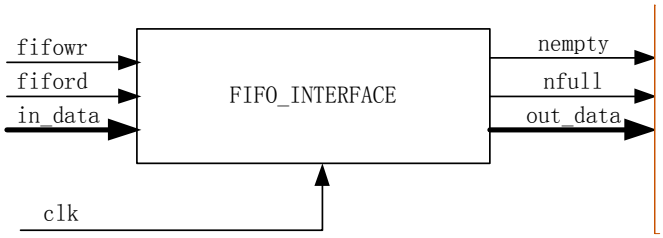
```
     end

always @(Data)
  if (WrSram)
    begin
         #TDVEH;
         if (SRW)
           begin
                WR_flag=0;
                $display("ERROR! Can't write!
                        Chip enable Data setup time is too short!");
           end
    end

always @(posedge SRW )
  begin
       #TWHDX;          //Data hold time
       if(DelayData !== Data)
        $display("Warning!  Data hold time is too short!");
  end

always @(DelayAddr or DelayData or WrSramDly)
  if (WrSram &&WR_flag)
    begin
         if(!Addr[5])
           begin
                #15 SramMem[Addr]=Data;
               // $display("mem[%h]=%h",Addr,Data);
                $fwrite(file,"mem[%h]=%h    ",Addr,Data);
                if(Addr[0]&&Addr[1])  $fwrite(file,"\n");
           end
         else
           begin
                $fclose(file);
                $display("Please check the txt.");
                $stop;
           end
    end

endmodule
```

练习十二　利用 SRAM 设计一个 FIFO

在本练习中，要求同学利用练习十一中提供的 SRAM 模型，设计 SRAM 读写控制逻辑，使 SRAM 的行为对用户表现为一个 FIFO（先进先出存储器）。

1）**设计要求：**

本练习要求同学设计的 FIFO 为同步 FIFO，即对 FIFO 的读/写使用同一个时钟。该 FIFO 应当提供用户读使能（fiford）和写使能(fifowr)输入控制信号，并输出指示 FIFO 状态的非空（nempty）和非满（nfull）信号，FIFO 的输入、输出数据使用各自的数据总线：in_data 和 out_data。下图为 FIFO 接口示意图。

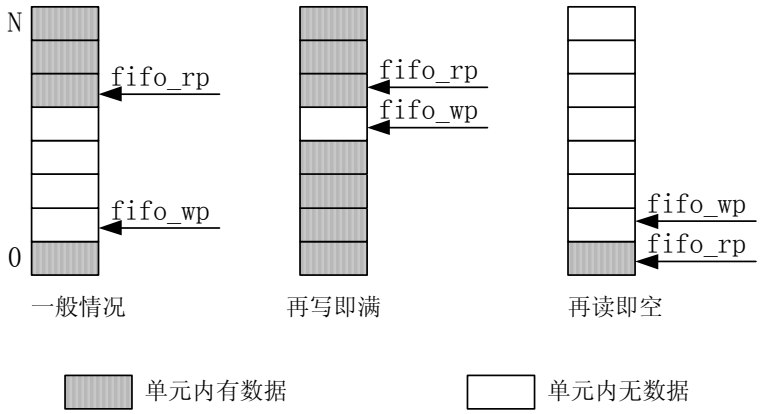


批注 [fwn1]: 这些不是设计的具体内容，而是为检验设计正确与否所提供的验证环境。此处如描述一下 SRAM 与 FIFO 的差异，并由此得到 FIFO 接口设计的关键在与 SRAM 地址产生这一结论会好一些。

2）FIFO 接口的设计思路

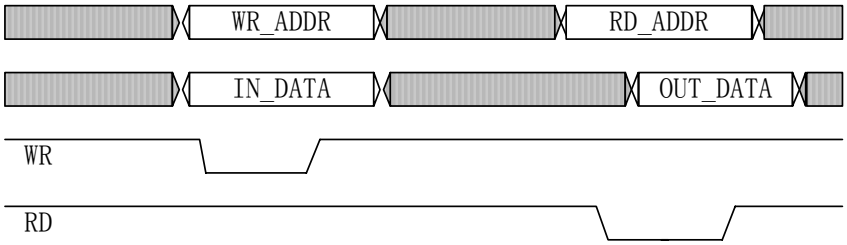
FIFO 的数据读写操作与 SRAM 的数据读写操作基本上相同，只是 FIFO 没有地址。所以用 SRAM 实现 FIFO 的关键点是如何产生正确的 SRAM 地址。

