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

因为我想接着在五级流水上实现这个 AXI，所以主要自己找了一些资料。

参考资料：我主要参考了 b 站上的一个视频，**【如何科学的设计 FPGA：实现 AXI 总线自由之 AXI 解读】**。首先照着这个手敲了一遍 AXI，然后想办法怎么把 AXI 接到 CPU 上。单独的 AXI 不是很难，但是接入到五级流水的时候改动非常大。

AXI 总线实现：首先系统梳理了 AXI 协议的五通道握手机制、突发传输模式及其在 CPU 访存中的应用场景；随后分别实现了 AXI4-Full Master 与 AXI RAM Slave，支持 INCR、FIXED 与 WRAP 突发方式、字节写使能以及读写并行操作。在与五级流水 CPU 集成过程中，对取指阶段（IF）与访存阶段（MEM）进行了全面改造，通过状态机、握手保持、忙/完成标志及流水线 over/allow 机制，实现了 CPU 与 AXI 总线的无缝交互。

CPU 加上总线：实验过程中解决了 PC 无法更新、指令无法进入流水线、访存阶段死锁、Slave 超读等多个问题，最后实现了 AXI–CPU 通路。最后 CPU 能够正确完成指令取指与数据访存，流水线各级协同工作良好，功能仿真结果与期望一致。

关键词： AXI 总线；五级流水 CPU

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

2.1 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。

3 AXI 的五个通道的信号

3.1 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

3.2 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

3.3 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)].

3.4 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。

3.5 读地址

几乎完全和3一样。

3.6 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。

3.7 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。

4 AXI RAM Master 实现

4.1 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。

4.2 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。

4.3 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;
```

4.4 协议状态机设计/实现

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。

4.5 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。

5 AXI RAM Slave 实现

5.1 目标效果

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];
```

5.2 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
```

5.3 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
```

5.4 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。

6 AXI 总线与 CPU 的集成

6.1 系统架构设计/实现

在五级流水 CPU 中，访存操作主要发生在两个阶段：

1. **取指阶段（IF）**：从指令存储器读取指令
2. **访存阶段（MEM）**：执行 load/store 指令，访问数据存储器

为此，我采用了了**双 AXI Master** 架构：（实际上就是替换掉之前的 ROM 和 RAM，都改成 RAM）

- **指令 AXI Master**：专门用于取指，连接到指令 RAM Slave
- **数据 AXI Master**：用于 load/store 操作，连接到数据 RAM Slave

顶层模块接口修改如下：

```
module pipeline_cpu(
    input clk,
    input resetn,

    // 指令AXI Master接口
    output [31:0] M_AXI_INSTR_ARADDR,
    output [7:0] M_AXI_INSTR_ARLEN,
    output M_AXI_INSTR_ARVALID,
    input M_AXI_INSTR_ARREADY,
    input [31:0] M_AXI_INSTR_RDATA,
    input M_AXI_INSTR_RVALID,
    output M_AXI_INSTR_RREADY,
    // ... 其他AXI信号

    // 数据AXI Master接口
    output [31:0] M_AXI_DATA_AWADDR,
    output [31:0] M_AXI_DATA_ARADDR,
    // ... 其他AXI信号
);
```

6.2 取指模块（Fetch）的 AXI 接口改造

原始的 fetch 模块直接连接 inst_rom，现在需要通过 AXI 总线获取指令。主要修改包括：

6.2.1 状态机设计

设计了一个 4 状态的取指状态机：

```
parameter IDLE = 2'b00, // 空闲
        REQUEST = 2'b01, // 发起AXI请求
        PENDING = 2'b10, // 等待AXI响应
        DONE = 2'b11; // 指令就绪
```

状态转移逻辑：

```
always @(*) begin
    case(current_state)
        IDLE: begin
            if (IF_valid && !axi_busy)
                next_state = REQUEST;
        end
        REQUEST: begin
            next_state = PENDING;
        end
        PENDING: begin
            if (axi_done)
                next_state = DONE;
        end
        DONE: begin
            if (next_fetch) // 流水线接受指令
                next_state = IDLE;
        end
    endcase
end
```

