计算机系统 I 实验报告 2

实验 2-1:64 位全加减法器的实现

姓 名: 洪 奕 迅

学 号: 3230102930

班 号: 工信 2319

计算机系统 I

(春夏, 2024)

浙江大学

计算机学院

2024 年 4 月 8 日

目录

1 64 位全加减法器的实现

1.1 电路代码设计

1.1.1 1-bit 全加器

Theorem 1.1.1 ▶ 1-bit 全加器的电路逻辑

在不考虑进位和运算溢出的情况下，对于输入 a, b, c_{in} 的和可以被直接表示为 $a \oplus b \oplus c_{in}$。
同时不妨参照超前进位加法器的设计，记 $P = ab$, $G = a \cdot b$，则进位 $c_{out} = P \cdot c_{in} + G$，即进位发生在输入全真或一真一假但进位为真时。

基于这一电路逻辑，设计了如下的 *Adder* 模块来实现 1-bit 加法器。

Code Snippet 1.1.2 ▶ src/lab2-1/adder.v

```
module Adder(
    input a,
    input b,
    input c_in,
    output s,
    output c_out
);
    assign s = a^b^c_in;
    assign c_out = ((a^b)&c_in) | (a&b);
endmodule
```

1.1.2 可变位数行波进位加法器

只需要将 $c_{i\ in}$ 作为 $c_{i+1\ out}$ 即可，同时利用 parameter 定义一个参数实现可变长度，以下为实现代码：

Code Snippet 1.1.3 ▶ src/lab2-1/adders.v

```
module Adders #(
    parameter LENGTH = 32
)(
```

```
    input [LENGTH-1:0] a,
    input [LENGTH-1:0] b,
    input c_in,
    output [LENGTH-1:0] s,
    output c_out
);

    wire [LENGTH:0] carry;
    assign carry[0]=c_in;

    genvar i;
    generate
        for(i=0;i<LENGTH;i=i+1)begin
            Adder adder0(.a(a[i]), .b(b[i]), .c_in(carry[i]), .s(s[i]),
             ↪  .c_out(carry[i+1]));
        end
    endgenerate

    assign c_out=carry[LENGTH];

endmodule
```

1.1.3 全加减法器

Theorem 1.1.4 ▶ n-bit 加减法逻辑

首先考虑文档中提到的 “n-bit 下的加减法事实上是 mod 2^N 下的加减法”。对于 $a, b \leq 2^N - 1$，注意到

$$b \equiv (2^N + b) \bmod 2^N$$

故对于任意 $b \in [-(2^N - 1), 2^N + 1]$，有

$$a + b \equiv a + (2^N + b) \bmod 2^N$$

对于正数情况下，电路特性保证了 $+2^N$ 被直接取模，而对于负数，需要利用逻辑门加以处理，从而避免幂运算带来的不便性，注意到：

$$a \oplus \underbrace{11\ \ldots 1}_{N} = 2^N - 1 - a$$

$$a \oplus \underbrace{00\ldots0}_{N} = a$$

将其带入上式子的 b，其中 $b < 0$ 说明进行的是减法，得到以下式子：

$$a + b \equiv \begin{cases} a + (b \oplus \underbrace{00\ldots0}_{N}) + 0 & b > 0 \\ a + (b \oplus \underbrace{11\ldots1}_{N}) + 1 & b < 0 \end{cases}$$

因此引入 *do_sub* 变量来替代上式的 0/1 变得顺理成章，以下为行波进位加法器的实现代码：

Code Snippet 1.1.5 ▶ src/lab2-1/add-subs.v

```
module AddSubers #(
    parameter LENGTH = 32
)(
    input [LENGTH-1:0] a,
    input [LENGTH-1:0] b,
    input do_sub,
    output [LENGTH-1:0] s,
    output c
);
    wire [LENGTH-1:0] temp;
    assign temp = {LENGTH{do_sub}};
    Adders #(
        .LENGTH(LENGTH)
    ) adders1(
        .a(a),
        .b(b[LENGTH-1:0]^temp),
        .c_in(do_sub),
        .s(s),
        .c_out(c)
    );
endmodule
```

