# 数字集成电路第11次作业

1. 选题回顾

* **基于Booth编码和Wallace树的乘法器设计**

**要求：**设计基于Booth编码和Wallace树的16 x 16无符号数乘法器电路。Booth算法对乘数从低位开始判断，根据两个数据位的情况决定进行加法、减法还是仅仅移位操作。利用Booth算法产生部分积，然后利用Wallace树结构进行部分积压缩，从而提高乘法器的性能。

二、电路结构



**图1：电路结构图**

本乘法器主要由三部分组成。首先，输入的两个16位信号进入Booth乘法器，并输出部分积和加1信号给Wallace树用于部分积压缩。最后，将压缩好的信号输入给32位超前进位加法器，并输出计算结果。

三、模块代码

**Top module**

module multiplier (

**input** **wire** [15:0]  InputX  ,//*输入数据，二进制补码*

**input** **wire** [15:0]  InputY  ,//*输入数据，二进制补码*

**output** **wire** [31:0] product //*输出乘积，二进制补码*

);

// *8个booth，得到8个扩展到32位的部分积*

**wire** [7:0] Cout\_in;

**wire** [31:0] Xout\_in[7:0];

Booth\_8 u\_Booth\_8(

    .x         (InputX    ),

    .y         (InputY    ),

    .addsignal (Cout\_in   ),

    .xout0     (Xout\_in[0]),

    .xout1     (Xout\_in[1]),

    .xout2     (Xout\_in[2]),

    .xout3     (Xout\_in[3]),

    .xout4     (Xout\_in[4]),

    .xout5     (Xout\_in[5]),

    .xout6     (Xout\_in[6]),

    .xout7     (Xout\_in[7])

);

// *32位的Wallace树，将8个部分积的和压缩成2个数的和*

**wire** [31:0] C\_in;

**wire** [31:0] S\_in;

Wallace\_32 u\_Wallace\_32(

    .xin0 (Xout\_in[0]  ),

    .xin1 (Xout\_in[1]  ),

    .xin2 (Xout\_in[2]  ),

    .xin3 (Xout\_in[3]  ),

    .xin4 (Xout\_in[4]  ),

    .xin5 (Xout\_in[5]  ),

    .xin6 (Xout\_in[6]  ),

    .xin7 (Xout\_in[7]  ),

    .Cin  (Cout\_in[5:0]),

    .C    (C\_in        ),

    .S    (S\_in        ),

);

// *使用32位超前进位加法器将计算乘法结果*

ADD\_32 u\_ADD\_32(

    .Input1(S\_in      )             ,

    .Input2({C\_in[30:0],Cout\_in[6]}),

    .C0    (Cout\_in[7])             ,

    .out   (product   )             ,

);

endmodule

**Booth编码**

module booth (

**input**  **wire** [15:0] x ,//*乘数x*

**input**  **wire** [2: 0] y ,//*来自乘数y的控制位*

**output** **wire**        addsignal,//*进位标志位*

**output** **wire** [16:0] partial\_product //*部分积*

);

**wire** x\_add1 = (~y[2]&~y[1]&y[0])|(~y[2]&y[1]&~y[0]);

**wire** x\_add2 = (~y[2]&y[1]&y[0]);

**wire** x\_sub2 = (y[2]&~y[1]&~y[0]);

**wire** x\_sub1 = (y[2]&~y[1]&y[0])|(y[2]&y[1]&~y[0]);

**assign** partial\_product = {17{x\_add1}} & {x[15],x}

                            |{17{x\_add2}} & {x[15:0],1'b0}

                            |{17{x\_sub1}} & {~x[15],~x}

                            |{17{x\_sub2}} & {~x[15:0],1'b1};

**assign** addsignal = x\_sub1|x\_sub2;

endmodule

这里使用的是基4的Booth编码，相比于基2可以大大加快运算速度。

**Wallace树压缩**

module wallace\_8(

**input**   [7: 0]     N   ,// *N个1bit数按相同的权重进行压缩*

**input**   [5: 0]     Cin ,// *来自右侧的进位*

**output**             C   ,// *最后一级计算的C*

**output**             S   ,// *最后一级计算的S*

**output**  [5: 0]     Cout // *传递到左侧的进位，没用到就是了*

);

**wire** [5: 0]    Input\_second;

    full\_adder u\_adder\_1(.Cin(N[7-:3]),.Cout(Cout[0]),.S(Input\_second[5]));

    full\_adder u\_adder\_2(.Cin(N[4-:3]),.Cout(Cout[1]),.S(Input\_second[4]));

    half\_adder u\_adder\_3(.Cin(N[1-:2]),.Cout(Cout[2]),.S(Input\_second[3]));

**assign** Input\_second[2:0] = {Cin[0],Cin[1],Cin[2]};

**wire** [3:0]    Input\_third;

    full\_adder u\_adder\_4(.Cin(Input\_second[5-:3]),.Cout(Cout[3]),.S(Input\_third[3]));

    full\_adder u\_adder\_5(.Cin(Input\_second[2-:3]),.Cout(Cout[4]),.S(Input\_third[2]));

**assign** Input\_third[1:0] = {Cin[3],Cin[4]};

**wire** [2:0]    Input\_forth;

    full\_adder u\_adder\_6(.Cin({Input\_third[3-:2],Cin[3]}),.Cout(Cout[5]),.S(Input\_forth[2]));

