lab4

lab4-1

实验目的

- 学习使用 SystemVerilog 的 struct、package 等高级语法
- 学习掌握卷积操作的设计和实现
- 学习 valid-ready 的握手协议

实验环境

- 操作系统：Windows 10+ 22H2，Ubuntu 22.04+
- VHDL：Verilog，SystemVerilog

实验过程

代码实现卷积仿真

顶层逻辑，Convunit

```
wire shift_valid;
    Conv::data_vector data_vec;
    Shift shift(
        .clk(clk),
        .rst(rst),
        .in_data(in_data),
        .in_valid(in_valid),
        .in_ready(in_ready),
        .data(data_vec),
        .out_valid(shift_valid),
        .out_ready(1'b1)
    );
    reg shift_valid_d;
    Conv::data_vector kernel_latched;
    wire conv_ready;
    wire [127:0]conv_result;
    always @(posedge clk or posedge rst)begin
        if(rst)begin
            shift_valid_d<=1'b0;
            kernel_latched<='{default: '0};
        end else begin
            shift_valid_d<=shift_valid;
            if(shift_valid)begin
                kernel_latched<=kernel;
            end
```

```
        end
    end
    ConvOperator conv(
        .clk(clk),
        .rst(rst),
        .kernel(kernel_latched),
        .data(data_vec),
        .in_valid(shift_valid_d),
        .in_ready(conv_ready),
        .result(conv_result),
        .out_valid(out_valid),
        .out_ready(out_ready)
    );
    assign result=conv_result;
```

- 相当于是先进行shift读入数据，再进行operate的数据的操作。
- 我的理解是这个的一个top文件
  Convoperator

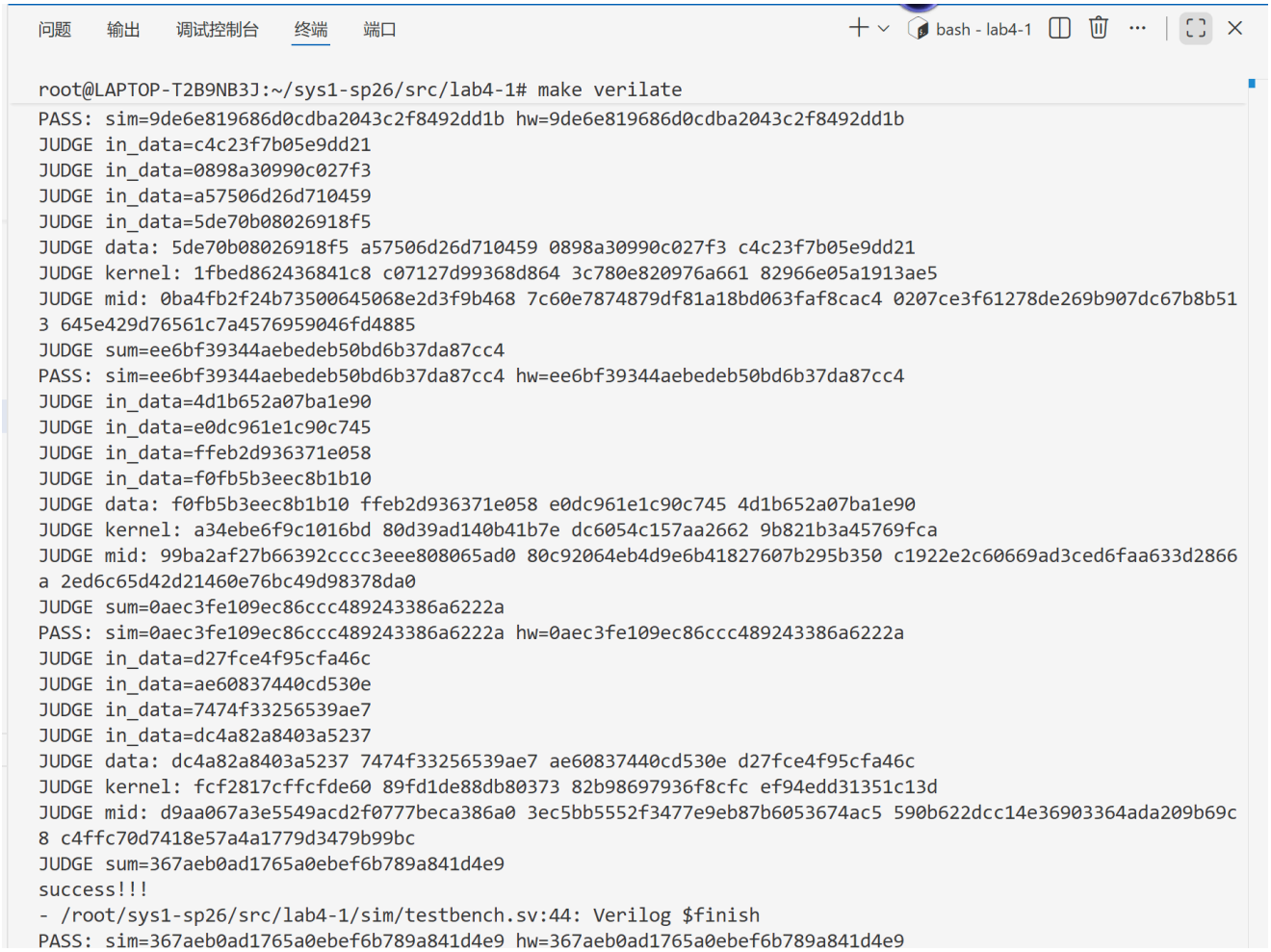
```
    wire [Conv::WIDTH*2-1:0]mul_out [0:3];
    wire [3:0]mul_inish;
    reg  mul_start;
    Conv::data_vector data_reg;
    Conv::data_vector kernel_reg;
    always @(posedge clk or posedge rst)begin
        if(rst)begin
            in_ready<=1'b1;
            data_reg<='{default: '0};
            kernel_reg <='{default: '0};
        end else begin
            in_ready <=1'b1;
            if(in_valid&&in_ready) begin
                data_reg<=data;
                kernel_reg<=kernel;
            end
        end
    end
    always @(posedge clk) begin
        if(in_valid) begin
            mul_start<=1'b1;
        end else begin
            mul_start<=1'b0;
        end
    end

    genvar m;
    generate
        for(m=0;m<4;m=m+1) begin: gen_mult
```

```
            Multiplier #(.LEN(Conv::WIDTH))mul(
                .clk(clk),
                .rst(rst),
                .multiplicand(data_reg.data[m]),
                .multiplier(kernel_reg.data[m]),
                .start(mul_start),
                .product(mul_out[m]),
                .finish(mul_finish[m])
            );
        end
    endgenerate
    wire [129:0]sum_full;
    assign sum_full={2'b0,mul_out[0]}+{2'b0,mul_out[1]}+{2'b0,mul_out[2]}+
{2'b0,mul_out[3]};
    always @(posedge clk or posedge rst)begin
        if(rst) begin
            out_valid<=1'b0;
            result<='0;
        end
        else begin
            if(&mul_finish)begin
                result<=sum_full[127:0];
                out_valid<=1'b1;
            end
            if(out_valid&&out_ready) begin
                out_valid<=1'b0;
            end
        end
    end
endmodule
```

