计算机系统 I 实验报告 4

实验 4-1：卷积模块

实验 4-2: 串口使用 (BONUS)

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 5 月 15 日

目录

1 实验 4-1：卷积模块

1.1 实验过程

在设计代码过程中碰到了许多关于数据类型的问题，因此在此处先对于预定义的卷积包 *conv_struct.vh* 以及各种预定义模块中设计的各种数据类型加以说明：

Code Snippet 1.1.1 ▶ repo/include/conv_struct.vh

```
1  //conv_struct
2  parameter WIDTH = 64;
3  parameter LEN   = 4;
4
5  typedef logic [WIDTH-1:0] data_t;
6  typedef logic [WIDTH*2-1:0] result_t;
7
8  typedef struct{
9      data_t data [LEN-1:0];
10 } data_vector;
11 //conv_operator
12 typedef struct {
13     Conv::result_t data;
14     logic valid;
15 } mid_vector;
```

这些数据类型的含义大致如下：

- *data_t*: 本次实验中的基本数据类型，用于存储一个 64 位长的操作数。
- *result_t*: 基于乘法器知识可知，操作数结果应保证至少两倍长于操作数，用于存储最大 128 位的操作结果。
- *data_vector*: 这一类型是第一种类型所构成的数组，因为卷积核和参与运算的部分都是一个数组，而其中每一个数所表示的向量则使用了 *data_t* 类型。
- *mid_vector*: 因为我们在第一阶段需要进行连续的乘法操作，在这个数据类型中引入了 *valid* 数据来标志乘法的完成，这在状态机的控制中有重要作用。

由于乘法器和上一次实验完全相同，接下来的模块代码设计中将其略过不谈（不知道为什么不改成 valid-ready，而是直接拿其他模组 valid-ready 部分信号作为 start 和 finish 信号）。

1.1.1 移位寄存器模块设计

根据提供要求，我们设计了如下的有限状态机：

Code Snippet 1.1.2 ▶ src/submit/Shift.sv

```
integer i;
always @(posedge clk or negedge rst) begin
    if(~rst)begin
        state_reg<=RDATA;
        for(i=0;i<Conv::LEN;i=i+1)begin
            data_reg[i]=0;
        end
    end else begin
        case(state_reg)
            RDATA:begin
                out_valid<=1'b0;
                in_ready<=1'b1;
                if(in_valid)begin
                    for(i=0;i<Conv::LEN-1;i=i+1)begin
                        data_reg[i]=data_reg[i+1];
                    end
                    data_reg[Conv::LEN-1]=in_data;
                    state_reg<=TDATA;
                end
            end
            TDATA:begin
                for(i=0;i<Conv::LEN;i=i+1)begin
                    data.data[i]=data_reg[i];
                end
                out_valid<=1'b1;
                if(out_ready)begin
                    state_reg<=RDATA;
                end
            end
        endcase
    end
```

```
32  end
```

文档中提供了详尽的条件跳转说明，此处略过不谈。但在 *TDATA* 中碰到了这样一个问题，经过测试发现并不能通过 *data.data = data_reg* 的方式赋值，因为结构体成员不能直接被赋值为一个非打包的向量，因此使用了循环赋值的方式实现数据的传输。