**assign** Input\_forth[1:0] = {Input\_third[0],Cin[5]};

    full\_adder u\_adder\_7(.Cin(Input\_forth[2-:3]),.Cout(C),.S(S));

endmodule

module Wallace\_32(

**input** **wire** [31:0]  xin0     ,

**input** **wire** [31:0]  xin1     ,

**input** **wire** [31:0]  xin2     ,

**input** **wire** [31:0]  xin3     ,

**input** **wire** [31:0]  xin4     ,

**input** **wire** [31:0]  xin5     ,

**input** **wire** [31:0]  xin6     ,

**input** **wire** [31:0]  xin7     ,

**input** **wire** [5:0]   Cin      ,//*最右侧的进位输入*

**output** **wire** [31:0] C        ,//*最上面的输出进位*

**output** **wire** [31:0] S        ,//*最上面的输出*

**output** **wire** [5:0]  Cout      //*最左侧的输出，最后没用到*

);

**wire** [5:0] c\_in[32:0];

**assign** c\_in[0] = Cin[5:0];

**genvar** temp;

**generate**

**for**(temp=0; temp<32; temp=temp+1)

**begin**:wallace

        wallace\_8 u\_wallace\_8(

            .N    ({xin7[temp],xin6[temp],xin5[temp],xin4[temp],xin3[temp],xin2[temp],xin1[temp],xin0[temp]}),

            .Cin  (c\_in[temp]   ),

            .C    (C[temp]     ),

            .S    (S[temp]     ),

            .Cout (c\_in[temp+1] )

        );

**end**

**endgenerate**

**assign** Cout = c\_in[32];

endmodule

Wallace树将输入的8个部分积按照3个一组进行压缩，从3-3-2压缩至2-2-2，重新分组后得到3-3，压缩为2-2，再分组为3-1，并压缩至2-1，再全部分到一组并压缩为2个数之和，即32位的S和C。

**32位超前进位加法器**

module ADD\_32(

**input**  **wire** [31:0]  Input1  ,//*输入数据1，补码*

**input**  **wire** [31:0]  Input2  ,//*输入数据2，补码*

**input**  **wire**         C0      ,//*来自低位的进位*

**output** **wire** [31:0]  out     ,//*输出和*

**output** **wire**         C        //*输出进位*

);

**wire** [31:0] p1 = Input1|Input2;

**wire** [31:0] g1 = Input1&Input2;

**wire** [31:0] c;

**wire** [7:0] p2, g2;

**wire** [1:0] p3, g3;

**assign** c[0] = C0;

**genvar** temp;

**generate**

**for** (temp = 0; temp<8; temp=temp+1)

**begin**

        Add\_4 u\_Add\_4\_first (.P1(p1[(4\*temp+3)-:4]),.G1(g1[(4\*temp+3)-:4]),.C0(c[temp\*4]),.P4(p2[temp]),.G4(g2[temp]),.C(c[(4\*temp+3)-:3]));

**end**

**endgenerate**

**generate**

**for** (temp = 0; temp<2; temp=temp+1)

**begin**

        Add\_4 u\_Add\_4\_second (.P1(p2[(4\*temp+3)-:4]),.G1(g2[(4\*temp+3)-:4]),.C0(c[temp\*16]),.P4(p3[temp]),.G4(g3[temp]),.C({c[temp\*16+12],c[temp\*16+8],c[temp\*16+4]}));

**end**

**endgenerate**

**assign** c[16]=g3[0]|(p3[0]&c[0]);

// *得到进位后计算加法和*

**assign** C = (Input1[31]&Input2[31]) | (Input1[31]&c[31]) | (Input2[31]&c[31]);

**assign** out = (~Input1&~Input2&c)|(~Input1&Input2&~c)|(Input1&~Input2&~c)|(Input1&Input2&c);

endmodule

本32位加法器使用组间顺序，组内超前进位结构。按照4位一组分为3层，其中最底层的8个4位超前进位加法器接入32位上层Wallace数两个输入和Booth编码器的进位，输出8位的两个超前进位信号及进位信号；第二层接入这8位信号，输入到2个4位超前进位加法器，输出2位的两个超前进位信号及进位信号，最顶层即将这些信号进行简单运算即可得到最终结果。此时输出的进位信号C由于没用到所以在顶层就没例化。

四、测试代码及仿真结果

**`timescale** 1ns / 1ps

module multiplier\_tb;

**reg** [15:0]  InputX     ;

**reg** [15:0]  InputY     ;

**wire** [31:0] product    ;

**initial**

**begin**

        InputX = 16'hFC18;

        InputY = 16'hCB;

        # 50 ;

        InputX = 16'h1;

        InputY = 16'hFFFF;

        # 50 ;

        InputX = 16'h0001;

        InputY = 16'h8000;

        # 50 ;

        InputX = 16'hAAAA;

        InputY = 16'hAAAA;

        # 50 ;

        InputX = 16'h7FFF;

        InputY = 16'h7FFF;