1.2 基于 for 语句和随机数的仿真设计

以下是 *testbench* 的随机数产生代码，使用了 for 循环，为了检测代码在加减法不同情况下的正确性，这里设置了 10 组随机数加法样例和 10 组随机数减法样例：

Code Snippet 1.2.1 ▶ src/lab1-2/sim/testbench.v

```
```
integer i;
    initial begin
        integer i;
        for(i=0;i<10;i=i+1)begin
            a=$random();
            b=$random();
            do_sub=1'b0;
            #20;
        end
        for(i=0;i<10;i=i+1)begin
            a=$random();
            b=$random();
            do_sub=1'b1;
            #20;
        end
        $finish;
    end
```
```

分析对拍器可知，在我们的模块运算结果与对拍器不符时，会输出 error 信息，以下为终端运行界面，没有 error 可以说明我们的全加减法器的正确性：

```
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include -I/usr/local/shar
e/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST
=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-
overloaded-virtual -Wno-shadow -Wno-sign-compare -Wno-uninitialized -Wno-unused-
but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused
-variable    -DVL_DEBUG -DTOP=Testbench -std=c++17 -DVL_TIME_CONTEXT   -fcorouti
nes -c -o VTestbench__ALL.o VTestbench__ALL.cpp
echo "" > VTestbench__ALL.verilator_deplist.tmp
Archive ar -rcs VTestbench__ALL.a VTestbench__ALL.o
g++    verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o VTes
tbench__ALL.a    -pthread -lpthread -latomic   -o Testbench
rm VTestbench__ALL.verilator_deplist.tmp
make[1]: Leaving directory '/home/forever/sys1-sp24/src/lab2-1/build/verilate'
cd /home/forever/sys1-sp24/src/lab2-1/build/verilate; ./Testbench
- /home/forever/sys1-sp24/src/lab2-1/sim/testbench.v:24: Verilog $finish
forever@forever:~/sys1-sp24/src/lab2-1$
```

