浙江大学

计算机组成与设计实验报告

课程名称：	计算机组成与设计
姓　　名：	周炜
学　　号：	3210103790
专　　业：	计算机科学与技术
联系电话：	18757321314
指导老师：	刘海风

2022 年 5 月 1 日

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Lab6 缓存设计

学生姓名：周炜 学号：3210103790

实验地点：紫金港东四 509 室 实验日期：2023 年 5 月 1 日

一、实验目的

(1) 理解高速缓存的基本概念和作用

(2) 掌握缓存的组织结构和映射方式、替换策略

(3) 理解缓存的工作原理、命中率和一致性问题

(4) 设计缓存的控制器模块和存储模块

二、实验目标及任务

目标：熟悉数据缓存的工作原理，了解存储单元的组织结构，掌握缓存控制器的设计方法，设计并测试两路组关联 Cache

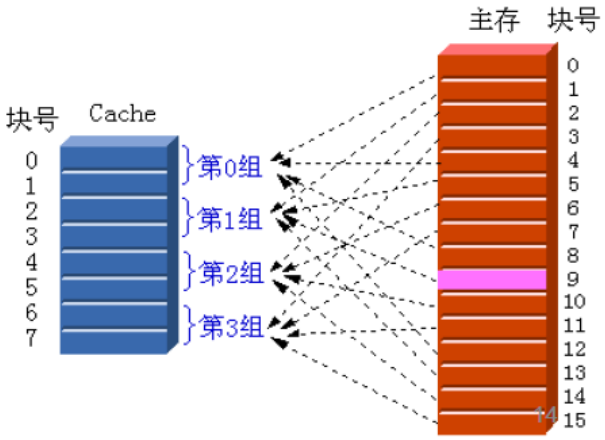
任务 1 数据缓存模块的设计

任务 2 数据缓存模块的仿真验证

本实验采用的映射方式为组相联映射方式

优点：命中率比直接相联高，冲突率比直接相联低，块的利用率提高

缺点：实现难度和造价比直接相联的cache高



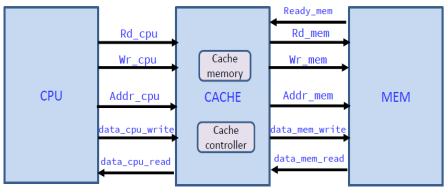
Cache的替换算法采用**近期最少使用（LRU）法**：较好反映了time locality

写hit的策略采用**写回法（write back）**：只写回cache，不写回memory，同时置dirty位为1，待这个块被替换出Cache时再将其写回memory。

写miss的策略采用**write allocate**：做替换，之后写cache，将dirty bit置为1

三、操作方法和实验步骤

任务 1 ：数据缓存模块的设计



Data Cache与CPU 和Memory 接口

Parameter	Value	Unit
Size of Cache	4	KB
Associativity	2-way	-
NUM of Sets	128	-
Blocks/Cache line	4	words
Address width	32	bits
Data width	32	bits
TAG width	23	bits
Index width	7	bits
Word offset	2	bits
Valid width	1	bit
Dirty width	1	bit
LRU width	1	bit

基本参数

（1） 编写有限状态机图

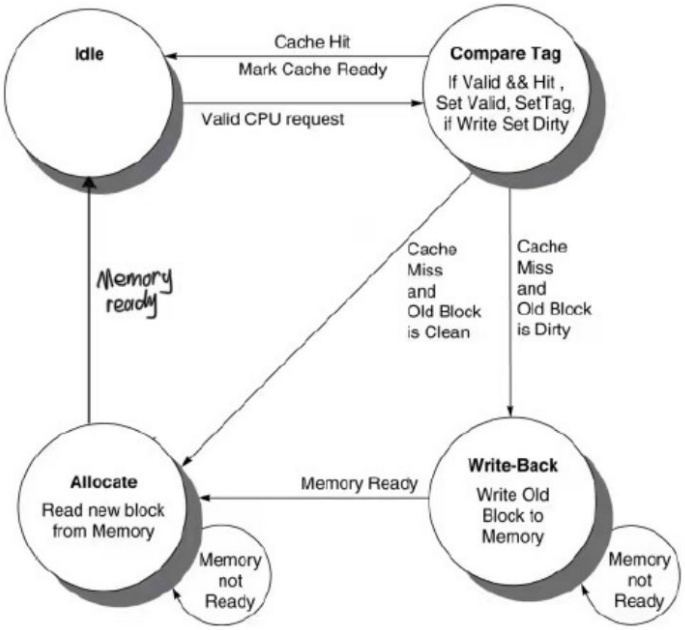
实现Cache时，我们使用有限状态机的方法实现，为了解决时钟同步的问题，我没有直接按照ppt的FSM来做，而是进行了一定更改：

Idle：闲置状态，不做任何事

Compare Tag：比较相同 Index 的 tag 是否相同，决定是否 hit。如果 hit，将 least used bit 置 1，同时将另一路相同 index 的数据的 least used bit 置 0