        # 50 ;

**$finish**;

**end**

    multiplier u\_multiplier(

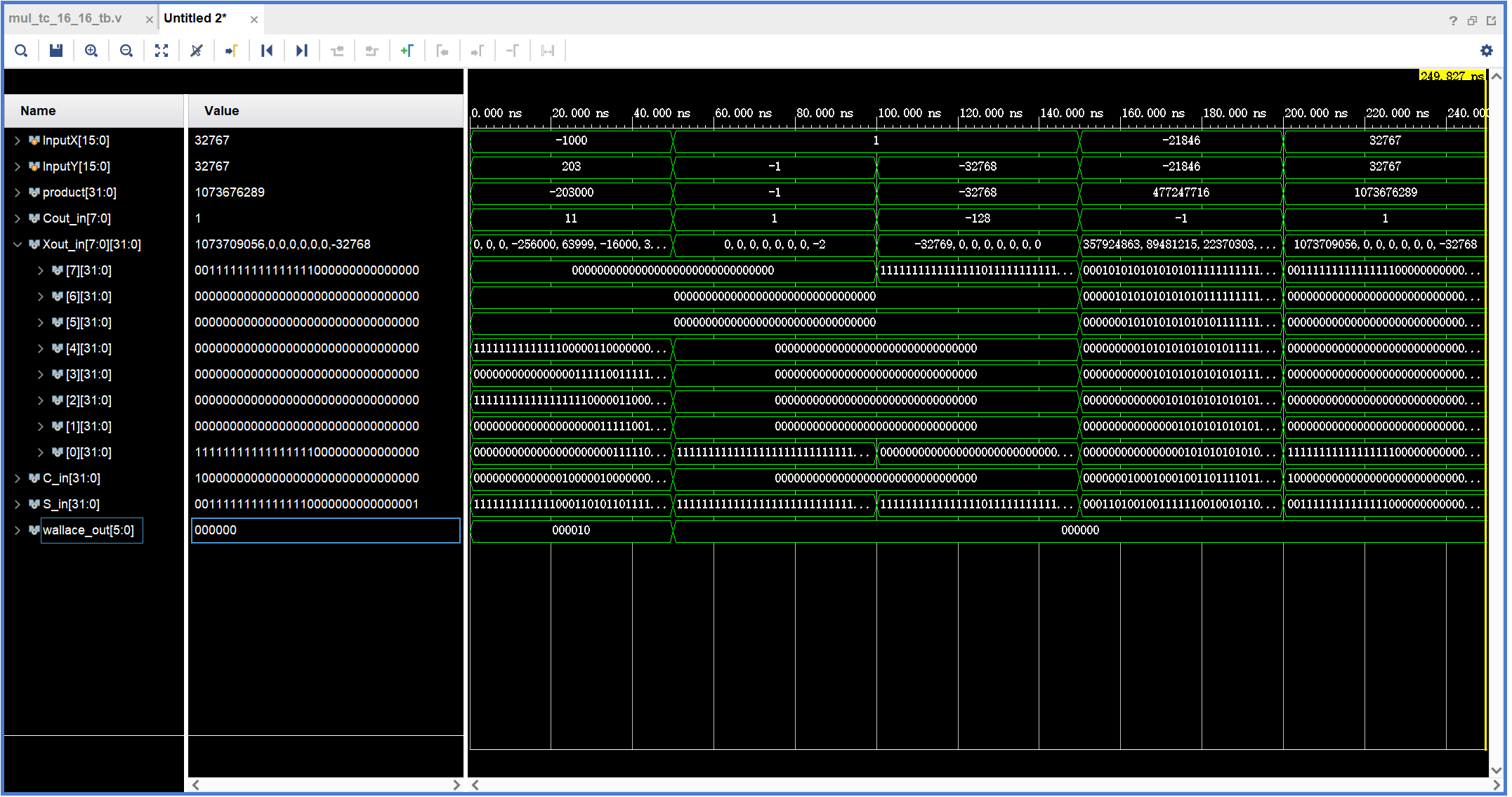
        .InputX  (InputX),

        .InputY  (InputY),

        .product (product)

    );

