在五级流水 CPU 上实现 AXI 总线

姓名：梁朝阳　**专业**：密码科学与技术

目录

摘要

AXI 总线实现
CPU 加上总线

关键词：

1 实验要求

参考《CPU 设计实战》的第 8 章，自己动手实现总线接口相关功能。要求：

1. 根据自己情况，类 SRAM 总线和 AXI 总线完成一项支持即可，建议选择 AXI 总线（需要查阅 AXI 总线文档）。
2. 虽然说不建议使用当前五级流水的代码了，但是我学习后想保持一个实验的连续性，而且我想把这个 CPU 实现完全，因为还是采用了这个 CPU 的代码。

2 AXI 原理

3 总览

AXI 总线是一种突发总线，突发传输。一直连续的传输，比如说突发 8 次传输，就是指传输数据连续的传输。（突发传输的地址是连续的，数据是连续的）

1. 读：地址控制（主机 (Master) 发送读地址）-> 从机 (Slave) 发送数据。
2. 写：地址控制（主机发送读地址）-> 主机发送数据 -> 接受完后 response。

4 AXI 的五个通道的信号

4.1 全局信号

Table 1: 全局信号 | Global Signals

信号	来源	描述
ACLK	时钟源	全局时钟信号，所有信号在时钟上升沿采样
ARESETn	复位源	全局复位信号，低电平有效

4.2 写地址

Table 2: 写地址通道信号 |Write address channel signals

通道	来源	宽度	描述
AWID[3:0]	Master	4	写地址 ID
AWADDR[31:0]	Master	32	写地址
AWLEN[3:0]	Master	4	给出了突发传输几次
AWSIZE[2:0]	Master	3	突发传输的数据宽度。例如有 32 个字节，那么 size 是 4 。
AWBURST[1:0]	Master	2	突发传输类型 †
AWLOCK[1:0]	–	2	AXI 不支持锁
AWCACHE[3:0]	Master	4	cache 类型
AWPROT[2:0]	Master	3	Protection type. (不是端口, 而是保护)
AWVALID	Master	1	写地址有效
AWREADY	Slave	1	写地址准备好 ‡

† 固定自增扫描打包，00: fixed, 01: increment, 10: wrapping.

‡ AWVALID & AWREADY 为 1 时可以数据有效

4.3 写数据

Table 3: 写地址通道信号 |Write address channel signals

通道	来源	宽度	描述
WID[3:0]	Master	4	写数据 ID
WDATA[31:0]	Master	32	写数据
WSTRB[3:0]	Master	4	（掩码/闪光）写数据有效字节 †
WLAST	Master	1	写数据最后一个
WVALID	Master	1	写数据有效
WREADY	Slave	1	写数据准备好

† 例如数据：32'H1234_5678 和掩码：4'BO111 代表：12 无效，其余有效。WSTRB[n] corresponds to WDATA[(8 × n) + 7:(8 × n)].

4.4 写响应

Table 4: 写响应通道信号 |Write response channel signals

通道	来源	宽度	描述
BID[3:0]	Slave	4	写响应 ID
BRESP[1:0]	Slave	2	写响应状态 †
BVALID	Slave	1	写响应有效
BREADY	Master	1	写响应准备好

† 写响应状态：00: OKAY, 01: EXOKAY, 10: SLVERR, 11: DECERR。

4.5 读地址

几乎完全和**??**一样。

4.6 读数据

把 RESP 信号和写数据信号并在一起了，功能类似写信号。

4.7 低功耗接口信号

Table 5: 低功耗接口信号（Low Power Interface Signals）

信号名称	来源	描述
CSYSREQ	时钟控制器（Clock controller）	SYS low-pow Request
CSYSACK	外设（Peripheral device）	ACK
CACTIVE	外设（Peripheral device）	时钟活跃指示。†

† 1= 需要时钟，0= 不需要时钟。表示外设是否需要其时钟信号。

5 代码实现历程

5.1 接口与参数定义

AXI Master 模块的关键参数如下所示：

```
module axi_full_master #(
    parameter C_M_AXI_ID_WIDTH = 1,
    parameter C_M_AXI_ADDR_WIDTH = 32,
    parameter C_M_AXI_DATA_WIDTH = 32,
    parameter C_M_TARGET_SLAVE_BASE_ADDR = 32'h00000000
)(
    input wire                          M_AXI_ACLK,
    input wire                          M_AXI_ARESETN,
```

Master 的接口包括五条 AXI 通道及用户控制接口，用户接口用于控制：操作类型（读/写）、启动信号、访问起始地址、突发长度、写入数据通道、读出数据通道

这使得 Master 能够作为上层模块（CPU 模拟器或测试逻辑）的直接驱动对象。

5.2 二进制位宽计算函数

AXI 协议中的 AWSIZE/ARSIZE 字段需要根据数据宽度自动计算。使用如下函数：

```
function integer clogb2;
    input integer number;
    integer i;
begin
    clogb2 = 0;
    for(i = number-1; i > 0; i = i >> 1)
        clogb2 = clogb2 + 1;
end
endfunction
```

