浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	梁晨
学　　院：	计算机科学与技术学院
专　　业：	计算机
邮　　箱：	liangchenwater@outlook.com
学　　号：	3180102160
电　　话：	15901260806
指导教师：	马德
报告日期：	2021 / 6 / 30

实验6：Two-way Associated Cache

- 实验目的
  - **理解高速缓存的基本概念和作用**
  - **掌握缓存的组织结构和映射方式、替换策略**
  - **理解缓存的工作原理、命中率和一致性问题**
  - **设计缓存的控制器模块和存储模块**

- 实验环境
  - Intel Core i7、32GRAM计算机，Windows10家庭版操作系统
  - Sword4.0开发板
  - VIVADO 2020开发工具

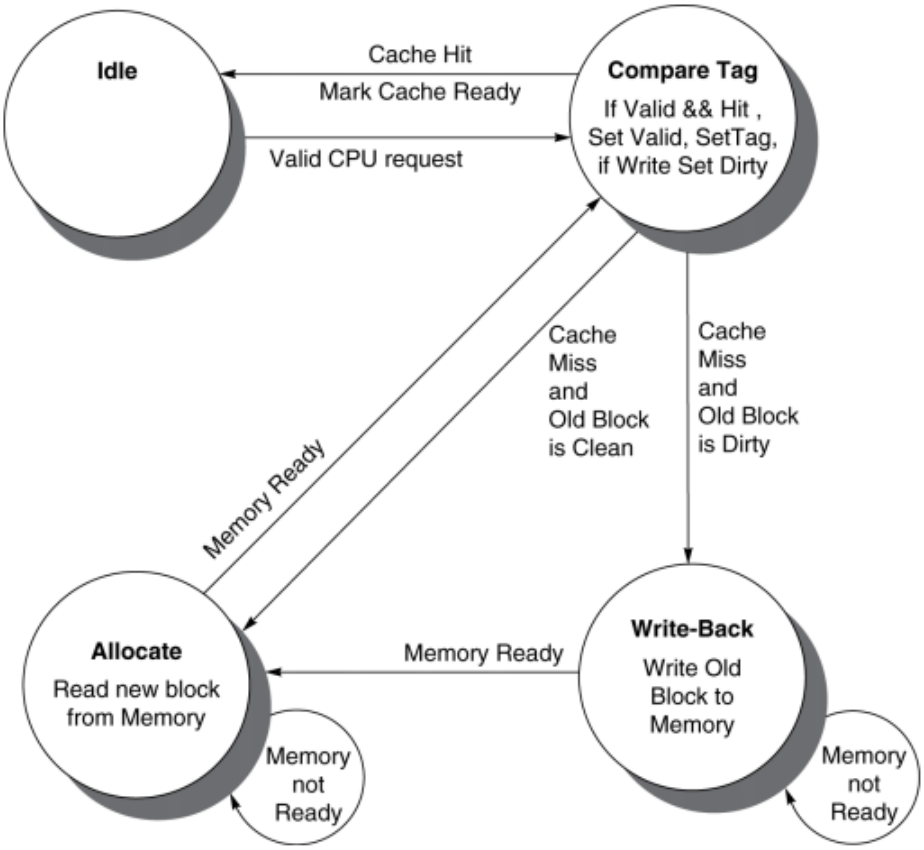
- 实验原理和实验内容
  - 在设计Cache时，我们一般需要考虑以下几个问题：
    - Cache与Memory之间的地址映射方式：
      - 全相联映射：
        - 优点： 命中率高，cache存储空间利用率高
        - 缺点：速度低，成本高，访问相关存储器时，每次都要与全部内容比较
      - 直接相联映射：
        - 优点：地址映射方式简单，访问速度快，硬件设备简单
        - 缺点：命中率低，替换操作频繁
      - 组相联映射
        - 优点：命中率比直接相联高，冲突率比直接相联低，块的利用率提高
        - 缺点：实现难度和造价比直接相联的cache高
    - Cache的替换算法：
      - 随即替换（RAND）法：简单、易于实现，但命中率较低
      - 先进先出（FIFO）法：命中率比随机法高，但因为不满足time locality，不如LRU
      - Least Recently Used（LRU）法：较好反映了time locality，一般选取LRU策略
    - 写hit的策略
      - write through：写回cache的同时写回memory。
        - 优点：可靠性更高，操作过程简单
        - 缺点：写操作速度过慢，CPU需等待的时间过长
      - 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。
        - 优点：速度较快，CPU不必等memory完成写操作后再继续执行
        - 缺点：可靠性差，控制操作较为复杂
    - 写miss的策略
      - write around：写memory，写完后并不把对应块搬回cache（不做替换）。与write through搭配使用。
      - write allocate：做替换，之后写cache，将dirty bit置为1。与write back搭配使用。