6.2.2 与流水线的握手机制

关键信号说明：

- `axi_start`：向 AXI Master 发起读请求（1 周期脉冲）
- `axi_done`：AXI 读事务完成，指令有效
- `IF_over`：通知下一级指令准备就绪
- `next_fetch`：下一级允许接收新指令

6.3 访存模块（MEM）的 AXI 接口改造

MEM 模块需要处理两类操作：

1. **非访存指令**（如算术运算）：直接通过，`MEM_over` 立即置 1
2. **访存指令**（load/store）：发起 AXI 事务，等待 `axi_done`

关键逻辑：

```
// MEM完成信号：区分访存/非访存指令
assign MEM_over = (inst_load | inst_store) ?
                  axi_done : MEM_valid;

// 防止重复发起AXI请求
reg axi_started;
always @(posedge clk) begin
    if (MEM_allow_in)
        axi_started <= 1'b0;

    if (do_load && !axi_started) begin
        axi_start <= 1'b1;
        axi_started <= 1'b1;
    end
end
```

7 调试过程与问题解决

7.1 问题 1：取指阶段无法获取指令

7.1.1 现象

仿真波形显示：

- IF_pc 始终为 0x00000034（初始值）
- IF_inst 始终为 0
- IF_over 始终为 0
- 流水线完全停滞

7.1.2 根本原因

PC 更新条件设置错误：

```
// 错误的逻辑（问题代码）
always @(posedge clk) begin
    if (next_fetch && axi_done) // 两个信号很少同拍
        pc <= next_pc;
end
```

分析：

- axi_done 是 AXI 读完成信号（脉冲）
- next_fetch 是流水线允许信号，依赖于 IF_over

- IF_over 又依赖于 axi_done
- 结果：两信号时序错位，永远无法同时为 1

7.1.3 解决方案

修改 PC 更新逻辑为”上一条指令完成且允许取下一条”：

```
// 正确的逻辑
always @(posedge clk) begin
    if (!resetn)
        pc <= `STARTADDR;
    else if (next_fetch && IF_over) // 指令完成+允许
        pc <= next_pc;
end
```

7.2 问题 2：指令无法传播到后续流水级

7.2.1 现象

波形显示：

- 第一条指令成功取到：IF_inst = 0x24010001
- IF_over = 1（正确）
- 但 ID_valid 始终为 0，指令未进入 ID 级

7.2.2 根本原因

IF_over 是瞬时脉冲信号，如果 ID 级当拍没有接受，信号就消失了：

```
// 问题代码：IF_over只维持1个周期
always @(posedge clk)
    IF_over <= axi_done;
```

7.2.3 解决方案

改为**握手保持**机制：

```
// 修正：保持IF_over直到被接受
always @(posedge clk) begin
    if (!resetn)
        IF_over <= 1'b0;
    else if (axi_done) // AXI完成时置位
        IF_over <= 1'b1;
    else if (next_fetch) // 被接受时清零
        IF_over <= 1'b0;
end
```

7.3 问题 3：ID 级缺少时钟信号

7.3.1 现象

综合时报错：

```
ERROR: [VRFC 10-2989] 'clk' is not declared
       [decode.v:332]
```

7.3.2 原因

为了解决时序对齐问题，需要在 decode 模块中打拍 `IF_over` 信号：

```
// decode.v 中新增的打拍逻辑
reg IF_over_d;
always @(posedge clk) begin
    if (!resetn)
        IF_over_d <= 1'b0;
    else
        IF_over_d <= IF_over;
end
```

但 decode 模块端口没有 `clk` 输入。

7.3.3 解决方案

修改 decode 模块接口和实例化：

```
// decode.v 端口增加
module decode(
    input clk, // 新增
    input resetn, // 新增
    input ID_valid,
    // ... 其他端口
);

// pipeline_cpu.v 实例化时连接
decode ID_module(
    .clk(clk), // 新增连接
    .resetn(resetn), // 新增连接
    .ID_valid(ID_valid),
    // ... 其他连接
);
```

