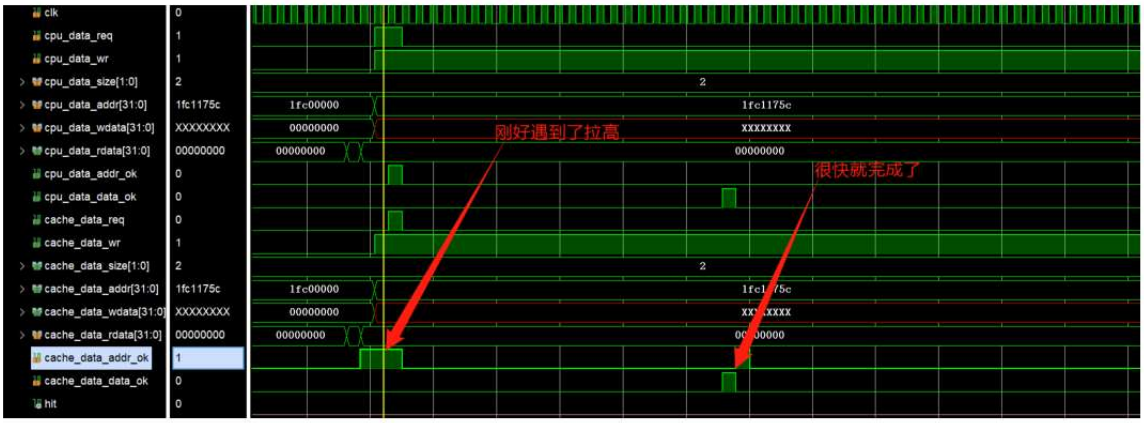
关于addr_ok

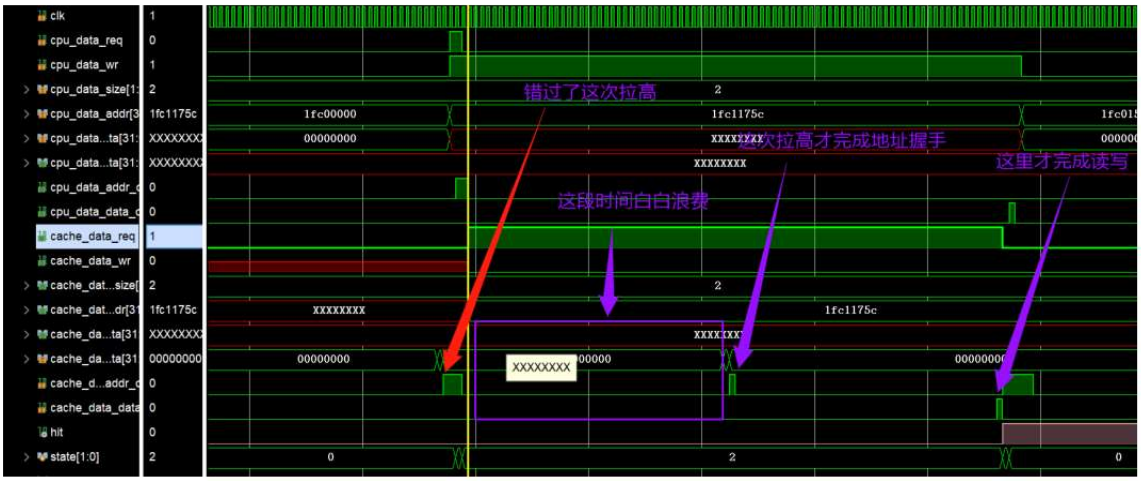
这里再解释一下。

如你所说，下图确实比较“幸运”，发出req后马上就收到addr_ok了。但是其实你发的两张图效率是一样的😀。

写直达：



上图虽然数据部分马上完成了，但CPU并没有取消暂停，因为指令还在取指呢！你看上图data_ok后后面是不是还有很长一串。那其实是指令cache在访存。对于指令cache来说它的情况和下图d_cache的处境就是一致的。发出req后很长时间才收到addr_ok。



所以，两张图区别就在于处理指令和数据请求的顺序不一样。正如lg的解释，这是由于你的数据cache比指令cache晚一个周期才发出请求，因而导致仲裁模块就先转发了指令cache的请求。

