lab 4-1: 卷积模块

1. 实验环境

1. 操作系统：Windows 10+ 22H2，Ubuntu 22.04+
2. VHDL：Verilog，SystemVerilog

2. 实验目的

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

3. 实验报告

3.1 每一步的过程&截图和解释 80%

1. ConvOperator部分代码

这部分主要说明卷积模块的状态切换方式：先接收输入，再等待内部乘法和加法完成，最后把结果统一送到输出端。

```
always@(posedge clk or posedge rst) begin
    if(rst) begin
        state_reg <= RDATA;
        in_ready <= 1;
        out_valid <= 0;
    end
    else begin
        case(state_reg)

            // RDATA (recieve data):
            // 初始状态；数据接收状态，和 shift 握手接收数据，接收到数据后进入 WORK 状态，
            // 并启动乘法器，清空所有 vector_stage 中间寄存器的 valid 值。

            RDATA: begin
                in_ready <= 1;
                out_valid <= 0;
                if(in_valid && in_ready) begin
                    state_reg <= WORK;
                    in_ready <= 1;
                    out_valid <= 0;
                end
            end

            // WORK (work):
```

```
            // 工作状态，等待乘法器、并行加法器依次执行计算，并把最后的计算结果存入 vector_stage2 电路。
            // 在 vector_stage2 寄存器的 valid 信号被设置为 1 之后进入 TDATA 状态。

            WORK: begin
                in_ready <= 0;
                out_valid <= 0;
                if(vector_stage2.valid) begin
                    //所有的乘法器都同时输出一个时钟刻的finish,导致vector_stage2.valid只有一个时钟周期有效，比较脆弱。
                    //可以考虑改成valid-finish的握手协议。
                    state_reg <= TDATA;
                    result <= vector_stage2.data;
                end
            end

            // TDATA (tansmit data):
            // 数据发送状态，和 caller 进行握手发送 result，发送完毕后返回 RDATA 状态。

            TDATA: begin
                in_ready <= 0;
                out_valid <= 1;
                if(out_ready && out_valid) begin
                    state_reg <= RDATA;
                    in_ready <= 0;
                    out_valid <= 0;
                end
            end

            default: state_reg <= RDATA;
        endcase
    end
end
//ConvOperator 输入的 data 和 kernel 送入乘法器执行乘法操作。
//在乘法计算结束后，将乘法器输出的结果保存到 vector_stage1 寄存器组中，寄存器的valid 设置为 1，表示数据有效。
//vector_stage1 寄存器中的结果通过并行加法树累加得到最终结果，保存到vector_stage2 寄存器中，valid 设置为 1.
//vector_stage2 的内容就是最后的 result，通过这个模块的输出端口输出给 caller。
Conv::result_t add_tmp [Conv::LEN-1:1] /* verilator split_var */;
logic valid_tmp [Conv::LEN-1:1] /* verilator split_var */;
generate
    for(genvar i = 0; i < Conv::LEN; i = i + 1) begin
        Multiplier #(
            .LEN(Conv::WIDTH)
        )mul(
            .clk(clk),
            .rst(rst),
            .multiplicand(kernel.data[i]),
```

```
            .multiplier(data.data[i]),
            .start(state_reg == WORK),
            .product(vector_stage1[i].data),
            .finish(vector_stage1[i].valid)
        );
    end
    for(genvar i = 1; i < Conv::LEN; i = i + 1) begin
        if(i<Conv::LEN/2)begin
            //assign add_tmp[i] = add_tmp[i*2] + add_tmp[i*2+1];
            Adders #(
                .LENGTH(VECTOR_WIDTH)
            ) add(
                .a(add_tmp[i*2]),
                .b(add_tmp[i*2+1]),
                .c_in(1'b0),
                .s(add_tmp[i]),
                .c_out()
            );
            assign valid_tmp[i] = valid_tmp[i*2] & valid_tmp[i*2+1];
            // 加法器的结果输出作为下一级加法器得输入
            // 参考了数据结构的 heap
        end else begin
            wire [VECTOR_WIDTH-1:0] data1 = vector_stage1[(i-
            Conv::LEN/2)*2].data;
            wire [VECTOR_WIDTH-1:0] data2 = vector_stage1[(i-
            Conv::LEN/2)*2+1].data;
            wire valid1 = vector_stage1[(i-Conv::LEN/2)*2].valid;
            wire valid2 = vector_stage1[(i-Conv::LEN/2)*2+1].valid;
            //assign add_tmp[i] = data1 + data2;
            Adders #(
                .LENGTH(VECTOR_WIDTH)
            ) add(
                .a(data1),
                .b(data2),
                .c_in(1'b0),
                .s(add_tmp[i]),
                .c_out()
            );
            assign valid_tmp[i] = valid1 & valid2;
            // 乘积运算结果两两相加
        end
    end
    assign vector_stage2.data = add_tmp[1];
    assign vector_stage2.valid = valid_tmp[1];
endgenerate
```