7.4 问题 4：非访存指令卡在 MEM 级

7.4.1 现象

波形分析：

```
Time=205000: addiu指令进入ID级, ID_over=1
Time=215000: addiu指令进入EXE级
Time=225000: addiu指令进入MEM级
Time=235000: MEM_valid=1, MEM_allow=0 ← 卡住!
             后续所有指令都停在EXE级之前
```

7.4.2 根本原因

MEM 模块的完成信号设置不当:

```
// 错误逻辑: 所有指令都等axi_done
assign MEM_over = axi_done;
```

分析: addiu 是算术指令，不需要访存，但 MEM 模块仍然等待 axi_done。由于没有发起 AXI 事务，axi_done 永远不会来，导致死锁。

7.4.3 解决方案

区分访存和非访存指令:

```
// 正确逻辑
assign MEM_over = (inst_load | inst_store) ?
                  axi_done : // 访存指令等AXI
                  MEM_valid; // 非访存立即完成
```

7.5 问题 5: 读取到错误的指令数据

7.5.1 现象

调试信息显示:

```
Time=125000 FETCH: Requesting PC=00000000
Time=155000 AXI_SLAVE: AR handshake, ARADDR=00000000
Time=165000 AXI_SLAVE READ: addr=0x00, idx=0,
            data=24010001 (正确)
Time=175000 AXI_SLAVE READ: addr=0x04, idx=1,
            data=00011100 (错误! 多读一次)
Time=195000 CPU收到: Inst=00011100 ← 错误数据
```

指令 RAM 内容:

- ram[0] = 0x24010001（正确）
- ram[1] = 0x00011100

CPU 应该读到 ram[0]，但实际读到了 ram[1]。

7.5.2 根本原因

AXI Slave 的读数据产生逻辑存在 bug：

```
// 问题代码
if (rd_active && (!rvalid_reg || r_hs)) begin
    // 没有检查是否已发完所有beat!
    idx_r = (rd_addr_reg >> 2);
    rdata_reg <= ram[idx_r];
    rd_cnt <= rd_cnt + 1;
    rd_addr_reg <= rd_addr_reg + 4; // 地址递增
end
```

问题分析：

当 `ARLEN=0`（只读 1 个 beat）时：

1. 第 1 个周期：`rd_cnt=0`，读 `ram[0]`，`rd_cnt` 变成 1，地址 +4
2. 第 2 个周期：条件 `(!rvalid_reg || r_hs)` 仍然满足，又读了 `ram[1]`
3. CPU 最终收到第 2 次读出的错误数据

7.5.3 解决方案

增加 beat 计数检查，防止超读：

```
// 修正后的逻辑
if (rd_active && (!rvalid_reg || r_hs)
    && (rd_cnt <= rd_len_reg)) begin // 增加检查

    idx_r = (rd_addr_reg >> 2);
    rdata_reg <= ram[idx_r];
    rvalid_reg <= 1'b1;
    rlast_reg <= (rd_cnt == rd_len_reg);

    rd_cnt <= rd_cnt + 1;

    // 只在非最后一拍时递增地址
    if (rd_cnt < rd_len_reg)
        rd_addr_reg <= rd_addr_reg + 4;
end

// 最后一拍握手后结束事务
if (rvalid_reg && rlast_reg && r_hs)
    rd_active <= 1'b0;
```