图 1.1: 仿真过程终端截图

1.3 综合实现全加减法器

1.3.1 Vivado 代码综合

在前面，我们已经得到了全加减法器 *add_subs.v*，而其余的约束文件都在仓库中给出，只需把他们全部加入 Vivado 后进行综合烧录即可。

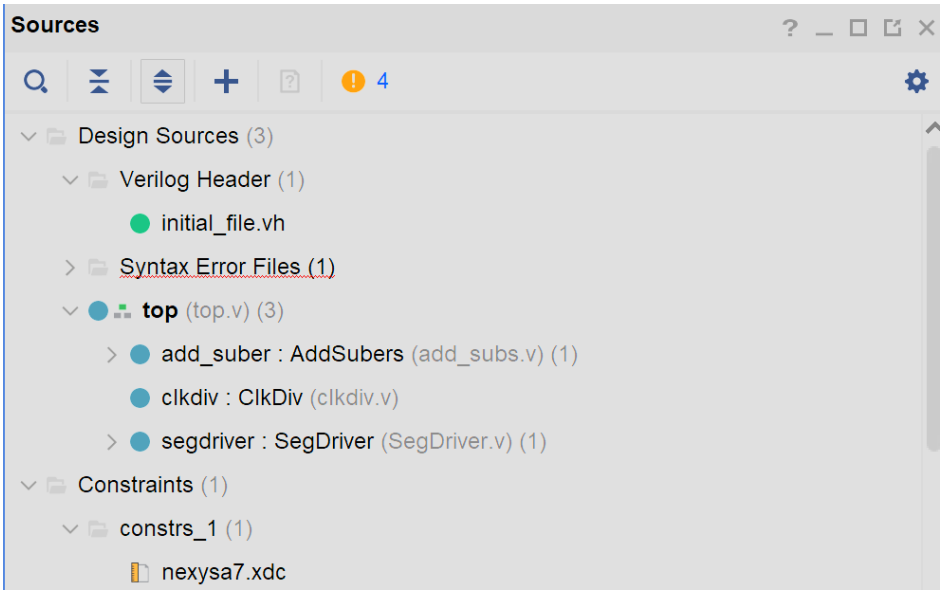

图 1.2: 综合文件

1.3.2 上板验证

文档的说明还是略为迷惑，这里根据 *top.v* 来分析实际上 FPGA 开关的功能：

Code Snippet 1.3.1 ▶ repo/sys-project/lab2-1/syn/top.v

```
```
  assign a=adder[{sw[1:0],1'b0}];
  assign b=adder[{sw[1:0],1'b1}];
```
  wire [31:0] display [7:0];
  assign display[0]=a[31:0];
  assign display[1]=a[63:32];
  assign display[2]=b[31:0];
  assign display[3]=b[63:32];
  assign display[4]=s[31:0];
  assign display[5]=s[63:32];
```

```
  assign display[6]={63'b0,c};
  assign display[7]=64'b0;
```
```
  wire [31:0] display_data;
  assign display_data=display[sw[5:3]];
```

分析后注意到 $SW[0]$，$SW[1]$ 的功能是选择测试样例，其中 $\{SW[1]SW[0],1'b0\}_2 = \{SW[1]SW[0]\}_2 \times 2$，$\{SW[1]SW[0],1'b1\}_2 = \{SW[1]SW[0]\}_2 \times 2 + 1$，以此从 adder 中选择数据，每两个一组，偶数下标为被减数，奇数为减数。此外 $SW[3]-SW[5]$ 的功能是根据二进制串 $\{SW[5]SW[4]SW[3]\}_2$ 的数值选择子串显示到数码管，具体为：0,2,4/1,3,5-a,b,s 低/高位，6-占位符 0 和最后一位进位输出，7-占位符。根据这一设定，以下是对于样例一的加法上板结果。 这组样例进位结果为 0。

