数字系统设计及FPGA入门实践 -- 11911303 吉辰卿

首先说明：有关所有的VHDL上板练习项目见E:\Desktop\DSD_practice、

2023.2.14

两条并行的学习路线：
1.理解rainbow算法的理论部分，能够自己用python代码实现加密和解密算法
2.学vhdl和简单的上板实践，能够慢慢理解项目加密和解密的vhdl程序

2023.2.15

1. 搞明白：单独用vivado开发并编程PL会不会动了BaseOverlay?

回答：

当然不是。

- 首先，要清楚FPGA可以被反复编程，且上电之后如果不把之前烧录的程序以**.bin**的形式存到FGPA逻辑门阵列外面(还在FPGA芯片里面但是在逻辑门外面)的ROM中，那么下一次上电程序会丢失且需要重新烧录。
- 其次，PYNQ、Zedboard有两种启动方式：**分别是通过SD卡启动和通过JTAG启动，这两种模式的切换是通过改变跳线帽的位置来实现的。两种模式就像进入了两种不同的操作系统一样。**通过SD卡启动直接进入默认的linux操作系统，里面的BaseOverlay等文件会烧到PL端从而编程FPGA（.bit文件），且PS端操作系统可以在操作系统上配置相关的软件（例如控制AD9361的软件）；而通过JTAG启动相当于是一个空白的系统什么也没有，需要我们对其进行FPGA编程。所以说，我们在vivado设计完了之后，在FPGA上编程要选择用JTAG启动，因为JTAG启动相当于一个空白的系统，什么也没有。所以说单独开发PL不可能动了BaseOverlay，BaseOverlay需要用SD卡启动，我用JTAG启动不会动SD卡上面的文件配置。

因此，这个项目就会有一个问题：

发射机必须得把通过PL加密后的数据，通过串口发回给PC存起来。之后插入SD卡上电进入系统，再通过串口把加密后的东西给PS发过去，再调用PS上AD9361上的软件发给接收机。

2. PYNQ允许使用12v电源供电时同时用Mirco-USB进行.bit程序的烧录（即不必为了烧录程序而改为使用USB供电)。

2023.2.16

- **看一段本学期DSD实验课 lab 1的代码分析，懂得VHDL代码怎么看，代码分析在下面代码块的注释中:**

counter.vhd:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```
entity Counter is
    Port ( Clock : in STD_LOGIC;
           Reset : in STD_LOGIC;
           Direction : in STD_LOGIC;
           Count_out : out STD_LOGIC_VECTOR (3 downto 0));
end Counter; -- counter是器件，相当于是一个只带输入和输出的黑箱子，而后面的结构相当于描述里面黑箱子的具体情况

architecture Behavioral of Counter is
    signal count_in, count_in_next: std_logic_vector(3 downto 0);
    signal delay, delay_next: std_logic_vector(24 downto 0);
begin
-- 注意，底下的代码执行的时候（不管是process部分还是其他部分）全为并行执行，所以理解代码的时候一定不要按着顺序来理解，一定要抓住一个信号去分析，看那些地方用到了它，再进一步地慢慢理解整个程序
process (reset, clock)  -- process里面只有时序逻辑，时钟信号都写在里面，process之外的是组合逻辑
-- 如何进入一个process？先看process的敏感信号列表，如果敏感信号列表里面任意一个信号发生变化，那么就进入这个process
-- if then elsif 只能出现在process里面，即描述时序电路
-- 这个process里面有两个敏感信号，分别为reset和clock。很显然，clock是时钟信号，reset是复位信号
begin
    if reset = '1' then
        -- 如果复位信号由0变成1，那么delay全部置为0且count_in全部置为0
        -- <=后面啥也不写表示第一位是0，加括号里面写other表示后面的剩余位全为0.总结一下就是把所有位都置为0
        delay <= (others=>'0');
        count_in <= (others=>'0');
    elsif clock='1' and clock'event then
        --此处代码表示当clock是1且clock跳变事件发生，也就是clock处于上升沿（看的是clock跳变之后的状态，才是1）
        --才运行下面的代码
        delay <= delay_next;
        count_in <= count_in_next;
    end if;
end process;
-- 这一部分在process外面，描述的是逻辑电路
    delay_next <= delay + 1;
    count_in_next <= count_in +1 when delay = 0 and direction = '1' else
    count_in -1 when delay = 0 and direction = '0' else
    count_in;
    count_out <= count_in;
end Behavioral;
```

所以结构体里面的代码如何看？首先所有信号初始值默认是0。当然开始的时候还没到时钟，因此先不要急着看process里面的内容，我们可以直接看一下代码块的内容：

