四位加法器实验报告

翁家翌 2016011446

2018.4.28

1 实验目的

1. 掌握组合逻辑电路的基本分析和设计方法。
2. 理解半加器和全加器的工作原理并掌握利用全加器构成不同字长的加法器的各种方法。
3. 学习元件例化的方式进行硬件电路设计。
4. 学会利用软件仿真实现对数字逻辑电路的逻辑功能进行验证和分析。

2 实验内容

1. 设计实现逐次进位加法器，进行软件仿真并在实验平台上进行测试。
2. 设计实现超前进位加法器，进行软件仿真并在实验平台上进行测试。
3. 使用 VHDL 自带的加法运算实现一个四位加法器。

3 代码及注释

3.1 1 位加法器元件定义

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity fp1 is
7          port(
8                  a,b,cin:in std_logic;
9                  s,cout:out std_logic;
10                 p,g:buffer std_logic
11         );
12 end fp1;
13
```

```
architecture plus of fp1 is
begin
        process(a,b)
        begin
                p<=a xor b;
                g<=a and b;
        end process;
        process(cin,p,g)
        begin
                s<=p xor cin;
                cout<=g or (cin and p);
        end process;
end plus;
```

工作原理：实现 1 位加法器元件，传入两个加数 a、b 和输入进位信号 c_{in}，传出结果 s、输出进位信号 c_{out}，p 和 g 是通过 a 和 b 计算出的中间变量，会用于 s 和 c_{out} 的计算，故设 p 和 g 为 buffer，根据书上公式求得所有输出和 buffer 的数值。

3.2 逐次进位加法器

```
entity fp4 is
        port(
                a,b:in std_logic_vector(3 downto 0);
                cin:in std_logic;
                s:out std_logic_vector(3 downto 0);
                cout:out std_logic
        );
end fp4;
architecture successive of fp4 is
        component fp1
                port(
                        a,b,cin:in std_logic;
                        s,cout:out std_logic;
                        p,g:buffer std_logic
                );
        end component;
        signal p,g,c:std_logic_vector(3 downto 0);
begin
        fa0:fp1 port map(a(0),b(0),cin,s(0),c(0),p(0),g(0));
        fa1:fp1 port map(a(1),b(1),c(0),s(1),c(1),p(1),g(1));
        fa2:fp1 port map(a(2),b(2),c(1),s(2),c(2),p(2),g(2));
        fa3:fp1 port map(a(3),b(3),c(2),s(3),cout,p(3),g(3));
end successive;
```

工作原理：使用元件例化，并按照级联的方式构造四位加法器，即在求出前一位进位之后进行这一位的计算。

3.3 超前进位加法器

```
architecture advance of fp4 is
        component fp1
                port(
                        a,b,cin:in std_logic;
                        s,cout:out std_logic;
                        p,g:buffer std_logic
                );
        end component;
        signal p,g,c:std_logic_vector(3 downto 0);
begin
        fa0:fp1 port map(a(0),b(0),cin,s=>s(0),p=>p(0),g=>g(0));
        fa1:fp1 port map(a(1),b(1),c(0),s=>s(1),p=>p(1),g=>g(1));
        fa2:fp1 port map(a(2),b(2),c(1),s=>s(2),p=>p(2),g=>g(2));
        fa3:fp1 port map(a(3),b(3),c(2),s=>s(3),p=>p(3),g=>g(3));
        process(p,g)
        begin
                c(0)<=g(0) or (p(0) and cin);
                c(1)<=g(1) or (p(1) and g(0)) or (p(1) and p(0)
                   and cin);
                c(2)<=g(2) or (p(2) and g(1)) or (p(2) and p(1)
                   and g(0)) or (p(2) and p(1) and p(0) and cin);
                cout<=g(3) or (p(3) and g(2)) or (p(3) and p(2)
                   and g(1)) or (p(3) and p(2) and p(1) and g(0))
                    or (p(3) and p(2) and p(1) and p(0) and cin);
        end process;
end advance;
```

工作原理：使用超前进位和元件例化，参考书本的简化公式，先计算 p、g，进位和结果的数值由 p、g 计算得出。由于并行的内部实现，超前进位的延时理论上应该比逐次进位小。

3.4 系统自带加法器