- 这个模块实现的是将收到的data数据先进行乘积的运算，再将data中的结果进行加法合并，再转存到result向量中。

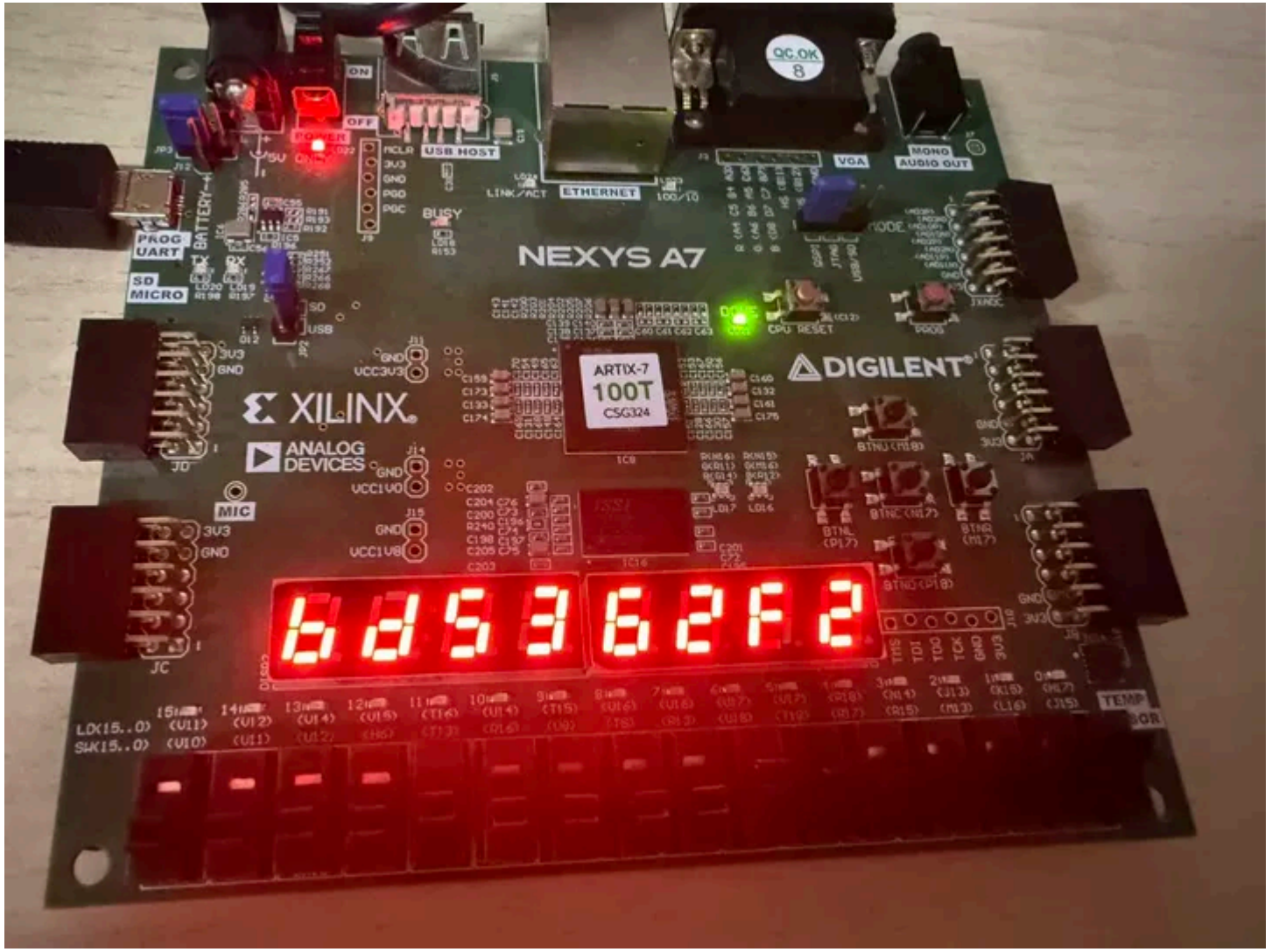
实现success



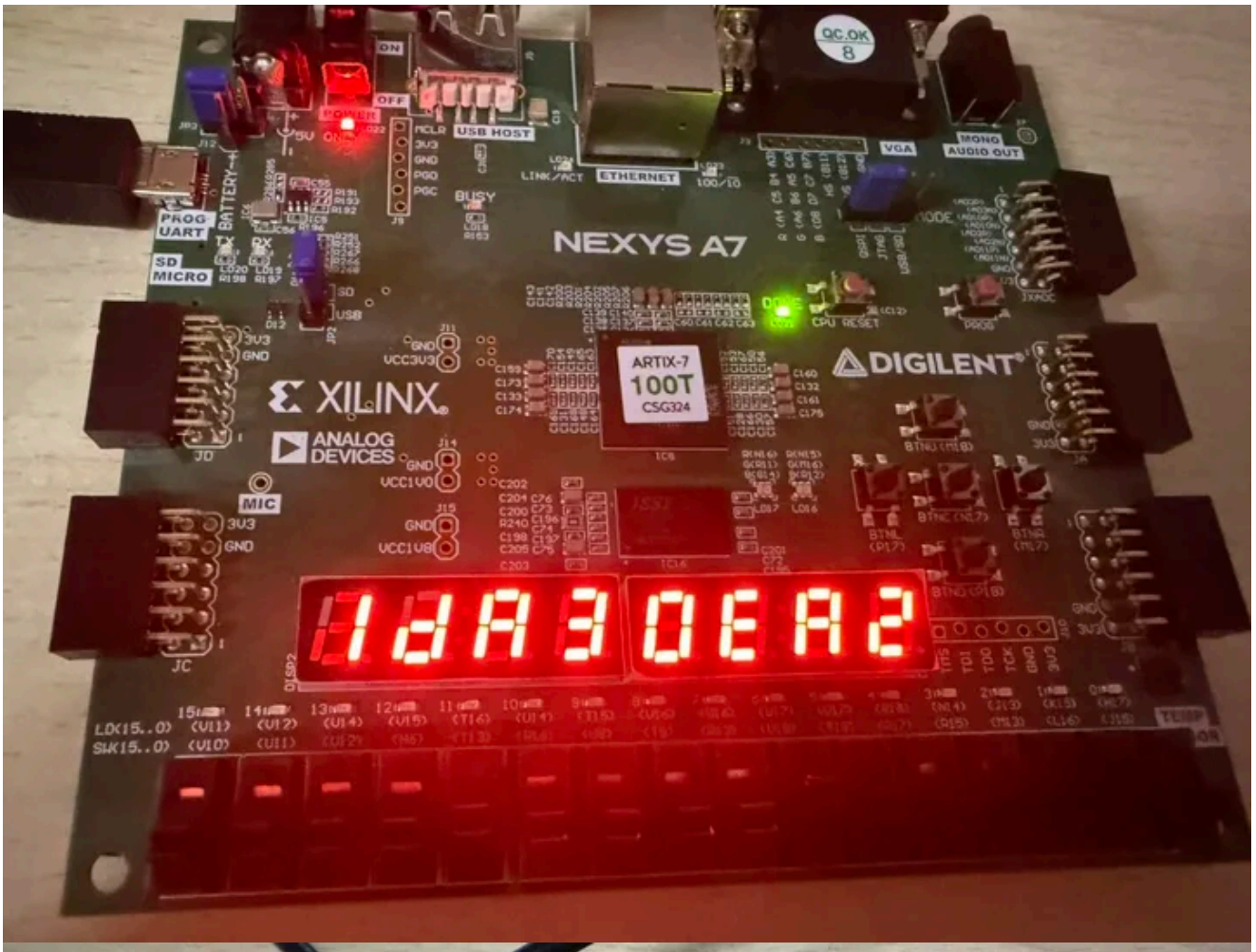

```
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab4-1# make verilate
PASS: sim=9de6e819686d0cdba2043c2f8492dd1b hw=9de6e819686d0cdba2043c2f8492dd1b
JUDGE in_data=c4c23f7b05e9dd21
JUDGE in_data=0898a30990c027f3
JUDGE in_data=a57506d26d710459
JUDGE in_data=5de70b08026918f5
JUDGE data: 5de70b08026918f5 a57506d26d710459 0898a30990c027f3 c4c23f7b05e9dd21
JUDGE kernel: 1fbed862436841c8 c07127d99368d864 3c780e820976a661 82966e05a1913ae5
JUDGE mid: 0ba4fb2f24b73500645068e2d3f9b468 7c60e7874879df81a18bd063faf8cac4 0207ce3f61278de269b907dc67b8b51
3 645e429d76561c7a4576959046fd4885
JUDGE sum=ee6bf39344aebedeb50bd6b37da87cc4
PASS: sim=ee6bf39344aebedeb50bd6b37da87cc4 hw=ee6bf39344aebedeb50bd6b37da87cc4
JUDGE in_data=4d1b652a07ba1e90
JUDGE in_data=e0dc961e1c90c745
JUDGE in_data=ffeb2d936371e058
JUDGE in_data=f0fb5b3eec8b1b10
