
Computer Organization & Design实验与课程设计

Lab06 缓存设计

Ma De (马德)

made@zju.edu.cn

2020

College of Computer Science, Zhejiang University

Course Outline

- 一、实验目的
- 二、实验环境
- 三、实验目标及任务

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

实验环境

□ 实验设备

1. 计算机（Intel Core i5以上，4GB内存以上）系统
2. Sword 2.0/Sword 4.0开发板
3. VIVADO 2017.4及以上开发工具

□ 材料

无

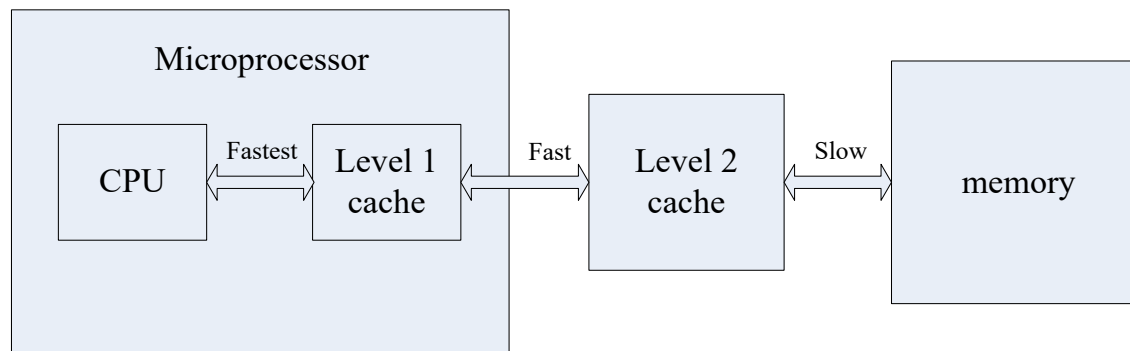
实验目标及任务

- **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（---two-way set associate cache）
- **任务2**：数据缓存模块的仿真验证

缓存的原理介绍

Cache的基本概念

- Cache又叫高速缓冲存储器，位于CPU与内存之间，是一种特殊的存储器子系统。
- 目前比较常见的是两极cache结构，即cache系统由一级高速缓存L1 cache和二级高速缓存L2 cache组成，L1 cache通常又分为数据cache（D-Cache）和指令cache（I-Cache），它们分别用来存放数据和执行这些数据的指令。



Cache的作用

- ❑ **Cache的作用就是为了提高CPU对存储器的访问速度。**
- ❑ **电脑的内存是以系统总线的时钟频率工作的，这个频率通常也就是CPU的外频。但是，CPU的工作频率(主频)是外频与倍频因子的乘积。因此，内存的工作频率就远低于CPU的工作频率了。导致的直接结果是：CPU在执行完一条指令后，常常需要“等待”一些时间才能再次访问内存，极大降了CPU工作效率。**

-----cache 诞生！

Cache的工作原理

- CPU运行程序是一条指令一条指令地执行的，而且指令地址往往是连续的，即CPU在访问内存时，在较短的一段时间内往往集中于某个局部，这时候可能会碰到一些需要反复调用的子程序。系统在工作时，可把这些活跃的子程序存入比主存快得多的cache中。

时间相关性（temporal locality）：如果一个数据现在被访问了，那么以后很有可能也会被访问；

空间相关性（spatial locally）：如果一个数据现在被访问了，那么它周围的数据在以后可能也会被访问

Cache的工作原理

- CPU在访问内存时，首先判断所要访问的内容是否在cache中，如果在，则称为**命中（hit）**，此时CPU直接从cache中调用该内容；否则称为**未命中（miss）**，CPU会通过cache对主存中的相应内容进行操作。