- 在本实验中，我们实现的是**两路主关联、LRU替换、write back+write allocate**的一级Cache。具体参数如下：

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

- 具体在实现Cache时，我们使用有限状态机的方法实现。我们将Cache当前state分为四个：
  - 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

  状态之间的转换如图所示：



特别注意，在我的设计中，为了解决时钟同步问题，Allocate在完成读取数据后，回到Idle状态，再从Idle状态转至Compare Tag，而不是直接进入Compare Tag状态

- 代码展示
  - Data_ram.v

```
`timescale 1ns / 1ps
module Data_ram(
input clk,
input en,
input wt,
input rst,
input[6:0] addr,
input[127:0] din,
output reg[127:0] dout
);
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
```

- Tag_ram.v

```
`timescale 1ns / 1ps
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
```

- 以上两个文件，Tag_ram.v和Data_ram.v并行地储存Cache的Data和Tag，设计参考了之前的Regs.v。注意Tag加上Valid、Dirty、Least-Used，三个bit，一共是23+3=26 bits。offset是word offset而不是byte offset，所以一个word是32bits，每个block4个words，4*32=128bits，为每个cache line的位宽。
- cache.v
  - 模块定义

```
module cache(
input clk,
input rst,
input[31:0] data_cpu_wt,
input[127:0] data_mem_rd,
input[31:0] addr_cpu,
input wt_cpu,
input rd_cpu,
input mem_ready,
output reg wt_mem,
output reg rd_mem,
output reg[127:0] data_mem_wt,
output reg[31:0] data_cpu_rd
);
```

  - 信号和储存单元的初始化

```
reg[1:0] state;
 reg[1:0] next_state;
 wire cpu_req_valid;
 wire dirty;
```

```
 wire[25:0] tag_out_orig0;
 wire[25:0] tag_out_orig1;
 wire[22:0] tag_out0;
 wire[22:0] tag_out1;
 reg[25:0] tag_in0;
 reg[25:0] tag_in1;

 wire[127:0] dout0;
 wire[127:0] dout1;
 reg[127:0] din0;
 reg[127:0] din1;

 wire[22:0] cpu_tag;
 wire[6:0] cpu_idx;
 wire[1:0] cpu_offset;


 reg wt0;
 reg wt1;
 reg wt_t0;
 reg wt_t1;
 reg hit;

 assign cpu_req_valid=rd_cpu|wt_cpu;
 assign tag_out0=tag_out_orig0[22:0],tag_out1=tag_out_orig1[22:0];
 assign cpu_tag=addr_cpu[31:9],
cpu_idx=addr_cpu[8:2],cpu_offset=addr_cpu[1:0];
 assign dirty=tag_out_orig1[23]?tag_out_orig0[24]:tag_out_orig1[24];

 Data_ram d0(
 .clk(~clk),
 .addr(cpu_idx),
 .rst(rst),
 .en(1),
 .wt(wt0),
 .din(din0),
 .dout(dout0)
 );

 Data_ram d1(
 .clk(~clk),
 .addr(cpu_idx),
 .rst(rst),
 .en(1),
 .wt(wt1),
 .din(din1),
 .dout(dout1)
 );

 Tag_ram t0(
 .clk(~clk),
 .addr(cpu_idx),
 .rst(rst),
 .en(1),
 .wt(wt_t0),
 .din(tag_in0),
 .dout(tag_out_orig0)
 );
```

```
Tag_ram t1(
.clk(~clk),
.addr(cpu_idx),
.rst(rst),
.en(1),
.wt(wt_t1),
.din(tag_in1),
.dout(tag_out_orig1)
);
```

