Simple_dma

一款基于学习目的的 DMAip 核

使用的大模型：ChatGPT-4o

一、 设计要求

根据题目要求的适合学习目的，我试着让大模型总结了如何才能让DMAip核适合学习目的：

1. 易于理解的结构

要求整个模块**逻辑清晰、状态明确**，便于初学者掌握 DMA 基本原理。
状态控制应使用**有限状态机（FSM）**分阶段实现，帮助学生理解数据流控制机制。

2. 支持基本的内存搬运

支持从指定源地址（src_addr）向目的地址（dst_addr）搬运连续数据。
搬运数据长度可配置（length），以 32 位字为单位。

3. 模拟真实硬件行为

要求引入握手机制以模拟内存访问延迟，例如：添加 mem_read_valid 信号，模拟“数据何时有效”。
便于后续扩展，例如 AXI 总线适配、中断触发、burst 搬运等。

4. 支持功能可观察性

提供 done 输出信号，表征 DMA 操作完成，有助于仿真和调试。
输出信号应尽量完整，方便波形观察：如读写地址、读写使能、写出数据等

二、 设计实现与分析

基于大模型自主分析的“如何才能让DMAip核适合学习目的”，我键入了如下提示词：“基于你自己总结的如何才能让DMAip核适合学习目的，请设计一款DMAip核，并给出相应代码”。

（该回答的代码仅仅是第一版，后面分析的时候发现有问题，

就做出了一些修改）

大模型的代码主要分为三个部分。

第一个部分是模块接口的说明以及四种寄存器，对于输入信号，大模型定义了如下5种变量：

1）clk，rst：时钟与复位信号

2）start：开始传输信号

3）src_addr，dst_addr：源地址和目标地址

4）length：传输的数据长度

5）mem_read_data， mem_read_valid：模拟内存读取返回数据与有效性信号

而输出信号则有4种：

1）done：DMA传输完成信号

2）mem_read_en，mem_write_en：控制内存访问信号

3）mem_read_addr，mem_write_addr：读写访问地址

4）mem_write_data：写入的数据

需要注意的是，simple_dma通过mem_read_addr/mem_read_en 发起读，通过mem_write_addr/mem_write_en 发起写，通过 mem_read_data/mem_read_valid 模拟外设返回数据有效。