Write Back：将替换出的 dirty 块写回主存

Allocate：将新的数据从主存读到 Cache



（2）编写 Data_ram.v 用于存储Data

一个word是32bits，每个block4个words，4*32=128bits，即output reg

Block_width=128

因而，Data_ram.v的代码如下：

```
module Data_ram(
    input clk,  // clock
    input en,  // enable
    input wt, // write enable
    input rst, // reset
    input[6:0] addr, // address
    input[127:0] din, // data in
    output reg[127:0] dout // data out
);
    reg[127:0] cache_data[0:127];
    reg i;
    always @ (posedge clk) begin
        if(en) begin
            if(wt) begin
                cache_data[addr]<=din;
            end
            dout<=cache_data[addr];
        end
    end
endmodule
```

（3） 编写 Tag_ram.v 用于存储Tag

首先需要计算input wire [TAG_width+V+U+D1:0] din, // data write in

Tag=23 bite，Valid、Dirty、Least-Used各1 bite，offset是word offset即32bits，每个block4个words，4*32=128bits

```
module Tag_ram(
    input clk,
    input en,
    input wt,
    input rst,
    input[6:0] addr,
    input[25:0] din,
    output reg[25:0] dout
);
    reg[25:0] cache_tag[0:127];
    reg i;
    always @ (posedge clk or posedge rst) begin
```

```
        if(en) begin
            if(wt) begin
                cache_tag[addr]<=din;
            end
        dout<=cache_tag[addr];
        end
    end
endmodule
```