我们可以借用软件中的方法，将 FIFO 抽象为环形数组，并用两个指针：读指针（fifo_rp）和写指针（fifo_wp）控制对该环形数组的读写。其中，读指针 fifo_rp 指向下一次读操作所要读取的单元，并且每完成一次读操作，fifo_rp 加一；写指针 fifo_wp 则指向下一次写操作时存放数据的单元，并且每完成一次写操作，fifo_wp 加一。由 fifo_rp 和 fifo_wp 的定义易知，当 FIFO 被读空或写满后，fifo_rp 和 fifo_wp 将指向同一单元，但在读空和写满之前 FIFO 的状态是不同的，所以如果能区分这两种状态，再通过比较 fifo_rp 和 fifo_wp 就可以得到 nempty 和 nfull 信号了。下图为 FIFO 工作状态的示意。



在得到 nfull 和 nempty 信号后，就需要考虑如何应用这两个信号来控制对 FIFO 的读写，使得 FIFO 在被写满后不能再写入，从而防止覆盖原有数据，并且在被读空后也不能再进行读操作，防止读取无效数据。

此外，在进 SRAM 读写操作时，应该注意建立地址、数据和控制信号的先后顺序。一般情况下，希望对 SRAM 读写的波形时序如下图所示：



即写 SRAM 时，先建立地址和数据，然后置写使能信号 WR 有效，在 WR 保持有效一定时间后，先复位 WR，然后释放地址和数据总线。而读取 SRAM 时，则先建立地址，然后置读使能 RD 有效，在 RD 维持有效一定时间后，复位 RD，同时读取数据总线上的值，然后再释放地址总线。在进行 FIFO 操作时，用户一般希望除了没有地址外，其它三个信号的时序关系能保持不变。请同学们在设计 FIFO 控制信号与 SRAM 控制信号间逻辑关系时注意这一点。

3）FIFO 接口的测试

在完成一个设计后，需要进行测试以确认设计的正确性和完整性。而要进行测试，就需要编写测试激励和结果检查程序，即测试平台（testbench）。在某些情况下，如果设计的接口能够预先确定，测试平台的编写也可以在设计完成之前就进行，这样做的好处是在设计测试平台的同时也在更进一步深入了解设计要求，有助于理清设计思路，及时发现设计方案的错误。

编写测试激励时，除了注意对实际可能存在的各种情况的覆盖外，还要有意针对非正常情况下的操作进行测试。在本练习中，就应当进行在 FIFO 读空后继续读取、FIFO 写满后继续写入、FIFO 复位后马上读取等操作的测试。

测试激励中通常会有一些复杂操作需要反复进行，如本练习中对 FIFO 的读写操作。这时可以将这些复杂操作纳入到几个 task 中，即减小了激励编写的工作量，也使得程序的可读性更好。