1.1.2 卷积运算模块设计

在使用移位寄存器后，我们在一个原始的输入数据序列中得到了类似如下一个滑动窗口的东西：

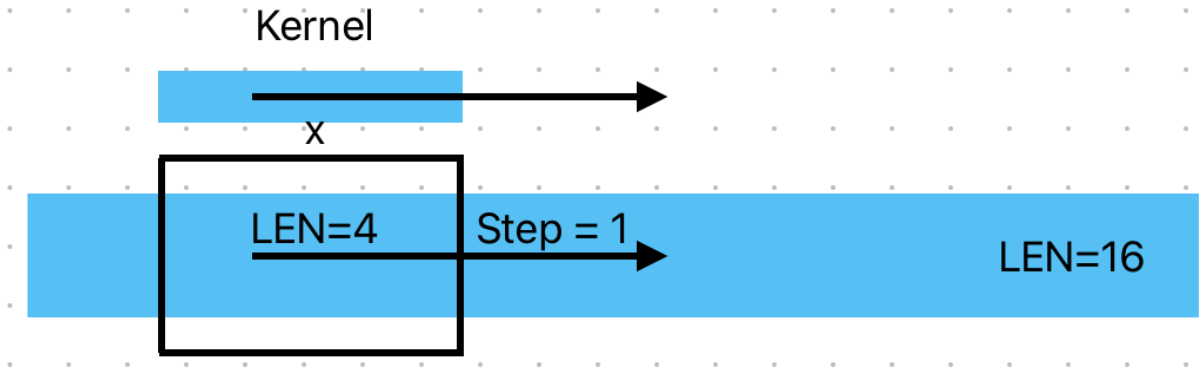

图 1.1: 移位寄存器的等效作用

这个滑动窗口会始终保持与卷积核对齐，并进行一个实际上类似于向量数量积操作的卷积运算。而我们的卷积运算模块的目的就是实现这个计算操作。这个状态机不同状态的说明同样在文档中有详尽说明，这里讨论几个遇到的问题：

1. 文档中的并行加法树似乎存在问题，我们应该从这个堆的底部开始逐步得到根的答案，因此循环应当从后向前进行，这在后面的代码中有所体现。
2. 同移位寄存器中相同的问题，对于 *vector_stage*1 ，由于其是现成的结构体，其成员不能被向量直接赋值，因此使用了一个 *logic* 数组 *vectoe_stage*1*_data* 来保存各个元素同卷积核运算的乘积。
3. 由于 *always* 块中并不能直接例化模组，因此 *data_vector* 中的 *valid* 发挥了重要作用，通过 *generate* 语句例化了逐级运算的乘法器，而 *valid* 则提供了从 *WORK* 跳转至 *TDATA* 的运算依据，即只有在滑动窗口中的最后一个元素参与运算后才会转向数据传输阶段。

以下是根据跳转逻辑设计的有限状态机：

Code Snippet 1.1.3 ▶ src/lab4-1/submit/ConvOperator.sv

```
    always @(posedge clk or negedge rst) begin
    if(~rst)begin
        state_reg<=RDATA;
    end else begin
        case(state_reg)
            RDATA:begin
                vector_stage2.valid <= 0;
                in_ready<=1'b1;
                out_valid<=1'b0;
                if(in_valid)begin
                    state_reg<=WORK;
                end
            end
            WORK:begin
                in_ready<=1'b0;
                out_valid<=1'b0;
                if(vector_stage1[Conv::LEN-1].valid)begin
                    for(j=Conv::LEN-1;j>=1;j=j-1)begin
                        if(j<Conv::LEN/2)begin
                            add_tmp[j]=add_tmp[j*2]+add_tmp[j*2+1];
                        end else begin
                        add_tmp[j]=vector_stage1_data[(j-Conv::LEN/2)*2]
                        +vector_stage1_data[(j-Conv::LEN/2)*2+1];//接上行
                        end
                    end
                    vector_stage2.data<=add_tmp[1];
                    vector_stage2.valid<=1'b1;
                    state_reg<=TDATA;
                end
            end
            TDATA:begin
                in_ready<=1'b0;
                out_valid<=1'b1;
                if(out_ready && vector_stage2.valid)begin
                    result<=vector_stage2.data;
                    state_reg<=RDATA;
                end
```

```
            end
            default:;
        endcase
    end
end
genvar i;
generate
    for(i=0;i<Conv::LEN;i=i+1)begin
        Multiplier #(
            .LEN(Conv::WIDTH)
        )mult1(
            .clk(clk),
            .rst(rst),
            .multiplicand(kernel.data[i]),
            .multiplier(data.data[i]),
            .product(vector_stage1_data[i]),
            .finish(vector_stage1[i].valid),
            .start(in_valid)
        );
    end
endgenerate
```

1.1.3 卷积模块设计

对于整个卷积模块，事实上只需要例化上述实现的移位寄存器和卷积运算单元即可，整个模块的 valid-ready 协议将由两个模块内部的协议实现保证，而对于两个模块的互相通信，我们引入了另一组 valid-ready 信号 *ex_valid*, *ex_ready* 作为外部通信信号，以下为实现代码的核心部分：