然后那个周期性的addr_ok确实是addr_ok变量和指令和数据请求都有关。这一点确实不对，但是你忽略掉这个addr_ok应该不会出错。你只需要关心你发出req后收到的addr_ok即可。

然后就是上面的情况其实是可以避免的。（需要换一个cpu_axi_interface）

我们知道mycpu_top顶层是axi接口。是通过cpu_axi_interface模块将两个类sram接口转换成了一个axi接口。

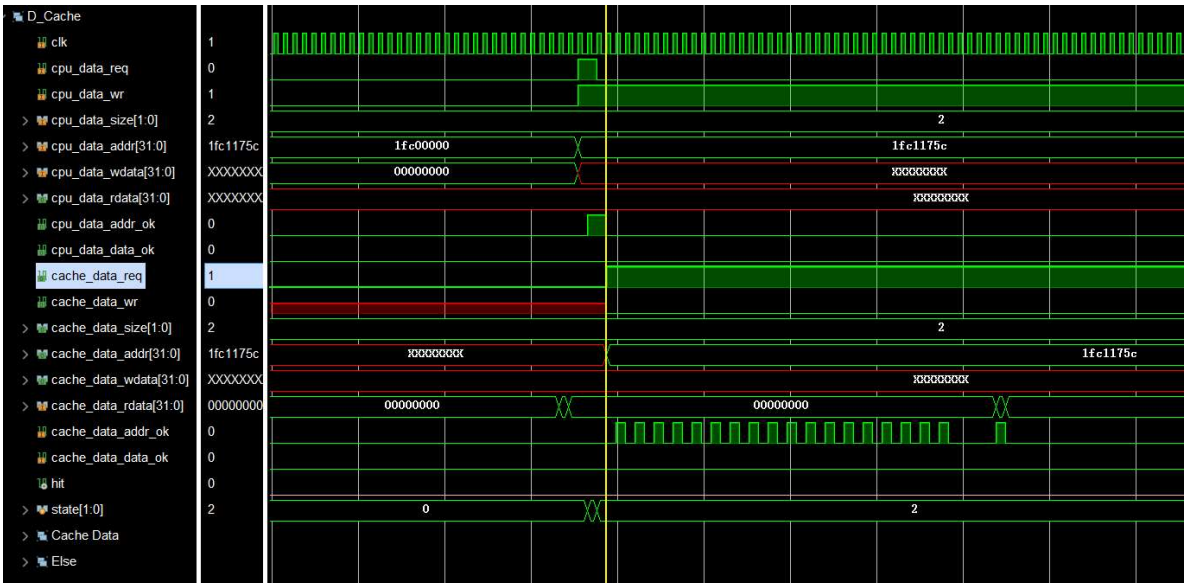
设计这个实验的时候为了防止出错，就用了龙芯杯比赛官方提供的cpu_axi_interface。但是他的实现比较简单，必须先将一个请求完全处理完毕后（data_ok）才会处理下一个请求。

但事实上其实可以连续的发送请求，即在一次请求addr_ok后马上发送下一个请求。这样的好处就是可以在等待一次请求返回数据的时候，发送下一次请求的地址，从而提高总线带宽。（从设备需要先处理地址，然后再处理返回的数据。因此处理返回数据的时候是可以同时处理下一次请求的地址的【流水线】）