```
  delay_next <= delay + 1;
    count_in_next <= count_in +1 when delay = 0 and direction = '1' else
    count_in -1 when delay = 0 and direction = '0' else
    count_in;
    count_out <= count_in;
```

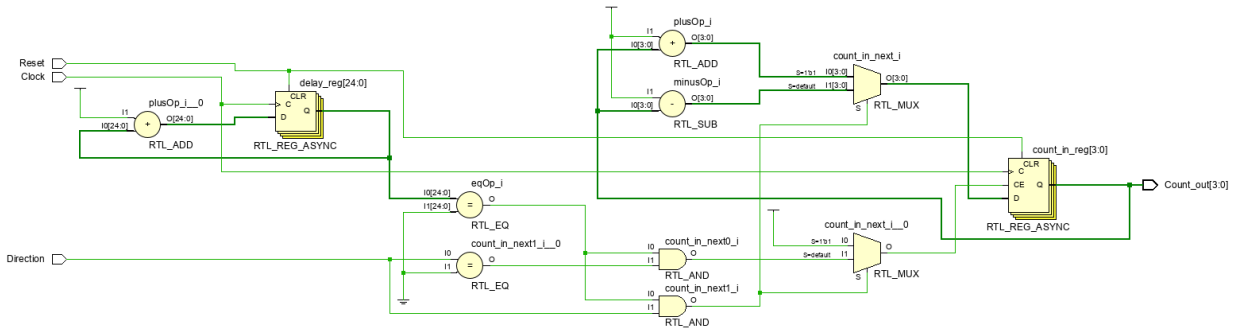
刚开始delay 是0, 把其加1赋给delay_next，所以delay_next变成1. 之后，由于delay为0，因此我们需要做出判断，如果direction是"1", 那么count_in_next变成了1（count_in默认是0，加了1变成1）。最后把count_in的值赋给count_out使得count_out的值变成1。

接着，系统就不动L1,静静等待时钟信号的到来,当时钟处于上升沿的时候，执行这两句：delay <= delay_next;

count_in <= count_in_next; 此时delay 是1，count_in 也变成了1。好了时钟结束，之后便跳到了process外面，delay_next变成2，count_in_next 1，且 count_out还是1. 下一次时钟到来，此时delay 是2，count_in还是1。之后，count_in_next 1，且 count_out还是1。一直加加加加加，直到最后一个时钟周期（2^25）时，delay变成了0。之后，count_in_next变成了2 且 count_out 立刻变成2。之后便进入下一个时钟大循环。

综上所述，我们可以看到，delay像一个时钟计数器，每次时钟之后+1. 而count_in像一个循环计数器，当delay每次过完了2^25之后，其才+1.

- **这里需要声明一点的是: 对于上面的代码描述，所有信号的初始值是不确定的，因FPGA内部的配置而异。为了简化问题，我才说所有信号的初始值为0.**
- 接下来我们看看上面设计得到的**RTL电路图**，看看是不是如我上所述那样子分析：



从这个原理图里面我们可以看到，delay和count_in已经被做成了一个寄存器，也就是说在process里面出现的信号(被赋值)，多半是寄存器。两个RTL_MUX为多路选择器，S为选择器的触发信号。S的值来确定每次选择左边的两条路中的那一条路通过。

然后基本上上面的电路图和我们上面写的代码是可以对上的，当然还有一个问题:上面的RTL电路中下面的那个RTL_MUX的多路选择器有什么用? 通过分析这个多路选择器的输入和输出我们可以清楚地看到：当delay是0，direction是1，底下那个MUX给CE送去1，而当delay是其他值的时候，不管direction是多少，MUX给CE送0。这里需要注意一点就是:右边寄存器输入的那个CE全称叫"clock enable",即**时钟使能信号**。当CE为0，则时钟信号无效，不管来多少个时钟上升沿都不会触发；而只有当CE的值为1时，时钟信号才会触发。

为了更好地说明这个问题，我们回到上面的代码中去。**我们可以惊奇地发现:** 只有当delay的值为0时，count_in_next的值才会增加1.等下一次时钟过来的时候，count_in的值才会改变。而delay等于其他值的时候，即便时钟过来了，count_in的值也不会改变(因为count_in_next的值没变)。所以说，当delay为非0值的时候，这个时钟也就没啥用（相当于啥也没做）。因此，为了节约资源，我们的RTL图才添加了一个时钟使

能信号，当delay为0的时候才开启时钟，让count_in的值发生改变；而在delay等于0的其他状态之下，这个时钟被停用。此时，无事发生。

但是上面的电路还是存在一个问题，当delay是非0的时候，上面的那个多路选择器的选择值是0，于是选择下一条也就是count_in_next = count_in -1,但是我代码里面明明写着count_in_next = count_in 在delay等于其他值的情况下, 这又该如何解释呢? 可以这样去理解：相当于CE是0，那个count_in_next它即使改变也相当于无效改变因为值送不过去默认就相当于不改变。

- 接下来，我们看看这个.vhd文件的testbench是怎么写的，如下所示：

tb_counter.vhd:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY counter_tbw IS
END counter_tbw;