JUDGE data: f0fb5b3eec8b1b10 ffeb2d936371e058 e0dc961e1c90c745 4d1b652a07ba1e90
JUDGE kernel: a34ebe6f9c1016bd 80d39ad140b41b7e dc6054c157aa2662 9b821b3a45769fca
JUDGE mid: 99ba2af27b66392cccc3eee808065ad0 80c92064eb4d9e6b41827607b295b350 c1922e2c60669ad3ced6faa633d2866
a 2ed6c65d42d21460e76bc49d98378da0
JUDGE sum=0aec3fe109ec86ccc489243386a6222a
PASS: sim=0aec3fe109ec86ccc489243386a6222a hw=0aec3fe109ec86ccc489243386a6222a
JUDGE in_data=d27fce4f95cfa46c
JUDGE in_data=ae60837440cd530e
JUDGE in_data=7474f33256539ae7
JUDGE in_data=dc4a82a8403a5237
JUDGE data: dc4a82a8403a5237 7474f33256539ae7 ae60837440cd530e d27fce4f95cfa46c
JUDGE kernel: fcf2817cffcfde60 89fd1de88db80373 82b98697936f8cfc ef94edd31351c13d
JUDGE mid: d9aa067a3e5549acd2f0777beca386a0 3ec5bb5552f3477e9eb87b6053674ac5 590b622dcc14e36903364ada209b69c
8 c4ffc70d7418e57a4a1779d3479b99bc
JUDGE sum=367aeb0ad1765a0ebef6b789a841d4e9
success!!!
- /root/sys1-sp26/src/lab4-1/sim/testbench.sv:44: Verilog $finish
PASS: sim=367aeb0ad1765a0ebef6b789a841d4e9 hw=367aeb0ad1765a0ebef6b789a841d4e9
```