- 时钟驱动，每一拍换状态

```
always @(posedge clk or posedge rst)
begin
if(rst) begin
state<=0;
wt0<=0;
wt1<=0;
wt_t0<=0;
wt_t1<=0;
end
else begin
state<=next_state;
end
end
```

- state的变化驱动，实现状态机的状态转换

```
always @(posedge clk or posedge rst)
begin
if(rst) begin
state<=0;
wt0<=0;
wt1<=0;
wt_t0<=0;
wt_t1<=0;
end
else begin
state<=next_state;
end
end

always @ (*) begin
case(state)
//IDLE
2'b00: begin
wt0<=0;
wt1<=0;
wt_t0<=0;
wt_t1<=0;
if(cpu_req_valid) begin next_state<=2'b01; end
else begin next_state<=2'b00;end
end
//Compare Tag
2'b01:begin
```

```
//hit
if(tag_out0==cpu_tag||tag_out1==cpu_tag) begin
hit<=1;
next_state<=2'b00;
wt_t0<=1;
wt_t1<=1;
if(tag_out0==cpu_tag) begin
if(rd_cpu) begin
tag_in0<={3'b101,tag_out0[22:0]};
tag_in1<={tag_out_orig1[25:24],1'b0,tag_out1[22:0]};
case(cpu_offset)
2'b00: begin data_cpu_rd<=dout0[31:0] ;end
2'b01:begin data_cpu_rd<=dout0[63:32]; end
2'b10: begin data_cpu_rd<=dout0[95:64]; end
2'b11: begin data_cpu_rd<=dout0[127:96]; end
endcase
end
else if(wt_cpu) begin
tag_in0<={3'b111,tag_out0[22:0]};
tag_in1<={tag_out_orig1[25:24],1'b0,tag_out1[22:0]};
 case(cpu_offset)
2'b00:begin din0<={dout0[127:32],data_cpu_wt}; end
2'b01:begin din0<={dout0[127:64],data_cpu_wt,dout0[31:0]}; end
2'b10:begin din0<={dout0[127:96],data_cpu_wt,dout0[63:0]}; end
2'b11:begin din0<={data_cpu_wt,dout0[95:0]}; end
endcase
wt0<=1;
end
end
else if(tag_out1==cpu_tag) begin
 if(rd_cpu) begin
 tag_in1<={3'b101,tag_out1[22:0]};
 tag_in0<={tag_out_orig0[25:24],1'b0,tag_out0[22:0]};
 case(cpu_offset)
2'b00: begin data_cpu_rd<=dout1[31:0] ;end
2'b01:begin data_cpu_rd<=dout1[63:32]; end
2'b10: begin data_cpu_rd<=dout1[95:64]; end
2'b11: begin data_cpu_rd<=dout1[127:96]; end
endcase
  end
else if(wt_cpu) begin
 tag_in1<={3'b111,tag_out1[22:0]};
  tag_in0<={tag_out_orig0[25:24],1'b0,tag_out0[22:0]};
case(cpu_offset)
2'b00:begin din1<={dout1[127:32],data_cpu_wt}; end
2'b01:begin din1<={dout1[127:64],data_cpu_wt,dout1[31:0]}; end
2'b10:begin din1<={dout1[127:96],data_cpu_wt,dout1[63:0]}; end
2'b11:begin din1<={data_cpu_wt,dout1[95:0]}; end
endcase
wt1<=1;
end //wt
end //the second way
end //hit
//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
if(mem_ready) begin
 next_state<=2'b11;
 wt_mem<=1;
data_mem_wt<=tag_out_orig1[23]?dout0:dout1;
  end
else begin
next_state<=2'b10;
end
end
//Allocate
2'b11:begin
if(mem_ready) begin
 next_state<=2'b00;
 rd_mem<=1;
if(tag_out_orig1[23])begin
din0<=data_mem_rd;
wt0<=1;
tag_in0<={3'b000,cpu_tag[22:0]};
wt_t0<=1;
end
else if(tag_out_orig0[23])begin
din1<=data_mem_rd;
tag_in1<={3'b000,cpu_tag[22:0]};
wt1<=1;
wt_t1<=1;
end
//the first data in
else begin
din0<=data_mem_rd;
wt0<=1;
tag_in0<={3'b000,cpu_tag[22:0]};
wt_t0<=1;
end
 end
else begin
next_state<=2'b11; end
end
default: begin next_state<=2'b00; end
 endcase
 end

endmodule
```

