电子科技大学示范性微电子学院

实验课程及实验室安全预习报告

（实验）课程名称 IC 综合实验 2

学号：2022340104032

姓名：徐子涵

实验地点：清水河校区国际创新中心 B523

实验日期：2024.11-2025.01

一. 实验目的与要求

实验目的：我认为主要有 3 个目的。第一：在过去两年的学习中，我们已经掌握了 Verilog 代码编写的基本方法和数字集成电路（数字 IC）设计的初步流程。本课程旨在进一步深化我们对 Verilog 的理解，包括行为建模、数据流建模等高级概念。我们也将亲自动手实践数字 IC 的设计流程，从而将理论知识与实际应用紧密结合，希望学生理论联系实际，把课本的重要知识运用到具体实践中。第二：培养自己独立解决问题的能力。在未来的职业生涯或研究生学习中，我们将面临众多挑战。在没有“永远在线”的导师的情况下，我们必须锻炼独立解决问题的能力。通过本次课程，我们将提升自己的综合素养，为未来的学术或职业发展打下坚实基础。第三：培养团队协作能力。在本次 IC2 实验课程中，我们将以四人小组的形式进行合作。在这一过程中，每位成员都将扮演关键角色，包括统领全局、合理分工、相互协作和互相支持。我们要学会从团队的角度出发，共同面对挑战，追求集体的成功而非个人利益，实现真正的合作共赢。

实验要求：在本次实验中我们将会依次进行 RTL 代码编写、门级仿真、DC 综合、ICC 布局布线、后仿以及 DRC 和 LVS 验证。最终达到在 550*550μm 面积的要求下，实现实验要求。

二. 实验相关原理预习

2.1. 前端设计

①系统设计：

在本次实验我们先了解了这次 ASIC 设计的**整个流程**，还分析了卷积神经网络数据流的变化，并将本次实验划分成各个模块，每个模块各司其职。确保每个模块能够独立完成指定功能的同时，考虑模块之间的交互和通信。具体可见下图 1 所示。

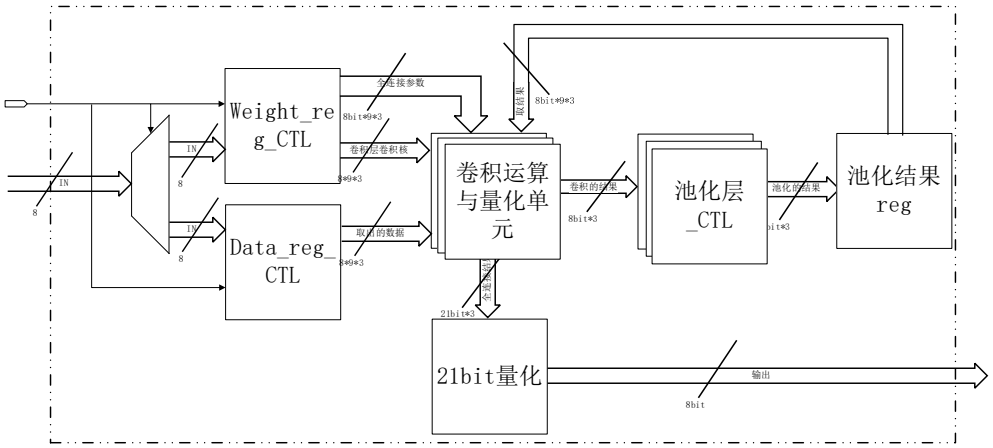

图 1 并行架构的模块图

②RTL 设计：

分析整个流程后，进入 RTL 级代码设计。RTL 涉及即利用硬件描述语言，如 VHDL，Verilog，System Verilog, 对电路以寄存器之间的传输为基础进行描述。我们这次实验主要是基于 Verilog 的代码设计。

其中一个我们仔细考量的细节在于**乘法器的设计**。若采用**并行架构**，在本次实验中我们将会使用 9 个乘法器，每个乘法器实现两个 8 bit 位的数据相乘，输出将有 16 bit 位。乘法器主要基于 3 种方法。