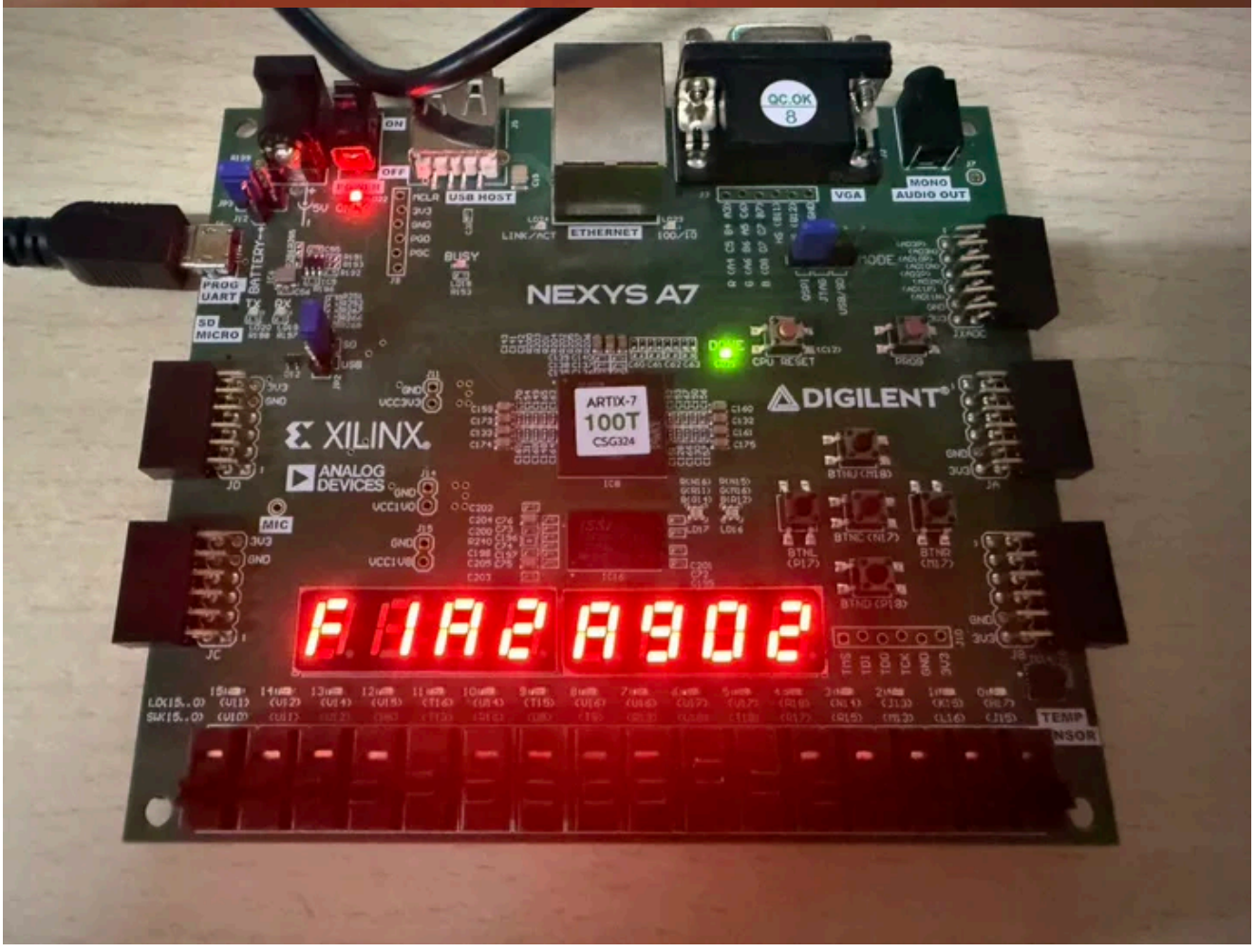
综合实现上板

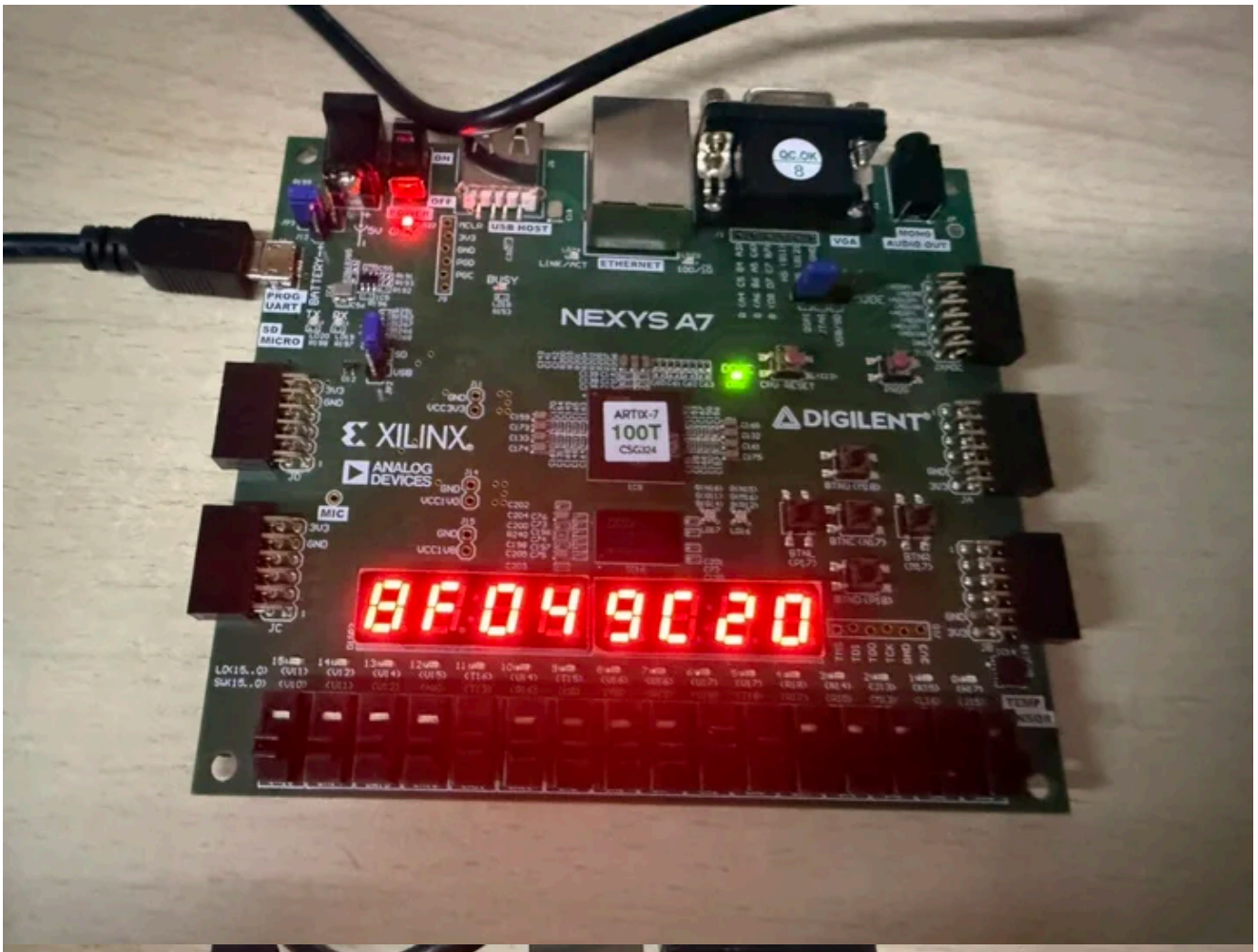
NEXYS A7
XILINX
ANALOG DEVICES
DIGILENT
ARTIX-7
100T
CSG324
ETHERNET
USB HOST
VGA
MONO AUDIO OUT
PMOD UART
SD MICRO
BATTERY
CPU RESET
QC OK
8
6 3 8 F C 6 D 0