Cache的映射方式

□ Cache与主存之间可以采取的地址映射方式有以下三种：

- 全相联映射方式
- 直接相联映射方式
- 组相联映射方式

Cache的映射方式

□ 全相联方式

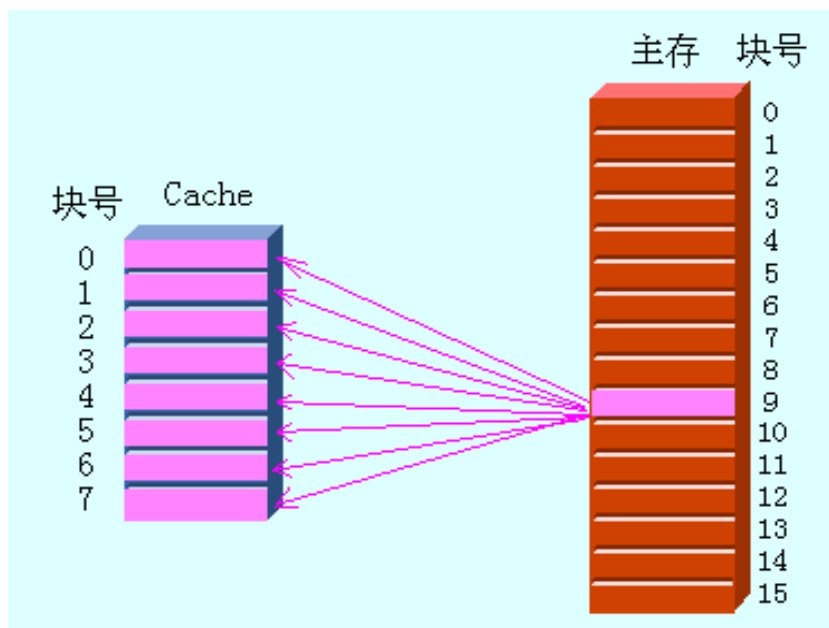
地址映射规则：主存的任意一块可以映射到cache中的任意一块

(1) 主存与cache分成相同大小的数据块。

(2) 主存的某一数据块可以装入cache的任意一块空间中。

优点：命中率比较高，cache存储空间利用率高。

缺点：速度低，成本高，访问相关存储器时，每次都要与全部内容比较。



Cache的映射方式

□ 直接相联方式

地址映射规则：主存储器中一块只能映射到cache的一个特定的块中。

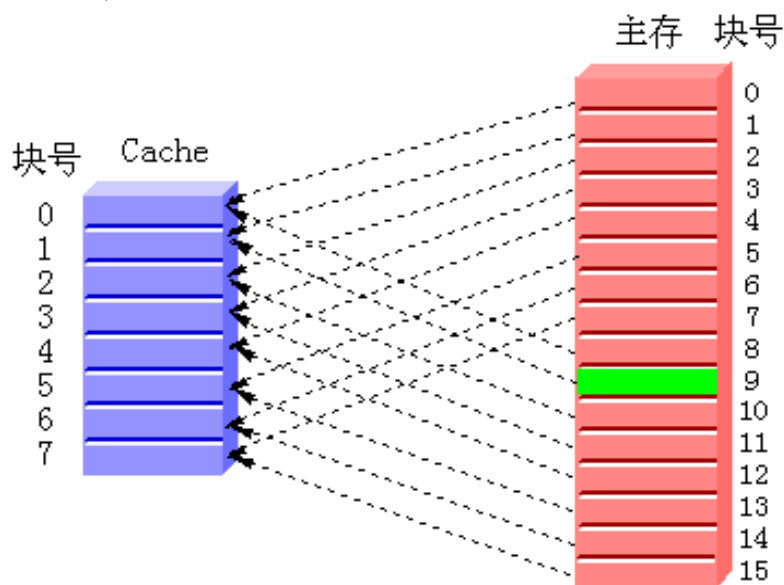
(1)主存与cache分成相同大小的数据块。

(2)主存容量应是cache容量的整数倍，将主存空间按cache的容量分分区，主存中每一区的块数与cache的总块数相等。

(3)主存中某区的一块存入cache时只能存入缓存中块号相同的位置。

优点：地址映射方式简单，数据访问时，只需检查区号是否相等即可，因而可以得到比较快的访问速度，硬件设备简单。

缺点：替换操作频繁，命中率比较低。



Cache的映射方式

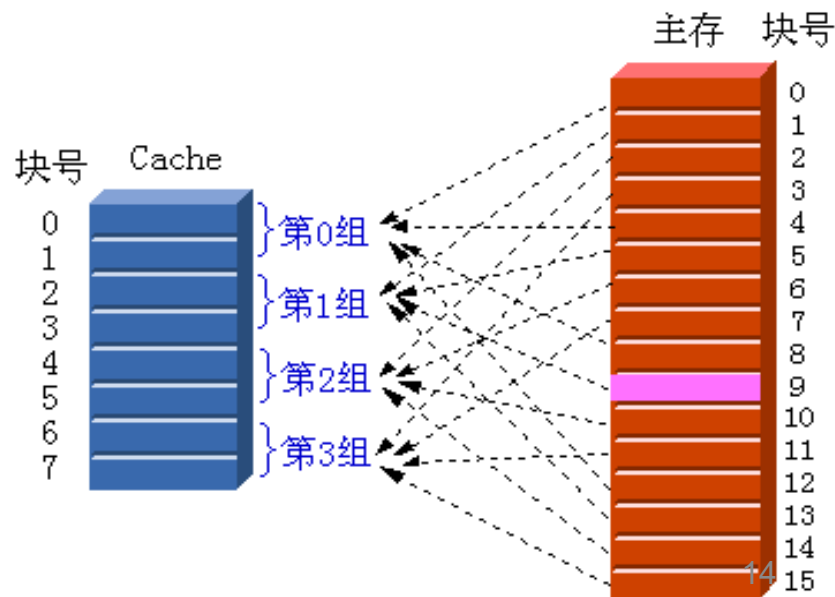
本实验采用此方式

□ 组相联映射方式 地址映射规则：

- (1) 主存和cache按同样大小划分成块。
- (2) 主存和cache按同样大小划分成组。
- (3) 主存容量是cache容量的整数倍，将主存空间按cache区的大小分成区，主存中每一区的组数与cache的组数相同。
- (4) 当主存的数据调入cache时，主存与cache的组号应相等，也就是各区中的某一块只能存入cache的同组号的空间内，但组内各块地址之间则可以任意存放，即从主存的组到cache的组之间采用直接映射方式；在两个对应的组内部采用全相联映射方式。

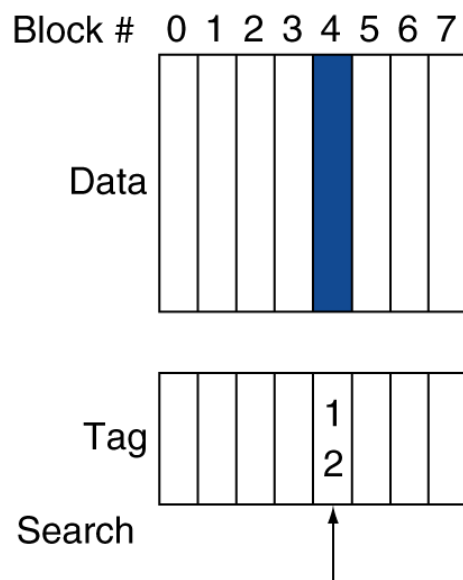
优点：块的冲突概率比较低，块的利用率大幅度提高，块失效率明显降低。

缺点：实现难度和造价要比直接映射方式高。



Cache的映射方式

Direct mapped



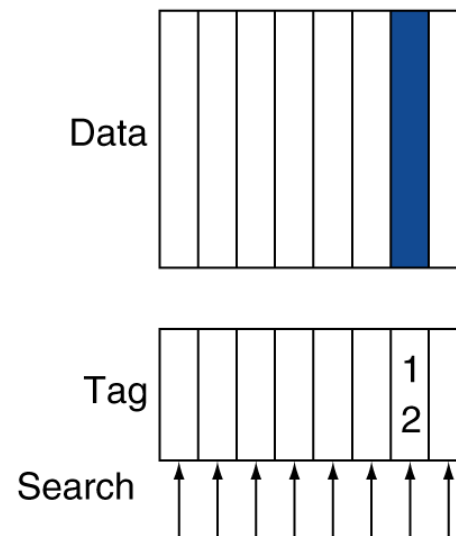
第4块
($12 \bmod 8$)

Set associative



第0组任意位置
($12 \bmod 4$)