2. Shift部分代码

Shift 模块负责接收外部输入并把它们保存在移位寄存器中，再把稳定的数据送给卷积模块。这样前后级就可以通过 ready/valid 对齐节拍。

```
always@(posedge clk or posedge rst) begin
    if(rst) begin
        state_reg <= RDATA;
        in_ready <= 1'b1;
        out_valid <= 1'b0;
    end
    else begin
        case(state_reg)

            // RDATA(receive data)：
            // 接收数据状态，Shift 模块和外部的 caller 握手载入 in_data 数据，
            // 此时不向后端的 ConvOperator 发送数据。当接收到数据之后进入 TDATA
            状态。

            RDATA: begin
                if(in_valid && in_ready) begin
                    data_reg <= {in_data, data_reg[Conv::LEN-1:1]};
                    if(out_ready) begin
                        state_reg <= TDATA;
                        in_ready <= 1'b0;
                        out_valid <= 1'b1;
                    end
                end
            end

            // TDATA(transmit data)：
            // 发送数据状态，Shift 模块和后端的 ConvOperator 握手发送数据
            data，
            // 此时不接收来自 caller 的数据。当发送数据之后进入 RDATA 状态。

            TDATA: begin
                if(out_ready && out_valid) begin
                    state_reg <= RDATA;
                    in_ready <= 1'b1;
                    out_valid <= 1'b0;
                end
            end

        endcase
    end
end

genvar i;
generate
    for (i = 0; i < Conv::LEN; i = i + 1) begin
        assign data.data[i] = data_reg[i];
    end
endgenerate
```

3.1.1 仿真通过，输出 `success!!!` 40%

```
cd /home/parallels/sys1-sp26/src/lab4-1/build/verilate; ./Testbench
simulate result: 5c1270934becfd6ac3a6633610ea16e0
simulate result: 9e8c3c630b84e51e41753456e3b97e60
simulate result: 54aeb66f2437bdd719a2fd235bc8d8b0
simulate result: 5495f9d066b1d0e5dc35af7a34fce8ec
simulate result: c4910c537ad47e034ab9e8b0599e739c
simulate result: d1e5171b8bf90217f590b91d7f38c0a2
simulate result: bc334ae758c313327ac7f3271738c01e
simulate result: a0c224698ea1170b8a165520557ca256
simulate result: 324398a189dc1f0dba48ff52c664694a
simulate result: 576573c1c181ea4c540c3b34a09a78c0
simulate result: 54e03c68901b18f3142f42fa3f534bf4
simulate result: 5495f9d066b1d0e5dc35af7a34fce8ec
simulate result: c4910c537ad47e034ab9e8b0599e739c
simulate result: d1e5171b8bf90217f590b91d7f38c0a2
simulate result: bc334ae758c313327ac7f3271738c01e
simulate result: a0c224698ea1170b8a165520557ca256
success!!!
- /home/parallels/sys1-sp26/src/lab4-1/sim/testbench.sv:51: Verilog $finish
```