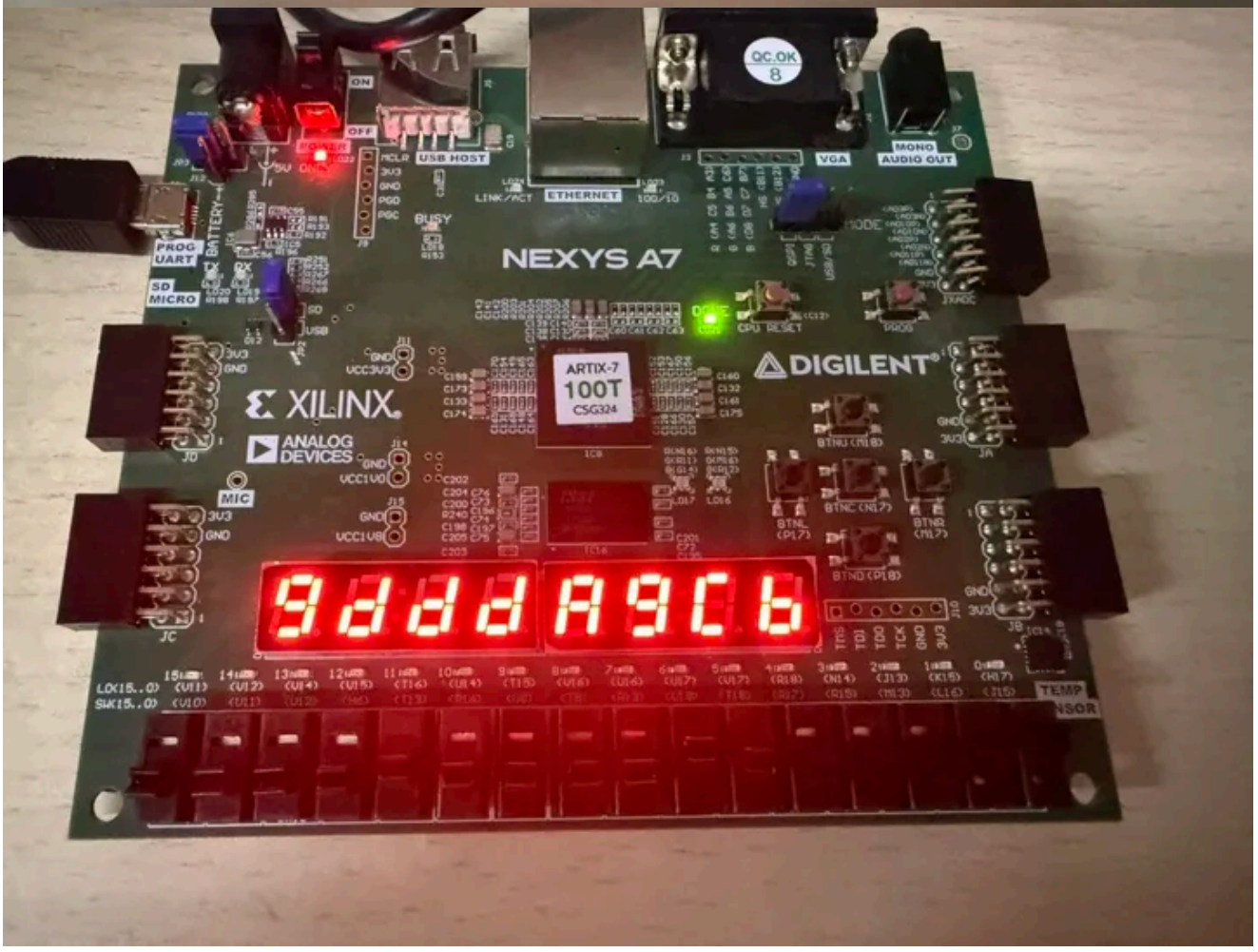

NEXYS A7
XILINX
ANALOG DEVICES
DIGILENT
ARTIX-7
100T
CSG324
ETHERNET
USB HOST
VGA
MONO AUDIO OUT
PROG UART
SD MICRO
MIC
CPU RESET
b d 5 3 6 2 F 2