整个逻辑流程是通过clk和异步rst控制。

三种寄存器：

1）src_ptr，dst_ptr：指向当前源地址和目标地址指针

2）count：剩余传输次数

3)data_buffer:缓存读取的数据

4)soft_rst：为了内部能够重置状态（第一次的代码没有，后面修改后添加的）

第二个部分是状态机的声明部分。大模型设立了5个状态：

IDLE：空闲等待启动信号；

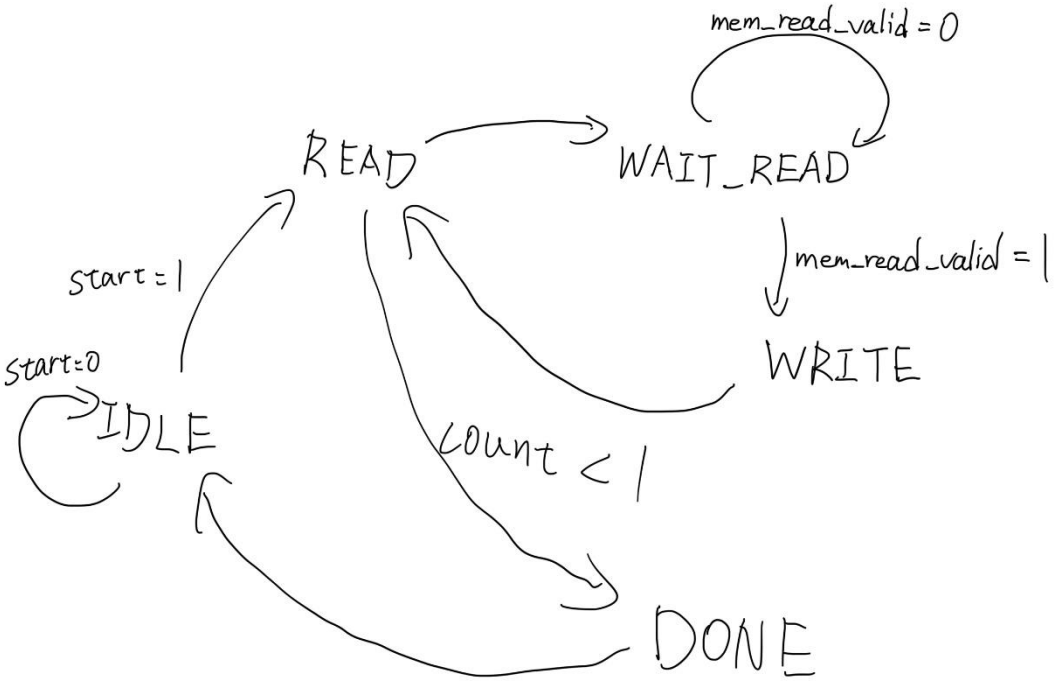
READ：发起读取请求；

WAIT_READ：等待 mem_read_valid 为高，数据有效；

WRITE：使用 data_buffer 写入数据；

DONE：搬运结束，发出 done 信号。

我自己根据大模型的代码画了一个大致的状态转换图（正好形式化验证学了自动机相关的理论）：



初始状态为IDLE空闲状态，箭头为状态转化条件。

第三个部分是状态机的实现部分和rst（异步复位）的实现。

首先说明rst（异步复位）的实现：

异步复位指的是**无论时钟状态如何，只要复位信号有效，电路立即进入复位状态**。与之相对的是**同步复位**，它只在时钟上升沿时起效。

因此，当检测到异步复位信号时，DMA应立即复位，便有了如下语句：

```
if (rst) begin
    rst <= 0;
    state <= IDLE;
    done <= 0;
    mem_read_en <= 0;
    mem_write_en <= 0;
    mem_read_addr <= 0;
    mem_write_addr <= 0;
    src_ptr <= 0;
    dst_ptr <= 0;
    count <= 0;
    data_buffer <= 0;
end
```

第一句always @(posedge clk or posedge rst) 表示电路将在 **clk 的上升沿**正常运行**或者**在 **rst 的上升沿**立即进入复

位状态（不等时钟），因此只要 rst 信号从低变高，不管当前时钟周期走到哪，马上执行 if（rst）的逻辑，重置所有寄存器，以此实现异步复位。

接下来是IDLE的实现：

```
IDLE: begin
    done <= 0;
    mem_read_en <= 0;
    mem_write_en <= 0;
    if (start) begin
        src_ptr <= src_addr;
        dst_ptr <= dst_addr;
        count <= length;
        state <= READ;
    end
end
```

实现了如下功能：

1）初始化状态（实际上就是不断更新读写的使能位状态，清除 done，禁止读写）；

2）等待 start 信号（该处使用忙等实现）；

3）加载起始地址和传输长度（if条件下的赋值语句）；

4）跳转到 READ 状态（下一个clk上升沿时触发）。

READ状态的实现：

```
READ: begin
    if (count > 0) begin
        mem_read_addr <= src_ptr;
        mem_read_en <= 1;
        state <= WAIT_READ;
    end else begin
        state <= DONE;
    end
end
```

实现了如下功能：

1）向外部内存发送读请求（使能内存读取信号）；

2）设置读地址为 src_ptr；

3)跳转到 WAIT_READ 状态(如果还有需要搬运的数据的话，下一个clk上升沿时触发)。

4)跳转到DONE状态（如果数据传输完毕的话，下一个clk上升沿时触发）

该状态有一个特别的设计点是count的判断，如果count不合法，程序仍然能够正常运行，而且WRITE状态就是依靠READ来实现的，READ使count小于1的搬运数量跳转到搬运完成（DONE）。后面WRITE的部分会再解释一次。

WAIT_READ状态的实现：

```
WAIT_READ: begin
    mem_read_en <= 0;
    if (mem_read_valid) begin
        data_buffer <= mem_read_data;
        src_ptr <= src_ptr + 4;
        state <= WRITE;
    end
end
```

实现了如下功能：

1）等待外设返回的数据有效（同样使用忙等实现）；

2）将 mem_read_data 存入 data_buffer；

3）跳转到WRITE状态。

可以从代码中看出，当且仅当内存数据准备好时，状态才能发生改变，这里我认为有一个问题，如果中间发生了任何事使mem_read_valid始终不为1，dma控制器就陷入了死循环，于是我询问了GPT该处为什么怎么设计，会不会有缺陷，GPT回答如下：

🎯 **设计这种等待的初衷**

✅ **1. 配合真实的总线/外设延迟**

在实际硬件中，例如 AXI、AHB、APB 等总线协议，**内存读操作是需要等待返回信号的。**
而且这个等待是不确定的 —— 你发出一个 `read` 请求后，数据可能在 1 个周期后回来，也可能 5 个周期才到，这取决于总线仲裁和内存响应速度。

➡️ 所以 `WAIT_READ` 这个状态，就是在**模拟或适配实际总线的延迟行为。**

✅ **2. 防止数据不一致**

如果不等 `mem_read_valid == 1` 就直接跳到写阶段，可能导致 `mem_read_data` 中的数据还是旧值，**搬运的数据错误或不一致。**
这个状态就是在实现**数据握手确认**：你不给我确认，我不写。

GPT同时给出了一个改进设计，即加入一个超时机制，超时则reset：

```
WAIT_READ: begin
    mem_read_en <= 0;
    if (mem_read_valid) begin
        data_buffer <= mem_read_data;
        state <= WRITE;
        wait_counter <= 0;
    end else begin
        wait_counter <= wait_counter + 1;
        if (wait_counter >= TIMEOUT_LIMIT)
            soft_rst <= 1;
    end
end
```

Soft_rst的作用此处就凸显出来，用于异常情况复位，原先的复位判断也要同步更改：

```
if (rst || soft_rst) begin
```

TIMEOUT_LIMIT设置为一个localparm，是一个常量。

WRITE状态的实现：

```
WRITE: begin
    mem_write_addr <= dst_ptr;
    mem_write_en <= 1;
    dst_ptr <= dst_ptr + 4;
    count <= count - 1;
    state <= READ;
end
```