下面的测试程序给同学们做为参考，希望同学们能先用这段程序测试所设计的 FIFO 接口，然后编写自己更全面的测试程序。

```
`define FIFO_SIZE 8
`include “sram.v”       // 有的仿真工具不需要加这句, 只要 sram.v 模块编译过就可以了
`timescale 1ns/1ns

module t;

reg [7:0]        in_data;                  //FIFO 数据总线
reg                fiford,fifowr;          //FIFO 控制信号

wire[7:0]         out_data;
```

```
wire                nfull, nempty;          //FIFO 状态信号

reg                 clk,rst;

wire[7:0]           sram_data;              //SRAM 数据总线
wire[10:0]          address;                //SRAM 的地址总线
wire                rd,wr;                  //SRAM 读写控制信号

reg [7:0]           data_buf[`FIFO_SIZE:0]; //数据缓存，用于结果检查
integer index;                              //用于读写 data_buf 的指针

//系统时钟
initial    clk=0;
always     #25 clk=~clk;

//测试激励序列
initial
    begin
        fiford=1;
        fifowr=1;
        rst=1;
        #40 rst=0;
        #42 rst=1;

        if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");

        //连续写 FIFO
        index = 0;
        repeat(`FIFO_SIZE) begin
            data_buf[index]=$random;
            write_fifo(data_buf[index]);
            index = index + 1;
        end

        if (nfull) $display($time,"Error: FIFO full, nfull should be low.\n");
        repeat(2) write_fifo($random);
        #200

        //连续读 FIFO
        index=0;
        read_fifo_compare(data_buf[index]);
        if (~nfull) $display($time,"Error: FIFO not full, nfull should be high.\n");

        repeat(`FIFO_SIZE-1) begin
```

```
            index = index + 1;
            read_fifo_compare(data_buf[index]);
        end

        if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");

        repeat(2) read_fifo_compare(8'bx);

        reset_fifo;

        //写后读 FIFO
        repeat(`FIFO_SIZE*2)
        begin
            data_buf[0] = $random;
            write_fifo(data_buf[0]);
            read_fifo_compare(data_buf[0]);
        end

        //异常操作
        reset_fifo;
        read_fifo_compare(8'bx);
        write_fifo(data_buf[0]);
        read_fifo_compare(data_buf[0]);

        $stop;
    end

fifo_interface    fifo_interface(
                      .in_data(in_data),.out_data(out_data),
                      .fiford(fiford),.fifowr(fifowr),
                      .nfull(nfull),.nempty(nempty),
                      .address(address),.sram_data(sram_data),
                      .rd(rd),.wr(wr),
                      .clk(clk),.rst(rst)
        );

sram m1( .Address(address),
            .Data(sram_data),
            .SRG(rd),                //SRAM 读使能
            .SRE(1'b0),              //SRAM 片选,低有效
            .SRW(wr));               //SRAM 写使能

task write_fifo;
input [7:0] data;
```

```
begin
            in_data=data;
    #50     fifowr=0;                   //往 SRAM 中写数
    #200    fifowr=1;
    #50;
end
endtask

task read_fifo_compare;
input [7:0] data;
begin
    #50     fiford=0;                   //从 SRAM 中读数
    #200    fiford=1;
            if (out_data != data)
                $display($time,"Error: Data retrieved (%h) not match the one stored (%h). \n",
                            out_data, data);

    #50;
end
endtask

task reset_fifo;
begin
    #40 rst=0;
    #40 rst=1;
end
endtask

endmodule
```

