中国科学院大学
《计算机体系结构》实验报告

组员 1 武弋 2022K8009929002

组员 2 王泽黎 2022K8009929011

组员 3 张家玮 2022K8009929010

实验项目编号 5 实验名称 AXI 总线接口设计专题实验

一、 实验内容简介

exp14：添加类 SRAM 总线支持，将 CPU 对外接口修改为类 SRAM 总线接口，在采用握手机制的 block RAM 的 SoC 验证环境中完成 exp14 对应 func 的随机延迟功能验证。

exp15：添加 AXI 总线支持，实现了一个类 SRAM-AXI 的 2x1 的转接桥，然后拼接上 exp14 完成的类 SRAM 接口的 CPU，将 myCPU 封装为 AXI 接口。

exp16：完善 AXI 总线接口设计使其在采用 AXI 总线的 SoC 验证环境里完成 exp16 对应 func 的随机延迟功能验证，成功通过了仿真和上板验证。

二、 RTL 源码及解释

- **添加类 SRAM 总线支持。**

本部分中，将 bram 接口改为了相对贴近真实内存的类 SRAM 接口，接口部分代码如下：

```
module mycpu_top(
    input wire      clk,
    input wire      resetn,
    // inst sram interface
    output wire     inst_sram_req, // 指令SRAM的请求信号
    output wire     inst_sram_wr,  // 指令SRAM的写使能信号
    output wire [ 1:0] inst_sram_size, // 指令SRAM的数据大小
    output wire [ 3:0] inst_sram_wstrb, // 指令SRAM的写位数信号
    output wire [31:0] inst_sram_addr, // 指令SRAM的地址
    output wire [31:0] inst_sram_wdata, // 指令SRAM的写数据
    input wire [31:0] inst_sram_rdata, // 指令SRAM的读数据
    input wire      inst_sram_addr_ok, // 指令SRAM的地址有效信号
    input wire      inst_sram_data_ok, // 指令SRAM的数据有效信号
    // data sram interface
    output wire     data_sram_req, // 数据SRAM的请求信号
    output wire     data_sram_wr,  // 数据SRAM的写使能信号
    output wire [ 1:0] data_sram_size, // 数据SRAM的数据大小
    output wire [ 3:0] data_sram_wstrb, // 数据SRAM的写位数信号
    output wire [31:0] data_sram_addr, // 数据SRAM的地址
    output wire [31:0] data_sram_wdata, // 数据SRAM的写数据
    input wire [31:0] data_sram_rdata, // 数据SRAM的读数据
    input wire      data_sram_addr_ok, // 数据SRAM的地址有效信号
    input wire      data_sram_data_ok, // 数据SRAM的数据有效信号
    // trace debug interface
    output wire [31:0] debug_wb_pc,
```

```
    output wire [ 3:0] debug_wb_rf_we,
    output wire [ 4:0] debug_wb_rf_wnum,
    output wire [31:0] debug_wb_rf_wdata
);
```

其原理是 CPU 给 SRAM 发送访存信号及访存数据信息，SRAM 根据访存信号及访存数据信息进行访存操作，然后将访存数据及其有效信息返回给 CPU，实现了 CPU 与 SRAM 之间的通信。

首先来看 CPU 输出给指令 SRAM 的信号：

```
assign inst_sram_req = pipe_allowin[0] && !((ex_WB || has_int_WB) && br_taken); // instruction
    memory enable
assign inst_sram_wr  = 1'b0; // instruction memory write enable
assign inst_sram_wstrb = 4'b0; // instruction memory strb
assign inst_sram_size = 2'b10; // instruction memory size
assign inst_sram_addr = nextpc; // instruction memory address
assign inst_sram_wdata = 32'b0; // instruction memory write data
```