Image 1

3.1.2 综合实现卷积单元 40%

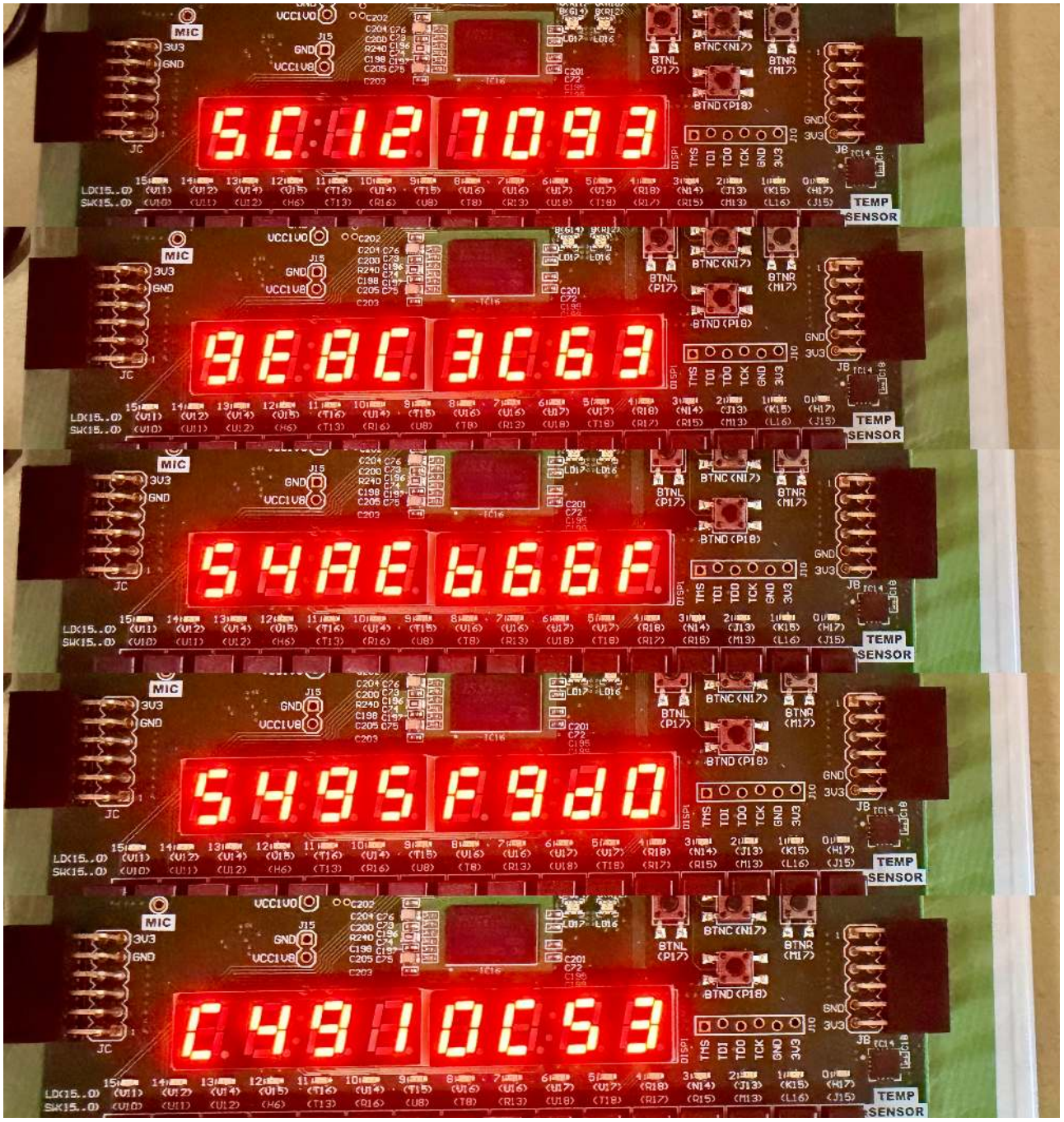

Image 2

3.2 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%

仿真测试样例中，输入端只有在 `in_valid` 和 `in_ready` 同时有效时才采样数据，采样后的值会被寄存器保存，所以不会丢失。输出端在结果稳定后才拉高 `out_valid`，并保持到 `out_ready` 到来为止。

下板时也是同样的逻辑：`Shift` 先缓存并移位，`ConvOperator` 再做计算。前后级靠 ready/valid 对齐，因此能保证数据完整并满足握手协议。