Code Snippet 1.1.4 ▶ src/lab4-1/submit/ConvUnit.sv

```
logic ex_valid;
logic ex_ready;
Shift shift1(
    .clk(clk),
    .rst(rst),
    .in_data(in_data),
    .in_valid(in_valid),
    .in_ready(in_ready),
    .data(data),
```

```
    .out_valid(ex_valid),
    .out_ready(ex_ready)
);
ConvOperator conv1(
    .clk(clk),
    .rst(rst),
    .kernel(kernel),
    .data(data),
    .in_valid(ex_valid),
    .in_ready(ex_ready),
    .result(result),
    .out_ready(out_ready),
    .out_valid(out_valid)
);
```

1.1.4 仿真上板

我们设计了如下的仿真代码，能够生成 16 组随机测试样例（每次改变一个数据），且每次运算的卷积核也进行随机生成。对于其为什么满足 valid-ready 协议将在下一章进行详细说明，以下为仿真代码：

Code Snippet 1.1.5 ▶ src/lab4-1/sim/testbench.sv

```
integer i,j,seed=1919810;
initial begin
    rst=0;
    #20;
    rst=1;
    for(i = 0; i < 16; i=i+1)begin
        for(j = 0; j < Conv::LEN; j=j+1)begin
            kernel.data[j] = {$random(seed),$random(seed)};
        end
        in_data = {$random(seed),$random(seed)};
        in_valid = 1'b1;
        #20;
        in_valid = 1'b0;
        #20;
        @(posedge out_valid)
    end
```

```
17      $display("success!!!");
18      $finish;
19  end
```

以下为仿真截图和部分上板图片：

```
simulate result: 0003a99de23c38df88c4f01788b649a0
simulate result: 00fab5fbf96b0013b1a9071765fbbdbf
simulate result: 1fd15fe325fe21ab98e50d5398dc9308
simulate result: 1adfda2d54e4f1f570e0ffbe652e0901
simulate result: da01e11950288daed0641c0730350a03
simulate result: fbf23a38f0d70fafeee6078df801a200
simulate result: 724f2ff6dad22ff31454602a4bd98026
simulate result: 0000356bff08a1b41f2070eb40200640
simulate result: af26b91f4a46f4971dc8ec4088ab5006
simulate result: c01ff6b33e60441e78416d98f8018028
simulate result: b63518042b8a4c0588fb490813762201
simulate result: 206c0158988407812f980cb0ff800605
simulate result: f7045494ce0e9f5ed10d8e7f2204c401
simulate result: a38e1ff736e380192a68008d95b30052
simulate result: ff006a933d2145e6fe80edd800801880
success!!!
```