（4） 编写顶层文件cache.v

```
`timescale 1ns / 1ps

module cache(
  input             clk,  //clk
  input             rst,  //reset
  input[31:0]       data_cpu_write, //data from cpu
  input[127:0]      data_mem_read, //data from mem
  input[31:0]       addr_cpu, //addr from cpu
  input             wr_cpu, //cpu write enable
  input             rd_cpu, //cpu read enable
  input             ready_mem, //mem ready
  output reg        wr_mem, //mem write enable
  output reg        rd_mem, //mem read enable
  output reg[127:0] data_mem_write, //data to mem
  output reg[31:0]  data_cpu_read,  // data to cpu
  output reg[31:0]  addr_mem //cache for mem addr
);

wire dirty;
wire cpu_valid;
wire[1:0] offset;
wire[6:0] cpu_id;
wire[22:0] cpu_tag;
wire[22:0] tag_out_new_0;
wire[22:0] tag_out_new_1;
wire[25:0] tag_out_original_0;
wire[25:0] tag_out_original_1;
wire[127:0] d_out_0;
wire[127:0] d_out_1;

reg wt_0;
reg wt_1;
reg wt_t0;
reg wt_t1;
reg hit;
```

```
reg[1:0] state;
reg[1:0] next_state;
reg[25:0] tag_in_0;
reg[25:0] tag_in_1;
reg[127:0] d_in_0;
reg[127:0] d_in_1;


assign cpu_valid = rd_cpu|wr_cpu;
assign tag_out_new_0 =
tag_out_original_0[22:0],tag_out_new_1=tag_out_original_1[22:0];
assign cpu_tag = addr_cpu[31:9];
assign cpu_id = addr_cpu[8:2];
assign offset = addr_cpu[1:0];
assign dirty =
tag_out_original_1[23]?tag_out_original_0[24]:tag_out_original_1[24];

Data_ram d0(
  .clk(~clk),
  .addr(cpu_id),
  .rst(rst),
  .en(1),
  .wt(wt_0),
  .din(d_in_0),
  .dout(d_out_0)
);
Data_ram d1(
  .clk(~clk),
  .addr(cpu_id),
  .rst(rst),
  .en(1),
  .wt(wt_1),
  .din(d_in_1),
  .dout(d_out_1)
);
Tag_ram t0(
  .clk(~clk),
  .addr(cpu_id),
  .rst(rst),
  .en(1'b1),
  .wt(wt_t0),
  .din(tag_in_0),
  .dout(tag_out_original_0)
);
Tag_ram t1(
```

```
  .clk(~clk),
  .addr(cpu_id),
  .rst(rst),
  .en(1'b1),
  .wt(wt_t1),
  .din(tag_in_1),
  .dout(tag_out_original_1)
);
// change state every posedge clk
always @(posedge clk or posedge rst) begin
  if(rst) begin
    state <= 0;
    wt_t0 <= 0;
    wt_t1 <= 0;
    wt_0 <= 0;
    wt_1 <= 0;
  end
  else
    state <= next_state;
end

always @ (*) begin
  case(state)
  //IDLE
  2'b00: begin
    wt_t0  <=  0;
    wt_t1  <=  0;
    wt_0  <=  0;
    wt_1  <=  0;
    if(cpu_valid)
      next_state <= 2'b01;
    else
      next_state <= 2'b00;
  end
  //Compare Tag
  2'b01:begin
  //hit begin
    if(tag_out_new_0 == cpu_tag||tag_out_new_1 == cpu_tag) begin
      hit <= 1'b1;
      next_state <= 2'b00;
      wt_t0 <= 1'b1;
      wt_t1 <= 1'b1;
      if(tag_out_new_0 == cpu_tag) begin
        if(rd_cpu) begin
```

```
          tag_in_0 <= {3'b101, tag_out_new_0[22:0]};
          tag_in_1 <= {tag_out_original_1[25:24], 1'b0,
tag_out_new_1[22:0]};
          case(offset)
            2'b00:  data_cpu_read <= d_out_0[31:0] ;
            2'b01:  data_cpu_read <= d_out_0[63:32];
            2'b10:  data_cpu_read <= d_out_0[95:64];
            2'b11:  data_cpu_read <= d_out_0[127:96];
          endcase
        end
        else if(wr_cpu) begin
          tag_in_0 <= {3'b111, tag_out_new_0[22:0]};
          tag_in_1 <= {tag_out_original_1[25:24],1'b0,
tag_out_new_1[22:0]};
          case(offset)
            2'b00: d_in_0 <= {d_out_0[127:32], data_cpu_write};
            2'b01: d_in_0 <= {d_out_0[127:64],
data_cpu_write,d_out_0[31:0]};
            2'b10: d_in_0 <= {d_out_0[127:96],
data_cpu_write,d_out_0[63:0]};
            2'b11: d_in_0 <= {data_cpu_write, d_out_0[95:0]};
          endcase
          wt_0 <= 1'b1;
        end
      end
      else if(tag_out_new_1==cpu_tag) begin
        if(rd_cpu) begin
          tag_in_1 <= {3'b101, tag_out_new_1[22:0]};
          tag_in_0 <= {tag_out_original_0[25:24], 1'b0,
tag_out_new_0[22:0]};
          case(offset)
            2'b00:  data_cpu_read <= d_out_1[31:0] ;
            2'b01:  data_cpu_read <= d_out_1[63:32];
            2'b10:  data_cpu_read <= d_out_1[95:64];
            2'b11:  data_cpu_read <= d_out_1[127:96];
          endcase
        end
        else if(wr_cpu) begin
          tag_in_1 <= {3'b111, tag_out_new_1[22:0]};
          tag_in_0 <= {tag_out_original_0[25:24],1'b0,
tag_out_new_0[22:0]};
          case(offset)
            2'b00: d_in_1 <= {d_out_1[127:32], data_cpu_write};
```

```
            2'b01: d_in_1 <= {d_out_1[127:64], data_cpu_write,
d_out_1[31:0]};
            2'b10: d_in_1 <= {d_out_1[127:96], data_cpu_write,
d_out_1[63:0]};
            2'b11: d_in_1 <= {data_cpu_write, d_out_1[95:0]};
          endcase
          wt_1 <= 1'b1;
        end //wt
      end //the second way
    end
    //hit end
    //not hit, replace
    else if(dirty) begin
      hit <= 0;
      next_state <= 2'b10;
      end
    else begin
      hit <= 0;
      next_state <= 2'b11;
      end
  end
  //Write Back
  2'b10:begin
    addr_mem = {cpu_tag, cpu_id, 4'b0000};
    if(ready_mem) begin //write back
      next_state <= 2'b11;
      data_mem_write <= tag_out_original_1[23]?d_out_0:d_out_1;
      wr_mem <= 1'b1;
    end
    else
      next_state <= 2'b10;
  end
  //Allocate
  2'b11:begin
    addr_mem = {cpu_tag, cpu_id, 4'b0000};
    if(ready_mem) begin
      next_state <= 2'b00;
      rd_mem <= 1'b1;
      if(tag_out_original_1[23])begin  //the second data
        d_in_0 <= data_mem_read;
        tag_in_0 <= {3'b000, cpu_tag[22:0]};
        wt_0 <= 1'b1;
        wt_t0 <= 1'b1;
      end
```

```
      else if(tag_out_original_0[23])begin   //the second data in
        d_in_1 <= data_mem_read;
        tag_in_1 <= {3'b000, cpu_tag[22:0]};
        wt_1 <= 1'b1;
        wt_t1 <= 1'b1;
      end
      else begin   //the first data in
        d_in_0 <= data_mem_read;
        tag_in_0 <= {3'b000, cpu_tag[22:0]};
        wt_0 <= 1'b1;
        wt_t0 <= 1'b1;
      end
    end
    else
      next_state <= 2'b11;
  end
  default:
    next_state <= 2'b00;
  endcase
end
endmodule
```

任务 2：：数据缓存模块的仿真验证

自己编写仿真代码，仿真的情况需要包含 Read hit，Write hit，Read miss，Write miss

仿真效果如下：

```
module cache_tb();
    reg clk;
    reg rst;
    reg[31:0] data_cpu_write;
    reg[127:0] data_mem_read;
    reg[31:0] addr_cpu;
    reg wr_cpu;
    reg rd_cpu;
    reg ready_mem;
    wire wr_mem;
    wire rd_mem;
    wire[127:0] data_mem_write;
    wire[31:0] data_cpu_read;
    wire[31:0]  addr_mem;
cache cache(
    .clk(clk),
    .rst(rst),
    .data_cpu_write(data_cpu_write),
    .data_mem_read(data_mem_read),
```

