情况说明

一份包含了目前仲裁器部分简单设计思路和实现方法的文档。

接口描述

为了正常的运行基于 AXI 总线的功能测试和性能测试，我们必须按照功能测试以及性能测试的顶层模块中规定的接口来编写顶层模块。

工程中的 `Top.v` 以及 `Top_old.v` 是两个顶层模块的简单实现，它们的结构是类似的，但由于目前只有 `Top_old.v` 是基本可用的，所以以下的叙述将基于 `Top_old.v` 来进行。

模块对外的接口大致分为四组，第一组是以下部分：

```
module Top(
    input         aclk,
    input         aresetn,
```

这两个输入信号分别是时钟信号和复位信号，其中复位信号低电平有效。

```
    input  [5:0]  int,
```

`int` 信号是外部中断，共有 6 个。此信号应当被直接输入给 CPU。

```
    output [3:0]  arid,
    output [31:0] araddr,
    output [3:0]  arlen,
    output [2:0]  arsize,
    output [1:0]  arburst,
    output [1:0]  arlock,
    output [3:0]  arcache,
    output [2:0]  arprot,
    output        arvalid,
    input         arready,

    input  [3:0]  rid,
    input  [31:0] rdata,
    input  [1:0]  rresp,
    input         rlast,
    input         rvalid,
    output        rready,

    output [3:0]  awid,
    output [31:0] awaddr,
    output [3:0]  awlen,
    output [2:0]  awsize,
    output [1:0]  awburst,
    output [1:0]  awlock,
    output [3:0]  awcache,
    output [2:0]  awprot,
    output        awvalid,
    input         awready,

    output [3:0]  wid,
    output [31:0] wdata,
    output [3:0]  wstrb,
    output        wlast,
    output        wvalid,
    input         wready,

    input  [3:0]  bid,
    input  [1:0]  bresp,
    input         bvalid,
    output        bready,
```

以上这些信号都是 AXI 总线所需要的输入和输出信号，用于总线和 RAM 之间的数据交互。

```
    output [31:0] debug_wb_pc,
    output [3:0]  debug_wb_rf_wen,
    output [4:0]  debug_wb_rf_wnum,
    output [31:0] debug_wb_rf_wdata
);
```

以上四个信号是功能测试时，测试程序判断整个 SoC 执行是否正确所需要的信号。

模块组成

顶层模块 `Top_old.v` 由三个模块组成，分别是 `AXI_master` 、 `arbiter` 和 `Uranus` 。

`AXI_master` 是 AXI 总线的核心实现，这部分负责接管模块中所有和 AXI 有关的接口。同时，为了方便和 CPU 之间的数据交互，这部分模块会暴露出以下这些接口给 CPU：

```
    .awid_i(awid_conn),
    .awaddr_i(awaddr_conn),
    .awlen_i(awlen_conn),
    .awsize_i(awsize_conn),
    .awburst_i(awburst_conn),
    .wdata_i(wdata_conn),
    .wstrb_i(wstrb_conn),
    .arid_i(arid_conn),
    .araddr_i(araddr_conn),
    .arlen_i(arlen_conn),
    .arsize_i(arsize_conn),
    .arburst_i(arburst_conn)
```

由于 CPU 只有一组用于读取指令的 ROM 接口，和一组用于读取数据的 RAM 接口，所以实际上 CPU 不能直接和 AXI 总线做交互。因为总线中只有一组数据读取接口和一组数据数据写入接口，但是 CPU 中的 RAM 和 ROM 都需要被读取。于是我们需要一个仲裁器来将 CPU 的 RAM、ROM 请求转换为发送给 AXI 总线的操作信号。仲裁器的接口如下：

```
arbiter arbiter_0(
    .clk(aclk),
    .rst(aresetn),

    .rdata(rdata),
    .rlast(rlast),
    .wlast(wlast),
    .rvalid(rvalid),

    .ram_en(ram_en_conn),
    .ram_write_en(ram_write_en_conn),
    .ram_write_data(ram_write_data_conn),
    .ram_addr(ram_addr_conn),

    .rom_en(rom_en_conn),
    .rom_write_en(rom_write_en_conn),
    .rom_write_data(rom_write_data_conn),
    .rom_addr(rom_addr_conn),

    .stall_all(stall_all_conn),

    .ram_read_data(ram_read_data_conn),
    .rom_read_data(rom_read_data_conn),

    .awid_o(awid_conn),
    .awaddr_o(awaddr_conn),
    .awlen_o(awlen_conn),
    .awsize_o(awsize_conn),
    .awburst_o(awburst_conn),
    .wdata_o(wdata_conn),
    .wstrb_o(wstrb_conn),
    .arid_o(arid_conn),
    .araddr_o(araddr_conn),
    .arlen_o(arlen_conn),
    .arsize_o(arsize_conn),
    .arburst_o(arburst_conn)
);
```

其中：

1. 第一组接口是时钟和复位信号；
2. 第二组接口用于读取目前 AXI 总线的一些状态（后面会说到）；
3. 第三组和第四组接口用于接管 CPU 的 RAM 和 ROM 的输入信号；
4. 第五组接口的名字是 `stall_all`，用于控制 CPU 的暂停。因为 AXI 总线在读写数据的时候均会产生延迟，而且目前并没有 Cache，所以在数据读写未完成之前，必须令 CPU 暂停；
5. 第六组接口用于将从 RAM 和 ROM 读取出的数据传递给 CPU；