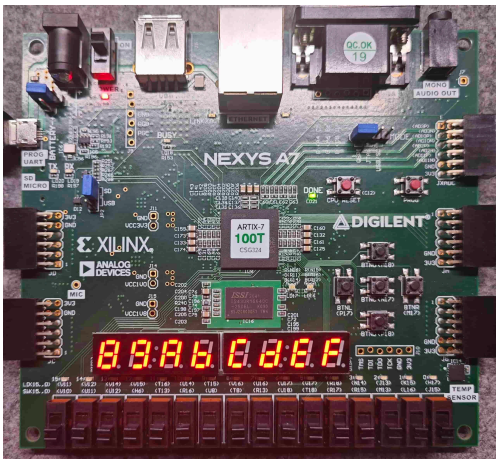

图 1.3: a 低位

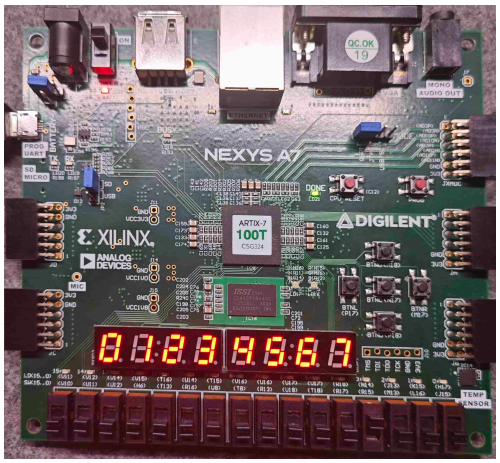

图 1.4: a 高位

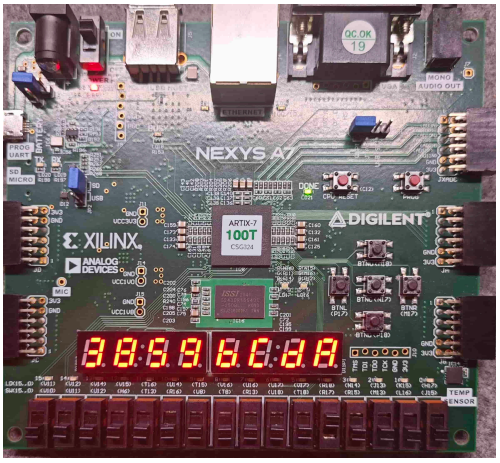

图 1.5: b 低位

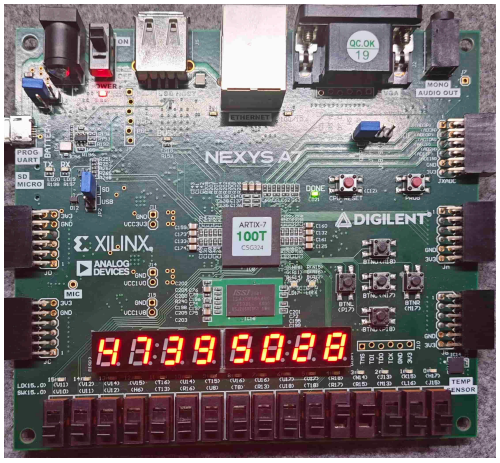

图 1.6: b 高位

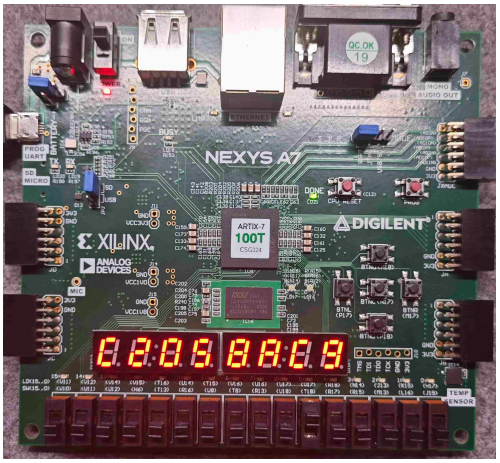

图 1.7: s 低位

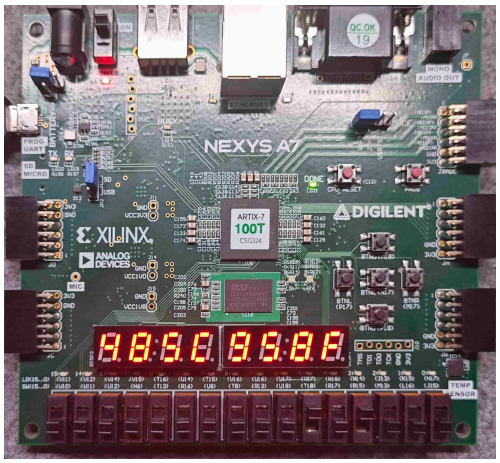

图 1.8: s 高位

1.4 adder 下标顺序对 $*readmemh* 初始化的影响

1.4.1 改变下标顺序的影响

首先需要搞清楚 [63 : 0]*adder*[7 : 0] 到底是什么意思。这其实由 [63 : 0]*adder*, *adder*[7 : 0] 组成，前者是 Verilog 中的线组向量，后者则是数组，因此在赋值时向量会按照其固定顺序被赋值，而数组下标则按照自然数序赋值。因此：

- 63/0 → 0/63：该数组中每一向量原本会按照 63 : bit_0, 62 : $bit_1 \cdots 0$: bit_63 的顺序赋值，而在更改后则变为 63 : bit_63, 62 : $bit_62 \cdots 0$: bit_0。
- 7/0 → 0/7：在变量后的数组下标的声明顺序并不会遵循其被声明的顺序，总是按自然数序赋值
- 63 → 127：数据被填入低位，空缺高位被 0 补齐，以下是使用 Vivado 仿真得到的波形图：