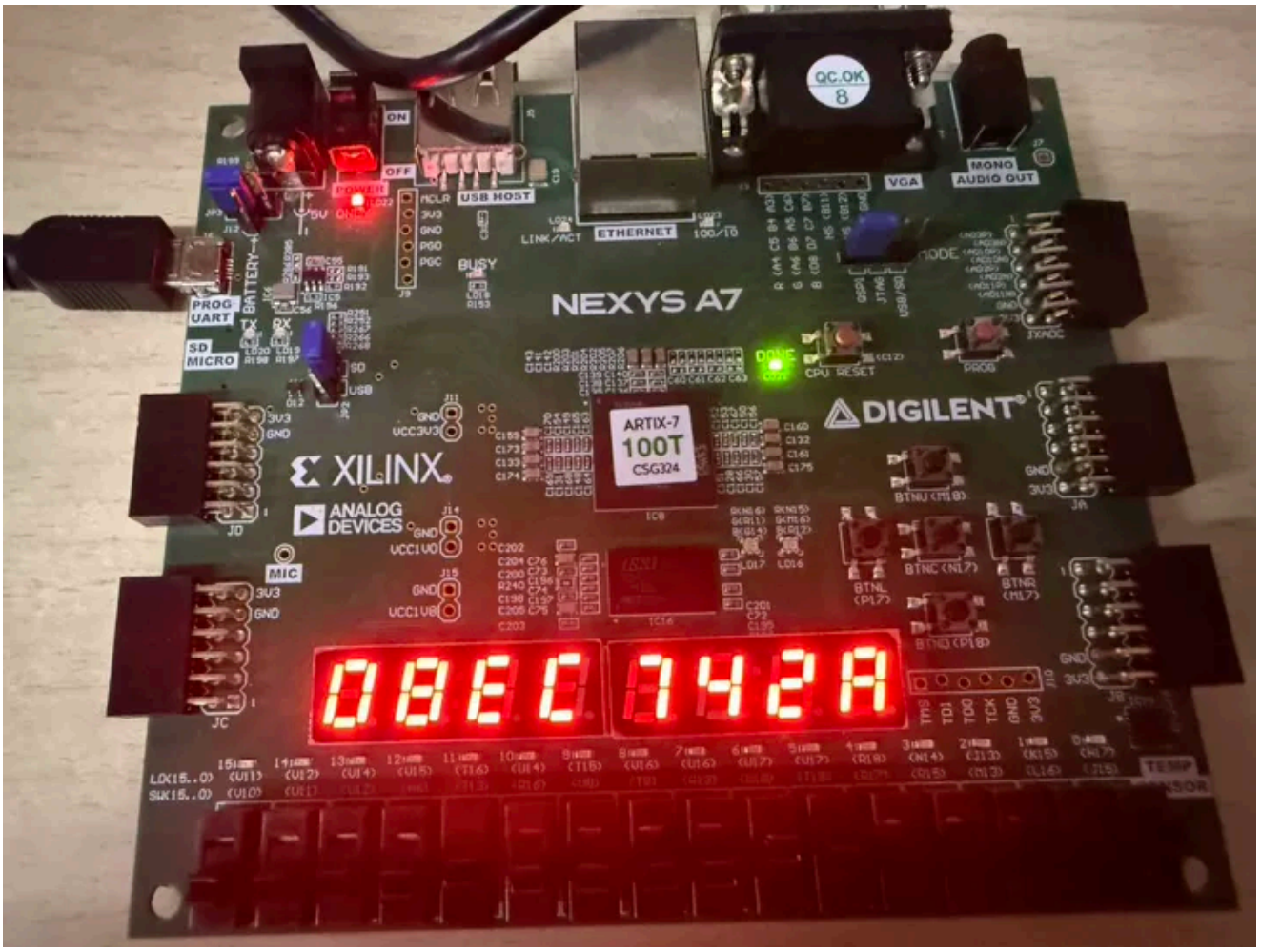
NEXYS A7
XILINX
ANALOG DEVICES
DIGILENT
ARTIX-7
100T
CSG324
BATTERY
ETHERNET
USB HOST
VGA
MIC
CPU RESET
JC
JD
TMS
TDI
TDO
TCK
GND
3V3