第一个是**采用移位相加的形式**，代码见**附录 1**，但经过 DC 逻辑综合，我们发现第一种设计的乘法器面积较大且延时较大，于是我们先从解决延时出发，查阅相关资料设计了一种 Booth 编码乘法器。

Booth 编码的**关键思想**是通过对乘数进行特殊编码，能够将传统的乘法中的多个部分积减少，从而优化乘法的步骤。它是通过对乘数的连续位进行两两比较，生成不同的部分积，从而减少加法操作的次数。具体来说，Booth 算法在进行乘法时，不是按位逐一相乘，而是对乘数进行编码，根据编码结果生成部分积，并通过加法合并这些部分积。从而节约了**时间，具体可见附录 2**。

我们还设计了一种流水线的 Booth 算法乘法器，通过插入寄存器进一步提高运行速率。具体可见**附录 3，**但由于本次实验面积的限制，以及我们的综合考量。我们最后总采用了最后一种乘法器。即，直接使用“*”，直接使用“*”号，在 DC 逻辑综合时，会调用元件库直接完成此功能，我们“惊奇”的发现，无论从**时间还是面积**来说，最后一种方法的实现效果是最好的。侧面反映了我们实现的有符号数的乘法操作应该还有**较大的优化空间**。

2.2. 功能验证

此次步骤旨在验证设计是否符合预期功能。可以通过编写测试平台（Test

bench）和测试用例（Test Case）来验证设计的功能是否符合预期。

使用仿真工具（如 ModelSim、VCS 等）验证 RTL 代码的功能正确性。本次实验我们主要使用 **VCS + DVE** 的测试。

VCS 是 Synopsys 提供的一款的硬件仿真工具，主要用于对数字电路的设计进行仿真。VCS 能够编译和执行 Verilog 代码，并进行功能验证和时序分析。**DVE** 是 VCS 的一个可视化工具，用于查看仿真波形和调试仿真过程。它提供了一个图形化界面，能够展示仿真过程中的信号波形、状态、变量等信息，使设计者能够直观地理解仿真结果，方便问题定位。

在这个过程中，我们先通过编写 Testbench 后，进行 VCS 仿真，用 DVE 观察波形，并纠正代码中的错误，最后成功验证实验代码的正确性。

2.3. DC 逻辑综合

我们将设计好的代码带入下一个步骤，即综合。综合是指将 RTL 代码转化为门级网表，并根据设计目标（如功耗、时序、面积等）进行优化。综合工具（**Synopsys Design Compiler**）将 RTL 代码映射到实际硬件的逻辑门和触发器上。

并根据时序、功耗、面积等要求对设计进行优化。可以通过**调节时序约束、修改设计结构等方式进行优化。**

用得到的网表进行接下来的门级仿真，门级仿真主要验证以下内容：

1.功能一致性：设计在门级网表中的功能是否与原始 RTL 描述一致。

2.时序一致性：设计的时序是否符合时钟要求和设计规范。

3.综合后的逻辑行为：验证综合后的逻辑是否能够实现预期的功能。

2.4. ICC 布局布线

布局和布线是一个非常重要的阶段，它直接影响到芯片的性能、功耗和面积。

在布局布线之前，先将经过综合的网表导入 ICC 工具，生成初步的设计信息，包括门级网表、时钟树、时序约束等。

布局（Placement）：首先进行的是**标准单元的放置，**将逻辑门和触发器等设计元素放置到芯片上，确保它们的位置符合时序和功耗要求。随后 ICC 会考虑时钟和数据路径的时序约束，尽量将时钟路径缩短，减少时钟延迟。时序优化的目标是保证在规定的时钟周期内，数据能够在各个单元之间顺利传递。

布线（Routing）：将各个逻辑单元通过金属线连接起来，以完成电路的功能。布线的目标是确保设计中的信号能够可靠地传输，并且满足时序要求。布线过程中需要优化信号完整性和功耗。