取指请求信号是这些信号中相对最复杂的，发出取指请求的条件为 IF 级允许进入，且当前不因存在跳转/存在异常/存在中断而处于阻塞状态。

CPU 不会向指令 SRAM 写数据，所以写使能信号为 0，写数据为 0，写位数信号为 0，作为固定信息。

size 信号也固定为 2'b10，表示数据大小为 32 位。

取指地址则固定取 nextpc，即下一条指令的地址，这是因为我们实现的 CPU 会在 pre-IF 级就发送取值地址，方便及时更新 PC。

接着是 CPU 输出给数据 SRAM 的信号：

```
assign data_sram_req = (inst_ld_w_MEM || inst_ld_b_MEM || inst_ld_h_MEM || inst_ld_bu_MEM
    || inst_ld_hu_MEM || inst_st_w_MEM || inst_st_b_MEM || inst_st_h_MEM)
    && pipe_valid[3] && ~has_int_MEM && ~ex_MEM && ~has_int_WB && ~ex_WB
    && (current_state == NR || current_state == WA);
assign data_sram_addr = alu_result_MEM;
assign data_sram_wstrb = (inst_st_b_MEM ? (4'h1 << data_sram_addr[1:0]) :
    inst_st_h_MEM ? (4'h3 << data_sram_addr[1:0]) :
    4'hf)
    & {4{mem_we_MEM && pipe_valid[3] && ~has_int_MEM && ~ex_MEM && ~has_int_WB && ~ex_WB}}; // If
        has exception in the pipeline, do not write to data_sram
assign data_sram_wr = |data_sram_wstrb;
assign data_sram_size = inst_ld_h_MEM || inst_ld_hu_MEM || inst_st_h_MEM ? 2'b01 :
    inst_ld_b_MEM || inst_ld_bu_MEM || inst_st_b_MEM ? 2'b00 :
    2'b10;
assign data_sram_wdata = inst_st_b_MEM ? {4{rkd_value_MEM[ 7:0]}} :
    inst_st_h_MEM ? {2{rkd_value_MEM[15:0]}} :
    rkd_value_MEM;
```

相比指令 SRAM，数据 SRAM 的访存信号更加复杂，因为数据 SRAM 既可以被读出，也可以被写入，且其访存信息会因指令不同而改变。

数据 SRAM 的请求信号会在 MEM 级正常工作（该流水级有效且无阻塞），且该级指令为访存类指令时有效。并且，为了确保数据正确性，数据 SRAM 请求信号会在接收到地址有效信号后拉低，此处使用状态机控制。

数据 SRAM 的地址信号为 alu_result_MEM，即 ALU 计算出的访存地址。

数据 SRAM 的写位数信号由指令类型决定，st.b 为 1，st.h 为 3，st.w 为 f，且在 store 指令生效时拉高，否则

为 0。为方便实现，写使能信号则为写位数信号的或。

数据 SRAM 的数据大小信号由指令类型决定，ld.h 和 st.h 为 01，ld.b 和 st.b 为 00，其他访存指令为 10，同样需要访存类指令生效执行。

数据 SRAM 的写数据信号由指令类型决定，st.b 为 rkd_value_MEM 的低 8 位，st.h 为 rkd_value_MEM 的低 16 位，st.w 为 rkd_value_MEM。

然后是使用指令 SRAM 返回给 CPU 的信号：

```
assign inst = first_IF || inst_sram_data_ok ? inst_sram_rdata : inst_IF_reg; // 当前指令
always @(posedge clk) begin // 指令的延迟寄存器
    if (reset) begin
        inst_IF_reg <= 32'h0;
    end
    else if (inst_sram_data_ok) begin
        inst_IF_reg <= inst_sram_rdata;
    end
end

assign pipe_ready_go_preIF = inst_sram_req && inst_sram_addr_ok; // preIF级的ready_go信号

assign pipe_ready_go[0] = pipe_valid[0] && (inst_sram_data_ok || inst_IF_reg_valid) &&
    !cancel_next_inst; // IF级的ready_go信号
```

返回的指令数据会在数据有效时组成指令信号，同时存入指令寄存器中，作为指令码。而地址和数据有效信号的主要功能是用作 ready_go 信号的判断逻辑，同时还会作为跳转阻塞时的判断条件（分散在多处过于零散，此处未给出）。