Fully associative



任意
位置

块12可
能被放
置的地
方

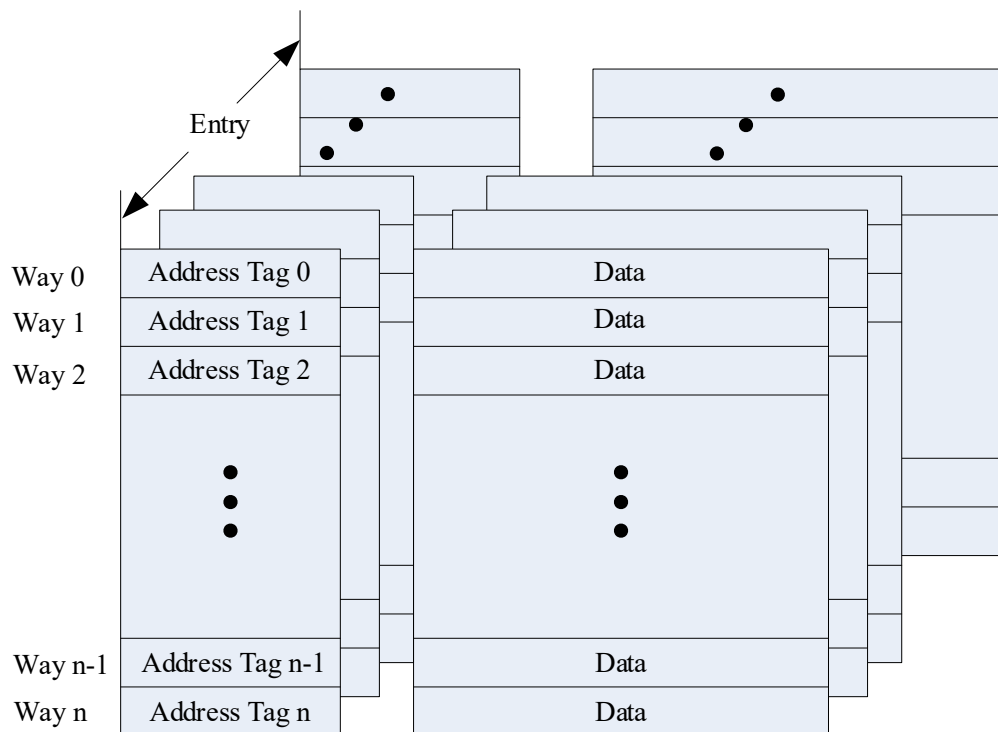
Cache的组织结构

- ❑ 块（block）：块是cache与主存的传输单位。
- ❑ 路（way）：路是组相联映射方式的cache结构中的基本存储单位，每一路存储一个块的数据。
- ❑ 组（set/entry）：组是组相联映射方式的cache对块进行管理的单位。
- ❑ 区（tag）：块的地址对应的主存储器中的区。
- ❑ 块内偏移地址（offset）：用来标示块内一个字（节）的地址。
- ❑ 组相联映射方式下主存储器的地址空间由，区，组和块内偏移地址组成：



Cache的组织结构

□ 组相联映射方式下cache的内部结构。



Cache的替换算法

□ Cache可以采用的替换算法主要有以下几种

- 先入后出(FILO)算法
- 随机替换(RAND)算法
- 先入先出(FIFO)算法
- 近期最少使用(LRU)算法



实验采用
LRU

所要解决的问题：当新调入一块，而Cache又已被占满时，替换哪一块？

Cache的替换算法

- ❑ **随机(RAND)法**是随机地确定替换的存储块。设置一个随机数产生器，依据所产生的随机数，确定替换块。这种方法简单、易于实现，但命中率比较低。
- ❑ **先进先出(FIFO)法**是选择那个最先调入的那个块进行替换。当最先调入并被多次命中的块，很可能被优先替换，因而不符合局部性规律。这种方法的命中率比随机法好些，但还不满足要求。
- ❑ **近期最少使用(LRU)法**是依据各块使用的情况，总是选择那个最近最少使用的块被替换。这种方法比较好地反映了程序局部性规律，命中率最高。

Cache的性能评估

- 提高cache的性能也就是要降低主存的平均存取时间
主存平均存取时间=命中时间+未命中率*未命中惩罚
- 提高cache的性能有以下三种方法：减少命中时间，减少未命中率，减少未命中惩罚。

影响命中率的硬件因素主要有以下四点：

- Cache的容量。
- Cache与主存储器交换信息的单位量（cache line size）。
- Cache的组织方式
- Cache的替换算法

Cache的一致性问题

- 在采用cache的系统中，同样一个数据可能既存在于cache中，也存在于主存中，两者数据相同则具有一致性，数据不相同就叫做不一致。
- Cache主要有两种写策略写直达法（write through）和写回法（write back）