设计规则检查（DRC LVS）：检查布局是否符合工艺设计规则，例如金属线的宽度、间距等。

在完成所有的设计优化、验证和检查后，最终会生成**带有布局信息的网表**。这个网表用于生产和测试阶段，包含所有标准单元、连线、时序信息等。

2.5. 后仿

后仿是数字集成电路设计中的一个关键阶段，它用于验证在综合之后的设计是否符合预期的功能和时序要求。后仿通过将综合后生成的网表与仿真环境结合，模拟设计在真实硬件实现后的行为，确保设计从逻辑到时序都没有问题。

带入返标的过程是将综合后的门级网表中的信息（如延迟、时序约束、信号和门的物理特性等）带入仿真环境，以确保仿真能够模拟出实际硬件的行为。后续通常会使用 VCS 来进行仿真。

2.6. 物理验证

布局与布线后仿真：在完成布局与布线后，需要进行仿真以验证时序和功能是否符合要求。

物理验证：使用 DRC、LVS 等工具进行物理设计验证，确保设计没有违反工艺规则，并且电路连接正确。

三. 实验内容与步骤预习纲要

3.1. 实验内容

给定的卷积核的结构包含卷积层、池化层、全连接层各一层，共计 3 层。各层的详细介绍如下。

卷积层：在 11×11 的矩阵数据外添加 Padding 后，对对应的矩阵进行卷积操作，其中，卷积核大小为 3x3，步长为 2，共有 3 个卷积核；

池化层：对输出的数据进行最大池化，其中池化核大小为 2x2，步长为 2；

全连接层：全连接层对应的权重矩阵大小为 3x3x3，与池化输出一一对应，需要完成一次乘加操作。

芯片的输入数据为 1x1 的矩阵，是位宽为 8bit 的有符号数补码，按照卷积核、全连接权重、100 组 11×11 的矩阵的数据依次输入。

芯片的输出数据为 1x1 的矩阵，位宽为 8bit，输出的为全连接的结果通过量

化所得到的内容。

由于在数据处理过程中得到的乘积数据为 16bit，加和后需要用 20bit（全连接 21bit）的数据记录才能保证数据不发生溢出，而后续进行池化、全连接和输出所需要的数据均为 8bit，因此需要一种量化规则来进行 20bit 或 21bit 的到 8bit 的量化处理。给定的卷积核的量化规则为：在保证符号正确的情况下拓展到 20 位来表示，然后将低 8 位抹除，剩余的 12 位若超过 int8 的表示范围，则取 int8 范围的最大值，反之，取这 12 位的低 8 位。由于数据均由补码表示，因此所得的结果只需要将最高位符号位向前扩展即可得到扩展数据。

3.2. 步骤预习纲要

3.2.1. 卷积层运算

从题目所述，首先依次从输入/输出（IO）端口送入待处理的图像数据以及卷积层的权重数据。随后，将 121 个权重数据与 3×3 大小的卷积核进行卷积运算，且该卷积运算的步长为 2，具体处理过程如下图 2 所示：