endmodule



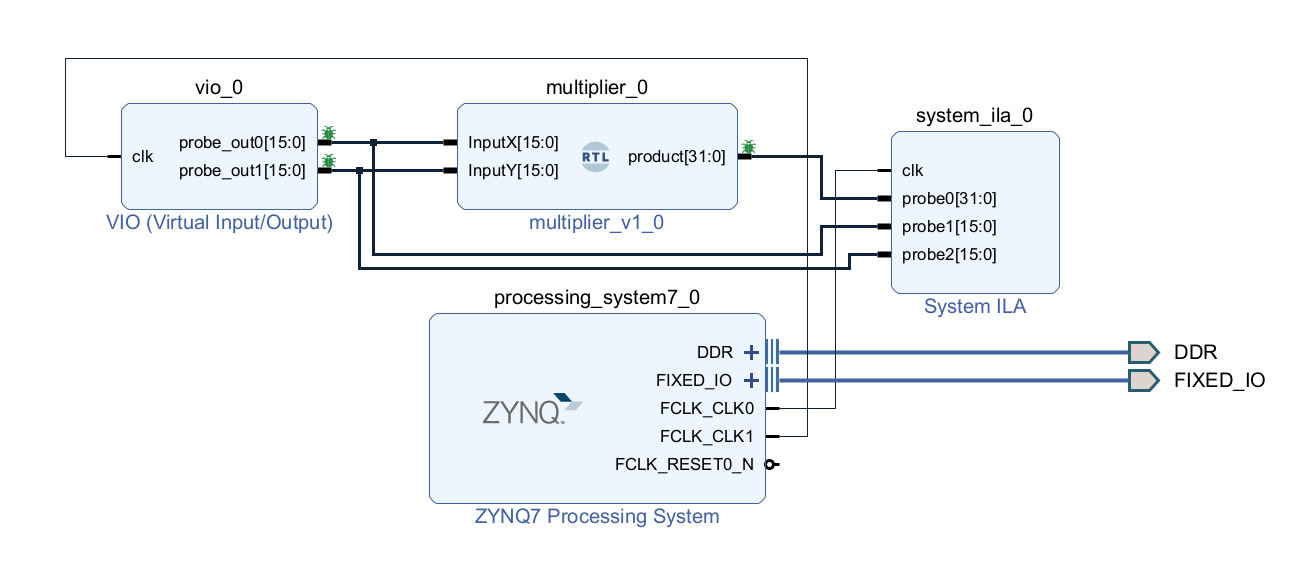
**图2：软件仿真结果图**

如上图所示，本乘法器计算结果符合要求，通过了软件仿真测试。