Cache的一致性问题

□ 写直达法（write through）

方法：在对cache进行写操作的同时，也将内容写到主存中。

优点：可靠性较高，操作过程比较简单。

缺点：写操作速度得不到改善，与写主存的速度相同。

□ 写回法（write back）

方法：在CPU执行写操作时，只写入cache，不写入主存。

优点：速度较高。

缺点：可靠性较差，控制操作比较复杂。

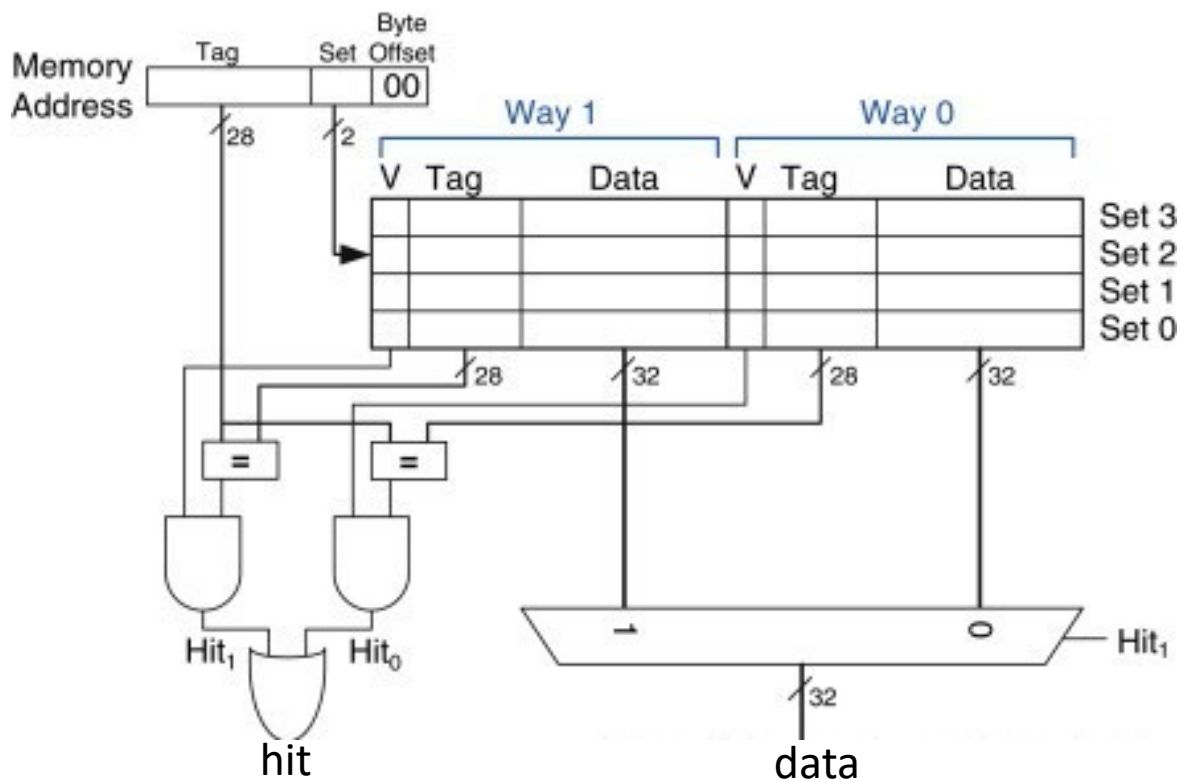
■ 任务1：数据缓存模块的设计

-----Cache存储体（cache memory）

-----Cache控制器（cache controller）

Cache设计实现

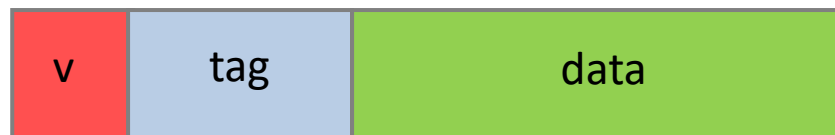
- 如图，为两路四组的组织结构：



Cache设计实现

■ two-way set associate cache

■ Cache的一行构成（cache line）



- **Tag:** 标签位用于比较在way1和way0两路中，哪一个cache块存了我们的目的数据
- **data block:** 数据位用来存储（内存和cache交换的数据块）目标tag的数据块。
- **v:** 有效位用来 表示该cache块是否有效。

Cache设计实现

■ two-way set associate cache

■ cache的数据对应的内存地址



•**Block offset:** 对于内存地址而言，其后block offset个字节的数据会构成一个和cache做数据交换的块，即cache块的大小。

•**index:** 对于内存地址而言，其应该被映射到cache里的哪一组，故该部分的位数代表的是整个cache能容纳多少组；上图，2路组相连，其中的MemoryAddress里的set是2位，那么这个内存地址就可以根据set的两位去找到自己在cache中对应的组是哪一组，如：当set的两位为10时，那么对应到第3组set2。

•**tag:** 在使用index选出cache位于哪一组后去比较具体的cache块位于哪一路。

Cache设计实现

■ two-way set associate cache

■ 本实验所实现的Data Cache功能要求:

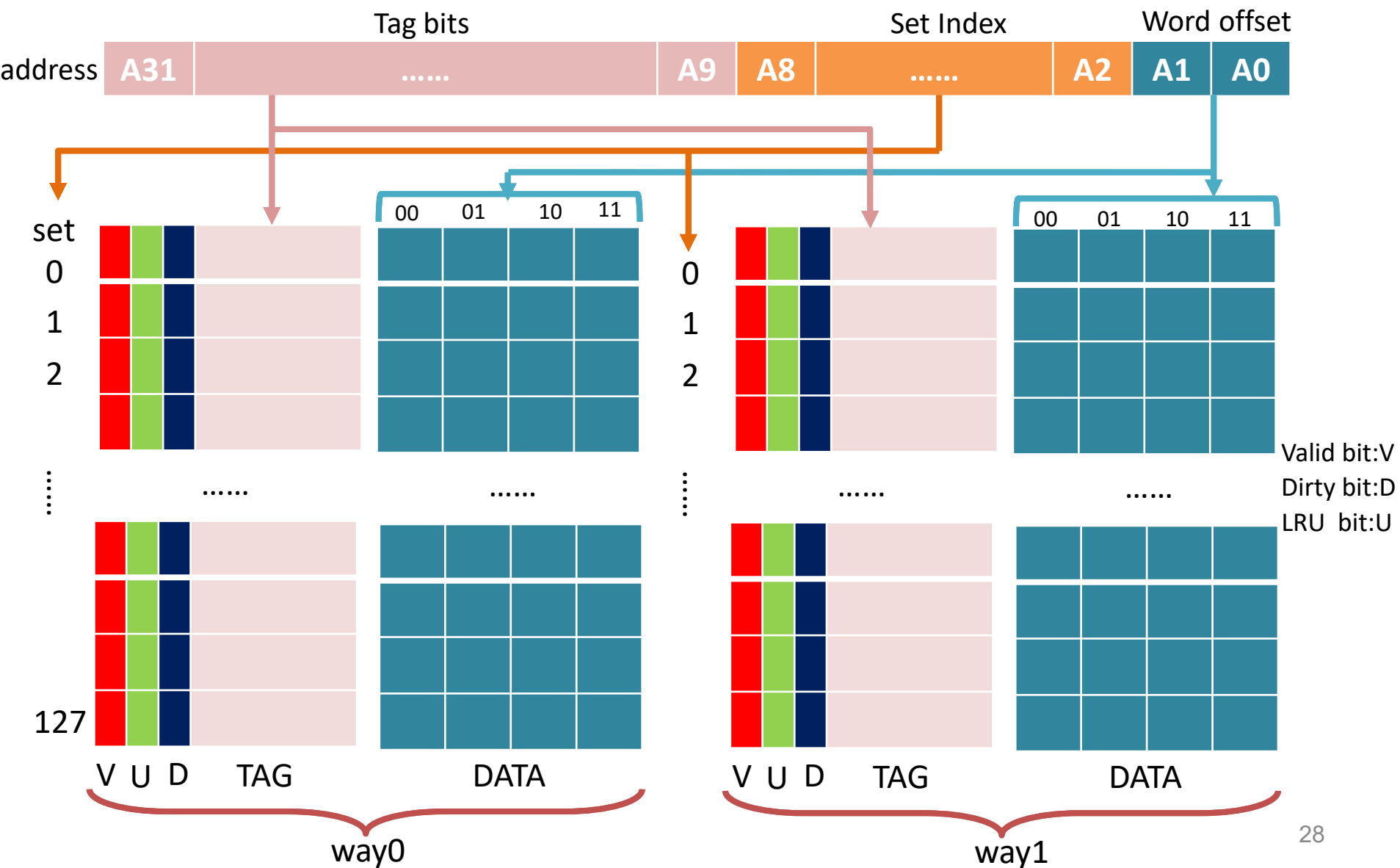
- two-way set associate cache
- Write Back
- Write Allocate
- Block size is four words(16bytes or 128bits)
- Cache size is 4KiB
- Replacement Policy: Least Recently Used

组数目、索引字段等为多少?