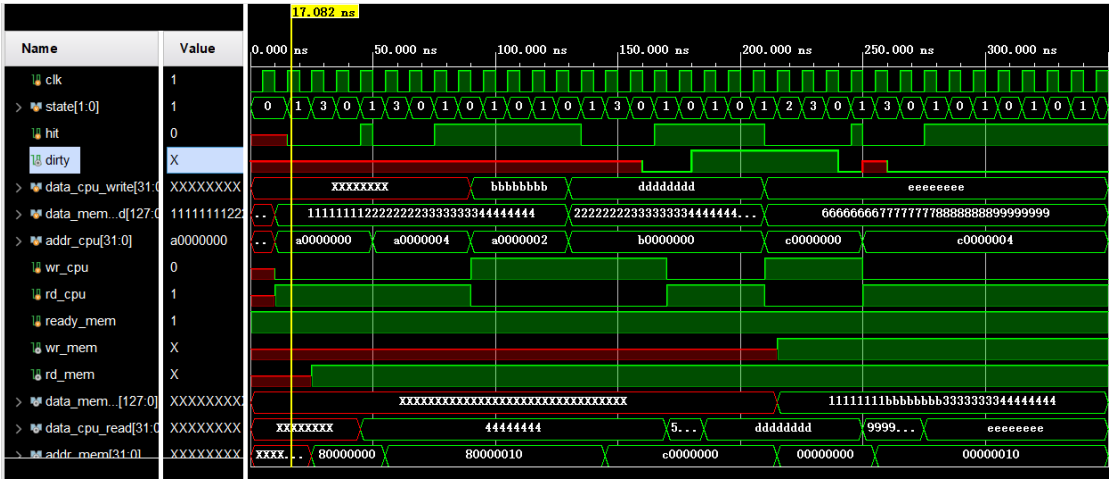
```
    .addr_cpu(addr_cpu),
    .wr_cpu(wr_cpu),
    .rd_cpu(rd_cpu),
    .ready_mem(ready_mem),
    .wr_mem(wr_mem),
    .rd_mem(rd_mem),
    .data_mem_write(data_mem_write),
    .data_cpu_read(data_cpu_read),
    .addr_mem(addr_mem)
);

always #5 clk=~clk;
initial begin
    clk=0;
    rst=1;
    ready_mem=1;
    #2;
    rst=0;
    //read miss
    #8;
    addr_cpu=32'hA0000000;
    wr_cpu=0;
    rd_cpu=1;
    data_mem_read=128'h11111111222222223333333344444444;
    //read hit
    #40;
    addr_cpu=32'hA0000004;
    wr_cpu=0;
    rd_cpu=1;
    //write hit
    #40;
    addr_cpu=32'hA0000002;
    wr_cpu=1;
    rd_cpu=0;
    data_cpu_write=32'hBBBBBBBB;
    //write miss
    #40;
    addr_cpu=32'hB0000000;
    wr_cpu=1;
    rd_cpu=0;
    data_cpu_write=32'hDDDDDDDD;
    data_mem_read=128'h22222222333333334444444455555555;
    //read hit
    #40;
```

```
    addr_cpu=32'hB0000000;
    wr_cpu=0;
    rd_cpu=1;
    //write miss + write back
    #40;
    addr_cpu=32'hC0000000;
    wr_cpu=1;
    rd_cpu=0;
    data_cpu_write=32'hEEEEEEEE;
    data_mem_read=128'h66666666777777778888888899999999;
    //read hit
    #40;
    addr_cpu=32'hC0000004;
    wr_cpu=0;
    rd_cpu=1;
end
endmodule
```