例如 10 最后算出的位宽就是 2。

5.3 Master 内部寄存器与信号

Master 的关键信号包括：

- 地址锁存寄存器（读/写独立）
- 长度计数器
- 当前突发类型
- 写数据 beat 计数器

- 状态机状态寄存器
- 用户侧 busy/done/error 状态

示例：

```
reg [C_M_AXI_ADDR_WIDTH-1:0] addr_reg;
reg [7:0]                    len_reg;
reg                          rw_reg;
reg [7:0]                    wbeat_cnt;
reg                          error_reg;
reg                          done_reg;
```

5.4 协议状态机设计

AXI Master 包含两套状态机：

1. 写事务状态机：
   - 写地址阶段（AW）
   - 写数据阶段（W）
   - 写响应阶段（B）
2. 读事务状态机：
   - 读地址阶段（AR）
   - 读数据阶段（R）

写状态机示例：

```
localparam ST_IDLE = 3'd0,
           ST_AW  = 3'd1,
           ST_W   = 3'd2,
           ST_B   = 3'd3,
           ST_DONE = 3'd4;
```

Master 仅在用户发出 start 信号时进入事务，并在 B/R 通道结束后回到空闲状态。
写地址的控制例：

```
always @(posedge M_AXI_ACLK) begin
    if (!M_AXI_ARESETN)
        M_AXI_AWVALID <= 1'b0;
    else if (state == ST_AW)
        M_AXI_AWVALID <= 1'b1;
    else if (M_AXI_AWVALID && M_AXI_AWREADY)
        M_AXI_AWVALID <= 1'b0;
end
```

读事务逻辑与其类似，但对应使用 AR/R 通道。

5.5 Master 写数据逻辑示例

写数据通道通过用户数据驱动：

```
assign M_AXI_WDATA = user_wdata;
assign M_AXI_WVALID = (state == ST_W) && user_wvalid;
assign M_AXI_WLAST = (wbeat_cnt == len_reg - 1);
```

写响应处理：

```
if (M_AXI_BVALID && M_AXI_BREADY) begin
    if (M_AXI_BRESP != 2'b00) error_reg <= 1'b1;
end
```

Master 将异常状态反馈到 user_error。

6 AXI RAM Slave 设计

6.1 设计目标

我实现的 Slave 做到了：

- 支持连续突发（INCR）、固定（FIXED）、回绕（WRAP）
- 支持字节写使能（WSTRB）
- 支持读写并发
- 可配置 RAM 深度

Slave 模块参数如下：

```
parameter C_S_RAM_DEPTH = 256;
reg [C_S_AXI_DATA_WIDTH-1:0] ram [0:C_S_RAM_DEPTH-1];
```

6.2 地址递增函数实现

支持三种突发模式：

```
function [C_S_AXI_ADDR_WIDTH-1:0] axi_next_addr;
    input [C_S_AXI_ADDR_WIDTH-1:0] addr;
    input [1:0] burst;
    input [2:0] size;
    input [7:0] len;
begin
    integer inc = (1 << size);
    integer bytes = inc * (len + 1);

    case (burst)
```

```
        2'b00: axi_next_addr = addr;       // FIXED
        2'b01: axi_next_addr = addr + inc; // INCR
        2'b10: begin                       // WRAP
            reg [31:0] base = addr & ~(bytes-1);
            axi_next_addr = base |
               ((addr + inc) & (bytes-1));
        end
        default: axi_next_addr = addr + inc;
    endcase
end
endfunction
```

6.3 写通道逻辑

写地址握手:

```
assign S_AXI_AWREADY = ~wr_active;
always @(posedge S_AXI_ACLK) begin
    if (aw_hs) begin
        wr_active <= 1;
        wr_addr_reg <= S_AXI_AWADDR;
        wr_len_reg <= S_AXI_AWLEN;
    end
end
```

写数据:

```
if (wr_active && w_hs) begin
    integer idx = wr_addr_reg[ADDR_LSB +: RAM_ADDR_WIDTH];
    for (i=0;i<C_S_AXI_DATA_WIDTH/8;i=i+1)
        if (S_AXI_WSTRB[i])
            ram[idx][8*i +: 8] <= S_AXI_WDATA[8*i +: 8];
end
```

6.4 读通道逻辑

读地址握手:

```
assign S_AXI_ARREADY = ~rd_active;
if (ar_hs) begin
    rd_active <= 1;
    rd_addr_reg <= S_AXI_ARADDR;
    rd_len_reg <= S_AXI_ARLEN;
end
```

读数据产生:

```
integer idx_r = rd_addr_reg[ADDR_LSB +: RAM_ADDR_WIDTH];
rdata_reg <= ram[idx_r];
rvalid_reg <= 1;
rlast_reg <= (rd_cnt == rd_len_reg);
```

Slave 按协议连续输出数据直到最后一个 beat。