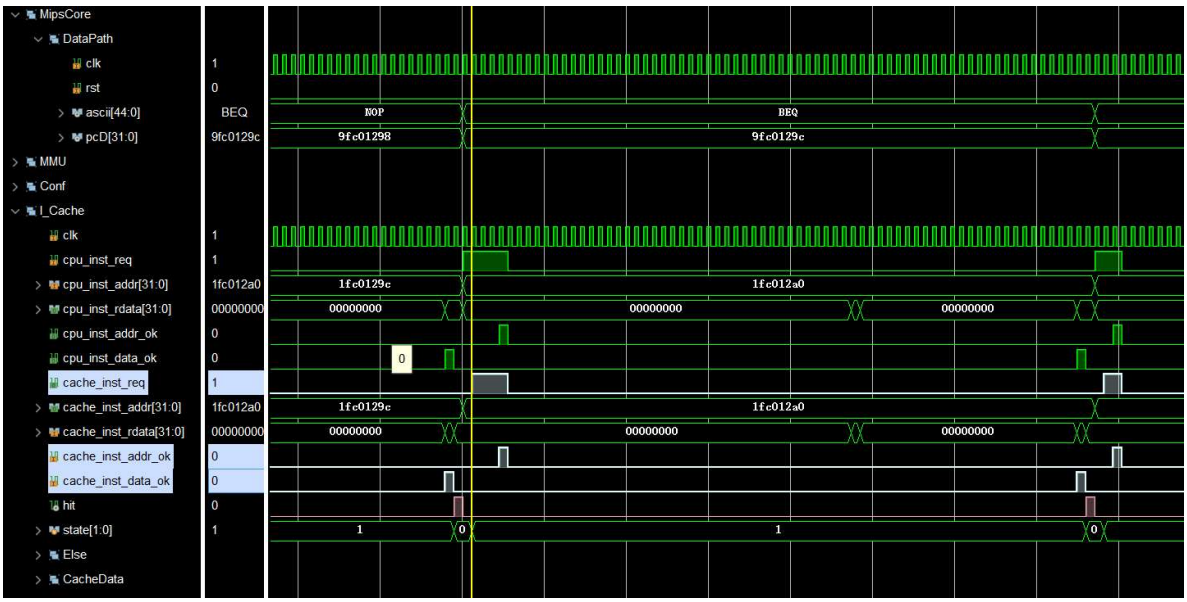
于是后来设计axi实验（）的时候就又自己又写了一个转换桥。（见附录）

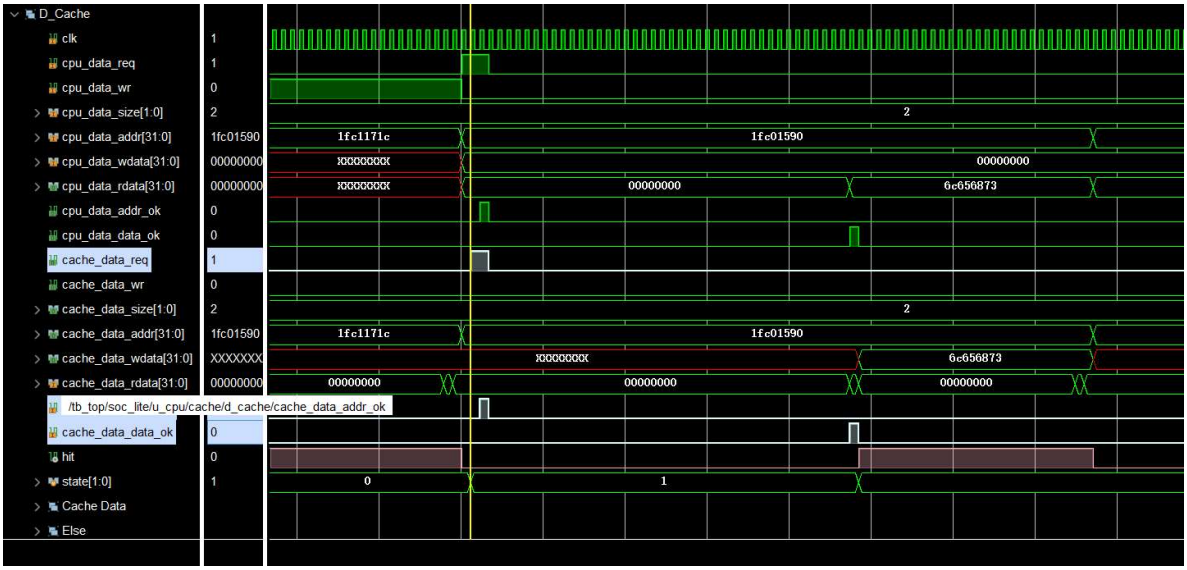
这是换上之后的波形图（这里是用的你的cache）。这里可以看到cache_data_req后，很快接收到了addr_ok。

这里也显示了你设计的一个错误，那就是在地址握手成功后没有拉低req信号。导致后面连续发送了多个数据请求，接收到了多个addr_ok。因此直接用这个转换桥会报错。



这里换上我写的一个写回cache，同一时刻指令和数据的波形图如下。可以看到指令和数据都很快返回了addr_ok。指令从req到data_ok花费的时间也小于两倍的数据req到data_ok的时间。