NEXYS A7
XILINX
DIGILENT
ANALOG DEVICES
ARTIX-7
100T
CSG324
1BA3 0EA2


NEXYS A7
XILINX
DIGILENT
ANALOG DEVICES
ARTIX-7
100T
CSG324
F1A2 A902


NEXYS A7
XILINX
DIGILENT
ARTIX-7
100T
CSG324
BF049C20


NEXYS A7
XILINX
DIGILENT
ARTIX-7
100T
CSG324
9ddda9C6



思考题

解释仿真测试样例和下板的顶层结构为什么满足 valid-ready 握手协议。20%

- 在testbench中，我们会有wait的模块来等待是否计算完毕，即`outvalid`，等待发送下一次的数据，具体如下：

```
initial begin
        clk=0; rst=1;
        in_valid=1'b0;
        #50;rst=0;#20;
        for(i=0;i<16;i=i+1)begin
            kernel.data[0]={$random(),$random()};
            kernel.data[1]={$random(),$random()};
            kernel.data[2]={$random(),$random()};
            kernel.data[3]={$random(),$random()};
            for(j=0; j<4; j=j+1) begin
                @(posedge clk);
                in_data={$random(),$random()};
                in_valid=1'b1;
                @(posedge clk);
                in_valid=1'b0;
            end
```

```
            wait(out_valid);
            @(posedge clk);
```

- 在我所理解的顶层模块中，convunit中也会有in_valid和out_valid两者进行握手协议，顶层模块展示如下：
-

```
module ConvUnit(
    input clk,
    input rst,
    input Conv::data_t in_data,
    input Conv::data_vector kernel,
    input in_valid,
    output in_ready,
    output Conv::result_t result,
    output reg out_valid,
    input out_ready
);
```

`ConvUnit` 模块被划分为 `Shift` 模块和 `ConvOperator` 模块，模块间用 valid-ready 协议传递数据。请思考能否对 `ConvOperator` 作类似上述的模块分割和数据交换，并给出这样分割后可能带来的性能提升。（bonus）5%

- convoperator 可以拆分为multiplier和adder。
- 性能我觉得可以实现流水话的操作，使得运行更加快速。

lab4-2

实验目的

- 学习使用 SystemVerilog 的 interface 等高级语法
- 学习串口的原理和使用
- 学习 FIFO 的原理和使用

实验环境

- 操作系统：Windows 10+ 22H2，Ubuntu 22.04+
- VHDL：Verilog，SystemVerilog

实验过程

代码的实现

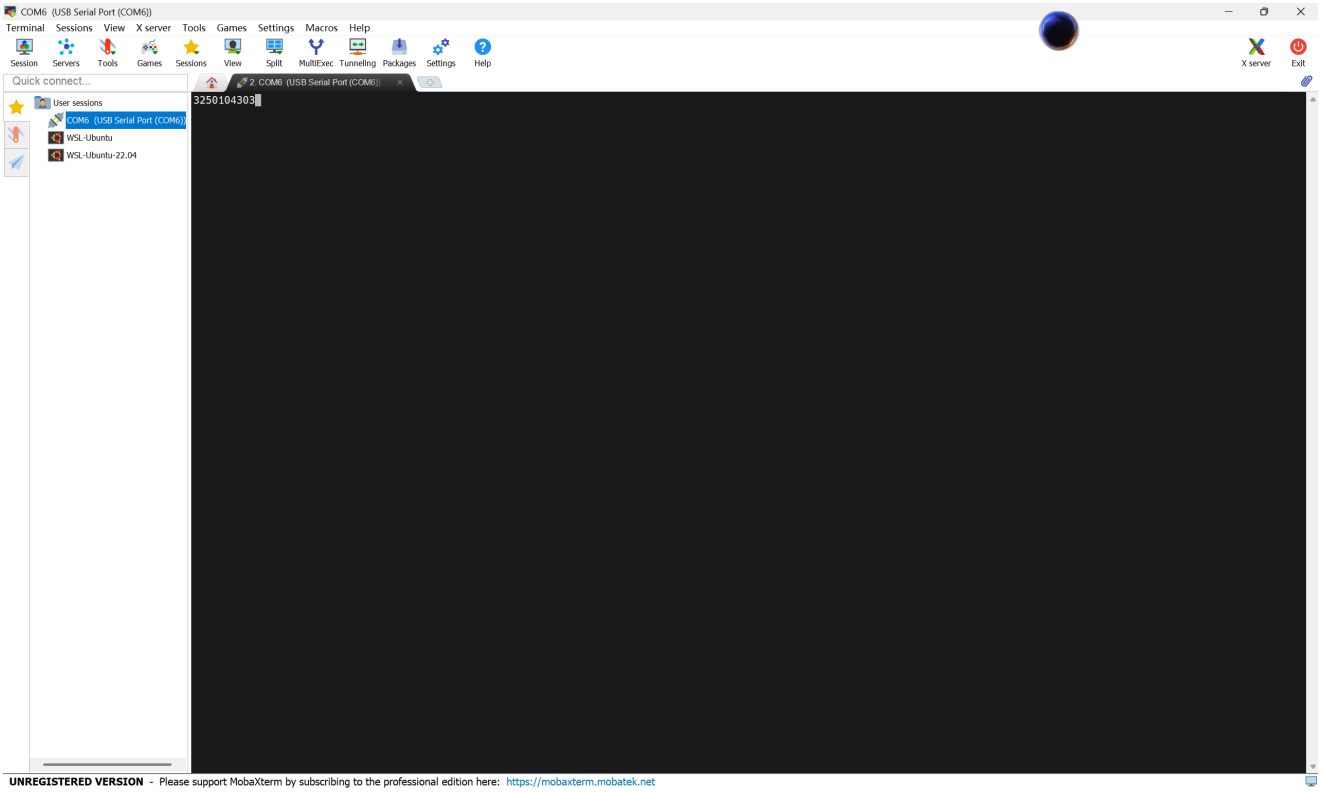
```
`include"uart_struct.vh"
module UartLoop(
    input clk,
    input rstn,
    Decoupled_ift.Slave uart_rdata,
    Decoupled_ift.Master uart_tdata,
    input UartPack::uart_t debug_data,
    input logic debug_send,
    output UartPack::uart_t debug_rdata,
    output UartPack::uart_t debug_tdata
);
    import UartPack::*;
    uart_t rdata;]
    uart_t tdata;
    uart_t rdata_reg;
    logic  rdata_reg_valid;
    reg [7:0]tdata_data_reg;
    reg tdata_valid_reg;
    always@(*)begin
        uart_rdata.ready=~rdata_reg_valid;
        if(debug_send)begin
            tdata_data_reg=debug_data;
            tdata_valid_reg=1'b1;
        end
        else begin
            tdata_data_reg=rdata_reg;
            tdata_valid_reg=rdata_reg_valid;
        end
        rdata=uart_rdata.data;
    end
    assign uart_tdata.data=tdata_data_reg;
    assign uart_tdata.valid=tdata_valid_reg;
    assign debug_rdata=uart_rdata.data;
    assign debug_tdata=tdata_data_reg;
    always@(posedge clk or negedge rstn)begin
        if(~rstn)begin
            rdata_reg<='0;
            rdata_reg_valid<=0;
        end
        else begin
            if(uart_rdata.valid&&uart_rdata.ready)begin
                rdata_reg<=uart_rdata.data;
```

```
                rdata_reg_valid<=1;
            end
            else if(rdata_reg_valid&&uart_tdata.ready)begin
                rdata_reg_valid<=0;
            end
        end
    end
endmodule
```