- 仿真验证
  - 在仿真文件中，
    - 我先引入了两次read miss来分别填充两路cache index==0的位置。
    - 接着是一次write hit，击中第一路cache index==0的位置。
    - 接着是一次read hit，击中第二路cache index==0的位置。
    - 接着是一次write miss，写入数据的地址对应cache index也是0，根据LRU原则，应替换第一路的数据块，第一路的数据块恰巧之后被写过，所以这个情况也测试write back+write allocate策略有没有被正确实现。

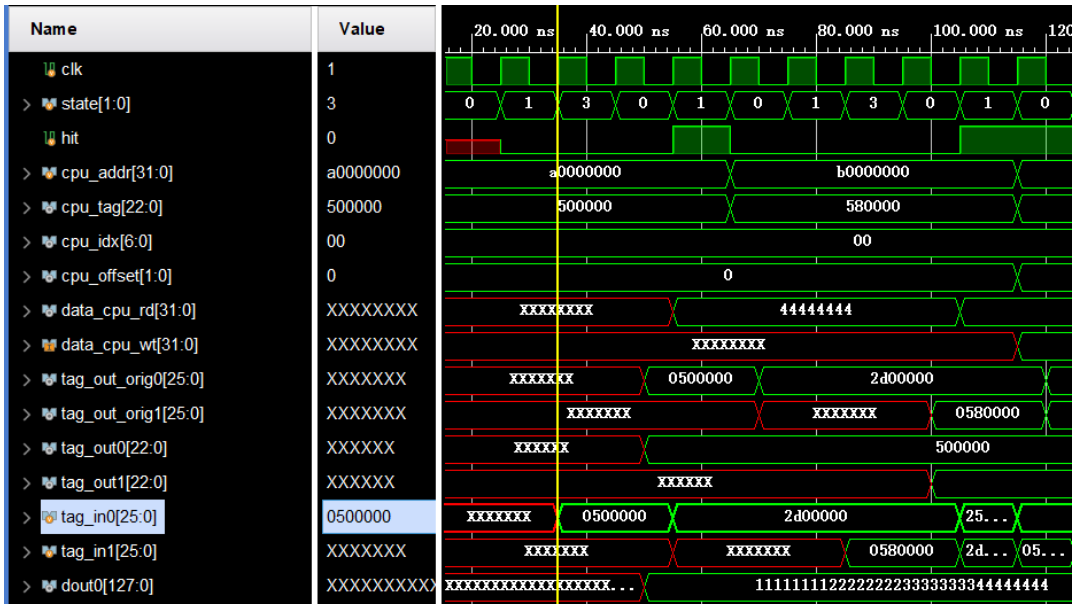
```
`timescale 1ns / 1ps
```

```
module tb();

reg clk;
reg rst;
reg wt_cpu;
reg rd_cpu;
reg mem_ready;
wire wt_mem;
wire rd_mem;
wire[127:0] data_mem_wt;
wire[31:0] data_cpu_rd;
reg[31:0] cpu_addr;
reg[31:0] cpu_data_wt;
reg[127:0] data_mem_rd;

cache cache(
.clk(clk),
.rst(rst),
.data_cpu_wt(cpu_data_wt),
.data_mem_rd(data_mem_rd),
.addr_cpu(cpu_addr),
.wt_cpu(wt_cpu),
.rd_cpu(rd_cpu),
.mem_ready(mem_ready),
.wt_mem(wt_mem),
.rd_mem(rd_mem),
.data_mem_wt(data_mem_wt),
.data_cpu_rd(data_cpu_rd)
);

always #5 clk=~clk;
initial begin
clk=0;
rst=1;
mem_ready=1;
#1;
rst=0;
//read miss
#14;
cpu_addr=32'hA0000000;
wt_cpu=0;
rd_cpu=1;
data_mem_rd=128'h11111111222222223333333344444444;
//read miss
#50;
cpu_addr=32'hB0000000;
wt_cpu=0;
rd_cpu=1;
data_mem_rd=128'h22222222333333334444444455555555;
//write hit
#50;
cpu_addr=32'hA0000002;
wt_cpu=1;
rd_cpu=0;
cpu_data_wt=32'hBBBBBBBB;
//read hit
#50;
cpu_addr=32'hB0000003;
```

```
wt_cpu=0;
rd_cpu=1;
//write miss+write back
#50;
cpu_addr=32'hF0000000;
cpu_data_wt=32'hEEEEEEEE;
wt_cpu=1;
rd_cpu=0;
data_mem_rd=128'h55555555666666667777777788888888;
end
endmodule
```