```
architecture system of fp4 is
        signal sys:std_logic_vector(4 downto 0);
begin
        process(a,b)
        begin
                sys<="00000"+a+b+cin;
        end process;
```

```
        process(sys)
        begin
                cout<=sys(4);
                s<=sys(3 downto 0);
        end     process;
end system;
```

工作原理：通过"00000"，使得四位加数变成五位的 vector，最终进位和结果分别取五位 vector 的最高位和末四位，即可完成加法运算。

4 仿真结果

4.1 逐次进位仿真结果

如图1所示。

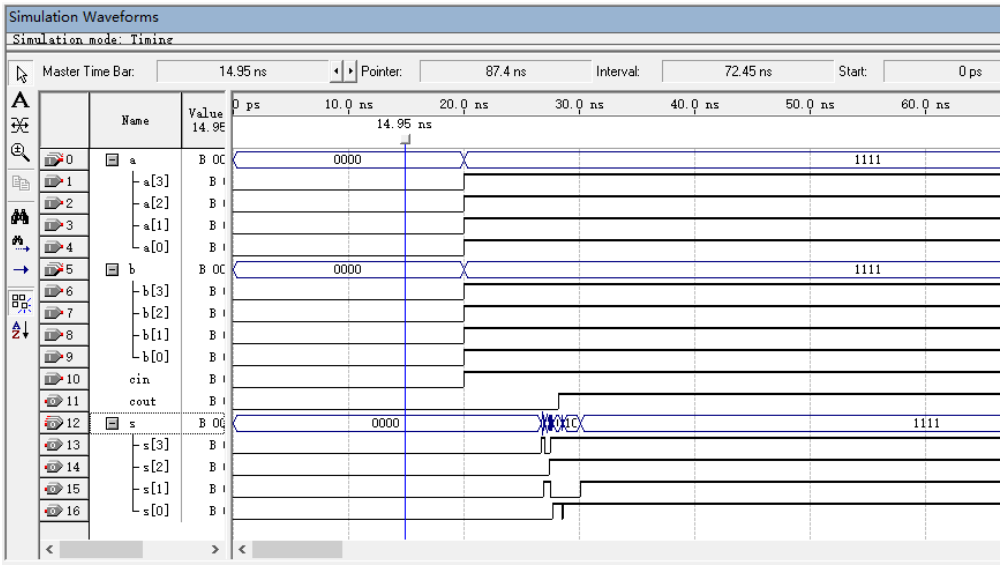

图 1: 逐次进位仿真结果

4.2 超前进位仿真结果

如图2所示。

4.3 系统自带加法器仿真结果

如图3所示。