仿真完成

```
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab4-2# make verilate
uild/verilate --top-module Testbench -o Testbench -I/root/sys1-sp26/src/lab4-2/include -I/root/sys1-sp26/src/lab4-2
/../../repo/sys-project/lab4-2/include -CFLAGS "-DVL_DEBUG -DTOP=Testbench -std=c++17"  /root/sys1-sp26/src/lab4-2/
../../repo/sys-project/lab4-2/sim/testbench.sv /root/sys1-sp26/src/lab4-2/../../repo/sys-project/lab4-2/sim/judge.s
v /root/sys1-sp26/src/lab4-2/submit/UartLoop.sv /root/sys1-sp26/src/lab4-2/../../repo/sys-project/lab4-2/general/Lo
opTest.sv /root/sys1-sp26/src/lab4-2/../../repo/sys-project/lab4-2/general/uart.sv /root/sys1-sp26/src/lab4-2/../..
/repo/sys-project/lab4-2/general/Decoupled_ift.sv /root/sys1-sp26/src/lab4-2/../../repo/sys-project/lab4-2/general/
async.v /root/sys1-sp26/src/lab4-2/../../repo/sys-project/lab4-2/general/fifo.sv +define+TOP_DIR=\"/root/sys1-sp26/
src/lab4-2/build/verilate\" +define+VERILATE
make -C /root/sys1-sp26/src/lab4-2/build/verilate -f VTestbench.mk Testbench
make[1]: Entering directory '/root/sys1-sp26/src/lab4-2/build/verilate'
make[1]: 'Testbench' is up to date.
make[1]: Leaving directory '/root/sys1-sp26/src/lab4-2/build/verilate'
cd /root/sys1-sp26/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!!!
- /root/sys1-sp26/src/lab4-2/../../repo/sys-project/lab4-2/sim/testbench.sv:21: Verilog $finish
root@LAPTOP-T2B9NB3J:~/sys1-sp26/src/lab4-2#
```

上板：



思考题

2. 阅读代码和理论，设计 async_transmitter 的有限状态机，并描述 async_transmitter 的大致工作流程

1. 状态机设计

状态机包含三个状态：

- **IDLE**：空闲态。等待 `TxD_start` 信号，清零位计数器和波特率计数器，发送线 `TxD` 保持高电平。
- **SEND**：发送态。通过波特率计数器控制节拍，依次发送 1 位起始位、8 位数据位和 1 位停止位。每发送一位，位计数器递增。
- **DONE**：完成态。拉高 `TxD_busy` 信号一个周期，通知上层模块发送已完成，随后无条件返回 IDLE 态，等待下一个字节的发送请求。

2. 工作流程

1. 上层模块将待发送数据放到总线上，并触发 `TxD_start` 脉冲。
2. 状态机从 IDLE 跳转到 SEND，`TxD_busy` 拉高表示进入忙碌状态。
3. 在 SEND 状态，由波特率计数器产生移位节拍，状态机按位索引将起始位、8 数据位、停止位逐位输出到 `TxD` 引脚。

uart 数据线不可避免存在毛刺和电平扰动，思考 async_receiver 可以用什么办法来规避接受数据的毛刺

- 我觉得可以在采样电路前端设置一个简单的数字滤波器