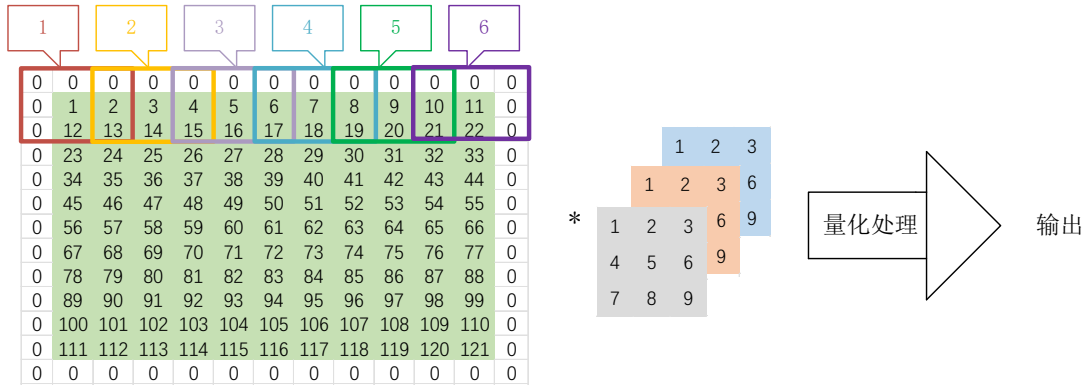

图 2 步进为 2 的卷积层处理

在对一个 3×3 的卷积核进行处理时，可以观察到，将 11×11 的图像层扩展为 13×13 后，再对图像中的数据执行卷积操作，最终会得到一个 6×6 的图像。具体到每一位，每个小方框代表 8 位的数据。经过 8 位×8 位的乘法运算后，得到 16 位的乘积。接着，将 9 个乘积进行累加运算，得到 20 位的数据。最后，对这 20 位的数据进行量化处理，并输出最终结果。

3.2.2. 池化层处理

借助前一级所得到的卷积层图像（尺寸为 6×6×3），进一步实施最大池化操作。具体而言，就是在每 4 个相邻的数据中筛选出最大的那个数据并予以输出，

其工作原理可参照图 3 所示的原理图。

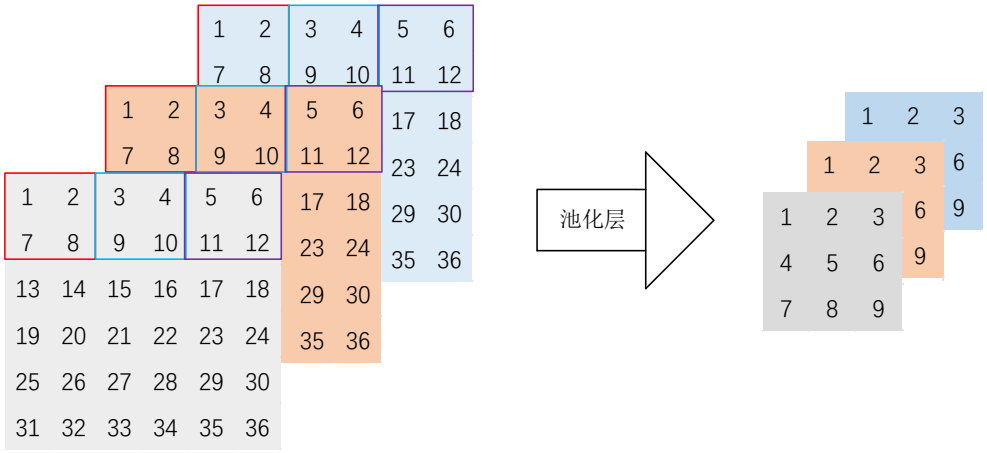

图 3 池化层处理

3.2.3. 全连接层计算

在全连接层中，我们需要两组数据：一组是池化层的输出结果，其维度为 3×3×3；另一组是全连接层所需的权重矩阵，同样具有 3×3×3 的维度。在全连接层的运算过程中，将再次执行类似于卷积层的乘加运算。因此，在后续的硬件实现阶段，可以考虑复用卷积层的模块，以此来达到优化芯片面积的目的。