事实上axi接口这边还有优化的余地。如axi接口读写通道是分开的，因此是可以同时进行读和写的！

因为指令只能读，所以同时读指令和写数据是可以的（当然，同时读指令和读数据就不行了）

关于性能

其实关于性能这块，我还没有仔细研究。比如我现在也不知道测试程序中sw指令占的比例是多少。

这是我用我写的写回cache，最后得分如下

```
//原始的axi_interface
================================================================
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2ccfb

Total Count(CPU count) = 0x1463e

================================================================
Test end!
----PASS!!!

//换了axi_interface，可以看出确实有提升
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2c780

Total Count(CPU count) = 0x143c0

================================================================
Test end!
----PASS!!!
```

但是我觉得其中有一个影响较大的点是，你设计的状态机，即使在cache命中的情况下，也需要至少延迟1个周期才能返回数据（因为收到req后，需要一个周期进入CompareTag）。

这对于指令cache来说更加严重，

因为假设指令cache命中率有95%，不命中需要23个周期，命中需要1个周期。那么

23*0.05 + 1*0.95 = 2.1

23*0.05 + 2*0.95 = 3.05

可以看到浪费一个周期，对IPC影响很大。

参考实验指导书中的指令cache命中的情况，你会发现指令cache在命中的情况会马上返回addr_ok和data_ok。

这是因为在示例cache中，在req为1的那个周期，直接进行了tag比较。

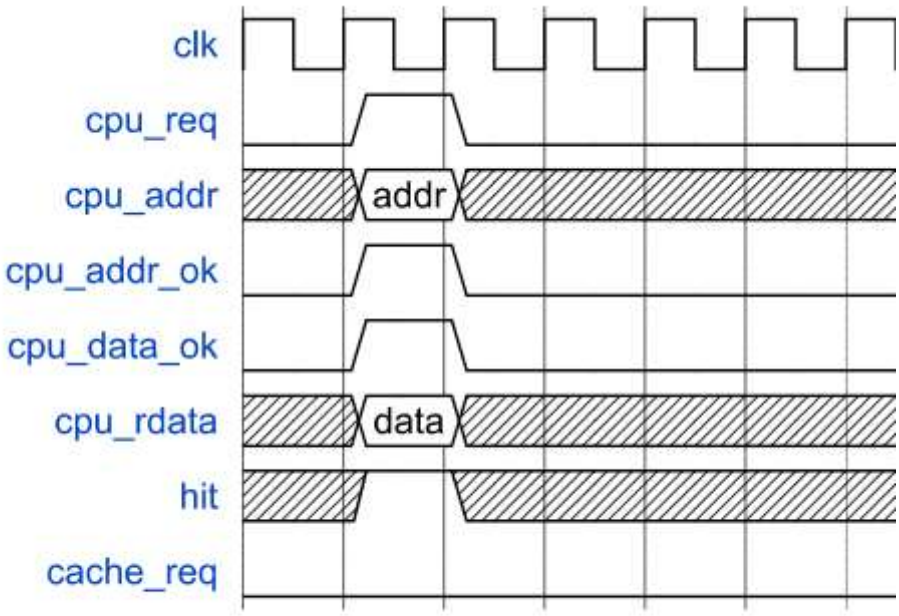


图 6: 指令 cache 命中时

当然这种写法会带来很大的时序问题，因为读取cache和tag比较全放在一个周期里了。

但实际实现中（指龙芯杯比赛），可以在CPU五级流水访存阶段的前一个阶段就将地址传给cache，进行cache行的读取，下个周期再进行tag比较。也就是将Cache分成了两级流水。