4) FIFO 接口的参考设计

FIFO 接口的实现有多种方案，下面给出的参考设计只是其中一种。希望同学们在完成自己的设计后，和参考设计做一下比较。

```
`define SRAM_SIZE 8    //为减小对 FIFO 控制器的测试工作量,置 SRAM 空间为 8Byte
`timescale 1ns/1ns

module fifo_interface(
                in_data,       //对用户的输入数据总线
                out_data,      //对用户的输出数据总线,
                fiford,        //FIFO 读控制信号，低电平有效
                fifowr,        //FIFO 写控制信号，低电平有效
                nfull,
```

```
                nempty,

                address,      //到 SRAM 的地址总线
                sram_data,    //到 SRAM 的双向数据总线
                rd,           //SRAM 读使能，低电平有效
                wr,           //SRAM 写使能，低电平有效

                clk,          //系统时钟信号
                rst);         //全局复位信号，低电平有效

//来自用户的控制输入信号
input           fiford, fifowr, clk, rst;

//来自用户的数据信号
input[7:0]      in_data;
output[7:0]     out_data;


reg[7:0]        in_data_buf,          //输入数据缓冲区
                out_data_buf;         //输出数据缓冲区

//输出到用户的状态指示信号
output          nfull, nempty;
reg             nfull, nempty;

//输出到 SRAM 的控制信号
output          rd, wr;

//到 SRAM 的双向数据总线
inout[7:0]      sram_data;

//输出到 SRAM 的地址总线
output[10:0]    address;
reg[10:0]       address;

//Internal Register
reg[10:0]       fifo_wp,              //FIFO 写指针
                fifo_rp;              //FIFO 读指针

reg[10:0]       fifo_wp_next,         //fifo_wp 的下一个值
                fifo_rp_next;         //fifo_rp 的下一个值

reg             near_full, near_empty;
```

```
reg[3:0]        state;                  //SRAM 操作状态机寄存器

parameter       idle            = 'b0000,
                read_ready      = 'b0100,
                read            = 'b0101,
                read_over       = 'b0111,
                write_ready     = 'b1000,
                write           = 'b1001,
                write_over      = 'b1011;

//SRAM 操作状态机
always @(posedge clk or negedge rst)
    if (~rst)
        state <= idle;
    else
        case(state)
            idle:                                   //等待对 FIFO 的操作控制信号
                if (fifowr==0 && nfull)     //用户发出写 FIFO 申请,且 FIFO 未满
                    state<=write_ready;
                else if(fiford==0 && nempty)//用户发出读 FIFO 申请,且 FIFO 未空
                    state<=read_ready;
                else                                //没用对 FIFO 操作的申请
                    state<=idle;

            read_ready:                             //建立 SRAM 操作所需地址和数据
                    state <= read;

            read:                                   //等待用户结束当前读操作
                if (fiford == 1)
                    state <= read_over;
                else
                    state <= read;

            read_over:                              //继续给出 SRAM 地址以保证数据稳定
                    state <= idle;

            write_ready:                            //建立 SRAM 操作所需地址和数据
                    state <= write;

            write:                                  //等待用户结束当前写操作
                if (fifowr == 1)
                    state <= write_over;
                else
                    state <= write;
```

```
            write_over:                 //继续给出 SRAM 地址和写入数据以保证数据稳定
                        state <= idle;

            default: state<=idle;
    endcase

//产生 SRAM 操作相关信号
assign rd = ~state[2];          //state 为 read_ready 或 read 或 read_over
assign wr = (state == write) ? fifowr : 1'b1;

always @(posedge clk)
    if (~fifowr)
        in_data_buf <= in_data;

assign sram_data = (state[3]) ? //state 为 write_ready 或 write 或 write_over
                            in_data_buf : 8'hzz;


always @(state or fiford or fifowr or fifo_wp or fifo_rp)
    if (state[2] || ~fiford)
        address = fifo_rp;
    else if (state[3] || ~fifowr)
        address = fifo_wp;
    else
        address = 'bz;

//产生 FIFO 数据
assign out_data = (state[2]) ?
                            sram_data : 8'bz;

always @(posedge clk)
    if (state == read)
        out_data_buf <= sram_data;


//计算 FIFO 读写指针
always @(posedge clk or negedge rst)
    if (~rst)
        fifo_rp <= 0;
    else if (state == read_over)
        fifo_rp <= fifo_rp_next;

always @(fifo_rp)
```

```
    if (fifo_rp == `SRAM_SIZE-1)
        fifo_rp_next = 0;
    else
        fifo_rp_next = fifo_rp + 1;

always @(posedge clk or negedge rst)
    if (~rst)
        fifo_wp <= 0;
    else if (state == write_over)
        fifo_wp <= fifo_wp_next;

always @(fifo_wp)
    if (fifo_wp == `SRAM_SIZE-1)
        fifo_wp_next = 0;
    else
        fifo_wp_next = fifo_wp + 1;


always @(posedge clk or negedge rst)
    if (~rst)
        near_empty <= 1'b0;
    else if (fifo_wp == fifo_rp_next)
        near_empty <= 1'b1;
    else
        near_empty <= 1'b0;

always @(posedge clk or negedge rst)
    if (~rst)
        nempty <= 1'b0;
    else if (near_empty && state == read)
        nempty <= 1'b0;
    else if (state == write)
        nempty <= 1'b1;

always @(posedge clk or negedge rst)
    if (~rst)
        near_full <= 1'b0;
    else if (fifo_rp == fifo_wp_next)
        near_full <= 1'b1;
    else
        near_full <= 1'b0;

always @(posedge clk or negedge rst)
    if (~rst)
```

```
        nfull <= 1'b1;
    else if (near_full && state == write)
        nfull <= 1'b0;
    else if (state == read)
        nfull <= 1'b1;

endmodule
```

参考资料：

[1] IEEE P1364.1 Draft Standard For Verilog Register Transfer Level Synthesis

[2] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995

[3] Clifford Cummings, "Correct Methods For Adding Delays To Verilog Behavioral Models," International HDL Conference 1999 Proceedings, pp. 23-29, April 1999.

[4] Mark Gordon Amald, "Verilog Digital Computer Design Algorithms into Hardware" 1999 by Prentice Hall PTR Prentice Hall Inc.