实现方法不限定, 以下仅提供一种思路参考

Cache设计实现

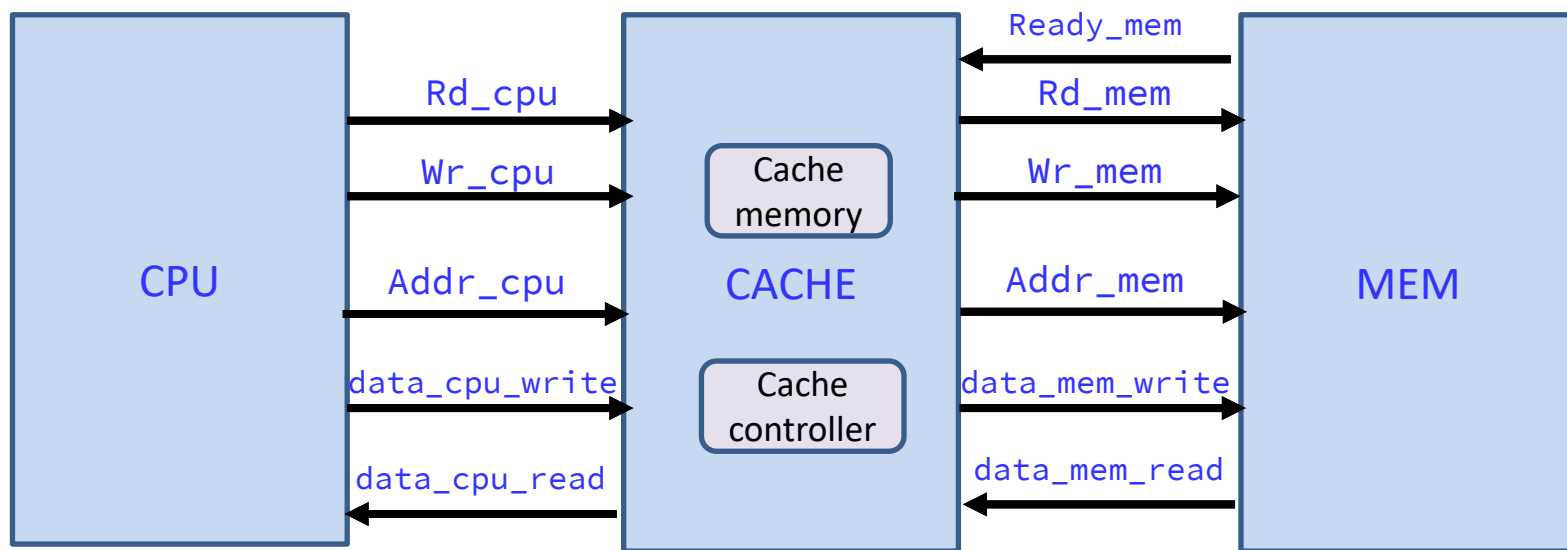
■ two-way set associate cache



Cache设计实现

■ two-way set associate cache

■ Data Cache与CPU、Memory接口：



Cache设计实现

■ two-way set associate cache

■ Data Cache基本参数:

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

?

Cache设计实现

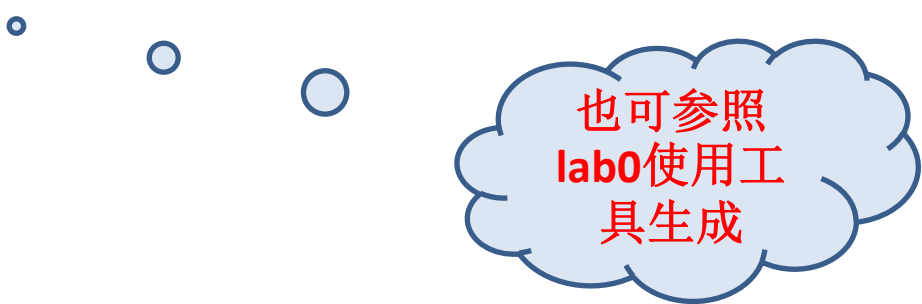
Cache Memory

- **Cache Memory**由**Cache line**构成，包含存储的数据字段，标签字段以及是否有效位；而本实验中还需设置脏位和**LRU**标志位。
- 由前**two-way set associate cache**的示意图得知，依据存储地址的**Index**索引到**Cache**中具体的哪一组，得到两路中的两个**Cache line**；而后通过**Tag**比较确定最终选择**line**；最后由**Block offset**定位所需数据。
- 此处将**TAG**和**Data**部分开存储，可以同时并行访问**TAG Ram**，**Data Ram**。

Cache设计实现

■ Cache Memory:Data_ram0/1

```
module Data_ram0(  
    input wire clk, // clock  
    input wire en, // enable  
    input wire [Index_width-1:0] addr, // address  
    input wire [Block_width-1:0] din, // data write in  
    output reg [Block_width-1:0] dout, // data read out  
);
```



也可参照
lab0使用工
具生成

Cache设计实现

■ Cache Memory:Data_ram0/1

```
.....  
//cache line memory: data for way0  
reg [Block_width-1:0] cache_data [0:NUM_of_sets-1];  
.....  
//memory initial  
Initial begin  
    $readmemh(".....", cache_data);  
end  
.....  
//Read and Write data to Cache  
dout <= cache_data[addr];  
.....  
cache_data[addr] <= din;  
.....
```

初始化操作
可参见VGA
模块

Way1的数据
存储实现过
程相同

Cache设计实现

■ Cache Memory:Tag_ram0/1

```
module Tag_ram0(  
    input wire clk, // clock  
    input wire en, // enable  
    input wire [Index_width-1:0] addr, // address  
    input wire [TAG_width+V+U+D-1:0] din, // data write in  
    output reg [TAG_width+V+U+D-1:0] dout, // data read out  
);
```

Cache设计实现

■ Cache Memory:Tag_ram0/1

.....

//cache line memory: tag,V,U,D for way0

reg [TAG_width+V+U+D-1:0] cache_TAG [0:NUM_of_sets-1];

.....

//memory initial

Initial begin

\$readmembh(".....", cache_data);

end

.....

//Read and Write TAG to Cache

dout <= cache_TAG[addr];

.....

cache_TAG[addr] <= din;

.....