四、实验结果

仿真效果如下图



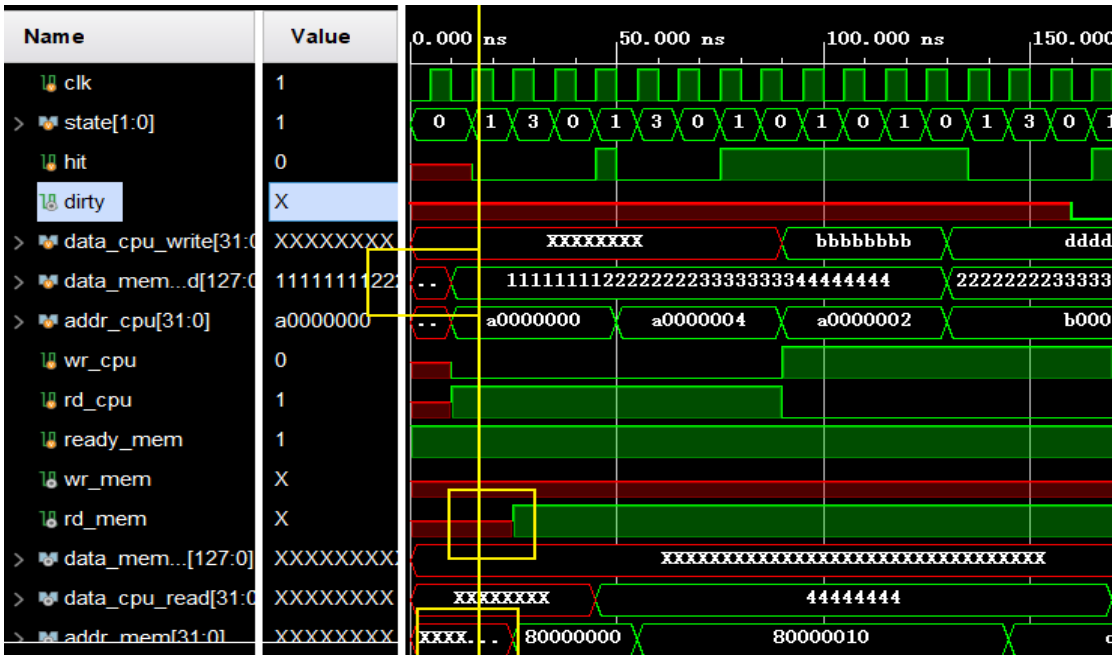
Read Miss：

addr_cpu = 32'hA0000000　wr_cpu = 0, rd_cpu = 1

由于请求的数据不在高速缓存中(`data_mem_read` 为空)，高速缓存必须从内存中读取数据。

结果，rd_mem 信号将变成高电平，addr_mem 将显示所请求的地址

状态机通过 Idle -> Compare Tag -> Allocate 进行转换

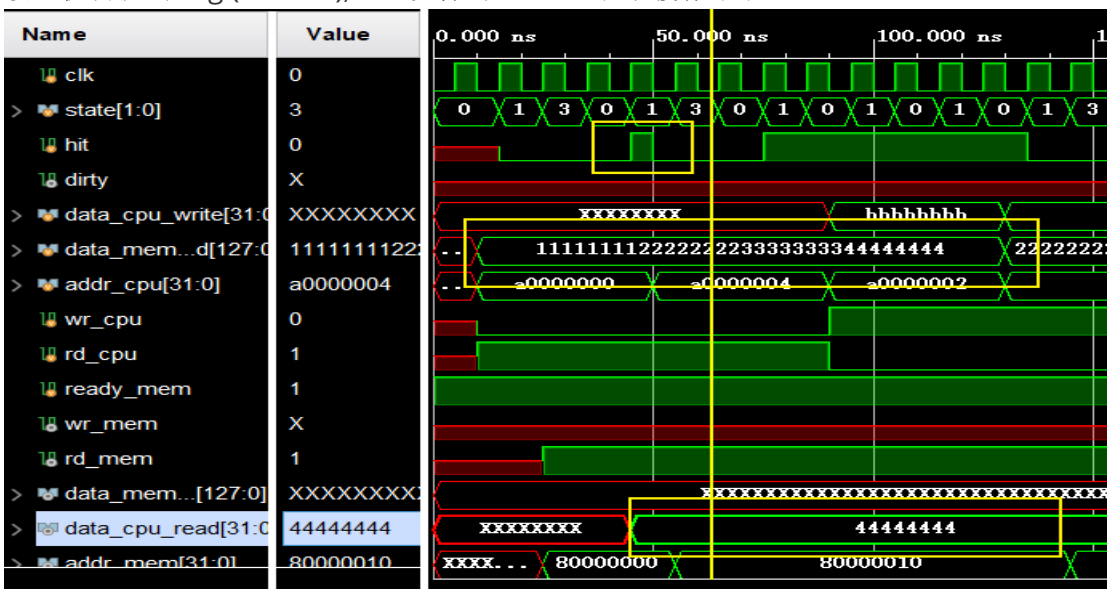


Read Hit:

addr_cpu = 32'hA0000004 wr_cpu = 0, rd_cpu = 1

这一次，要求的数据已经在高速缓存中（已经令 `data_mem_read=128'h11111111222222223333333344444444;`），所以高速缓存将提供数据而不访问内存。data_cpu_read 将显示来自高速缓存的数据（截取了低 32 位）

状态机转换到 Tag (state=1),，可以看到 Hit=1，表示缓存命中



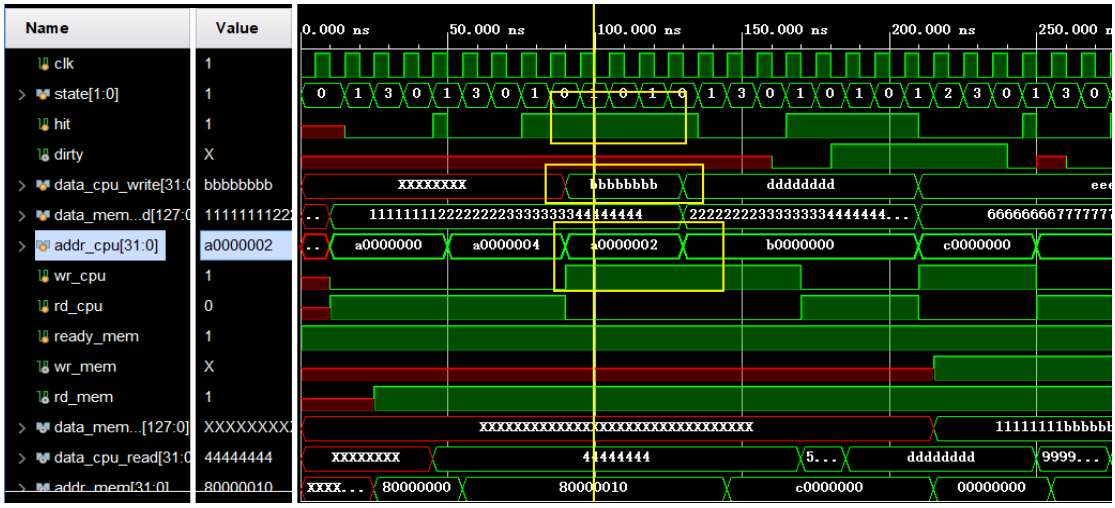
Write Hit:

addr_cpu = 32'hA0000002 wr_cpu = 1, rd_cpu = 0

data_cpu_write = 32'hBBBBBBBB

在这种情况下，缓冲区有请求的地址，所以它用新的数据更新相应的缓冲区行，而不访问内存。wr_mem 信号将保持低电平(由于我在仿真中没有初始化，因此它位 x 状态)。状态机转

换到 Tag (state=1),，Hit=1，表示缓存命中。数据被写入指定地址的高速缓存中



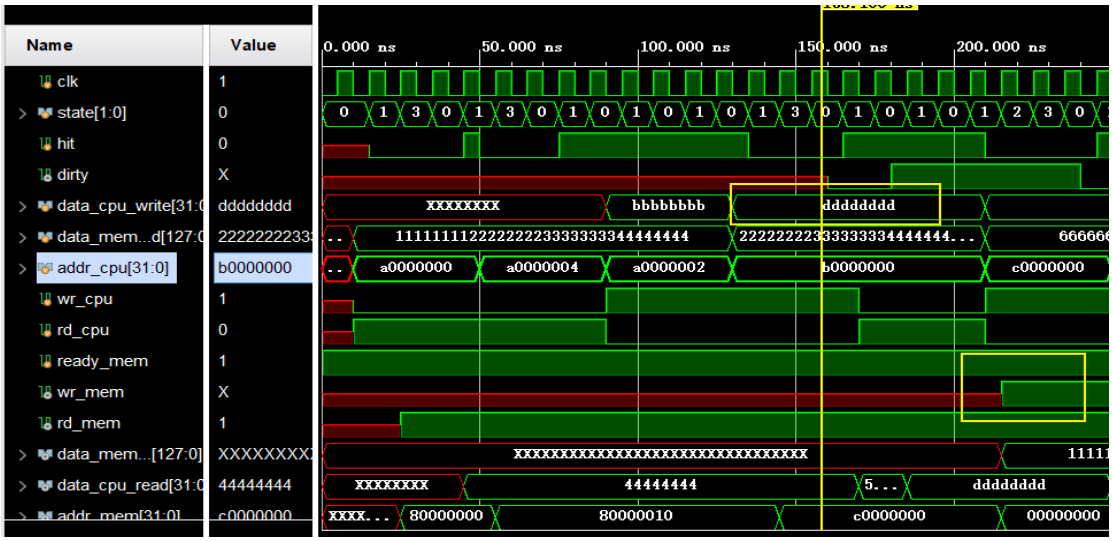
Write Miss：

addr_cpu = 32'hB0000000　wr_cpu = 1　rd_cpu = 0

data_cpu_write = 32'hDDDDDDDD

缓存没有请求的地址，所以它从内存中获取数据，并用新的数据更新缓存行。wr_mem 信号在几个周期后变为高电平以更新内存。

状态机通过 Idle -> Compare Tag -> Allocate 进行转换。在 Allocate(state=3)状态下，数据被写到指定地址的缓存中，并且标签被相应地更新