四. 使用的实验仪器（设备、元器件）

主要使用的是软件，例如 VS Code、VCS、DVE、DC、ICC 以及 Virtuoso。

五. 实验室安全操作事项

1. 计算机与软件操作安全：确保重要的设计文件、仿真结果和项目数据定期备份，**以防数据丢失**。

2. 确保计算机上安装并更新有效的**防病毒软件**，防止恶意软件入侵计算机。

3. **正确连接电源**：确保所有计算机设备、服务器和测试设备都连接到合适的电源插座和电源防护设备，避免因电压不稳定或电源问题引发设备损坏或火灾。

4. **长时间使用电脑**：保持正确的坐姿，避免长时间不休息，定时休息眼睛，避免眼疲劳和颈椎不适。

六. 思考题（至少 3 道）

6.1. 我们本次任务不包括静态时序分析（STA），请分析 STA 的作用，如果不进行 STA 会有什么影响？

STA 通过分析电路中所有信号路径的时序，确保数据在时钟周期内能够稳定**传输**并且没有**时序违规**。

如果不进行 STA 会有什么影响？最直接的后果是可能存在时序违规，特别是**建立时间和保持时间违规**。例如，信号可能无法在时钟周期内传播完整，导致数据错误。没有 STA 的支持，设计师可能无法及时发现这些问题。

时序问题会直接导致时序问题会直接导致**功能**上的错误，例如信号在正确的时钟周期内没有按预期到达目标位置，或逻辑单元的输入信号错误，从而导致电路逻辑不符合设计意图。

可是 **DC 如果修复了时序问题还需要 STA 吗？**

是的，即便 DC 修复了时序问题，仍然需要进行 STA。原因是：虽然 DC 在进行综合时可以通过优化逻辑、调整触发器位置、重新排列门级单元等方式来改善时序问题，但 DC 的时序修复并不是万无一失的，可能还会存在其他未被识别的时序问题。STA 能够帮助识别在布线阶段或物理设计中产生的新时序违规。

其次，DC 的时序修复主要是在逻辑层面进行的，而物理设计和时钟树的设计可能会引入**新的时序问题**。STA 工具可以分析整个设计的时序，不仅仅是综合后的逻辑，还包括物理实现后的时序。

6.2. 遇到时序闭合困难时，如何选择合适的**优化策略**来实现时序闭合？在追求时序闭合的过程中，如何确保不牺牲其他设计目标？

策略：

1.**时钟树合成**（CTS）优化是时序闭合的一个关键步骤。通过优化时钟树结构，减少时钟偏移和时钟延迟，可以有效提高时序闭合的机会。

2.**优化长路径**，在长路径中插入缓冲器可以减少信号延迟，提升时序性能。还可以通过合并或拆分逻辑，简化路径。通过替换复杂的逻辑门或使用更高效的逻辑结构来缩短路径。最后**调整布局中的元素**，减少信号线的长度，优化信号传输时间。

3.调整扇出，高扇出的路径（即驱动多个负载的路径）通常会导致较大的信号延迟，进而影响时序。通过控制扇出，可以降低路径延迟。

4.时序路径的局部优化，对于特定的时序违规路径，可以进行局部优化，通常是在时序分析工具的帮助下识别出具体的违规路径，并针对性地进行优化。

如何确保不牺牲其他设计目标：

1.**权衡设计目标：**在进行时序优化时，始终需要权衡功耗、面积、时序和性能。例如，虽然插入缓冲器可以有效地修复时序，但会增加功耗和面积；因此，选择高效的缓冲器并控制数量是减少影响的关键。

2.**多目标优化：**使用现代 EDA 工具，如 **ICC** 或 **Innovus**，可以进行多目标优化。这些工具允许在时序闭合过程中同时考虑功耗、面积和时序，帮助设计人员在多个目标之间找到平衡。

6.3. DRC 违规与时序违规之间的关系是什么？在解决 DRC 违规时，如何平衡设计规则与时序要求之间的冲突？如何确保设计在物理验证过程中没有漏掉任何问题？

DRC 违规：设计规则检查（DRC）主要是确保设计符合工艺制造规则，这些规则包括布线间距、金属层宽度、接触孔尺寸等，目的是确保设计可以在制造过程中正确、可靠地被加工。DRC 违规通常指在布局中存在违反这些物理规则的情况，例如布线过密、金属层间距过小等。

时序违规：时序分析关注的是信号在电路中的传播时间。时序违规通常发生在信号传输需要的时间超过了时钟周期或时序要求时，导致信号在到达目的地时已不再有效，从而影响电路的正确性和性能。