最后是使用数据 SRAM 返回给 CPU 的信号：

```
assign mem_result = inst_ld_b_MEM ? data_sram_addr[1:0] == 2'h0 ? {{24{data_sram_rdata[ 7]}},
    data_sram_rdata[ 7: 0]} :
   data_sram_addr[1:0] == 2'h1 ? {{24{data_sram_rdata[15]}}, data_sram_rdata[15: 8]} :
   data_sram_addr[1:0] == 2'h2 ? {{24{data_sram_rdata[23]}}, data_sram_rdata[23:16]} :
   {{24{data_sram_rdata[31]}}, data_sram_rdata[31:24]} :
   inst_ld_h_MEM ? data_sram_addr[1:0] == 2'h0 ? {{16{data_sram_rdata[15]}}, data_sram_rdata[15:
       0]} :
   {{16{data_sram_rdata[31]}}, data_sram_rdata[31:16]} :
   inst_ld_bu_MEM ? data_sram_addr[1:0] == 2'h0 ? {{24'b0, data_sram_rdata[ 7: 0]}} :
   data_sram_addr[1:0] == 2'h1 ? {{24'b0, data_sram_rdata[15: 8]}} :
   data_sram_addr[1:0] == 2'h2 ? {{24'b0, data_sram_rdata[23:16]}} :
   {{24'b0, data_sram_rdata[31:24]}} :
   inst_ld_hu_MEM ? data_sram_addr[1:0] == 2'h0 ? {{16'b0, data_sram_rdata[15: 0]}} :
   {{16'b0, data_sram_rdata[31:16]}} :
   data_sram_rdata;

always @(*) begin
    case(current_state)
        NR: // 没有收到返回信号
            if (!data_sram_req) begin
                next_state = NR;
            end
```

```
            else if (!data_sram_addr_ok) begin
                next_state = WA;
            end
            else if (data_sram_addr_ok && data_sram_data_ok) begin
                next_state = RD;
            end
            else begin
                next_state = WD;
            end
        WA: // 收到地址有效信号
            if (data_sram_addr_ok && !data_sram_data_ok) begin
                next_state = WD;
            end
            else if (data_sram_addr_ok && data_sram_data_ok) begin
                next_state = RD;
            end
            else begin
                next_state = WA;
            end
        WD: // 写入数据有效
            if (data_sram_data_ok) begin
                next_state = RD;
            end
            else begin
                next_state = WD;
            end
        RD: // 读出数据有效
            if (pipe_allowin[4]) begin
                next_state = NR;
            end
            else begin
                next_state = RD;
            end
        default:
            next_state = NR;
    endcase
end
```

数据 SRAM 返回的数据与之前实验并未有太多变动，仍然是根据位数信息组合成 load 指令所需的数据。地址和数据有效信号则是作为了状态机的状态转换判断条件，而状态则是用来控制数据 SRAM 的请求和组成 MEM 级的 ready_go 信号。

- **实现类 SRAM-AXI 转接桥**。

本实验中，我们实现了一个转接桥模块“bridge.v”，用于将类 SRAM 接口转换为 AXI 接口。其首先包含所有的 AXI 输入输出接口信号，此处不再给出，唯一要说明的是 CPU 作为 master 端，存储模块作为 slave 端。除此之外，还要包含 CPU 的类 SRAM 接口信号，其在 bridge 中的数据传输方向与 exp 实验 CPU 中的类 SRAM 接口信号数据传输方向相反，代码如下：

```
//SRAM interface
// inst sram interface
```

```
input wire      inst_sram_req, // chip select signal of instruction sram
input wire      inst_sram_wr,
input wire [ 1:0] inst_sram_size,
input wire [ 3:0] inst_sram_wstrb,
input wire [31:0] inst_sram_addr,
input wire [31:0] inst_sram_wdata,
output wire [31:0] inst_sram_rdata,
output wire     inst_sram_addr_ok,
output wire     inst_sram_data_ok,
// data sram interface
input wire      data_sram_req, // chip select signal of data sram
input wire      data_sram_wr,
input wire [ 1:0] data_sram_size,
input wire [ 3:0] data_sram_wstrb,
input wire [31:0] data_sram_addr,
input wire [31:0] data_sram_wdata,
output wire [31:0] data_sram_rdata,
output wire     data_sram_addr_ok,
output wire     data_sram_data_ok,
```