6. 最后一组接口用于和 AXI 总线模块连接，控制 AXI 读取或者写入合适的数据。

CPU 的接口部分描述从略。

所以目前 AXI 和 CPU 的实现基本不需要做任何改动，只有仲裁器的内部控制逻辑需要重新设计。

仲裁器的设计思路

在不考虑 Cache 的情况下，仲裁器的设计思路很简单。

AXI 总共有两组控制端，一组负责数据读取，另一组负责数据写入。由于 CPU 永远不会向 ROM 中写入数据，所以我们可以直接将 RAM 的地址传递给 AXI 的写端，并且配置特定的控制信号。这些信号可以从 `AXI_arbiter.v` 中找到：

```
assign  awid_o = 4'b0000;
assign  awaddr_o = ram_write_flag ? ram_addr : 32'h0;
assign  awlen_o = 4'b0000;
assign  awsize_o = 3'b010;
assign  awburst_o = 2'b00;
assign  wdata_o = ram_write_flag ? ram_write_data : 32'h0;
assign  wstrb_o = ram_en ? ram_write_en : 4'b0000;
```

- `awid` 是一个 ID，我们为它设置一个固定的值即可，比如 `4'b0000`；
- `awaddr` 是要写入的地址，之前说过 CPU 只会向 RAM 中写数据，所以当 RAM 的写有效时，向 `awaddr` 传递 RAM 的写入地址，其余情况传 0；
- `awlen` 是要写入的数据的总条数。这个信号和 burst 写入有关，但是因为目前的 AXI 好像并不能进行 burst 写入，所以我们直接传 `4'b0000`，代表一次只写一条数据；
- `awsize` 是每条数据占用的字节数，直接传 `3'b010`；
- `awburst` 是 burst 类型，同样因为不能进行 burst 写入，传递 `2'b00`；
- `wdata` 是要写入的数据，解释略；
- `wstrb` 可以理解为写使能信号，其意义和 CPU 传递的 RAM 的写使能信号完全相同：`4'b0000` 时写无效，`4'b0001` 时写入字节，`4'b0011` 时写入半字，`4'b1111` 时写入字。

以上我们就完成了写信号的传递。

在 AXI 读取时，我们需要注意一些问题。目前的功能测试和性能测试中，RAM 和 ROM 事实上是同一个存储设备。也就是总线上只挂载了一个 RAM，这个 RAM 同时充当了存储数据和指令的功能，我们只需要给它不同的地址，它就能取回不同的数据。

这样的话，因为 CPU 时刻需要从 ROM 中读取指令，完成取指操作，有时有需要从 RAM 中读取数据进行处理，所以我们应当做一个判断：如果 RAM 需要读取数据，就优先将 RAM 的地址传输到总线上，读出 RAM 的数据。其余情况下，将 ROM 的地址传递到总线上，读出指令数据交给 CPU。在所有的数据读取未完成之前，CPU 都应当暂停（也就是让 `stall_all` 信号为高电平）。

于是我们可以看到目前的仲裁器模块中对 AXI 读信号的传递做出了如下处理：

```
assign  arid_o = 4'b0000;
assign  araddr_o = ram_read_flag ? ram_addr : rom_en ? rom_addr : 32'h0;
assign  arlen_o = 4'b0000;
assign  arsize_o = 3'b010;
assign  arburst_o = 2'b00;
```

其中各个信号的意义和写相关的信号类似，不做解释。读地址信号方面，在 RAM 要读取时，传递 RAM 的地址，否则如果 ROM 要读取，则传递 ROM 的地址。两者皆非时传递 0。

除此之外，仲裁器必须向 RAM 和 ROM 输出两组数据，来充当 CPU 读取 RAM 和 ROM 时返回的数据。在仲裁器模块中的代码如下：

```
assign  ram_read_data = ram_read_flag ? rdata : 0;
assign  rom_read_data = !ram_read_flag && rom_en ? rdata : 0;
```

`rdata` 信号是从 AXI 总线返回的读取出的数据，我们对 RAM 和 ROM 目前的状态做出一些判断，然后在合适的时机将这个信号分发给 RAM 和 ROM 的读数据段即可。

这就是仲裁器的设计思路。但是实际实现起来会遇到许多问题，比如 CPU 的暂停信号应当如何发出？这就需要在仲裁器的其余部分编写合适的控制逻辑了。

目前的实现

目前的实现中有一些内容不可以删掉。

```
reg[1:0] wready_delay;
always @(posedge clk) begin
    if (!rst) begin
        wready_delay <= 0;
    end
    else begin
        wready_delay[0] <= wready;
        wready_delay[1] <= wready_delay[0];
    end
end
assign wready_out = wready | wready_delay[0] | wready_delay[1];
```

这段代码负责控制让 `wready` 信号延长两个周期。 `wready` 信号是 AXI 总线和从设备之间的一个握手信号，会在有数据写入时发出。由于 AXI 总线设计上的问题， `wready` 信号持续的时间太短，会导致数据无法写入，所以必须对这个信号作出延迟处理。

其他内容主要是负责让 CPU 在合适的时机暂停的。但是因为种种原因，现有的逻辑并不能在所有情况下都奏效。建议注释掉这些代码，然后从头写起 = =

其他资料

AXI 协议的描述可以在 MY-AXI 仓库中找到。