不过，作为实验，并不要求cache的跑分。你这样写状态机，反而更加清晰，容易理解，因此还是可取的。

最后，感谢你这么认真地做这个实验！有收获就好😊

附录

新的axi_interface

```
module cpu_axi_interface
(
```

```
    input         clk,
    input         resetn,

    //inst sram-like
    input         inst_req     ,
    input         inst_wr      ,
    input  [1 :0] inst_size    ,
    input  [31:0] inst_addr    ,
    input  [31:0] inst_wdata   ,
    output [31:0] inst_rdata   ,
    output        inst_addr_ok ,
    output        inst_data_ok ,

    //data sram-like
    input         data_req     ,
    input         data_wr      ,
    input  [1 :0] data_size    ,
    input  [31:0] data_addr    ,
    input  [31:0] data_wdata   ,
    output [31:0] data_rdata   ,
    output        data_addr_ok ,
    output        data_data_ok ,

    //axi
    //ar
    output [3 :0] arid         ,
    output [31:0] araddr       ,
    output [7 :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      ,
    //r
    input  [3 :0] rid          ,
    input  [31:0] rdata        ,
    input  [1 :0] rresp        ,
    input         rlast        ,
    input         rvalid       ,
    output        rready       ,
    //aw
    output [3 :0] awid         ,
    output [31:0] awaddr       ,
    output [7 :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      ,
    //w
    output [3 :0] wid          ,
    output [31:0] wdata        ,
    output [3 :0] wstrb        ,
    output        wlast        ,
```

```
    output        wvalid       ,
    input         wready       ,
    //b
    input  [3 :0] bid          ,
    input  [1 :0] bresp        ,
    input         bvalid       ,
    output        bready
);

wire rst;
assign rst = ~resetn;

reg arvalid_r;  //检测到取指或读数据时拉高，地址握手成功后拉低
reg read_sel;   //标识取指和读数据 0: inst, 1: data

always @(posedge clk) begin
    arvalid_r <= rst ? 1'b0 :
                 arvalid & arready ? 1'b0 :
                 inst_req | (data_req & ~data_wr) ? 1'b1 : arvalid_r;
    read_sel <= rst ? 1'b0 :
                data_req & ~data_wr ? 1'b1 :
                inst_req ? 1'b0 : read_sel;
end

reg write_req;  //发出写请求整个期间
reg waddr_rcv;  //写地址握手成功后
reg wdata_rcv;  //写数据握手成功后
wire write_finish; //写请求完成

always @(posedge clk) begin
    write_req <= rst                ? 1'b0 :
                 data_req & data_wr ? 1'b1 :
                 write_finish       ? 1'b0 : write_req;
    waddr_rcv <= rst               ? 1'b0 :
                 awvalid & awready ? 1'b1 :
                 write_finish      ? 1'b0 : waddr_rcv;
    wdata_rcv <= rst                     ? 1'b0 :
                 wvalid & wready & wlast ? 1'b1 :
                 write_finish            ? 1'b0 : wdata_rcv;
end

assign write_finish = waddr_rcv & wdata_rcv & (bvalid & bready);

//SRAM-like
assign inst_addr_ok = ~read_sel & arvalid & arready;
assign inst_data_ok = rvalid & (rid==4'd0);
assign inst_rdata   = rdata;

assign data_addr_ok = read_sel & arvalid & arready | data_req & waddr_rcv &
wdata_rcv;    //读数据 | 写数据
assign data_data_ok = rvalid & (rid==4'd1) | write_finish;
assign data_rdata   = rdata;

//AXI
//ar
assign arid    = read_sel ? 4'd1 : 4'd0;
assign araddr  = read_sel ? data_addr : inst_addr;
assign arlen   = 8'd0;      //一次事务只进行一次数据传输(no burst)
```

```
assign arsize  = read_sel ? data_size : inst_size;
assign arburst = 2'd0;      //默认值
assign arlock  = 2'd0;      //默认值
assign arcache = 4'd0;      //默认值
assign arprot  = 3'd0;      //默认值
assign arvalid = arvalid_r;
//r
assign rready  = 1'b1;      //简化为单向握手

//aw
assign awid    = 4'd0;
assign awaddr  = data_addr;
assign awlen   = 8'd0;
assign awsize  = data_size;
assign awburst = 2'd0;
assign awlock  = 2'd0;
assign awcache = 4'd0;
assign awprot  = 3'd0;
assign awvalid = write_req & ~waddr_rcv;
//w
assign wid    = 4'd0;
assign wdata  = data_wdata;
assign wstrb  = awsize==2'd0 ? 4'b0001<<awaddr[1:0] :           //控制写哪些字节
                awsize==2'd1 ? 4'b0011<<awaddr[1:0] : 4'b1111;
assign wlast  = 1'd1;
assign wvalid = write_req & ~wdata_rcv;
//b
assign bready  = 1'b1;

endmodule
```