编号	地址	数据	情形
1	0xa0000000	0x44444444	read miss
2	0xb0000000	0x55555555	read miss
3	0xa0000002	0xbbbbbbbb	write hit
4	0xb0000003	0x22222222	read hit
5	0xf0000000	0xeeeeeeee	write miss 、write back

- 测试点1：read miss



可以看到，当第一次read miss时，状态机变化为Idle->Compare Tag->Allocate。

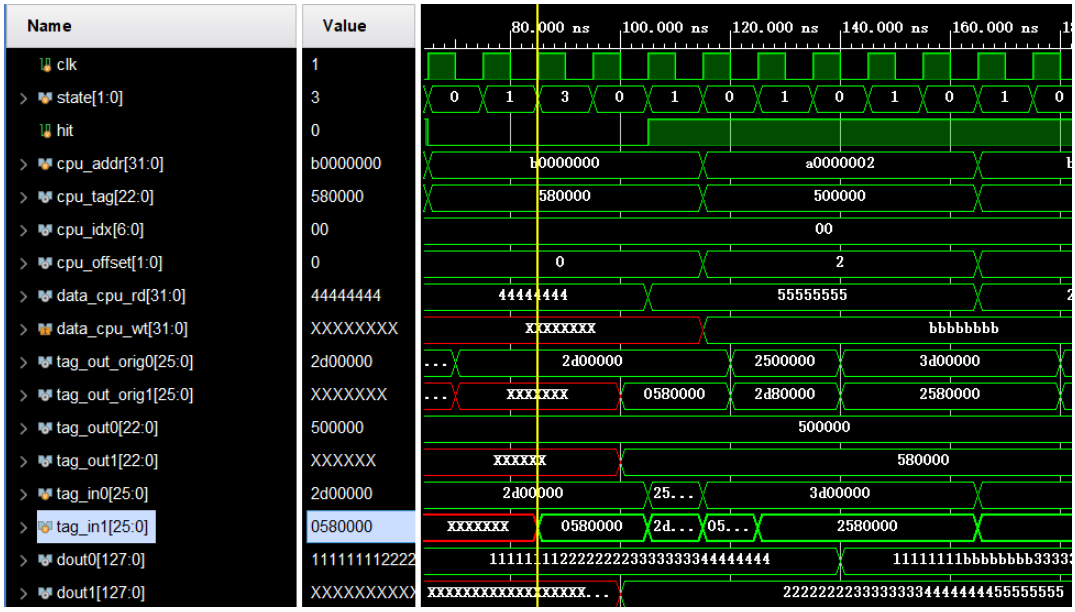
当state==3，即状态为Allocate时，tag_in0为cpu_addr的前23位，即0500000，可见数据已经写入第一路cache index==0的位置中。

