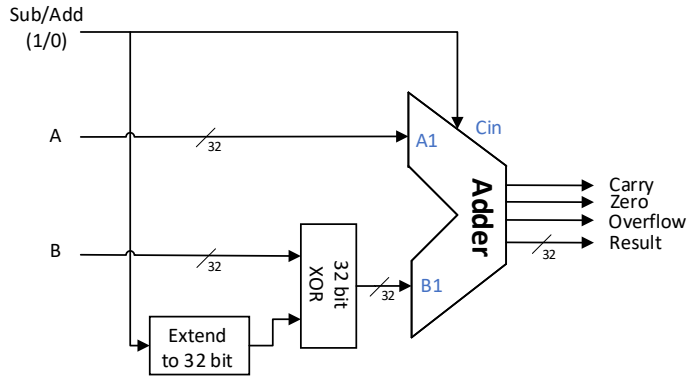


图 3-2: 简单加减 ALU

对于补码而言，－B 的补码就是将 B 连同符号位在内全部取反再加一的过程。

请查找资料，自行设计具有加减功能的加法器。

☞ **减法 Overflow**

虽然减法也是利用加法器实现的，但减法运算在减数是最小负数时溢出判断需要特殊处理。考虑以下两种实现，那一种是正确的?

方法一:

```
assign t_no_Cin = {n{ Cin }}^B ;
assign {Carry,Result} = A + t_no_Cin + Cin;
assign Overflow = (A[n-1] == t_no_Cin[n-1]) && (Result [n-1] != A[n-1]);
```

方法二:

```
assign t_add_Cin =( {n{Cin}}^B )+ Cin ;  //  在这里请注意^运算和+运算的顺序
assign { Carry, Result } = A + t_add_Cin;
assign Overflow = (A[n-1] == t_add_Cin[n-1]) && (Result [n-1] != A[n-1]);
```

☞ **Zero 输出**

在判断输出结果是否为零的时候有两种判断方式，一种是用 if 语句，将 Result 和 “0” 相比较，这样在硬件上会产生一个比较器。还可以使用如下语句:

```
assign zero = ~(| Result);
```

“| Result” 操作称为一元约简运算，这个运算在硬件上几个逻辑门就可以实现了，请查阅 Verilog 相关语法资料，了解此运算的操作过程。

选择你认为好的方式来进行结果是否为 “0” 的判断。

☞ **关于加法器测试值的设计**

在本实验的测试过程中，操作数比较多，位数也比较长，如果使用枚举的方式对每一个值都进行测试，当然测试得非常充分，但是，测试结果过长，阅读不方便而且也没有完全的必要。因此在测试的时候，选取一些关键、边界和具有典型意义的值很重要。例如这里我们要求设计一个 4 位的补码加减法器，输入和输出全部是补码，操作数的取值范围为-8 到 +7，我们可以对两个操作数进行如下取值：7，6，2，1，0，-1，-2，-7，-8 等。观察仿真结果，然后根据需要再添加其他的测试值。在测试加法器中，可以利用 test bench 中的 task 功能来进行自动测试。注意：task 主要用于测试中对结果进行计算和验证，不建议在实现电路时使用 task。

```
task check;  //测试任务
  input [3:0] results;  //ALU的结果预期正确输出
  input resultof, resultc, resultz;  //ALU的预期溢出，进位，零位
  begin
    if(outputs!=results)  //比较预期结果和测试单元输出的outputs
      begin                //出错时显示
              $display("Error:x=%h,y=%h,ctrl=%b,s should be %h,
                 get %h", inputa, inputb, inputaluop, results, outputs);
      end
    //自行添加溢出，进位和零位的比较
  end
endtask
```

在测试时，可以调用 check 任务来判断结果是否正确。

```
for(i=-8;i<=7;i=i+1) //建议i和j可以是5位以上的带符号数
  for (j=-8;j<=7;j=j+1)
    begin
       inputa=i;
       inputb=j;  //设置两个输入
       inputaluop=4'b0000; //ALU的操作码
       k = * ;     //此处自行计算正确的输出，填入*处
       of= * ;     //可分不同情况手工填写
       z = * ;
       c = * ;
       #20 check(k[3:0],of,c,z);
    end
```

3.4 ALU 设计

在 CPU 中，ALU 的功能除了加减法运算之外，往往还包含逻辑运算、移位、乘除法、比较大小等等。我们这里按照 RISC-V 中基础指令集 RV32I 的 ALU 的设计要求来进行介绍。

RISC-V 基础指令集 RV32I 只支持 32 位整型数值的操作。操作数可以是带符号补码整数或无符号数。ALU 不需要完成乘除法，不需要进行溢出判断，相关操作由软件来完成。RV32I 的 ALU 需要完成以下操作：

- **加减法操作**：完成带符号补码数和无符号数的加减法操作，无需判断溢出和进位。此条件下，可以统一处理带符号数和无符号数。
- **逻辑运算**：完成 XOR、AND 及 OR 操作，其他操作采用软件实现。例如，NOT 可以使用与全零操作数进行 XOR 实现。
- **移位运算**：完成逻辑左移，逻辑右移、算术右移等功能。移位运算将在后面的移位寄存器实验中介绍。
- **比较运算**：完成带符号数与无符号数的全等和大小比较。此类运算均可利用减法实现。

  全等可以用减法 Zero 输出判断。

  带符号数的大小比较，可以用减法比较，即比较 A、B 两数大小时，首先 B 取反加一，然后与 A 相加。在不溢出时，结果的符号位为 1 则 A 小于 B。如果减法溢出，则 A 和 B 原始符号一定不同。此时，如果结果符号位为 0，说明 A 为负数，B 为正数，B 取反加一后为负，两者相加为正，所以 A 应小于 B。在溢出时如果结果符号位为 1，则 B 小于等于 A。所以，可以用 $Less_s = out\_s_{n-1} \oplus Overflow$ 来进行判断。具体原因请自行分析。

  无符号数大小比较也可以用减法判断。此时，如果最高位进位，则 A 大于或等于 B，否则 A 小于 B。实际电路中常常用 $Less_u = cin \oplus Carry$ 来实现。具体原因请自行分析。

3.5 实验验收内容

3.5.1 上板实验

实现一个带有逻辑运算的简单 ALU

设计一个能实现如下功能的 4 位带符号位的**补码** ALU：

表 3-1: ALU 功能列表

功能选择	功能	操作
000	加法	A+B
001	减法	A-B
010	取反	Not A
011	与	A and B
100	或	A or B
101	异或	A xor B
110	比较大小	If A<B then out=1; else out=0;
111	判断相等	If A==B then out=1; else out=0;

ALU 进行加减运算时，需要能够判断结果是否为 0，是否溢出，是否有进位等。这里，输入的操作数 A 和 B 都已经是补码。比较大小请按带符号数的方式设置。

执行逻辑操作时不需要考虑 overflow 和溢出。

由于开发板上输入有限，可以使用 SW 作为数据输入，button 作为选择端。

3.5.2 在线测试

必做 四位补码加减法器

必做 简单四位 ALU