五、硬件验证及评价

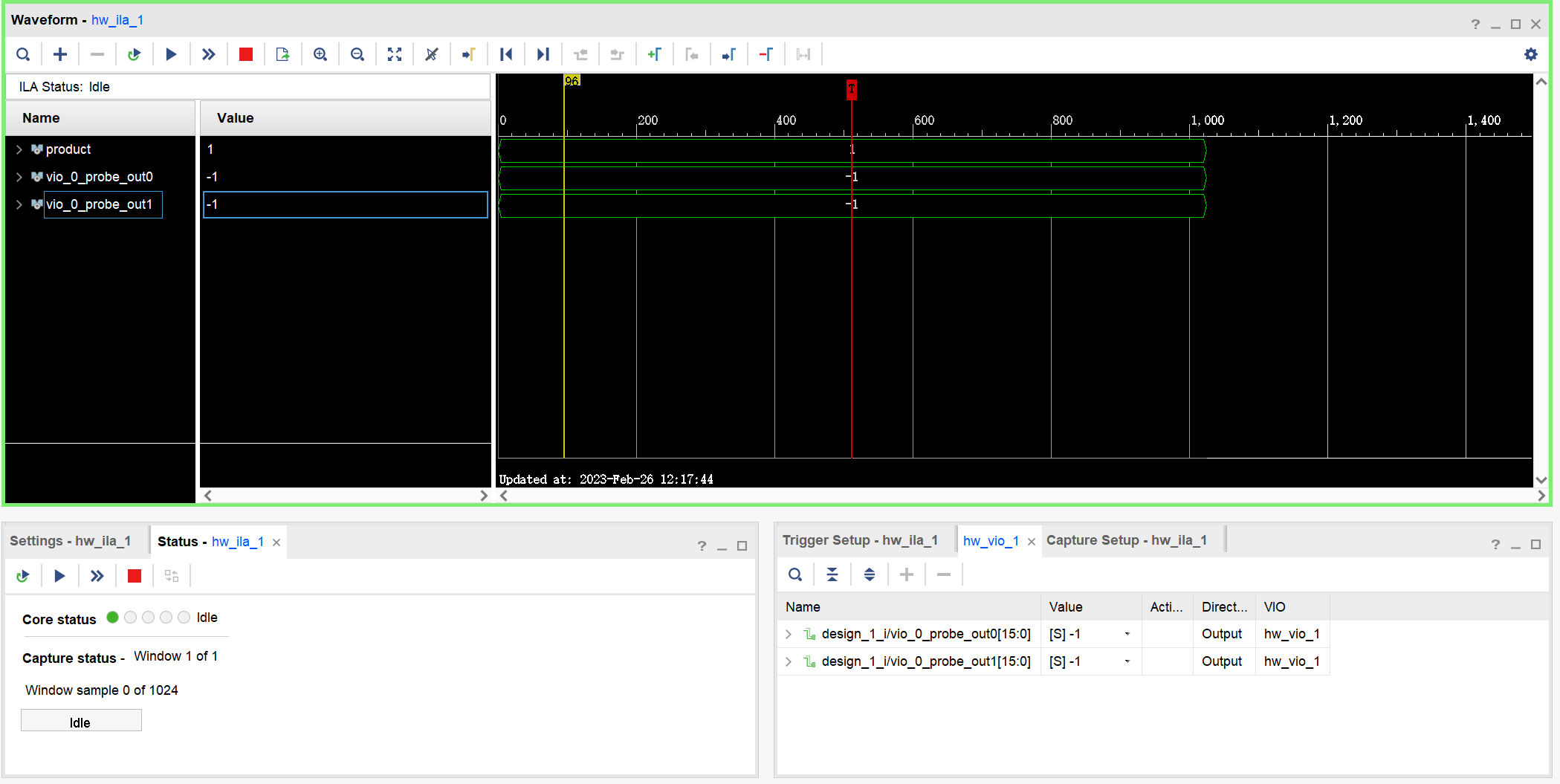
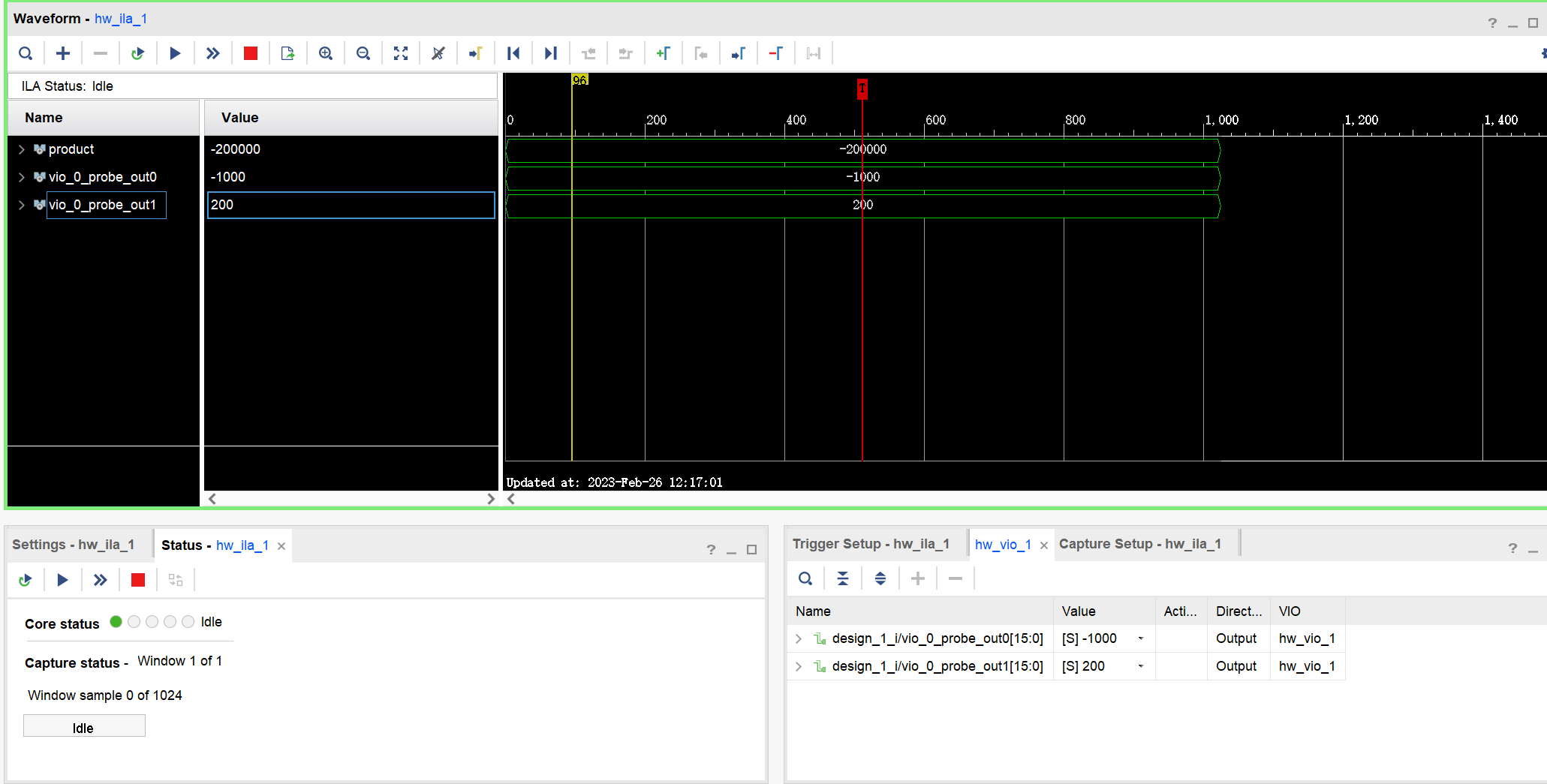
* **硬件验证**