在布局布线时，**DRC 违规**可能导致信号路径过长、信号质量差或信号干扰增大，这可能间接影响时序。比如，过于密集的布线可能增加信号的传播延迟，或导致电磁干扰，影响时序。

如何确保设计在物理验证过程中没有漏掉任何问题？

全流程验证：物理验证应该贯穿整个设计流程，从前端设计到后端布局布线，再到时序分析和功耗分析，确保每个步骤都经过验证。通过 DRC、LVS（版图与电路一致性检查）、Antenna Effect（天线效应检查）等多项物理验证，确保设计没有漏掉任何问题。

多角度验证：除了常规的 DRC 和 LVS，还应该进行 ERC（电气规则检查）和 PEX（后仿电参数提取）。ERC 可以帮助发现电路中的电气问题（如驱动能力不足、漏电流过大等），PEX 可以确保后仿与前仿的参数一致性，确保设计在不

同的工作条件下都能正常工作。

6.4. 当 DC 综合后的设计出现时序违规时，除了使用 compile_ultra 命令进行优化外，还可以从哪些方面入手来改善时序？

当 DC 综合后的设计出现时序违规时，除了使用 compile_ultra 命令进行优化外，还可以从以下几个方面入手来改善时序：

1. **时序高努力脚本**

使用 compile_ultra -timing_high_effort_script 命令，该命令会将时序的优先级提升到超过设计规则检查（DRC）的优先级，从而更积极地优化时序。

2. **重定时（Retiming）**

重定时是一种通过将时序不满足的部分组合逻辑转移到有余量的地方来改善时序的方法。具体命令如下表 1 DC 常见命令。

表 1 DC 常见命令

命令	用途
compile_ultra -retime	将时序不满足的部分组合逻辑转移到有余量的地方，从而改善时序
optimize_registers	适用于包含寄存器的门级网表，通过将后级的部分组合逻辑移到前级，使所有寄存器与寄存器之间的时序路径延迟都小于时钟周期，满足寄存器建立时间的要求
pipeline_design	适用于纯组合电路的门级网表，通过管道传递（pipeline）技术，使设计的传输量更快

3. 时序路径分组

通过合理划分时序路径分组（path groups），可以更有效地优化时序。默认情况下，每个时钟域对应一个 path group，但在只有一个时钟的情况下，可以进一步细分为 InputToReg、RegToReg、RegToOutput、InputToOutput 四个 path group。优化时，先优化关键路径（最差的时序路径），然后优化次差路径。