[5] Thomas D E, Moorby P R. "The Verilog Hardware Description Language." 2 second ed 1995 by Kluwer Academic Publishers

[6] Quick Works Version 7.1 User's Guide with SpDE Reference 1998

[7] CADENCE Version 9502 Verilog-XL Reference 1997

[8] Synplify- Lite Synthesis User's Guide 1997

[9] 刘宝琴 "数字电路与系统" 北京清华大学出版社，1993

[10]夏宇闻 "复杂数字电路与系统的 Verilog HDL 设计技术" 北京航空航天大学出版社，1998

作者编后记

三十三年前我从清华大学自动控制系计算技术与装置专业毕业时，国内数字逻辑电路设计刚刚才开始采用半导体电路。数字逻辑电路的分析和综合还是采用传统的方法：即先在纸上画真值表,做布尔代数化简，画波形图，画有限状态机流程图，画静态和动态卡诺图等方法来设计电路。在实验板上先用晶体三极管、二极管、电阻、电容等搭出门电路和触发器电路，做成线路板，测出各项参数。每块线路板上只有几个触发器和几个与、或、非门。再用这样的线路板来构成数字逻辑电路。以现在的眼光看，当时的数字逻辑系统是相当简陋的。做一个简单的可控制步进马达按照输入做 X 和 Y 方向的直线、斜线和圆弧运动的数字控制系统，就需要用 20 多块线路板，设计和调试这样的系统需要花费很长的时间和很多的精力。

二十年前国内逐步开始使用微处理机，中规模的集成电路的使用也逐步普及，大学里的电子和计算机类学科普遍开设了汇编语言课程和一些常用的中规模的集成电路的使用课程，大大缩短了开发复杂专用数字系统所需的时间。

近十年来，国外先进工业国家由于计算机电路辅助设计技术和半导体集成工艺技术的快速进步,在生产的电子系统中，专用集成电路(ASIC)和 FPGA 的使用越来越多，特别在先进的电讯设备、计算机系统和网络设备中更是如此。这不仅是因为有不少实时的 DSP（数字信号处理）芯片是一般微处理机所无法替代的，而且也因为市场对电子产品的要求越来越高。在电子设计和制造领域我们与国外先进国家的技术差距越来越大。

作为一名在大学讲授复杂专用数字系统设计课程的老师深深感到自己身上责任的重大。我个人觉得这与大学的课程设置和教学条件有关，因为我们没有及时把国外最先进的设计技术介绍给同学们，也没有给他们创造实践的机会。

1992 年我受沈校长和系领导的委托，与董金明老师一起筹建世行贷款的电路设计自动化(EDA)实验室。在有限的经费中，沈士团校长为我们挤出十五万美圆，其中三万美圆购买了一套 CADENCE 设计环境，其余的购买工作站和网络设备。其中 CADENCE 设计环境中数字设计部分由我负责。自 1995 年起，工作站和 CADENCE 软件逐步到货，由于经费有限我们没有机会到美国去学习，只好自己在工作站上一边看着参考手册一边学着干，先掌握了利用电路图输入的方法，再逐步掌握了利用 Verilog HDL 设计复杂数字电路的仿真和综合技术。在此基础上我们为有关单位设计了一万门左右的复杂数字电路，提供给他们经前后仿真验证的 Verilog HDL 源代码，得到很高的评价。 我们也为我们的科研项目，小波(Wavelet)图象压缩,设计了小波卷积器和改进零修剪树(EZW)算法(即 SPIHT 算法)的硬线逻辑的 Verilog HDL 模型，并成功地进行了仿真和综合，在 Altera 10k50 系列的 CPLD 上成功地布线和通过的后仿真，并制成了带 PCI 接口的电路板。近年来我们为航天部 501 所完成了多项五万门级以上的编码/解码和加密电路的设计都取得很好的效果。这一类设计很难用传统的电路图输入方法来设计的，这些设计的成功得益于我们对于 Verilog HDL 设计方法的掌握。