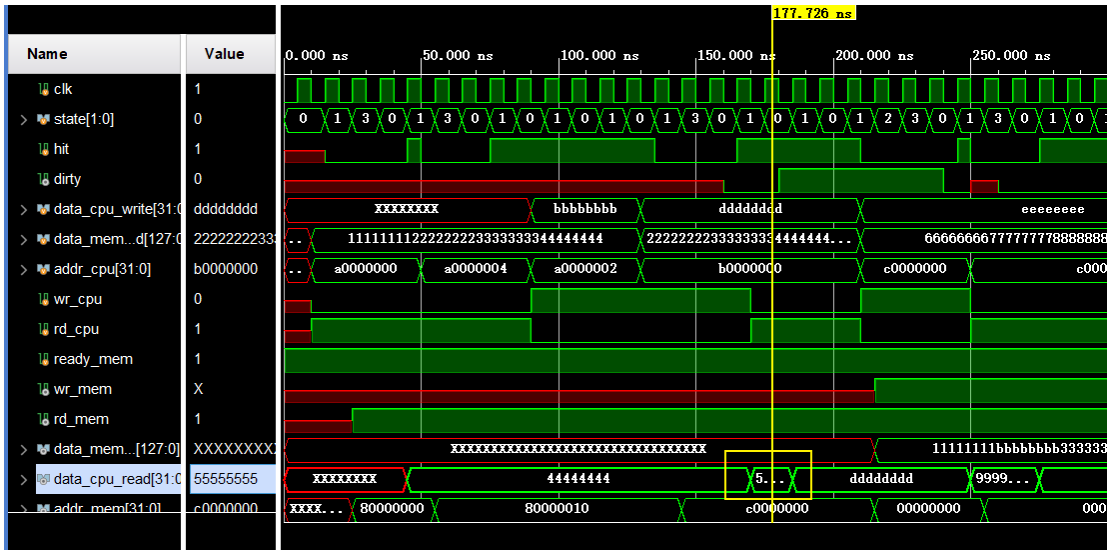


Write Hit：

addr_cpu = 32'hB0000000

wr_cpu = 0, rd_cpu = 1

这一次，要求的数据已经在高速缓存中，所以高速缓存将提供数据而不访问内存,data_cpu_read 将显示缓存中的数据



Write Miss + Write Back：

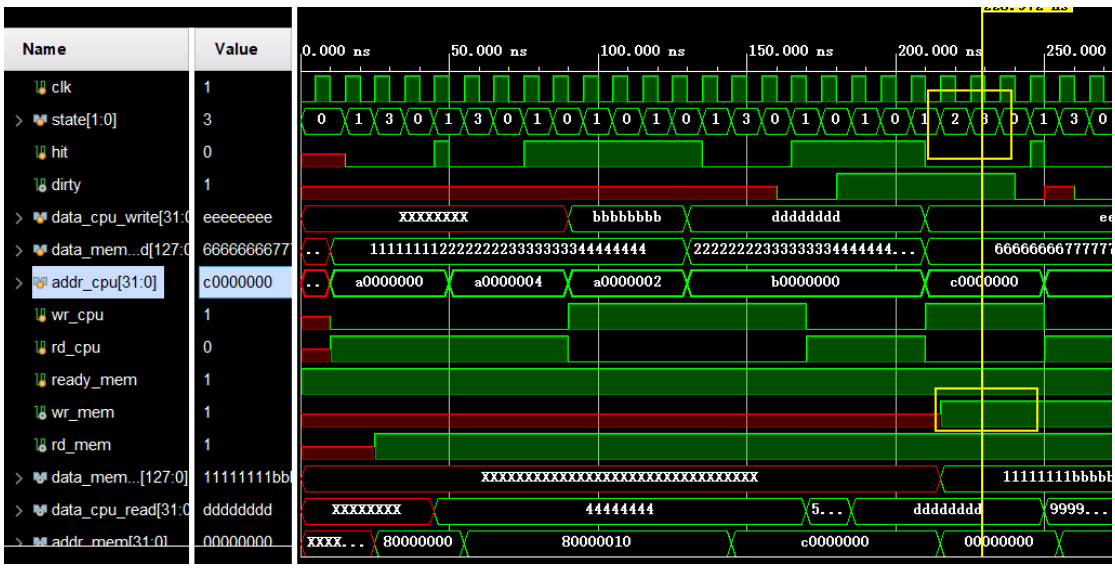
addr_cpu = 32'hC0000000

wr_cpu = 1, rd_cpu = 0

data_cpu_write = 32'hEEEEEEEE

缓存没有请求的地址，需要驱逐一个现有的缓存行来容纳新的数据。缓存首先将驱逐的数据写回内存（回写），然后从内存中获取新数据。wr_mem 信号将在几个周期后变为高电平，以更新内存。

状态机通过 Idle -> Compare Tag -> Write Back -> Allocate 进行转换。在 Write Back (state=2)状态下，数据被写回内存。然后，在 Allocate (state=3)状态下，新数据被写入缓存，标签被更新