实现了如下功能：

1）将 data_buffer 写入目标地址 dst_ptr；

2）更新地址指针和计数器；

3）判断是否完成搬运。

此处的设计需要和READ状态的设计联动理解，由于在设计要求部分就已经假定以一个字（32位）为搬运单位，指针是4个字节4个字节进行增长的，同时，在READ状态会对count进行判断，这样就知道是否搬运完毕，跳转到DONE状态。

DONE状态的实现：

```
DONE: begin
    done <= 1;
    mem_write_en <= 0;
    state <= IDLE;
end
```

实现了如下功能：

1）所有数据搬运完成；

2）拉高 done 信号，通知外部模块；

3）自动回到 IDLE，等待下次搬运。

此处拉低了内存写信号，防止了潜在的漏洞。

三、 设计预期成果

1）支持基本的32位数据块传输；

2）支持可配置的搬运长度；

3）提供 done 信号指示 DMA 操作完成；

4）采用 FSM 分阶段执行，便于分析；

5）模拟读取延迟（mem_read_valid）机制，贴近实际硬件行为；

6）输出读写地址、写数据、控制信号，方便仿真波形观测。

7）可拓展为支持 burst 模式、写握手、AXI 接口兼容版本；

四、 总结

由于我是大二,尚未学习FPGA相关课程,网上搜索相关FPGA环境教程均较为复杂，花费时间较多且环境无法配成功，此处无法测试gpt代码的可行性,但是,在分析gpt代码的过程中,还是学到了不少dma实际实现相关的知识，并头一次接触到verillog相关语言,使我进一步加深了对dma的认识,也对gpt现在的代码水平有了新的认知。gpt是一个很好的学习工具,在理解这些技术的过程中，没有gpt的帮忙，所花的时间肯定会更多。

五、 代码附录

```
module simple_dma (
    input        clk,              // 时钟信号
    input        rst,              // 异步复位信号
    input        start,            // 启动DMA
    output reg   done,             // 传输完成标志

    input  [31:0] src_addr,        // 源地址起点
    input  [31:0] dst_addr,        // 目的地址起点
    input  [31:0] length,          // 数据长度（单位:
字）
```

```
    input  [31:0] mem_read_data,      // 内存读取数据
    input         mem_read_valid,     // 读取数据有效信
号（模拟内存延迟）
    output [31:0] mem_write_data,     // 写入内存的数
据
    output reg    mem_read_en,        // 启用内存读取
    output reg    mem_write_en,       // 启用内存写入
    output reg [31:0] mem_read_addr,  // 当前读地址
    output reg [31:0] mem_write_addr  // 当前写地址
);

    // 内部寄存器
    reg [31:0] src_ptr, dst_ptr;
    reg [31:0] count;
    reg [31:0] data_buffer;
    reg soft_rst;

    // 超时次数
    localparam TIMEOUT_LIMIT = 100;
```

```
// 状态机状态定义
typedef enum logic [2:0] {
    IDLE,
    READ,
    WAIT_READ,
    WRITE,
    DONE
} state_t;

state_t state;

// 输出数据来自内部buffer
assign mem_write_data = data_buffer;

always @(posedge clk or posedge rst) begin
    if (rst || soft_rst) begin
        rst <= 0;
        state <= IDLE;
        done <= 0;
        mem_read_en <= 0;
```

```
        mem_write_en <= 0;
        mem_read_addr <= 0;
        mem_write_addr <= 0;
        src_ptr <= 0;
        dst_ptr <= 0;
        count <= 0;
        data_buffer <= 0;
    end
    else begin
        case (state)
            IDLE: begin
                done <= 0;
                mem_read_en <= 0;
                mem_write_en <= 0;
                if (start) begin
                    src_ptr <= src_addr;
                    dst_ptr <= dst_addr;
                    count <= length;
                    state <= READ;
                end
```

```
            end

            READ: begin
                if (count > 0) begin
                    mem_read_addr <= src_ptr;  // 设置
读地址
                    mem_read_en <= 1;          // 使能读
取
                    state <= WAIT_READ;        // 等待数
据有效
                end else begin
                    state <= DONE;             // 传输完成
                end
            end

          WAIT_READ: begin
            mem_read_en <= 0;
            if (mem_read_valid) begin
                data_buffer <= mem_read_data;
                state <= WRITE;
```

```
                wait_counter <= 0;
            end else begin
                wait_counter <= wait_counter + 1;
                if (wait_counter >= TIMEOUT_LIMIT)
                    soft_rst <= 1;                  // 触
发软复位
            end
        end

            WRITE: begin
                mem_write_addr <= dst_ptr;     // 设置
写地址
                mem_write_en <= 1;             // 启用写
操作
                dst_ptr <= dst_ptr + 4;
                count <= count - 1;
                state <= READ;                 // 返回继续
读取
            end

            DONE: begin
```

```
                done <= 1;
                mem_write_en <= 0;
                state <= IDLE;              // 结束，等
待下一次 start
            end
        endcase
    end
end
endmodule
```