8 验证结果

8.1 整体功能验证

最终波形显示 CPU 正常工作：

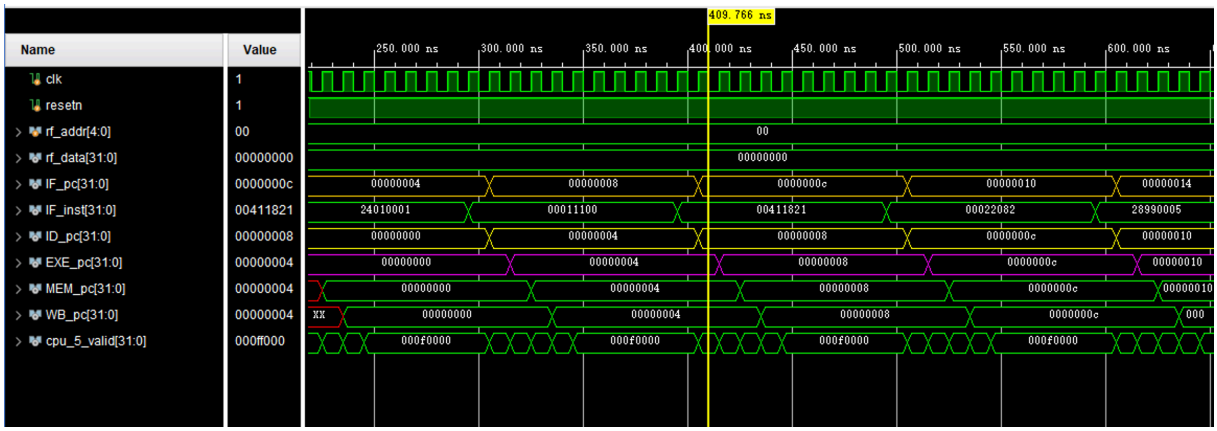

图 8.1: 整体流程

[caption=调试的时候增加的输出信息]

```
Time=195000: PC=0x00, Inst=0x24010001 (addiu $1,$0,1)
Time=295000: PC=0x04, Inst=0x00011100 (sll $2,$1,4)
Time=395000: PC=0x08, Inst=0x00411821 (addu $3,$2,$1)
Time=495000: PC=0x0C, Inst=0x00022082 (srl $4,$2,2)
...
```

可以看到之前是两周期的变成了 7 个周期，其具体为：

- AXI Master 状态转换：2 周期
- AXI 握手：1 周期
- AXI Slave 读 RAM：1 周期
- 数据传输：1 周期
- 流水线锁存：2 周期

流水线的效率反而更低了。但可能我认为扩展性应该会好一些，因为之后的 cache 等等也要加上去。

8.2 cpu_5_valid 信号变化

下面我想分析我自己写的这个信号：`cpu_5_valid` 是一个 32 位的流水线状态指示信号，其编码方式如下。

```
assign cpu_5_valid = {12'd0, // [31:20] 保留位
                      {4{IF_valid}}, // [19:16] IF级有效
```

```
                {4{ID_valid}}, // [15:12] ID级有效
                {4{EXE_valid}}, // [11:8] EXE级有效
                {4{MEM_valid}}, // [7:4] MEM级有效
                {4{WB_valid}}}; // [3:0] WB级有效
```

从波形图中可以清晰观察到流水线的**充填过程**（Pipeline Filling）：

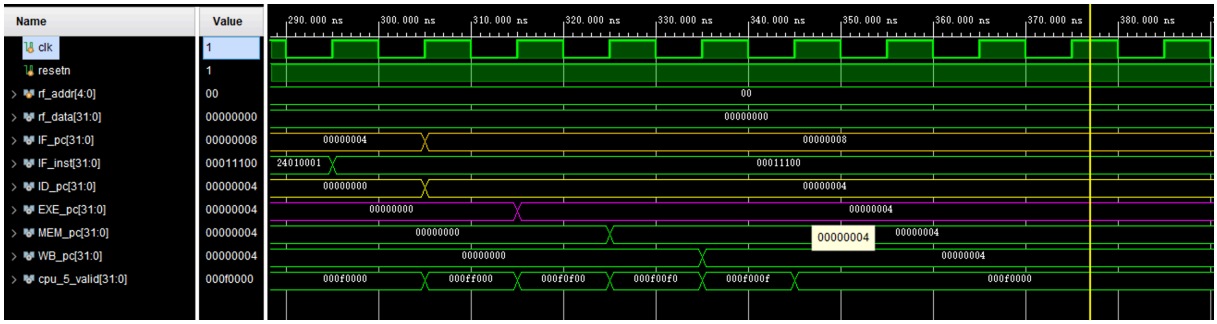

图 8.2: 流水线充填过程波形图

8.2.1 信号变化分析

1. 初始状态（t=290ns-300ns）

- `cpu_5_valid = 0x000F0000`
- 解析：仅 IF 级有效（`IF_valid=F`，其余为 0）
- 说明：第一条指令进入取指阶段，后续流水级均空闲