初始化操作
可参见VGA
模块

Way1的TAG
存储实现过
程相同

Cache设计实现

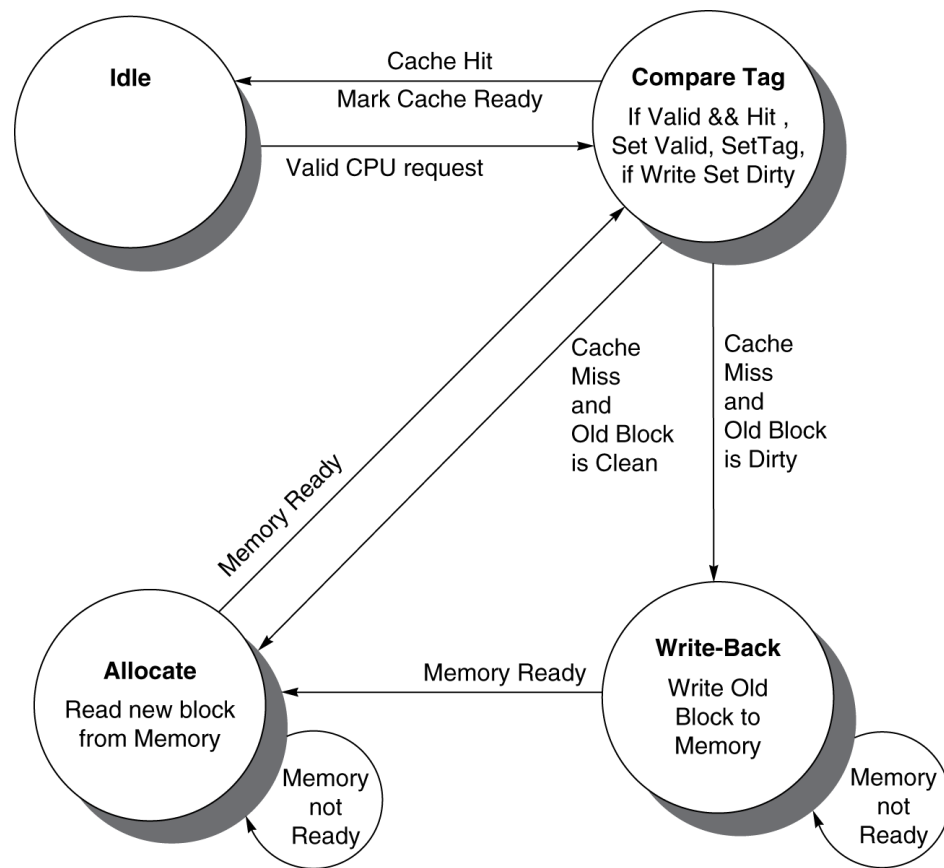
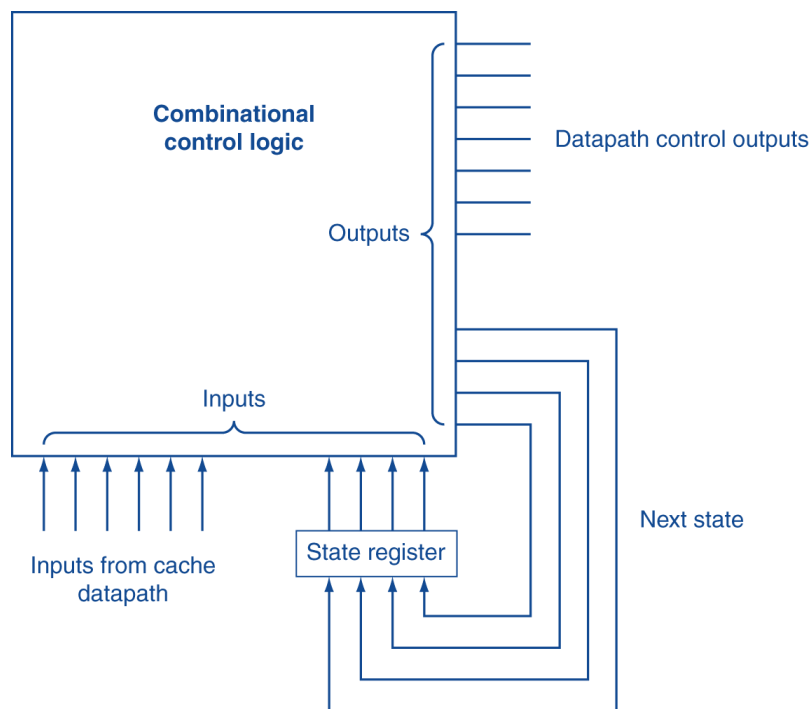
■ Cache Controller

- Cache控制采用FSM实现。
- 实现标签比较，检测读写命中与否。
- 实现Cache的数据更新，当miss后采用LRU完成数据替换。
- 实现Mem的数据更新，当write miss后采用Write Back，被改写的数据块在被替换出cache时写回到Mem。同时实现Write Allocate。

Cache设计实现

Cache Controller

Cache控制采用FSM实现



Cache设计实现

■ Cache Controller

■ Cache控制部分采用FSM实现

```
always@(posedge clk or posedge rst)
if(rst)
    state<=IDLE;
else
    state<=next_state;

always@(*)
case(state)
    IDLE:if(cpu_req_valid)
        .....;
    else
        next_state=IDLE;
    CompareTag:if(hit)                //若hit
        .....
    else
        next_state=Allocate;
```

```
Allocate:if(mem_ready)
    next_state=CompareTag;
else
    next_state=Allocate;
WriteBack:if(mem_ready)
.....
.....
    default:next_state=IDLE;
Endcase
.....
```



FSM的具体
步骤可参照
lab01

Cache设计实现

■ Cache.v 顶层端口

■ 顶层完成存储体的调用和控制器模块模块的实现

```
module cache(  
    input wire clk, // clock  
    input wire rst, // reset  
    input wire [31:0] data_cpu_write, // data write in  
    input wire [31:0] data_mem_read, // data read in  
    input wire [31:0] addr_cpu,      // cpu addr  
    input wire        wr_cpu,        // cpu write enable  
    input wire        rd_cpu,        // cpu read enable  
    input wire        ready_mem,     // memory ready  
    output reg        wr_mem,        // memory write enable  
    output reg        rd_mem,        // memory read enable  
    output reg [31:0] data_mem_write, // data to mem  
    output reg [31:0] data_cpu_read,  // data to cpu  
    2021/6/8 );
```

Cache设计实现

■ Cache.v 顶层端口

■ 顶层完成存储体的调用和控制器模块模块的实现

```
// Cache Controller State Machine and Logic
```

```
always@(posedge clk or posedge rst)
```

```
if(rst)
```

```
    state<=IDLE;
```

```
else
```

```
    state<=next_state;
```

```
.....
```

```
.....
```

```
// Instantiation Data RAM for Way 0
```

```
Data_ram0 d0 (.clock(clock),
```

```
               .addr(index),
```

```
               .din(wdata),
```

```
               .en(en0),
```

```
               .dout(rdata0));
```

```
// Instantiation of Tag RAM for Way 0
```

```
Tag_ram0      t0 (.clock(clock),  
                  .addr(index),  
                  .din(wtag0),  
                  .en(ent0),  
                  .dout(rtag0)  
                  );
```

```
.....
```

Way1的存储
例化过程相
同

■ 任务2：数据缓存模块的仿真验证

Cache仿真验证

■ 本实验所实现的Data Cache测试要求:

- Read hit
- Write hit
- Read miss
- Write miss




各种情况测试至少一组

Cache仿真测试

■ Cache_tb.v

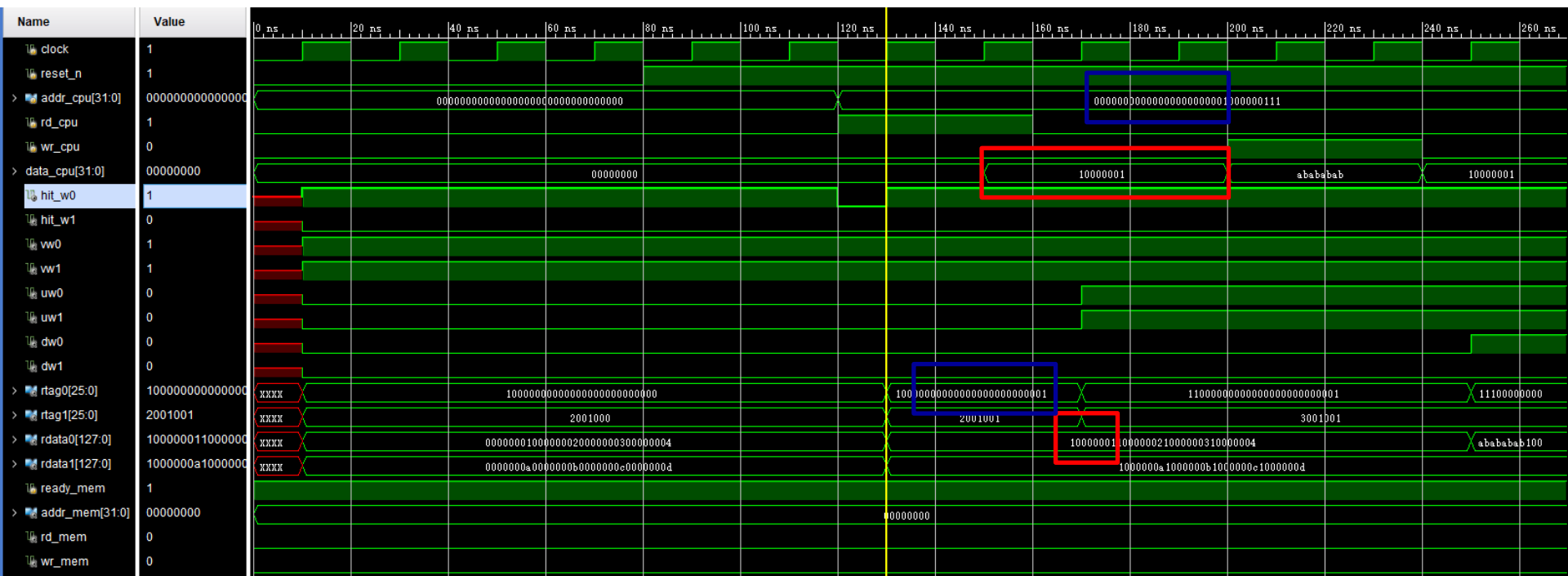
```
module cache_tb();  
.....  
.....  
//read hit  
rd_cpu = 1'd1;  
addr_cpu = 32'h00000207;  
.....  
// Write to same location  
wr_cpu = 1'd1;  
wcpu = 32'hbababab;  
addr_cpu = 32'h00000207;  
.....  
// Read from same location to check updated data  
rd_cpu = 1'd1;  
addr_cpu = 32'h00000207;
```



读的目的数据
已在**cache**存储
体中初始化过，
读命中

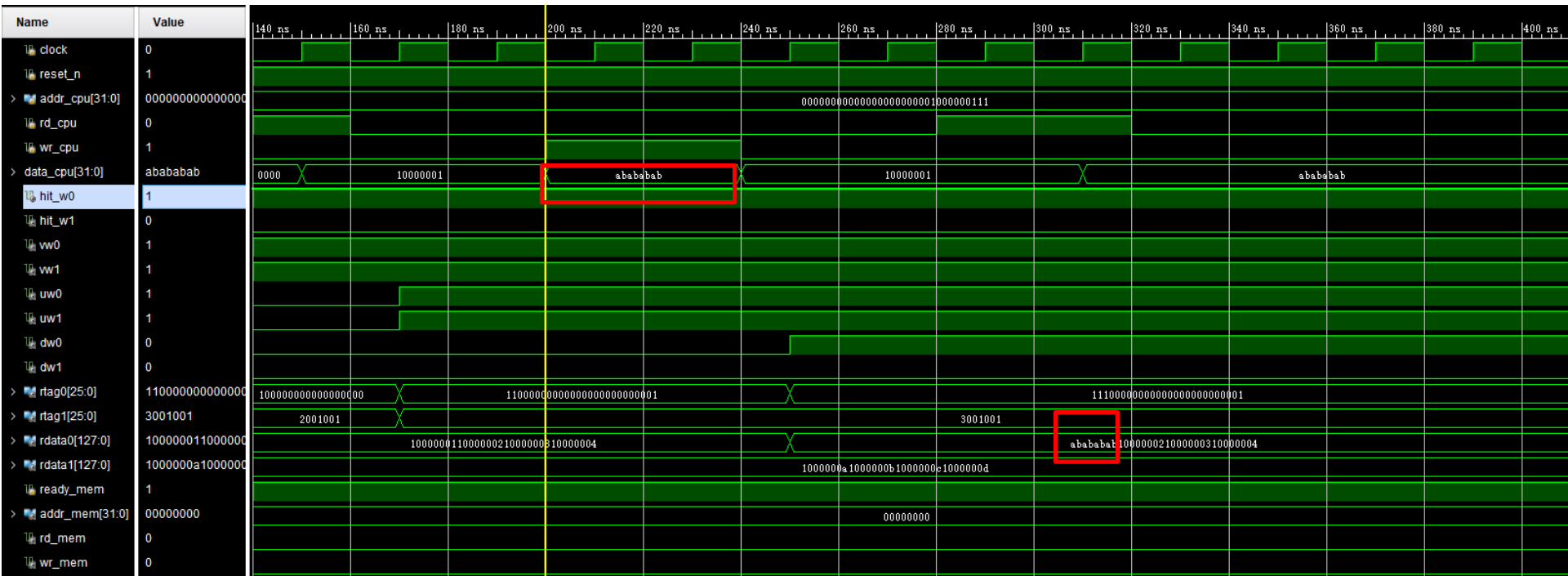
Cache仿真测试

- Read hit at 0x000000207



Cache仿真测试

- **Write hit at 0x000000207**



Cache仿真测试

■ Cache_tb.v

.....
//Read Miss with dirty bit 0 policy check, reads data from Main Memory

rd_cpu = 1'd1;

addr_cpu = 32'h0000020A;

.....

dmem= 32'h11111111;

delay

dmem= 32'h22222222;

delay

dmem= 32'h33333333;

delay

dmem= 32'h44444444;

.....