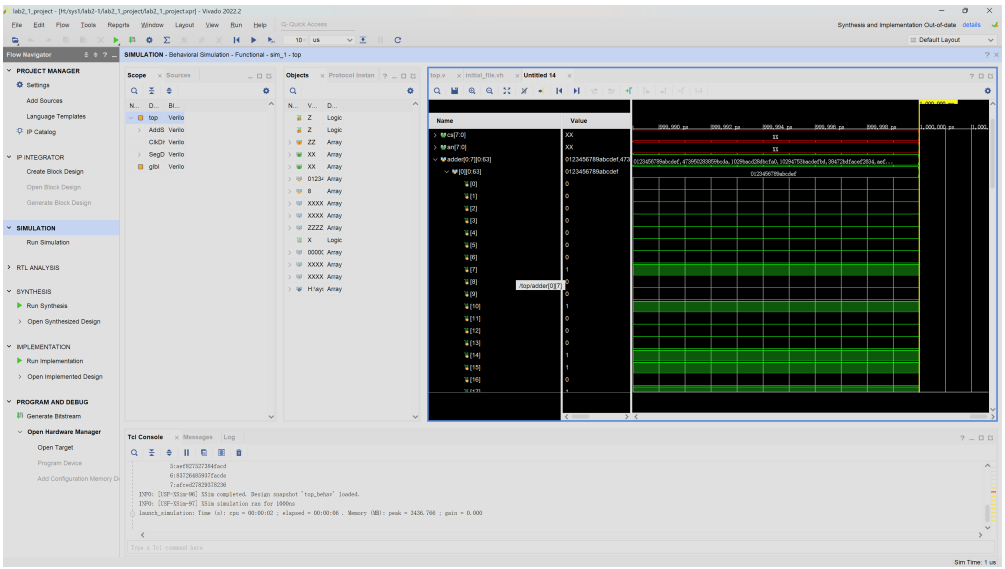

图 1.9: 模拟一

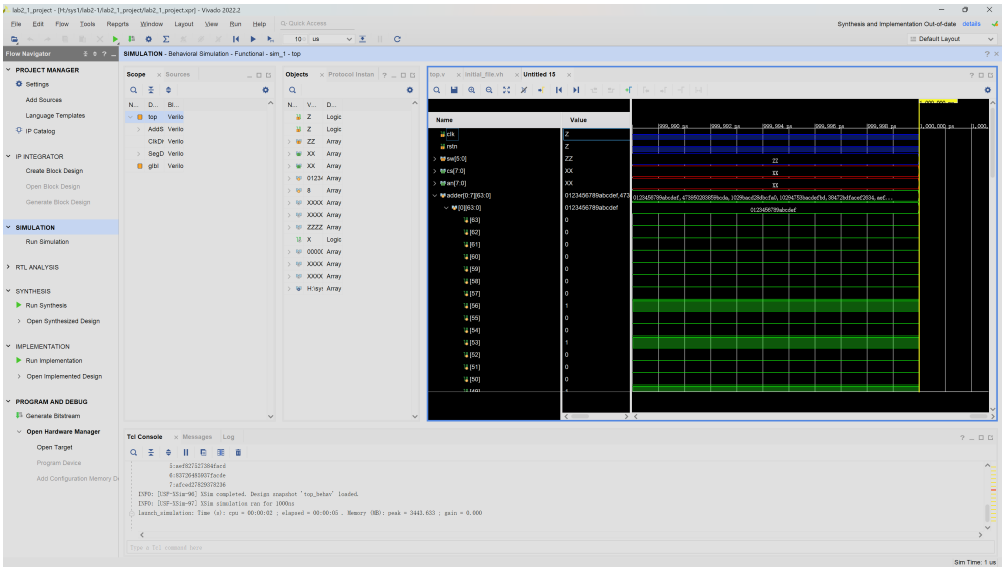

图 1.10: 模拟二

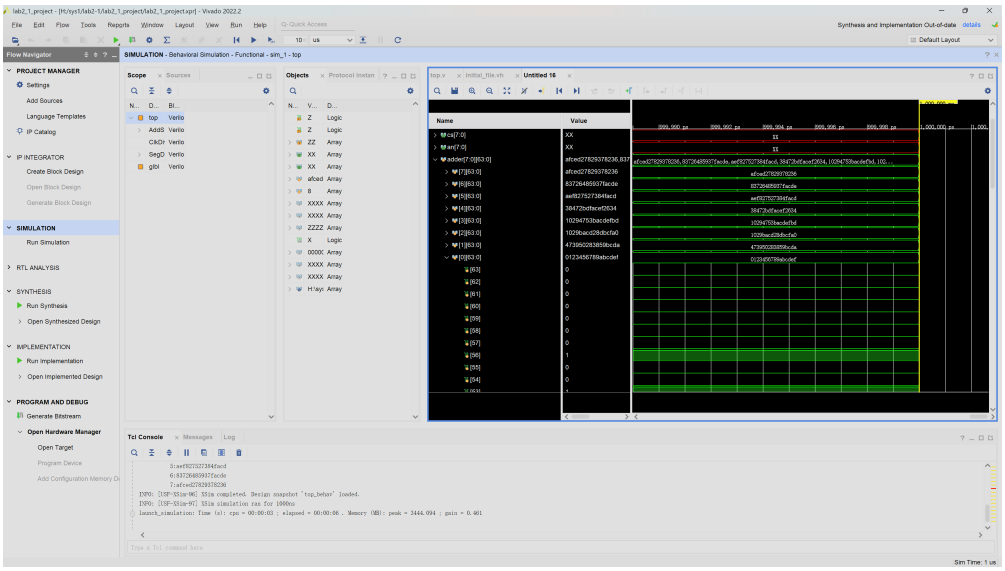

图 1.11: 模拟三

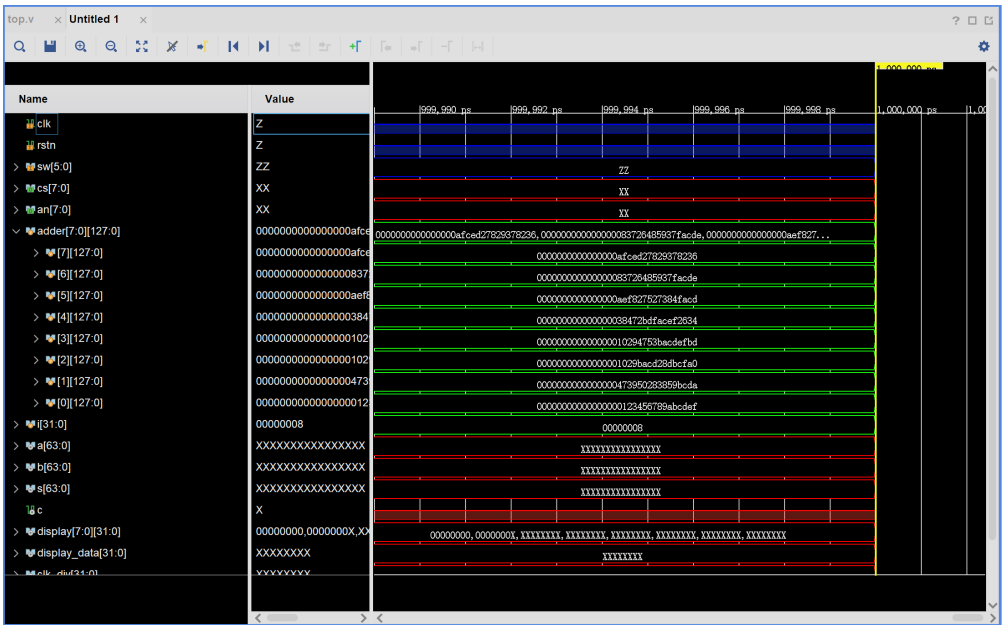

图 1.12: 模拟四

1.4.2 $*readmemh* 怎么初始化数组

$*readmemh* 从文件中读取 **十六进制** 的数据，其函数写法为 $*readmemh*("*PATH*", *variable*, (*start*, *end*))，在不直接给出起始和结束地址时，函数会按照十六进制文件中的顺序存放顺序将十六进制数据逐位存入数组，若不指定地址或在十六进制文件中设置地址跳转，将默认按照数组本身顺序存放，而这一顺序在前面已经提到过，而对于位宽不足，如 [127:0] 这种情况，将用 0 填补缺失位（查阅资料了解到会用未知阻态 x 填补空位，但是 Vivado 中确实显示是 0。

1.5 超前进位和行波进位加法器的优缺点分析

1.5.1 行波进位加法器