之后，状态机变化为Allocate->Idle->Compare Tag。

当state==0，即状态为Idle时，dout0读出第一路cache index==0位置的数据，为128bits（4 words）的0x11111111222222223333333344444444，可见数据、tag均以写入

当state==1，即状态为Compare Tag时，可以看到hit信号由0变1，代表命中，同时更新tag，将least_used bit置为1，tag由0500000变为了2d00000.

- 测试点2：read miss



可以看到，当第二次read miss时，状态机变化为Idle->Compare Tag->Allocate。

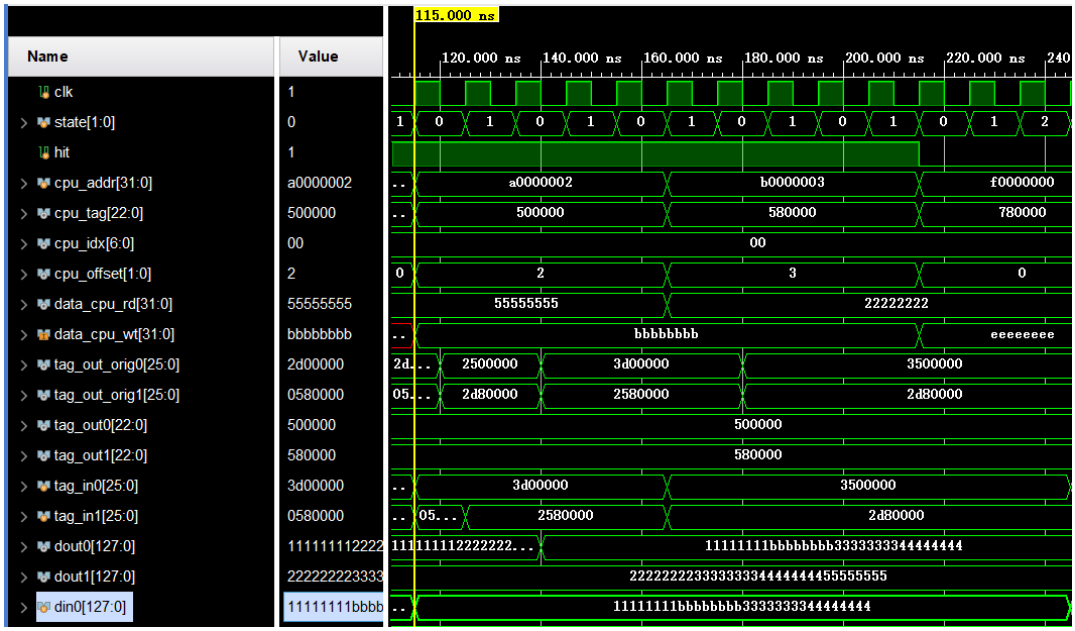
当state==3，即状态为Allocate时，tag_in1为cpu_addr的前23位，即0580000，可见数据已经写入第二路cache index==0的位置中。

之后，状态机变化为Allocate->Idle->Compare Tag。

当state==0，即状态为Idle时，dout1读出第一路cache index==0位置的数据，为128bits（4 words）的0x22222222333333334444444455555555，可见数据、tag均以写入

当state==1，即状态为Compare Tag时，可以看到hit信号由0变1，代表命中，同时更新tag，将least_used bit置为1，tag由0580000变为了2d80000.