此外，还有一些用于与 CPU 同步状态的信号，代码如下：

```
input wire      data_waddr_ok, //数据SRAM写地址传输完成
input wire      data_wdata_ok, //数据SRAM写数据传输完成
input wire      data_write_ok, //数据SRAM写入完成
input wire      data_raddr_ok, //数据SRAM读地址传输完成
input wire      data_rdata_ok, //数据SRAM读数据传输完成
input wire      inst_raddr_ok, //指令SRAM读地址传输完成
input wire      memory_access, //内存访问使能
input wire      inst_sram_using //指令SRAM正在使用标志
```

接下来是这些信号的组成逻辑，代码如下：

```
assign arid = (!memory_access | (memory_access && (data_write_ok | data_rdata_ok)) |
    inst_sram_using) ? 4'b0000 : 4'b0001; //0指令, 1数据
assign araddr = (arid == 4'b0) ? inst_sram_addr : data_sram_addr; //读地址
assign arlen = 8'b00000000; //固定为0
assign arsize = (arid == 4'b0) ? inst_sram_size : data_sram_size; //读大小
assign arburst = 2'b01; //固定为01
assign arlock = 2'b00; //固定为0
assign arcache = 4'b0000; //固定为0
assign arprot = 3'b000; //固定为0
assign arvalid = (inst_sram_req) | (data_sram_req & ~data_sram_wr); //读请求有效

assign rready = (data_raddr_ok & !data_rdata_ok) | (inst_raddr_ok & (!memory_access ||
    (memory_access && (data_write_ok || data_rdata_ok))))
              | inst_sram_using & inst_raddr_ok; // 数据读完成握手或指令读完成握手

assign awid = 4'b0001; //固定为1
assign awaddr = data_sram_addr; //写地址
```

```
    assign awlen = 8'b00000000; //固定为0
    assign awsize = data_sram_size; //写大小
    assign awburst = 2'b01; //固定为01
    assign awlock = 2'b00; //固定为0
    assign awcache = 4'b0000; //固定为0
    assign awprot = 3'b000; //固定为0
    assign awvalid = data_sram_req & data_sram_wr; //写请求有效

    assign wid = 4'b0001; //固定为1
    assign wdata = data_sram_wdata; //写数据
    assign wstrb = data_sram_wstrb; //写掩码
    assign wlast = 1'b1; //固定为1
    assign wvalid = data_waddr_ok & !data_wdata_ok; //写请求数据有效

    assign bready = data_wdata_ok; // 写数据完成二次握手

    assign inst_sram_rdata = rdata; //指令码
    assign inst_sram_addr_ok = arvalid & arready & (arid == 4'b0); //指令地址有效
    assign inst_sram_data_ok = rvalid & rready & inst_raddr_ok; //指令数据有效

    assign data_sram_rdata = {32{arid == 4'b1}} & rdata; //读数据
    assign data_sram_addr_ok = arvalid & arready & (arid == 4'b1) & ~data_sram_wr | awvalid &
        awready & (arid == 4'b1) & data_sram_wr & ~inst_sram_using; //数据地址有效
    assign data_sram_data_ok = rvalid & rready & ~data_sram_wr | bvalid & bready & data_sram_wr &
        ~inst_sram_using; //读写数据有效
```

数据类的信号直接由类 SRAM 接口信号连接，而握手信息信号则要根据 AXI 的握手逻辑进行设计。简单来说，这些握手信号的控制逻辑有以下这些原则:“握手请求信号在握手之前持续拉高、握手请求信号在握手完成之后立即并保持拉低、同一时刻只存在一个生效的握手请求信号”根据本人在上学期《计算机组成原理研讨课》中设计流水线 CPU 访存接口的经验，遵循这些原则可以保证访存不会因为冲突而出现问题。