对于行波进位加法器，有以下优缺点：

- 优点：
  1. 行波进位加法器本身结构较为简单，而且每一级原理完全一致，因此在逻辑设计上较为简单。
  2. 由于每级相同的原理，行波进位加法器的逻辑电路只需要不断重复每一级的门电路构成，因此电路复杂度是线性的，并且需要的门电路数量相当有限。可以实现对电路元件数量和成本的控制。
  3. 在位数较少的情况下，由于传递的延迟并不明显，因此速度同超前进位加法器没有显著区别。
- 缺点：
  1. 相比于超前进位加法器，行波进位加法器后一级的进位输入依赖于前级的进位输出，因此行波进位加法器在大规模的加减法运算上，存在极为明显的运算延迟，同时计算速度也相应的比较慢。

1.5.2 超前进位加法器

对于超清进位加法器，有以下优缺点：

- 优点：
  1. 超前进位加法器的进位是直接逻辑运算的结果，因此不存在行波进位加法器那样依赖前级的延迟，运算速度取决于逻辑电路本身的复杂性，适用于大规模加法运算，性能较好。
- 缺点：
  1. 由引入 P_i, G_i 的逻辑表达式可知，超前进位类似于在逻辑上列举了一切产生结果的可能，因此在高位上，超前进位加法器的逻辑电路会变得极为复杂，这本身会有一定的逻辑延迟，但仍然小于行波进位加法器的延迟。
  2. 这种复杂的逻辑电路的结果就是电路硬件成本和规模都相应升高。

1.6 运算溢出的布尔代数表示

1.6.1 无符号数

由于 Lab 里用的似乎都是无符号数，所以这里讨论无符号数的溢出。

- 对于无符号数加法，溢出只需观察最高位是否发生了进位（对于减法是 mod 2^N 后的“进位”），即：

$$c \to OVERFLOW$$

- 对于不溢出的无符号数减法，对于 $Theorem$1.1.4 中的运算结果事实上总是 $\geq 2^N$，这是因为在 $a \geq b$ 的情况下运算才不会溢出，而此时：

$$b \oplus \underbrace{11\cdots1}_{N} \geq a \oplus \underbrace{11\cdots1}_{N}$$

$$RHS \geq a + (a \oplus \underbrace{11\cdots1}_{N}) + 1 = 2^N$$

因此仍然只需判断最高位是否发生进位，但在无符号减法中，没有发生进位才代表了运算溢出，即：

$$\neg c \to OVERFLOW$$

因此对于在代码中使用的 c 参数也有了意义，这事实上代表了运算溢出标识符，对于加法 1 代表预运算溢出，而对于减法，0 代表运算溢出。