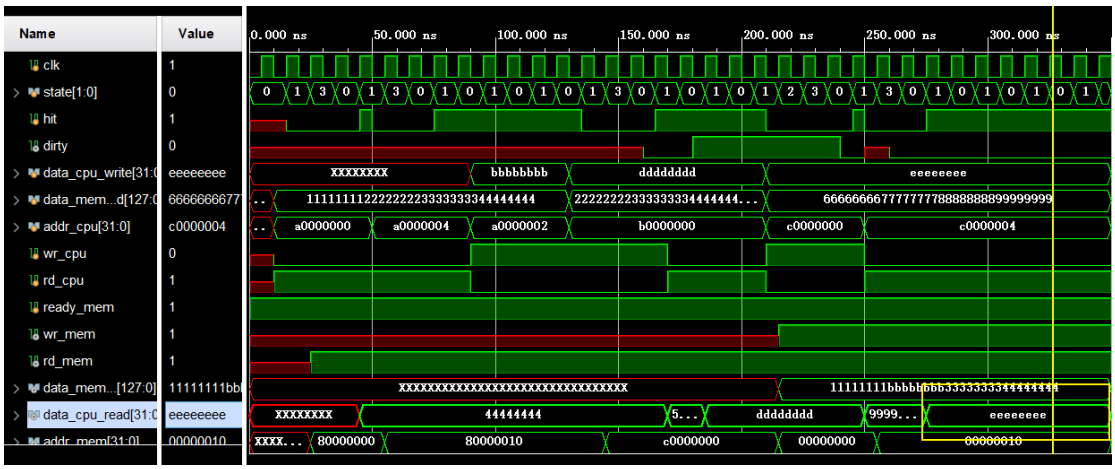


Read Hit：

addr_cpu = 32'hC0000004

wr_cpu = 0, rd_cpu = 1

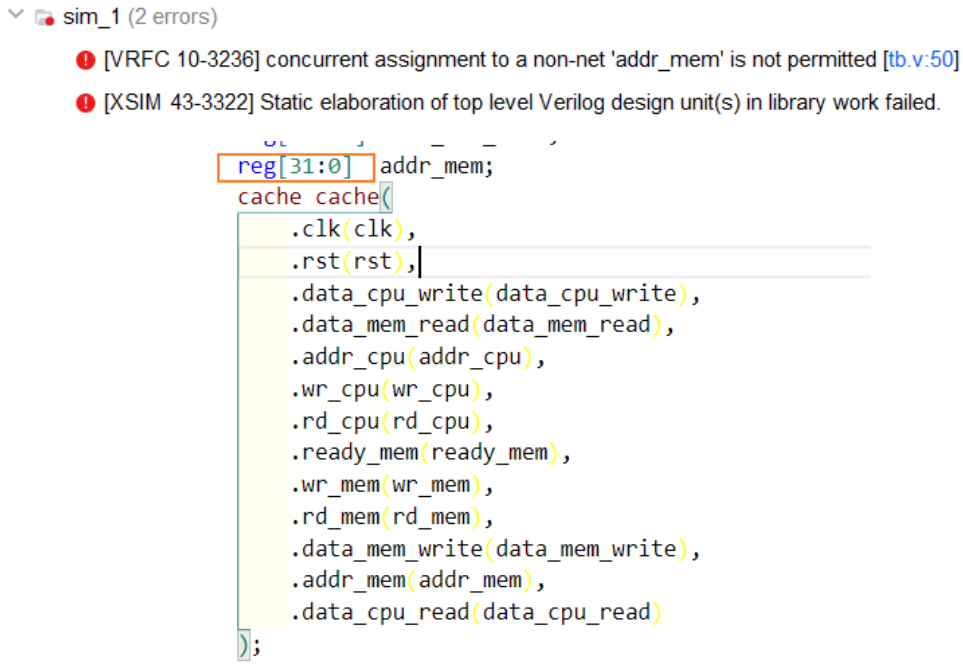
在最后的情况下，要求的数据已经在高速缓存中，所以高速缓存将提供数据而不访问内存,data_cpu_read 将显示来自高速缓存的数据。



五、遇到的问题及解决方法

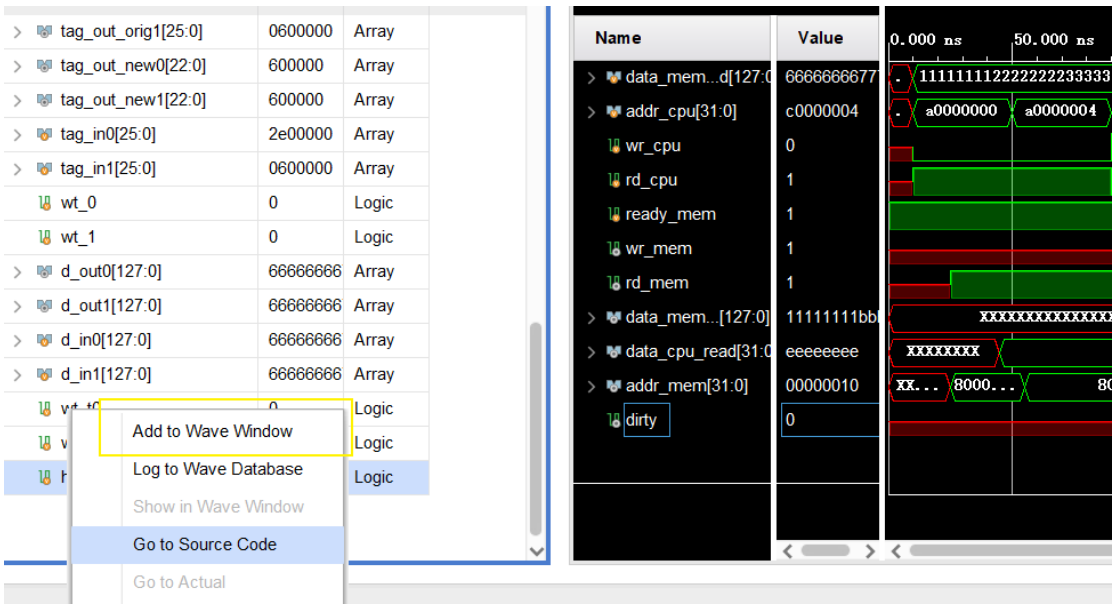
一个低级错误

仿真的时候 output 只能是 wire，写错了 orz



仿真小妙招

为什么写这个，问就是以前显示中间变量，我都是把他们一个个通过改接口的方式接出来的 QAQ



先点击添加，然后再点击 reset