// Read from same location to check updated data

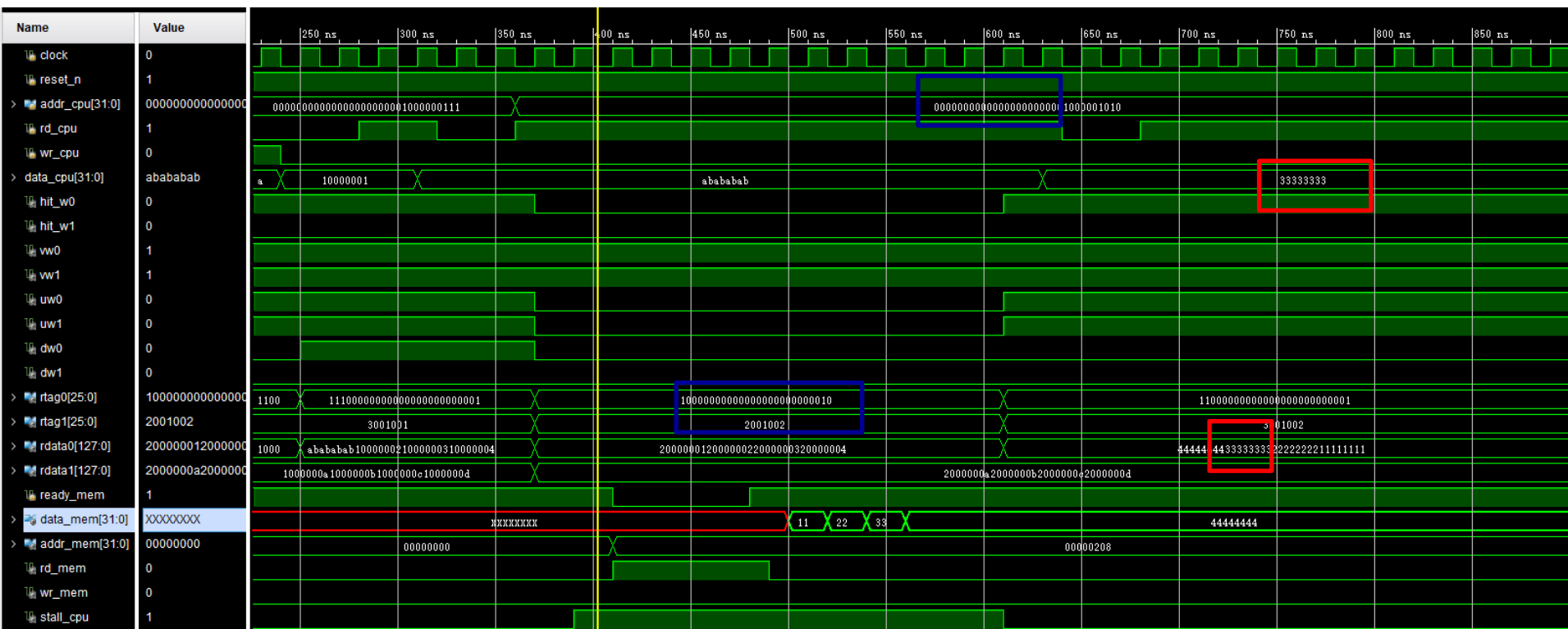
rd_cpu = 1'd1;

addr_cpu = 32'h0000020A;



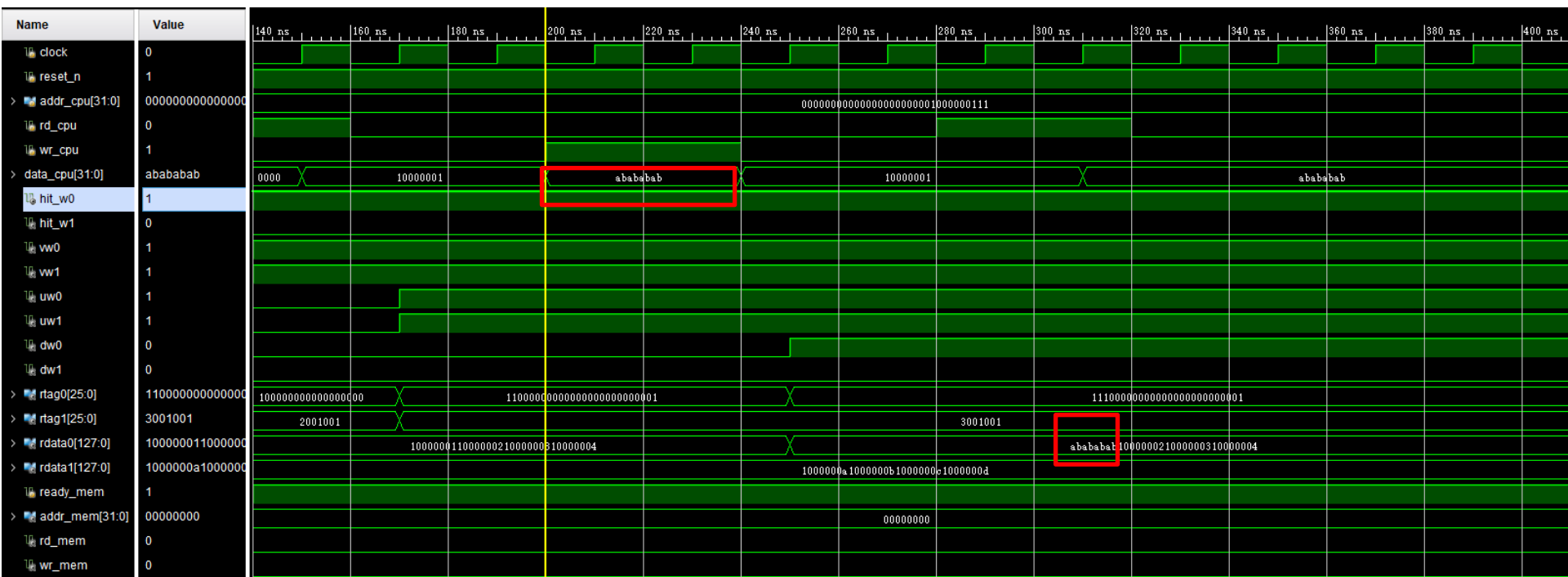
Cache仿真测试

- **Read miss at 0x00000020A and after read from RAM, read hit**



Cache仿真测试

- **Write hit at 0x000000207**



思考题

- 实验只设计实现数据缓存，若实现指令缓存，设计方法是否一样？指令缓存也会存在写回、写分派现象吗？指令缓存的内容如果需要修改，如何操作？
- 带缓存的流水线CPU如何实现，当发生缺失的情况时CPU应该如何应对？

 **END**