2. 第一条指令推进（t=300ns-310ns）

- `cpu_5_valid = 0x000FF000`
- 解析：IF 和 ID 级同时有效
- 说明：第一条指令（`PC=0x04`，`Inst=0x24010001`）进入 ID 级译码，同时第二条指令进入 IF 级

3. 流水线继续充填（t=310ns-330ns）

- `cpu_5_valid = 0x000F0F00`（t 315ns）
- 解析：IF 和 EXE 级有效，ID 级已空
- 说明：第一条指令进入执行阶段，流水线出现**气泡**（bubble）
- `cpu_5_valid = 0x000F00F0`（t 325ns）
- 解析：IF 和 MEM 级有效
- 说明：第一条指令进入访存阶段
- `cpu_5_valid = 0x000F000F`（t 335ns）

- 解析：IF 和 WB 级有效
- 说明：第一条指令进入写回阶段

4. 稳定运行阶段（t>340ns）

- `cpu_5_valid = 0x000F0000`（周期性变化）
- 说明：由于 AXI 访存延迟，流水线未能达到理想的 5 级满载状态
- 观察到 MEM 级的 PC 停留在 `0x00000004` 较长时间（高亮显示）
- 表明该指令在 MEM 阶段等待 AXI 事务完成

8.2.2 关键现象解读

流水线气泡（Pipeline Bubble）

从 `cpu_5_valid` 的跳变可以看出，流水线并非连续满载，而是存在明显的气泡：

- 从 `0x000FF000`（IF+ID）直接跳到 `0x000F0F00`（IF+EXE），跳过了完整的 3 级同时有效状态
- 原因：**AXI 取指延迟**导致下一条指令无法及时进入流水线

MEM 级停顿

波形中 MEM_pc 长时间保持 `0x00000004`：

- 说明该指令为**访存指令**（load/store）
- MEM 级等待数据 AXI Master 完成总线事务
- 此时 `MEM_allow=0`，阻塞了后续指令的推进