CPU 与 bridge 状态同步信号逻辑如下：

```
    always @(posedge aclk)begin
    if(reset) begin
        inst_sram_using <= 1'b0;
    end
    else if(inst_sram_using == 1'b0 & inst_sram_req) begin
        inst_sram_using <= 1'b1;
    end
    else if(inst_sram_using == 1'b1 & inst_sram_data_ok) begin
        inst_sram_using <= 1'b0;
    end
end

always @(posedge aclk) begin
    if(reset) begin
        data_waddr_ok <= 1'b0;
    end
    else if(awready && awvalid) begin
```

```
        data_waddr_ok <= 1'b1;
    end
    else if(pipe_ready_go[3]) begin
        data_waddr_ok <= 1'b0;
    end
end

always @(posedge aclk) begin
    if(reset) begin
        data_wdata_ok <= 1'b0;
    end
    else if(wready && wvalid) begin
        data_wdata_ok <= 1'b1;
    end
    else if(pipe_ready_go[3]) begin
        data_wdata_ok <= 1'b0;
    end
end

always @(posedge aclk) begin
    if(reset) begin
        data_write_ok <= 1'b0;
    end
    else if(bvalid && bready) begin
        data_write_ok <= 1'b1;
    end
    else if(pipe_ready_go[3]) begin
        data_write_ok <= 1'b0;
    end
end

always @(posedge aclk) begin
    if(reset) begin
        data_raddr_ok <= 1'b0;
    end
    else if(data_sram_addr_ok) begin
        data_raddr_ok <= 1'b1;
    end
    else if(pipe_ready_go[3]) begin
        data_raddr_ok <= 1'b0;
    end
end

always @(posedge aclk) begin
    if(reset) begin
        data_rdata_ok <= 1'b0;
    end
    else if(inst_sram_using) begin
        data_rdata_ok <= 1'b0;
    end
```

```
    else if(rvalid & rready & memory_access & ~inst_sram_using) begin
        data_rdata_ok <= 1'b1;
    end
    else if(pipe_ready_go[3]) begin
        data_rdata_ok <= 1'b0;
    end
end

always @(posedge aclk) begin
    if(reset) begin
        inst_raddr_ok <= 1'b0;
    end
    else if(inst_sram_addr_ok) begin
        inst_raddr_ok <= 1'b1;
    end
    else if(pipe_ready_go[0]) begin
        inst_raddr_ok <= 1'b0;
    end
end
```

这类信号是在一个流水级周期之内握手的完成标志，用于 bridge 中握手信号原则的实现。由于这类信号需要用到时序逻辑，而我们不想在 bridge 中引入时序逻辑，因此将这些信号的逻辑放在了 CPU，传入到 bridge 中。

除此之外，由于 CPU 的状态逻辑与 exp14 中产生了变动，所以有许多控制信号出现了错误，所以进行了大量的调试和修改，此处不再给出具体代码，后面将给出个别典型例子。

三、 调试过程中遇到的重点问题

- **完成 exp14 时遇到的问题。**

第一个问题是分支指令阻塞时，由于一个流水级周期从之前实验 BRAM 的单周期变为了 SRAM 的多周期，导致不能正确控制阻塞的时序。通过调整 cancel 信号适配多周期的 SRAM，解决了这个问题。

第二个问题是在调试过程中，有些种子能够通过而有些无法通过。经过重设种子仿真排查，发现是在特定时间延迟下，触发中断时有信号发生了时序冲突导致错误，优化控制逻辑后问题得以解决。

- **完成 exp15 时遇到的问题。**

本实验中我们主要遇到了两类问题，第一类是指令访问和数据访问会发生冲突，解决方案是根据 bridge 中握手信号的设计原则来调整 CPU 的控制逻辑，也是为了这个目的，我们在 CPU 中添加了同步控制信号传到 bridge 中。

第二类也是上一实验中出现的阻塞时序问题，由于 cancle 信号的设计存在一定缺陷，并不能保证在各种需要跳转的情况都能正确完成阻塞，于是我们通过取到错误指令时将其转化为空指令的方式来解决这个问题。

四、 实验总结

本次实验是第二个小组合作实验，经过讨论，达成了一致同意的分工方案，每个人都完成了自己的任务。在遇到困难时，同学之间的讨论和共同探索起到了很大的作用，我们的团结协作使得实验能够顺利完成。本次实验难度相比上次也有了一定提升，每个人都花费了更多时间和精力在实验上，也学到了很多新的知识。

五、 实验分工

exp14 代码编写：张家玮、王泽黎

exp15 代码编写：武弋、王泽黎

prj5 实验报告撰写：武弋

除完成各自的任务外，小组成员之间在遇到问题时相互讨论，共同解决了实验中遇到的问题。