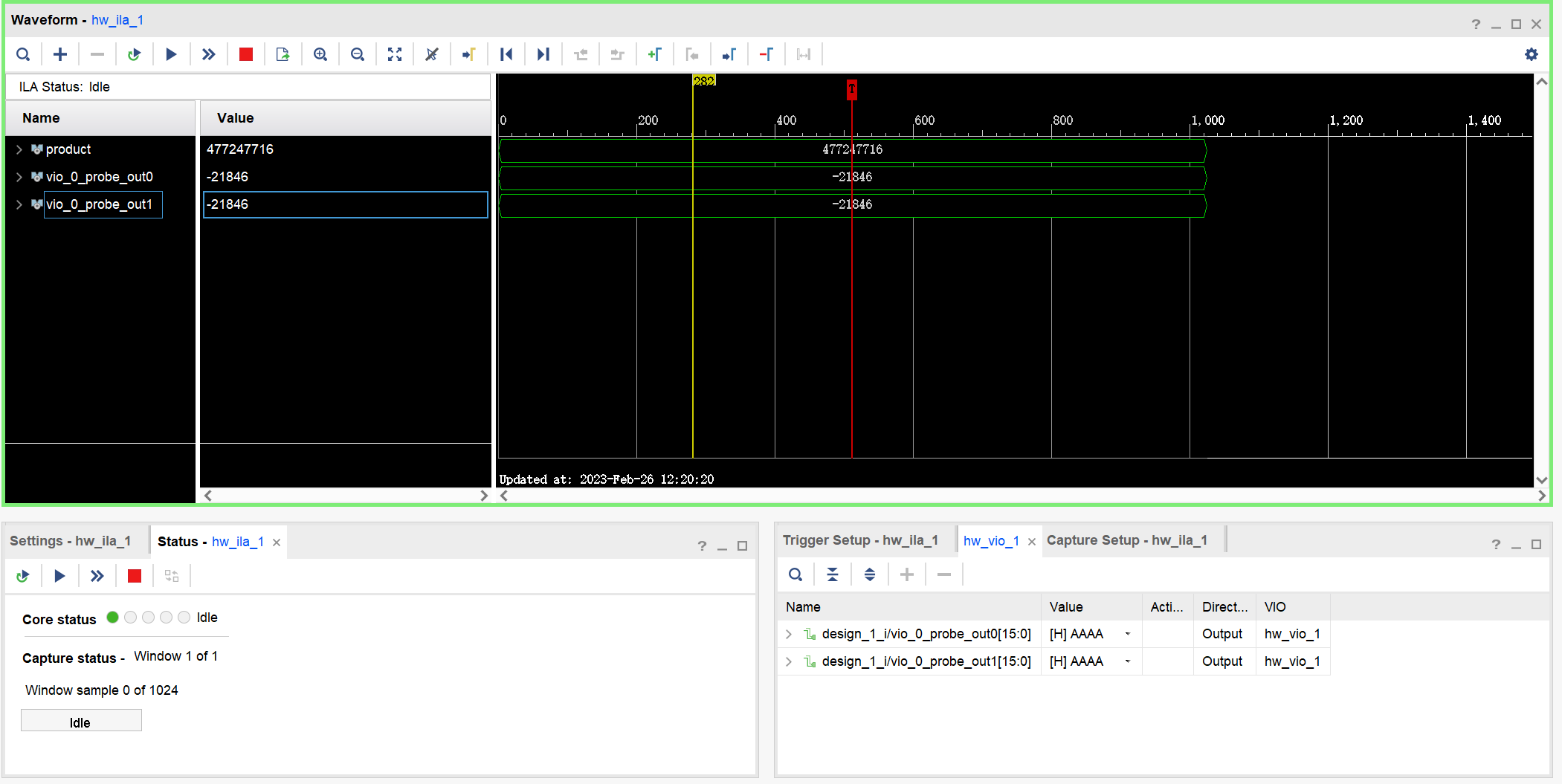
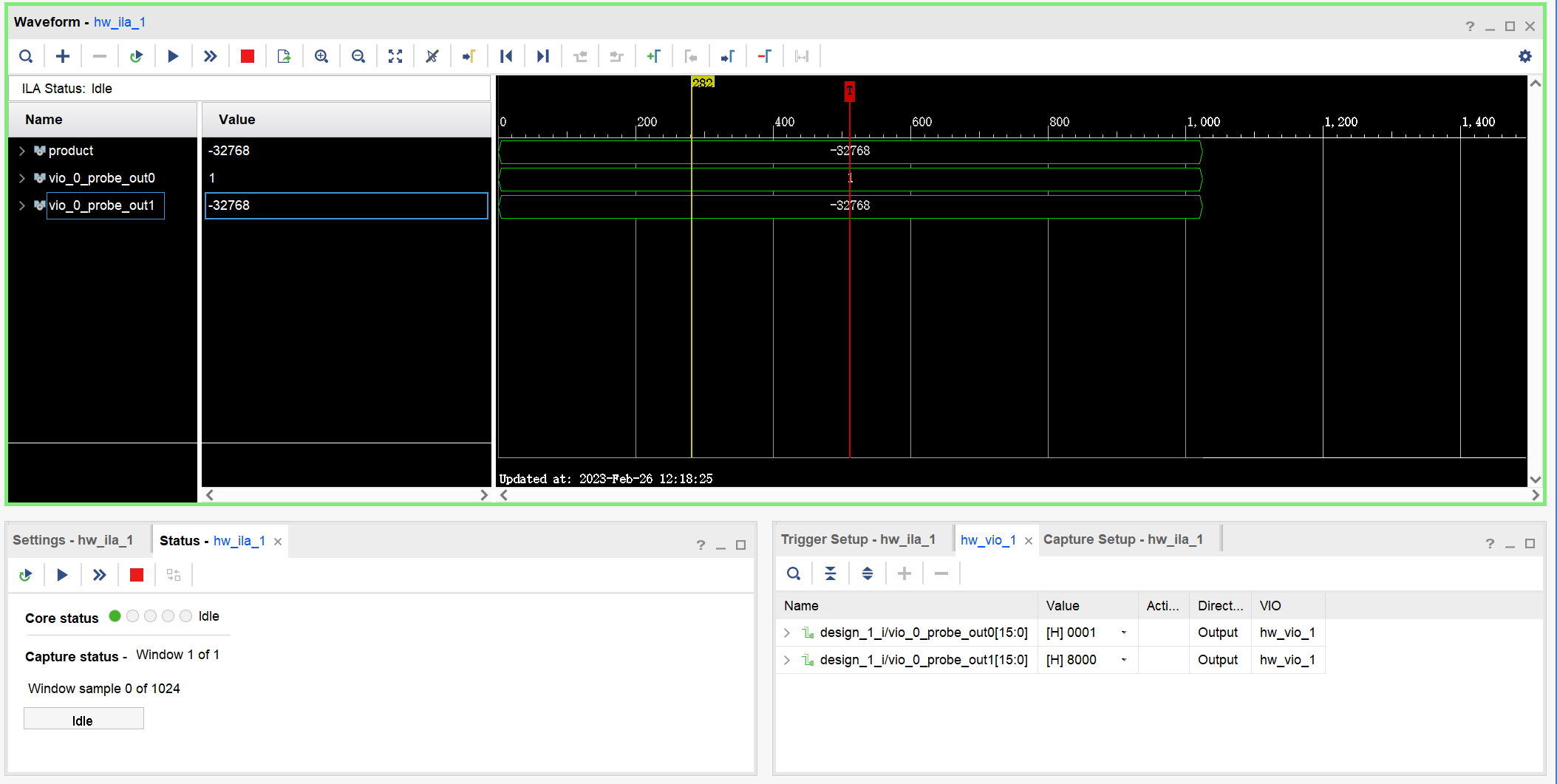
我们基于zynq7010进行了硬件FPGA仿真，结构图如下所示，使用vio输入两个加数，并通过ila实时显示输入输出值。