8.3 AXI 指令总线信号分析

从波形图中可以观察到指令 AXI Master 的关键信号行为，这些信号反映了 CPU 取指过程中的 AXI 总线交互细节。

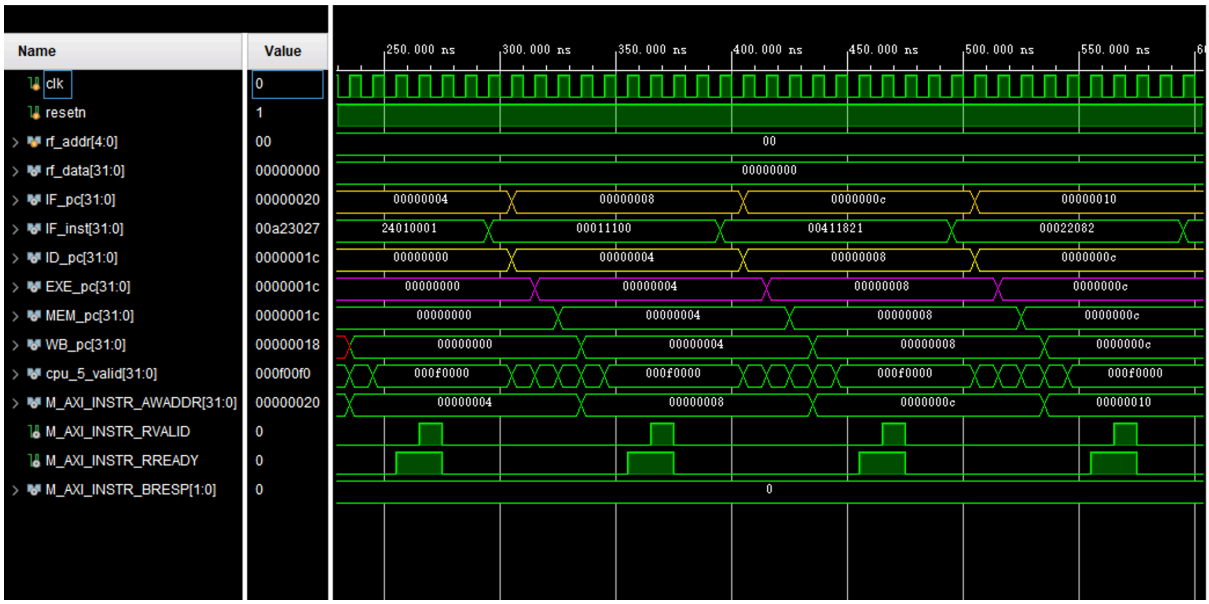

图 8.3: AXI 详细信号

8.3.1 M_AXI_INSTR_AWADDR[31:0] - 读地址信号

信号特征：

- 波形显示地址按**字对齐**递增：`0x04 → 0x08 → 0x0C → 0x10`
- 每次取指发起一个新的读地址请求
- 地址递增步长为 4 字节（32 位指令宽度）

时序关系：

- 地址变化领先于指令数据到达约 50-70ns
- 例如：`AWADDR=0x08` 时刻（t 350ns），对应的指令 `0x00011100` 在约 400ns 时出现在 `IF_inst`
- 说明从地址发出到数据返回经历了完整的 AXI 读事务周期

与流水线 PC 的对应关系：

- `M_AXI_INSTR_AWADDR` 总是与 `IF_pc` 保持同步
- 表明 Fetch 模块的 PC 值正确传递给 AXI Master
- 地址连续性验证了流水线取指的正确性

8.3.2 M_AXI_INSTR_RVALID - 读数据有效信号

信号特征：

- 表现为**周期性脉冲**，每个脉冲对应一次成功的指令读取
- 脉冲宽度约 1-2 个时钟周期
- 脉冲间隔约 100ns，与取指周期一致

AXI 握手分析：

```
// AXI读数据通道握手条件
读数据传输成功 = RVALID & RREADY
```

从波形可以看出：

- RVALID 拉高后，需要等待 RREADY 同时为高才完成握手
- 每次 RVALID 脉冲后，IF_inst 立即更新为新的指令数据
- 例如：t 300ns 处 RVALID 有效，IF_inst 更新为 0x24010001

与 cpu_5_valid 的关联：

- RVALID 有效时，cpu_5_valid 中 IF 级标志保持有效
- 说明取指模块正确响应了 AXI Slave 返回的数据
- 验证了 axi_done 信号与 RVALID 的逻辑关系

8.3.3 M_AXI_INSTR_RREADY - 读数据准备信号

信号特征：

- 与 RVALID 时序**紧密配合**，形成握手对
- 波形显示 RREADY 几乎总是在 RVALID 之前或同时拉高
- 说明 CPU 端（Master）始终准备好接收指令数据

实现分析：

在实现中，RREADY 被设置为常高：

```
// AXI Master 中的实现
.user_rready(1'b1) // 指令读取通道永远ready
```

握手时序验证：

- 从波形看，每次 RVALID & RREADY 同时为高时
- IF_inst 准确更新，PC 递增
- 流水线推进，cpu_5_valid 变化
- 完整验证了 AXI 读数据通道的正确性

8.3.4 M_AXI_INSTR_BRESP[1:0] - 写响应信号

信号特征：

- 波形显示该信号始终为 0（即 2'b00 = OKAY）
- 在整个仿真过程中无变化

原因分析：
指令 AXI 接口是**只读通道**，因此写响应信号不会被使用：

```
// 指令AXI Master配置
.user_rw(1'b1), // 固定为读操作
```