ARCHITECTURE behavior OF counter_tbw IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT counter  --这里的原件表示把待测试的电路黑盒子--counter拿过来，因此里面定义的端口和Counter这个实体里面的定义完全一样，且元件名字必须和待测试结构体的实体名字一样程序才能识别相应的实体。
 PORT(
 CLOCK : IN std_logic;
 reset : IN std_logic;
 DIRECTION : IN std_logic;
 COUNT_OUT : OUT std_logic_vector(3 downto 0)
 );
 END COMPONENT;
 --Inputs --输入激励信号，因为我们测试的时候没有外界信号源，因此我们必须为每个输入端口添加激励信号以便进行测试，当然激励信号的波形行为我们可以自己定义
 signal CLOCK : std_logic := '0';
 signal reset : std_logic := '0';
 signal DIRECTION : std_logic := '0';
 --Outputs
 signal COUNT_OUT : std_logic_vector(3 downto 0);
 -- Clock period definitions
 constant CLOCK_period : time := 40ns;

BEGIN --开始测试，首先把激励信号源和对应的输入输出端口连接上
 -- Instantiate the Unit Under Test (UUT)
 uut: counter PORT MAP (  --这里前面必须写一个类似于uut:的注释，不写就会报错（比如这里前面写yingshe: 也行）
 CLOCK => CLOCK,
 reset => reset,
 DIRECTION => DIRECTION,
 COUNT_OUT => COUNT_OUT
 );
```

```
 -- 下面就是三个输入的激励信号的行为特性定义，描述时钟、复位和direction信号每个周期的波形特征(为了测试这块可以自己随便写)
 -- Clock process definitions
 CLOCK_process :process  -- process前面不写注释不会报错(CLOCK_process:)
 begin
 CLOCK <= '0';
 wait for CLOCK_period/2;
 CLOCK <= '1';
 wait for CLOCK_period/2;
 end process;
 -- reset process definitions
 reset_process:process
 begin
 reset <= '1';
 for i in 1 to 2 loop
 wait until clock='1';
 end loop;
 reset <='0';
 wait;
 end process;
 -- Stimulus process
 stim_proc: process
 begin

 DIRECTION <= '1';
 wait for 1400ns; --current Time:1400ns

 DIRECTION <='0';
 wait for 800ns; -- current Time:2200ns
 DIRECTION <= '1';
  wait;
 end process;
end;
-- 有了输入信号的波形特征测试就可以开始跑了，我们打开simulation即可看到输出输出信号对应的波形图
```

由于测试代码写起来比较简单，我就把代码的解读直接放到代码的注释里面去了。

总结一下，写testbench文件需要的四个要素：定义元件(把待测试的黑盒子拿过来)、定义激励信号、把激励信号和对应的输入输出端口映射上（端口映射）、定义输入激励信号的波形行为特征

2023.2.17

- 踩了一个小坑，就是vivado写完.xdc文件之后发现切到 I/O Planning之后底下对应的引脚没有同步分配且电压没改成3.3V.生成比特流自然会报错。**这是因为.xdc里面的信号端口要和.vhd里面定义的信号端口(输入和输出)大小写必须一样，如果不一样软件就识别不出来。我以为.xdc和VHDL一样可以忽略大小写，没想到他还看大小写。**把信号大小写改对（和.vhd）中的一样之后，就可以正常生成比特流啦，开心!

- 还有一个需要注意的点，vivado当你改完文件左上角显示合成或者实现过期，这时候你可以点detail进去选择Force Up_todate强制更新。**但是这个强制更新并不会重新跑一遍合成或者实现，而是在显示上强制显示当前是最新状态。因此改完某些文件要重跑一下合成或者实现的话还是得在左边的Project manager里面依次跑一下合成和实现才行。**