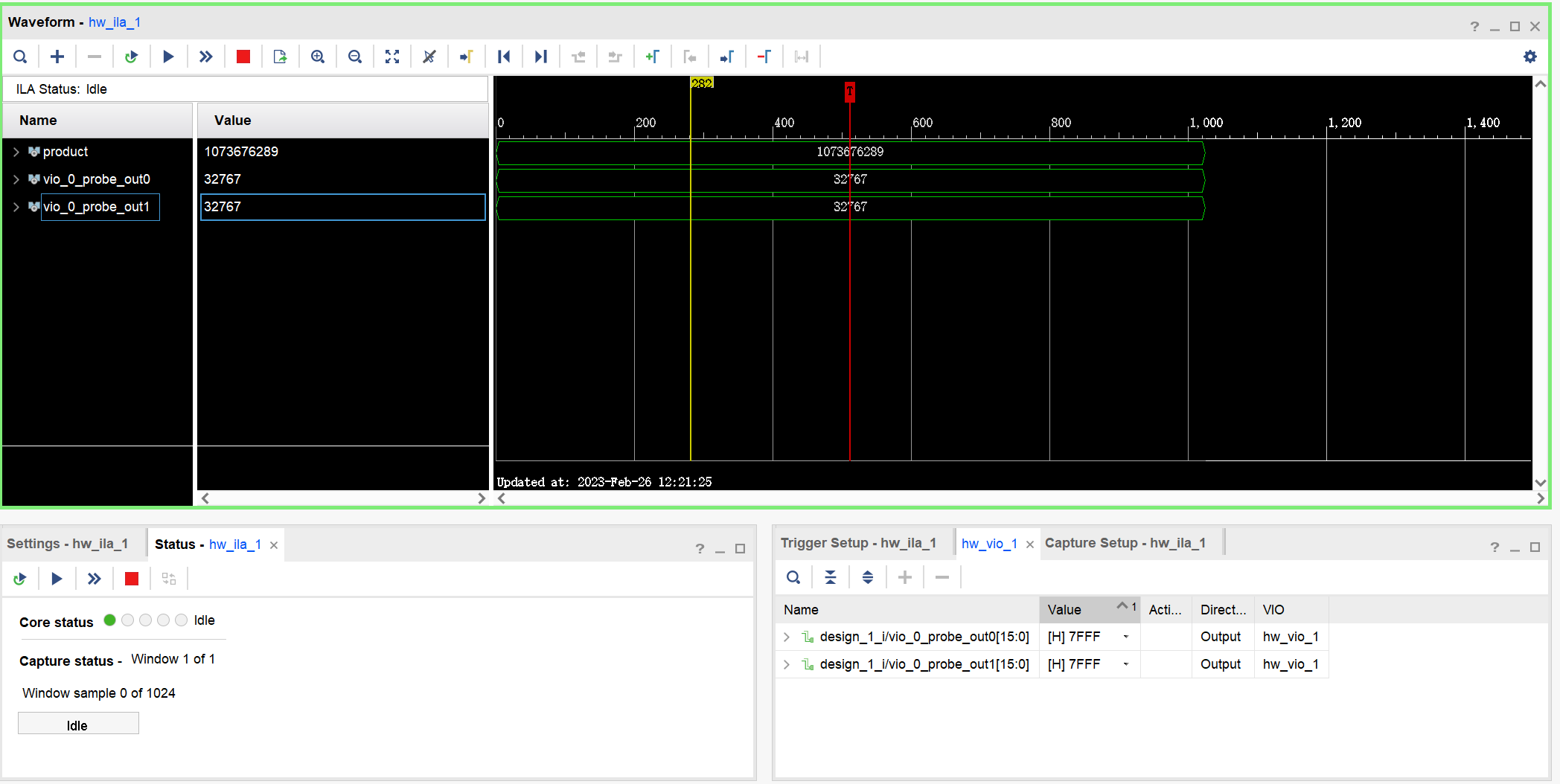


**图3：硬件仿真结构图**

我们按照上节设置的输入进行了硬件测试，结果如下图所示。



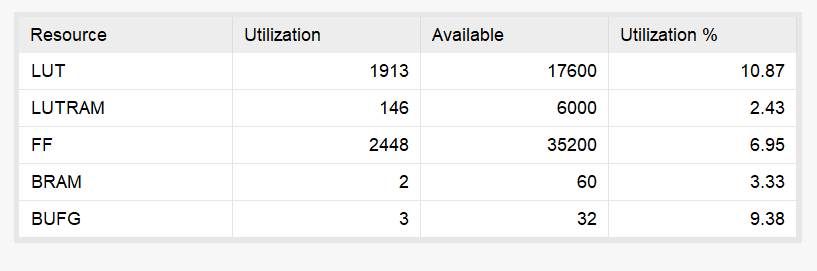
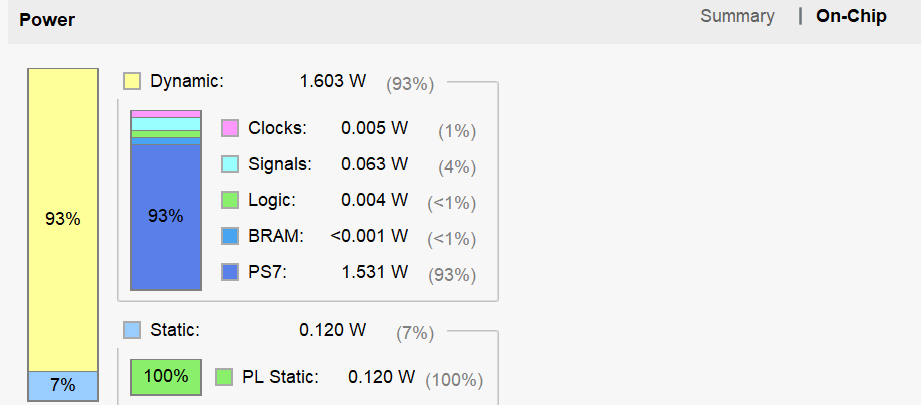




**图4：硬件仿真结果图**

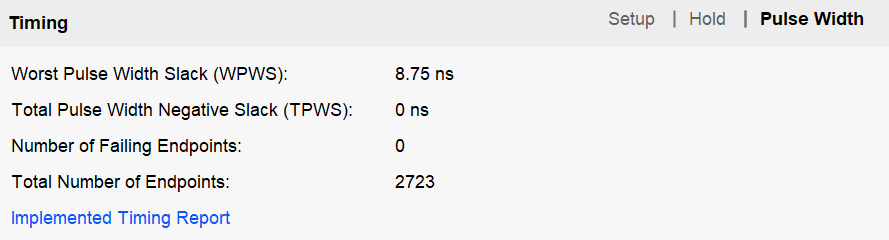
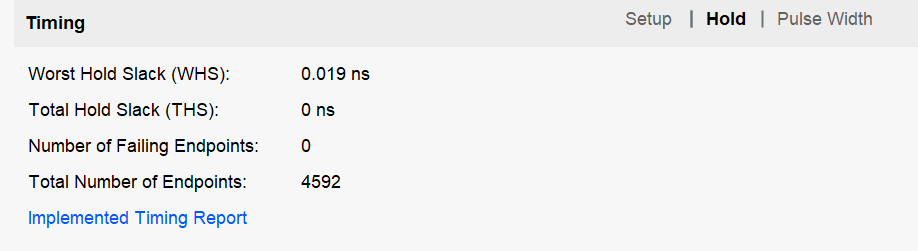
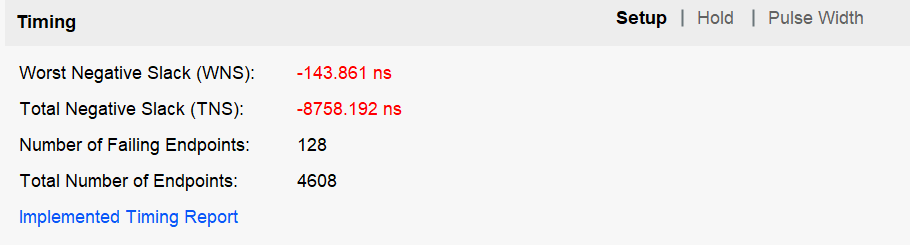
如图所示，硬件计算结果正确，符合要求。因此认为乘法器设计满足设计功能要求。

### 评价

**图5：资源消耗及功耗图**

如图所示，该乘法器消耗了不算少的硬件资源，我认为这是由于该乘法器确实用到了较多的结构来加快运算速度，因此在资源上相对消耗更多。而由于基本没有使用到zynq的高速性能，因此功耗不怎么高。



**图6：时序分析图**

如图所示，本设计在建立时间上出现了较大的违例。我认为这是因为在模块中有大量的子模块输入输出相互依赖的原因。目前还没想到怎么去优化这一问题。

总的来说，我较好的完成了相关设计任务。

对应测试工程已上传Github：<https://github.com/XS-dev/digital-integrated-circuit2023.git>