- BRESP 属于 AXI **写响应通道**（B 通道）
- 取指只使用**读地址通道**（AR）和**读数据通道**（R）
- 写地址（AW）、写数据（W）、写响应（B）三通道在指令总线中未激活
- BRESP=0 表示默认的 OKAY 状态，符合协议要求

对比：数据 AXI 总线
与之相对应，数据 AXI Master 需要支持 store 指令，因此：

- 数据总线的 BRESP 会在 store 操作时产生有效响应
- 波形中如果有 store 指令执行，会看到 M_AXI_DATA_BRESP 的变化
- 验证了指令总线和数据总线的**功能分离**设计/实现

8.3.5 AXI 信号时序总结

通过对这四个关键信号的分析，可以总结出完整的 AXI 取指时序流程：

1. **地址阶段**（AR 通道）：
   - Fetch 模块发起请求，AWADDR 更新为当前 PC
   - AXI Master 进入读地址状态，发送 ARVALID（波形中未显示）
   - Slave 响应 ARREADY，完成地址握手
2. **数据阶段**（R 通道）：
   - 经过若干周期延迟（Slave 从 RAM 读取）
   - RVALID 拉高，指令数据同步输出
   - CPU 端 RREADY 已准备好，立即完成握手
   - IF_inst 更新，axi_done 有效
3. **流水线推进**：

- `IF_over` 置 1，通知下一级
- PC 递增 +4，准备下一次取指
- `cpu_5_valid` 变化，指令进入 ID 级

从波形的 AXI 信号可以清楚看出，每个取指周期严格遵循上述流程，验证了 AXI 总线与 CPU 流水线的正确集成。

9 总结

9.1 实验成果

1. 成功实现了符合 AXI4 协议的 Master 和 Slave 模块
2. 将 AXI 总线集成到五级流水 CPU 中
3. 解决了多个复杂的时序和握手问题
4. 实现了完整的指令取指和数据访存通路

9.2 关键技术点

- **握手协议**：AXI 的 VALID/READY 双向握手机制
- **突发传输**：理解 ARLEN/AWLEN 与 beat 计数的关系
- **流水线控制**：over/allow 信号的协调
- **状态机实现**：避免死锁，保证事务完整性

附录（补充说明）

9.3 编址方法

```
93  // 初始化RAM，从地址0开始存放指令
94  initial begin
95      for(j = 0; j < C_S_RAM_DEPTH; j = j + 1) begin
96          case(j)
97              0:  ram[j] = 32'h24010001;
98              1:  ram[j] = 32'h00011100;
99              2:  ram[j] = 32'h00411821;
100             3:  ram[j] = 32'h00022082;
101             4:  ram[j] = 32'h28990005;
102             5:  ram[j] = 32'h0721000E;
103             6:  ram[j] = 32'h00642823;
104             7:  ram[j] = 32'hAC050014;
105             8:  ram[j] = 32'h00A23027;
106             9:  ram[j] = 32'h00C33825;
107             10: ram[j] = 32'h00E64026;
108             11: ram[j] = 32'h11030002;
109             12: ram[j] = 32'hAC08001C;
110             13: ram[j] = 32'h0022482A;
111             14: ram[j] = 32'h8C0A001C;
112             15: ram[j] = 32'h15450002;
113             16: ram[j] = 32'h00415824;
114             17: ram[j] = 32'hAC0B001C;
115             18: ram[j] = 32'h0C000026;
116             19: ram[j] = 32'hAC040010;
117             20: ram[j] = 32'h3C0C000C;
118             21: ram[j] = 32'h004CD007;
119             22: ram[j] = 32'h275B0044;
120             default: ram[j] = 32'h00000000; // nop
121         endcase
122     end
```

图 9.4: 硬编址

我把代码写死在了 slave 模块的初始模块中，最开始采取 AI 建议的一种加载方式，发现不行。这样的一个缺点是，因为我 mem.v 实例化的也是这个模块，导致了 mem RAM 初始的时候不是全 0，不过这个不影响正常运行。