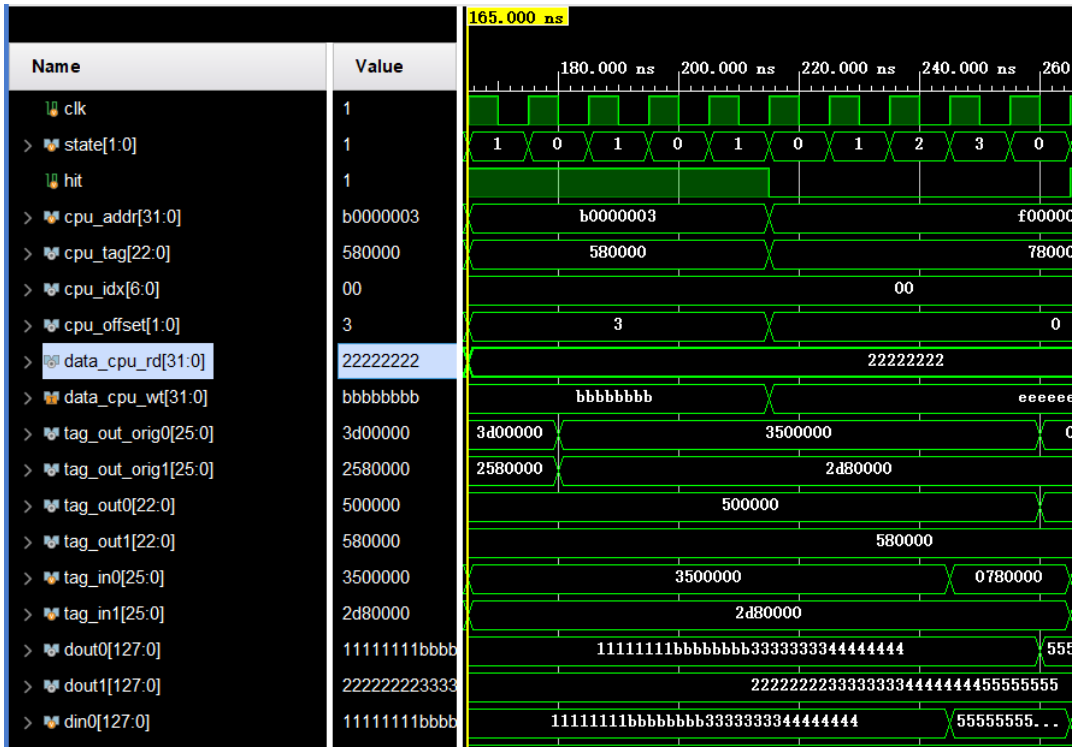
- 测试点3：write hit



可以看到，当write hit时，状态机的变化为Idle->Compare Tag->Idle，hit信号持续为1。

写入数据0xbbbbbbbb后，可以看到第一路的tag变成了3d00000，可见dirty bit已被置为1。

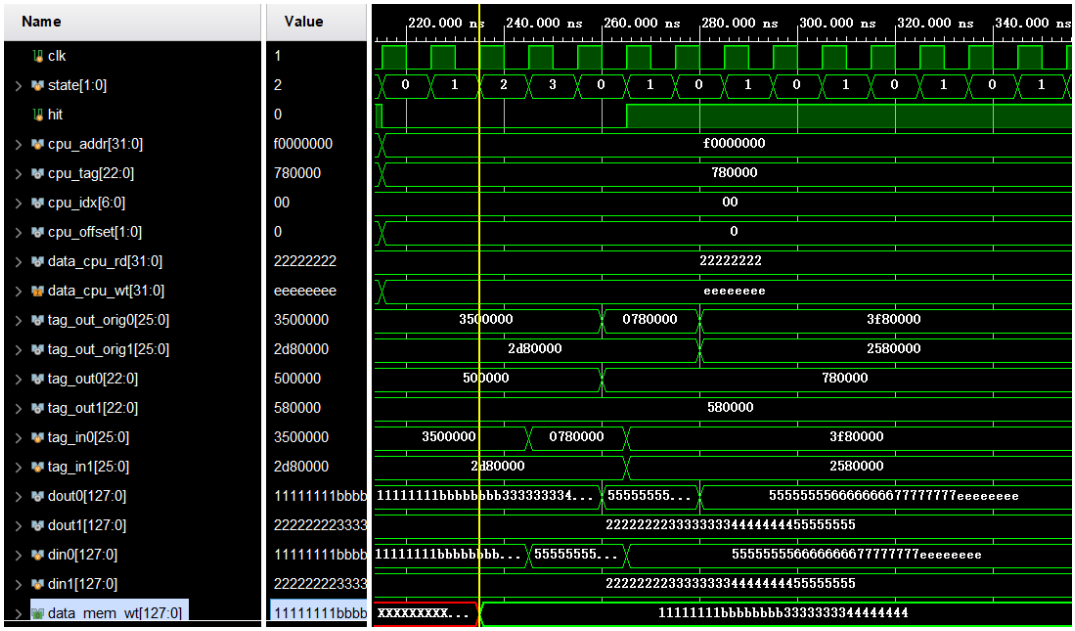
- 测试点4：read hit



可以看到，当read hit时，状态机的变化为Idle->Compare Tag->Idle，hit信号持续为1。

观察data_cpu_rd信号，可以确定被读出的数据为0x22222222，符合预期。同时tag0变为3500000，tag1变为2d80000，代表着tag0的least used bit被置为0，tag1的least used bit被置为1。

- 测试点5：write miss+write back



可以看到，当write miss且被替换出的块为dirty时，状态机的变化为Idle->Compare Tag->Write Back->Allocate->Idle，这段时间里hit信号持续为0。

当state==2，即状态为Write Back时，观察data_mem_wt信号变为了0x11111111bbbbbbbb3333333344444444，恰为被write后的第一路cache index==0的位置对应的数据。

之后，状态机从Idle变为Compare Tag，hit转为1，证明原来write miss的块已经被读入cache第一路，观察tag0变为0780000也能证明这一点。

经过这拍Compare Tag后，观察dout0，可以发现0xeeeeeeee已经被写入cache。