图 1.2: 仿真成功截图

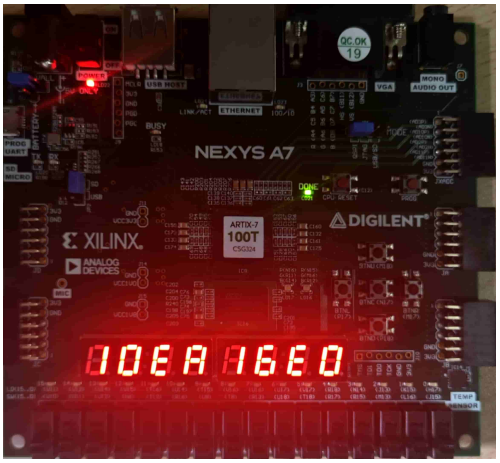

图 1.3: $n = 1,\ SW = 00$

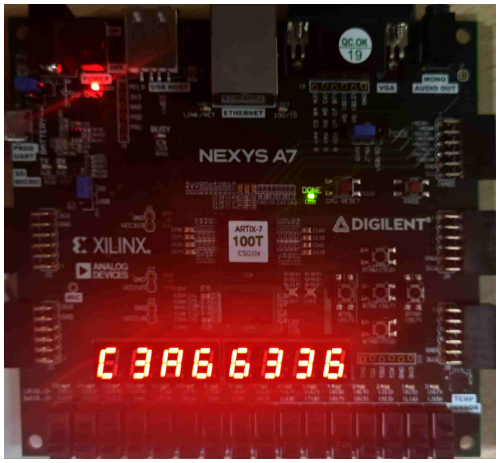

图 1.4: $n = 1,\ SW = 01$

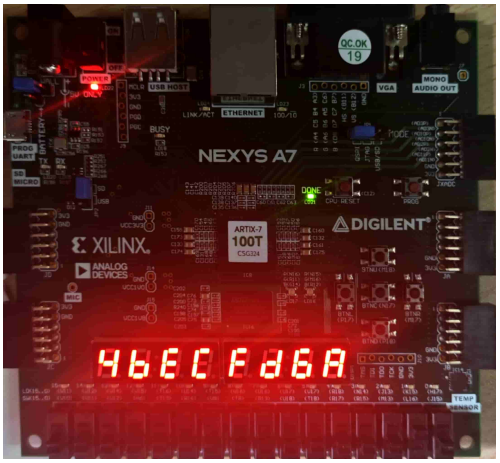

图 1.5: $n = 1,\ SW = 10$

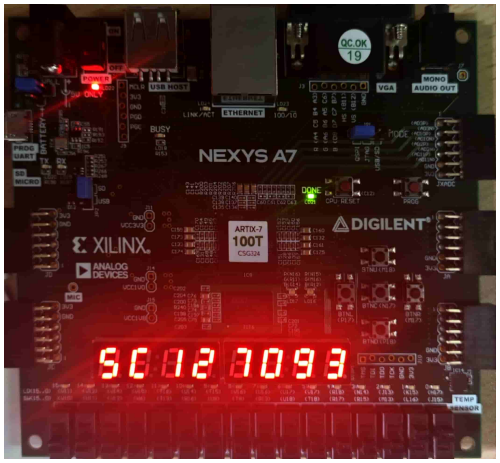

图 1.6: $n = 1,\ SW = 11$

1.2 握手协议解释

1.2.1 仿真样例

这里给出仿真样例设计的 valid-ready 信号波形图：

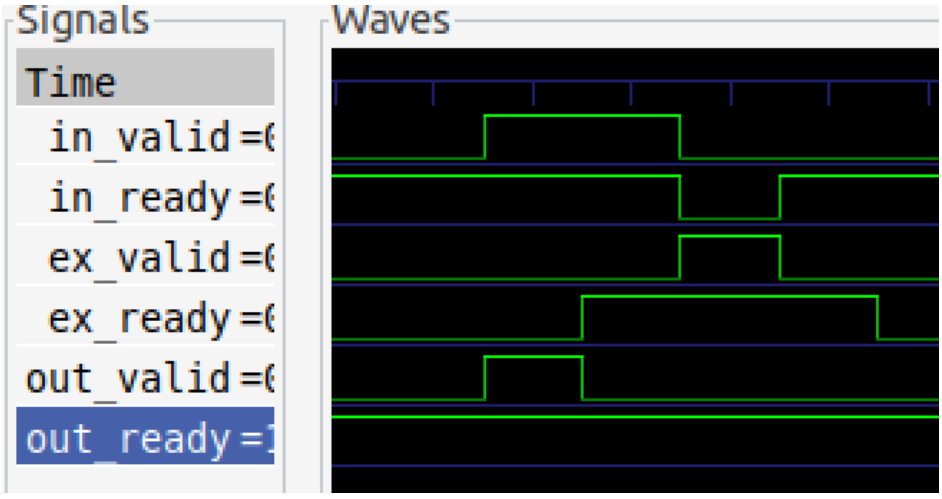

图 1.7: 握手信号波形图

对于每一组握手信号，都可以发现其高电平有重叠部分，其中重点是中间交互信号 *ex_valid*, *ex_ready* 可以发现两个模块通信符合 valid-ready 协议，一方面双方总是在高电平重合时进行数据交换，仿真成功代表这种数据交换正确。并且两者上升沿下降沿均不重合，没有互相依赖关系。

在我们的 *testbench.sv* 中，唯一相关信号是 *in_valid* ，通过设置延时切换高低电平使其实现使能效果且与 *in_ready* 没有依赖关系，而 @(*posedgeout*$_v$*alid*) 是为了避免延时设置导致结果在终端输出之前就已进入下一计算周期。

而对于 *out_ready* 长时间置高电平的原因是在仿真顶层模块中始终将其置高电平，而对于 *in_ready*，由于 *Shift* 模块只有两个工作状态，每次进入 *RDATA* 状态即将其置高电平，由于状态问题会造成其长期处于高电平状态。