4. 时钟约束调整

检查时钟不确定性（Clock Uncertainty）：如果时钟不确定性设置过大，可能会导致时序违规。适当减小时钟不确定性可以改善时序。

设置虚拟时钟：对于跨时钟域的输入信号，可以设置虚拟时钟来帮助时序收敛

七. 预习遇到的问题：

1. 在面对复杂的数字 IC 设计流程时，很容易迷失在具体细节中，难以从整体上把握设计的流向和步骤。

2.预习过程中，感觉自己 Verilog 代码的编写不是很熟练，害怕影响后续完成实验，故决定先通过基本的学习，提高自己代码编写能力，并在其中注重代码编写风格。

3.因为本次实验要求我们快速掌握数字 IC 设计的软件，所以我需要快速掌握软件的基本操作，因此我需要快速查找软件的学习笔记，使自己快速上手。

八. 附录

附录 1：移位相加乘法器

```
module sign_mult8 (out,a,b,clk);
output [15:0] out;
input [7:0] a,b;
input clk;
wire [15:0] out;
wire [14:0] out1, c1;
wire [12:0] out2;
wire [10:0] out3, c2;
wire [8:0] out4;
reg [14:0] temp0;
reg [13:0] temp1;
reg [12:0] temp2;
reg [11:0] temp3;
reg [10:0] temp4;
reg [9:0] temp5;
reg [8:0] temp6;
reg [7:0] temp7;

function [7:0] mult8x1;
    input [7:0] operand;
    input sel;
    begin
        mult8x1 = (sel) ? operand : 8'b00000000;
    end
endfunction

always @(posedge clk)
```

```
begin
    temp7 <= mult8x1(a, b[0]);
    temp6 <= (mult8x1(a, b[1]) << 1);
    temp5 <= (mult8x1(a, b[2]) << 2);
    temp4 <= (mult8x1(a, b[3]) << 3);
    temp3 <= (mult8x1(a, b[4]) << 4);
    temp2 <= (mult8x1(a, b[5]) << 5);
    temp1 <= (mult8x1(a, b[6]) << 6);
    temp0 <= (mult8x1(a, b[7]) << 7);
end

assign out1 = temp0 + temp1;
assign out2 = temp2 + temp3;
assign out3 = temp4 + temp5;
assign out4 = temp6 + temp7;
assign c1 = out1 + out2;
assign c2 = out3 + out4;
assign out = c1 + c2;

endmodule
```

附录 2：Booth 算法乘法器

```
module booth2_multiplier (
    input signed [7:0] a,   // 被乘数 (Multiplicand)
    input signed [7:0] b,   // 乘数 (Multiplier)
    output signed [15:0] product // 16 位乘积输出
);

    //assign product = (a * b);
     reg signed [15:0] multiplicand; // 扩展的被乘数
     reg signed [15:0] acc;          // 累加器，存储部分积
     reg [8:0] booth_multiplier;    // 扩展乘数到 9 位，用于两位 Booth 编码
     integer i;

     always @(*) begin
         // 初始化
         multiplicand = {{8{a[7]}}, a}; // 符号扩展被乘数到 16 位
         acc = 16'b0;                   // 累加器清零
         booth_multiplier = {b, 1'b0};  // 乘数扩展为 9 位，最低位补 0 用于 Booth
编码

         // Booth 两位乘法主循环
         for (i = 0; i < 8; i = i + 1) begin
```

```
            // 检查 Booth 编码 (两位编码 booth_multiplier[1:0])
            case (booth_multiplier[1:0])
                2'b01: acc = acc + multiplicand; // 加被乘数
                2'b10: acc = acc - multiplicand; // 减被乘数
                default: ; // 其他情况不操作
            endcase

            // 右移操作：累加器算术右移一位，符号位保留
            booth_multiplier = {booth_multiplier[8],
booth_multiplier[8:1]}; // 乘数算术右移一位
            multiplicand = multiplicand << 1; // 被乘数左移一位
        end

        // 最终输出累加器的值
        product = acc;
    end
endmodule
```

附录 3：加入流水线的乘法器

```
module a04108_clk_mult8 (
    input wire clk,
    input wire reset,
    input signed [7:0] a,
    input signed [7:0] b,
    output reg signed [15:0] product
);

    reg signed [15:0] multiplicand;
    reg signed [16:0] acc;
    reg [8:0] booth_multiplier;
    reg [3:0] counter;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            multiplicand <= 16'b0;
            acc <= 17'b0;
            booth_multiplier <= 9'b0;
            counter <= 4'b0;
            product <= 16'b0;
        end else begin
            if (counter == 4'b0) begin
                multiplicand <= {{8{a[7]}}, a};
                acc <= 17'b0;
                booth_multiplier <= {b, 1'b0};
```

```
                counter <= 4'b1000;
            end else begin
                case (booth_multiplier[1:0])
                    2'b01: acc <= acc + multiplicand;
                    2'b10: acc <= acc - multiplicand;
                    default: ; // 不操作
                endcase

                acc <= {acc[16], acc[16:1]};
                booth_multiplier <= {booth_multiplier[8],
booth_multiplier[8:1]};
                counter <= counter - 1;

                if (counter == 4'b0001) begin
                    product <= acc[15:0];
                end
            end
        end
    end
endmodule
```