从 94 年拿到一些有关 Verilog HDL 的资料起，我就在我所讲授的研究生课程“复杂专用数字系统设计”中，逐步增加有关利用 Verilog HDL 进行复杂数字系统设计的内容。1996 年春，我受张凤言老师的邀请，到国家教委电路教学委员会召集的华北区讨论会上作了一个三小时的有关 EDA 和 HDL 设计方法的讲座。会后张凤言老师就一直鼓励我写一本有关 HDL 设计方法的书。当时我虽然逐渐学会了一些 Verilog HDL 的设计方法，但是很不系统，也找不到好的教材作参考，总觉得很难下手。1996 年春夏之交，校园网接通，我从 Internet 网络上找到一些网址，陆续找到一些有关 Verilog HDL 的素材，但好的完整教材和光盘需要上千美圆才能购得，而我们没有外汇，也无法去购买，这使我感到很沮丧。1996 年秋，我为

QuickLogic 公司的 FPGA 芯片和设计工具的讲座作翻译后，外商送我一套 QuickLogic 设计工具,可以在 PC586 平台上运行,这套工具包括电路图输入和 Verilog HDL 输入工具、Verilog HDL 仿真器、一个小巧的综合器(Synplify)等。这套工具价格并不贵，工作平台是 PC586 机，在光盘上还有一套比较完整而简单的教学资料。我仔细地阅读了这些资料并使用了这套工具后，觉得在大多数学校里推广 Verilog HDL 设计方法是很有可能的。从此我就更积极地从 Internet 网络上找一些有关 Verilog HDL 设计方法的资料片段和有代表性的样板程序为写一本 Verilog HDL 设计方法的入门书而作准备。1997 年由于教学的需要，经过近一年的努力，98 年夏在北航出版社出版了“复杂数字逻辑与系统的 Verilog HDL 设计技术与方法”。出版后我总觉得有许多问题还没有叙述清楚,特别在算法系统的总体结构考虑和组成上与语法没有任何联系，状态机的概念引入也太突然，因此做了一些大的改动，补写了与电路结构有关的章节。加上了第一章，引入算法硬件实现的概念，也补充了较完整的 Verilog 语法作为附录可供设计参考，还加上一个上机练习的十个阶段的作业，便于同学通过自学掌握。

由于科研和实验室的各项工作很忙，我只能利用零碎时间，在我的研究生帮助下一点一点地把材料输入到计算机中并逐步加以整理。到现在两年又过去了，书总算初步有了一些新的面貌。我们使用 Verilog HDL 设计复杂数字逻辑电路已有近五年的时间，虽积累了一些经验，但水平并不高，书中谬误之处在所难免，敬请读者及时把意见反馈给我。我之所以在原学校出版的教材的基础上把这本新书推出，并起名为《从算法到硬线逻辑的实现-*复杂数字逻辑与系统的 Verilog HDL 设计技术与方法*》， 目的是想把我们在近两年来在 Verilog HDL 教学和设计方法上积累的一些新经验与读者分享,并把旧版本中许多没讲清楚的概念尽量阐述明白些，把系统设计的主要思路连贯起来。在大学生和研究生中加快 Verilog HDL 设计技术的推广，尽快培养一批掌握先进设计技术的跨世纪的人才。期望本书能在这一过程中起到抛砖引玉的作用。

读者如果配合一套可在 PC586 平台上运行的 Verilog HDL 仿真和综合工具，如 QuickLogic 公司的 Spade 的教学软件包(Verilog HDL 版)、Mentor 公司的 Modelsim 等开发环境下，只要带 Verilog HDL 仿真器(如 Verilog-XL)和综合器（如 Synplify），就可以运行本书所有的 Verilog HDL 程序，可在这一环境下通过做各种仿真和综合的练习，学会并掌握 Verilog HDL 设计技术，把设计思想逐步转变为万门级的具体的电路。在掌握了基本设计技术后再购买如 CADENCE、SYNOPISYS 等高级的设计环境，在设计几十万门以上电路时就容易成功，为进入 ASIC 设计做充分的准备，不会造成浪费。希望读者能通过电子邮件跟我交流设计的心得和经验，有条件时我将在北航的网页上开设 Verilog HDL 设计经验交流角，共同来促进这一新设计技术的成长和发展。

编者
2000 年 8 月 30 日
于北京航空航天大学 EDA 实验室

电子邮箱：xyw@dept2.buaa.edu.cn
通信地址：北京 100083 北京航空航天大学 205 信箱 夏宇闻