而对于代码层面的协议符合可以参考前两章，在各个状态机中，对于传输等操作，我们总是在握手信号握手后才进行，并且并不会因为握手不成功导致程序陷入死循环，这是代码层面满足 valid-ready 的方式。

1.2.2 顶层结构

顶层结构使用了两个模块 *ConvUnit* 和 *DataGenerator*，对于前者我们在前面章节中给出了其组成模块代码，这能够说明其内部是满足 valid-ready 协议的，而对于后者，这里给出其部分代码：

Code Snippet 1.2.1 ▶ repo/sys-project/lab4-1/syn/DataGenerator.sv

```
always@(posedge clk or negedge rstn)begin
    if(~rstn) index <= 3'b111;
    else if(next_test) index <= index+3'b001;
end

always@(posedge clk or negedge rstn)begin
    if(~rstn) valid <= 1'b0;
    else if(next_test) valid <= 1'b1;
    else if(valid & ready) valid <= 1'b0;
end
```

根据此前写的代码不难发现这里同样是满足了 valid-ready 握手的要求，而整个顶层结构由这两个模块构成，互相均满足 valid-ready 协议，因此总的来说这个顶层结构也是满足 valid-ready 协议的。

1.3 BONUS：模块分割设计

Technique 1.3.1 ▶ 一个可能的优化方式

我认为这种分割类似于我们用 valid-ready 协议代替此前使用的 start-finish 协议的原因，即我们希望整个处理过程的并行性。因为通过 case 方式实现的有限状态机其本质是类似于串行的，而在进行了模块化分割之后我们能够提升不同模块之间运行的并行性，从而达到性能上的优化，这或许还有些类似于组合逻辑和时序逻辑的区别。

基于这个想法，我可能会把 *ConvOperator* 的数据处理部分和传输部分分离，即将现有状态机中的 *WORK* 状态与乘法器进行合并，然后将状态机转为一个单纯的传输模块，两者通过 valid-ready 协

议进行传输。在 *ConvUnit* 中我们事实上也在进行类似的操作，*Shift* 充当的是对于输入数据的处理，*ConvOperator* 则负责计算数据并传输。

而对于性能上的提升，在不添加异步传输缓冲队列（Lab 4-2）等模块情况下，我目前能想到的就只有协议改进导致的并行性增强。

2 实验 4-2：串口使用 (BONUS)

2.1 实验过程

2.1.1 代码设计

我在回环测试中设计了这样的一个有限状态机：

1. RDATA: 接收数据阶段，按照握手协议进行数据从 *uart_rdata* 到 *rdata* 的传输，握手成功后传输并进入下一阶段
2. TDATA1: 数据传输阶段一，按照握手协议从 *tdata* 到 *rdata* 的传输，握手成功后传输并进入下一阶段，原本设计的根据 *debug_send* 来确定传输 *debug_data* 还是 *tdata*，但是最后硬件实现上发现这样的代码似乎不能实现预期效果，遂作罢
3. TDATA3: 数据传输阶段二，进行最后一步传输，握手后传输信号并回到最开始阶段

以下为状态机代码

Code Snippet 2.1.1 ▶ src/lab4-2/submit/UartLoop.sv

```
always @(posedge clk or negedge rstn) begin
    if(debug_send)begin
        debug<=1'b1;
    end
    if(~rstn)begin
        rdata_valid<=1'b0;
        tdata_valid<=1'b0;
        rdata<=0;
        tdata<=0;
        debug<=1'b0;
        uart_rdata.ready<=1'b0;
        uart_tdata.valid<=1'b0;
        state_reg<=RDATA;
    end else begin
        case(state_reg)
            RDATA:begin
                uart_rdata.ready<=1'b1;
```

```
                uart_tdata.valid<=1'b0;
                if(uart_rdata.valid && uart_rdata.ready)begin
                    rdata<=uart_rdata.data;
                    rdata_valid<=1'b1;
                    state_reg<=TDATA1;
                end
            end
            TDATA1:begin
                uart_rdata.ready<=1'b0;
                if(debug)begin
                    tdata<=debug_data;
                    tdata_valid<=1'b1;
                    state_reg<=TDATA2;
                end else if(rdata_valid)begin
                    tdata<=rdata;
                    tdata_valid<=1'b1;
                    state_reg<=TDATA2;
                end
            end
            TDATA2:begin
                if(tdata_valid && uart_tdata.ready)begin
                    uart_tdata.data<=tdata;
                    tdata_valid<=1'b0;
                    uart_tdata.valid<=1'b1;
                    state_reg<=RDATA;
                end
            end
            default:state_reg<=RDATA;
        endcase
    end
end
```