4.4 仿真结果的比较

输入算式均为 0000+0000+0=0000 到 1111+1111+1=11111，四位加法器的延时大约在 8 ~ 9ns，由于位数结果较少，超前进位进行的计算较多，因此没有体现出明显的优势；而

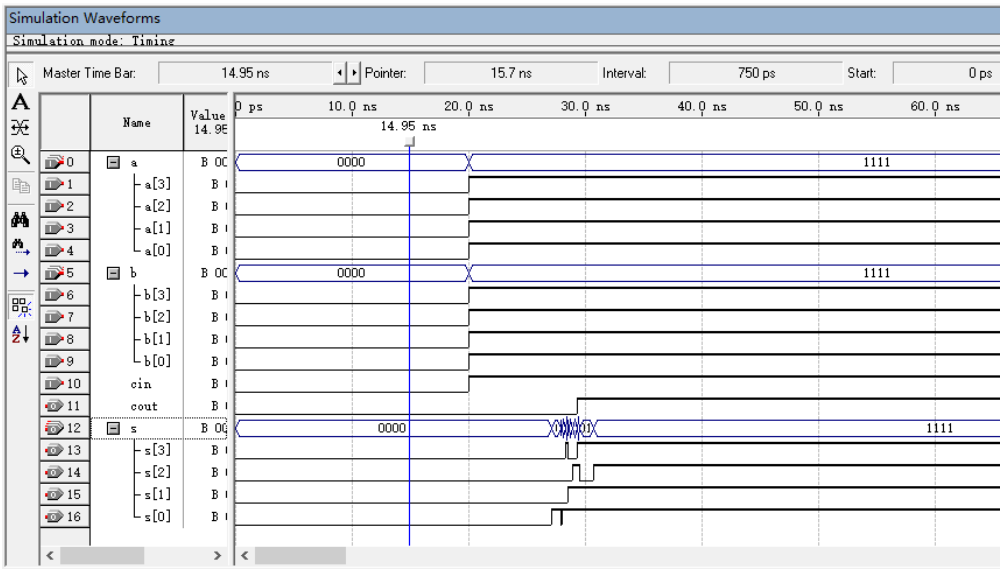

图 2: 超前进位仿真结果

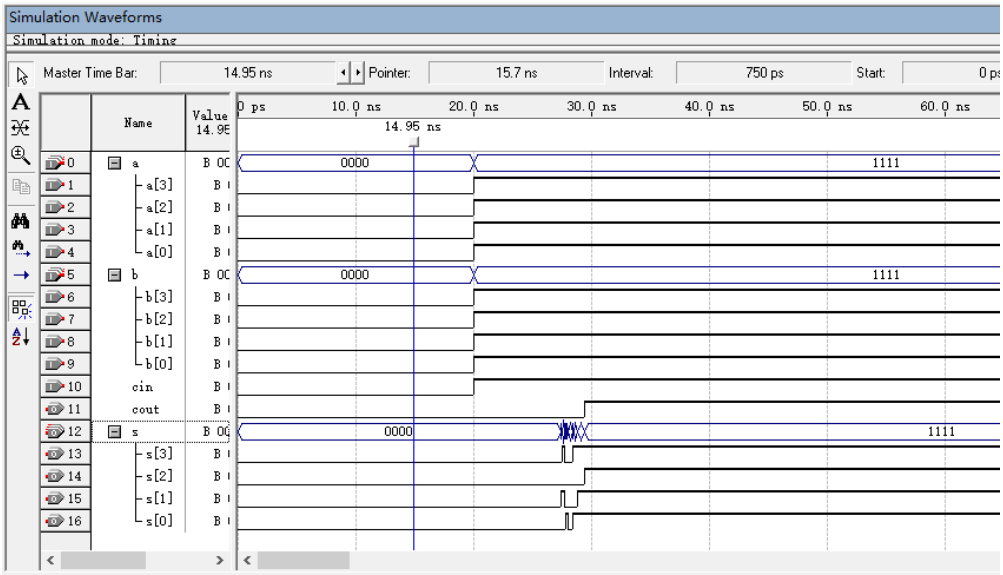

图 3: 系统自带加法器仿真结果

系统自带加法器进位信息和各位数值延时几乎相同，在平衡延时中优势显著，猜想系统自带加法器有一定的底层优化。

5 实验小结

这是我第二次进行 CPLD 实验，由于有“点亮数字人生”实验作铺垫，这次我使用 Quartus 的时候更加熟练。本次实验让我掌握了仿真的技能，仿真之后再将代码导入芯片，可以相对更安全地操作硬件，烧坏的几率大大降低。

本次实验让我对于全加器、半加器的理解更加透彻，对于逐次进位全加器和超前进位全加器的效果有了更直观的认识，通过仿真也了解到全加器延时的大致数量级，以及有些结果需要仿真实验才能得出，并不像理论结果那样显然。同时我也学习到了元件例化的技巧，增加了代码复用率，也使得代码可读性更强，有更好的层次性和结构性。