2.1.2 仿真上板

本次没有仿真代码需要设计，以下为仿真成功截图和上板图：

```
make[1]: Leaving directory '/home/forever/sys1-sp24/src/lab4-2/build/verilate'
cd /home/forever/sys1-sp24/src/lab4-2/build/verilate; ./Testbench
receive data c4
receive data 9c
receive data 02
transmit data c4
receive data e4
transmit data 9c
receive data 78
transmit data 02
receive data bc
transmit data e4
receive data b6
transmit data 78
receive data e4
transmit data bc
receive data b7
transmit data b6
receive data 53
transmit data e4
success!!!
```

图 2.1: 仿真成功截图

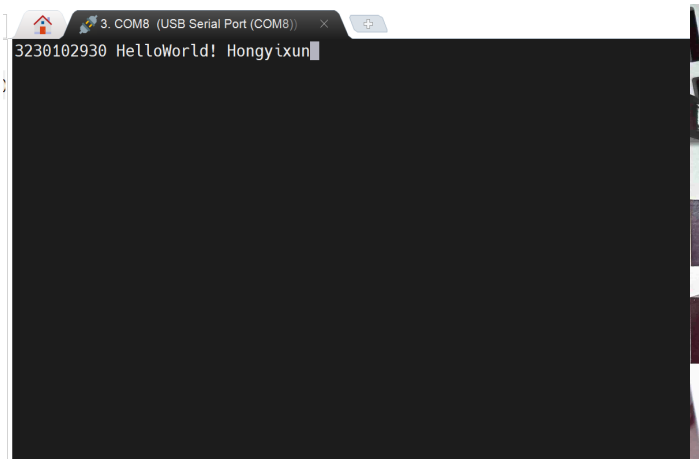

图 2.2: 串口通话截图

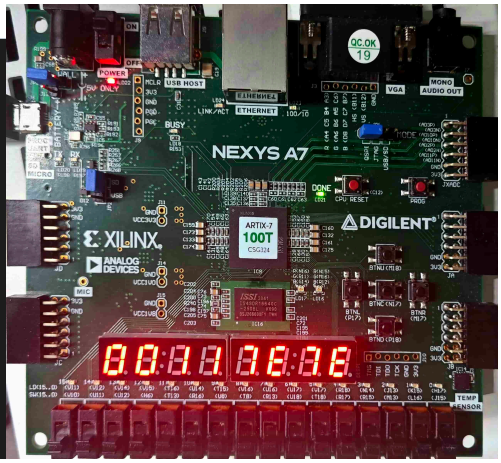

图 2.3: $S[1:0] = 00$

2.2 状态机设计

按照 *asyc_transmitter* 的工作逻辑，我设计了这样的有限状态机：

1. IDLE: 初始化状态，将 *TxD_busy* 置低电平表示可以传输数据，等待使能信号 *TxD_start* 后进入传输状态

2. TRANSMIT: 传输状态，从低到高逐位传输数据，同时使用计数器在传输 8 位后转入停止阶段，传输期间。*TxD_busy* 信号需要置高电平，表示无法传输新数据。

3. STOP: 停止阶段，发送两位停止位后回到 IDLE 状态。

这样文字化的状态机也同时大致描述了异步传输的工作流程，即在每组 *TxD_start* 信号间总是连续传输自身数据而不传输新数据。

2.3 减少毛刺干扰方法

参考此前学习的按键去抖器和边沿采样，我们可以做出类似的操作：

1. 调整 *async_transmitter* 发送数据的间隔，例如每位信号连续发送两个周期，而在 *asyn_receiver* 端则只需要检查输入信号是否在两个信号内绝大多数时间保持不变并取其值就可以筛去大多数毛刺，如果发送时间更长可以更大程度减少误差，实现去抖。

2. 调整 *async_transmitter* 发送数据的间隔，例如每位信号间间隔一个时钟周期，而在 *asyn_receiver* 端则只需要引入边沿采样就可以避免瞬间毛刺造成的影响