- 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？

    答：设计方法基本相同。但由于指令在进程执行中较少出现被修改的情况，对指令的操作绝大多数以read为主，所以使用write back+write allocate和使用write through+write around两种策略对Icache性能影响不大。我们两种策略都可以选用。即使我们选择速度更快的write back策略，由于指令顺序执行且每条指令仅被执行1次或几次，不久之后这条被写的指令就会被替换出cache，数据被写回memory，也会在短期内带来较大的写memory的开销，并不会比write through快多少。同时，指令一般都是顺序执行，space locality较好，所以命中率较dcache更高。

  - 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？

    答：首先，引起缓存缺失的指令一定在mem段，为了解决同步问题，应该向WB段插入NOP，使流水线暂停。

    read miss时，要看有没有数据竞争，如果有，则必须要让流水线stall，待数据从memory中取回，再让流水线继续执行；如没有数据竞争，则先可以让流水线继续，待到发生数据竞争的指令即将进入EX段，如果数据还没从memory中取回，则必须stall直到数据被取回。

    write miss时，如果是write back策略的cache，则流水线可以正常执行；如果是write through策略的cache，则可能需要stall，解决方法与read miss时相同。

    同时，使用write back策略时，如果被替换出的块是dirty的，且后续流水线中指令又有读这个地址的数据的操作，则流水线也必须stall，等待dirty块中数据写入memory中。当然，我们也可以设计更聪明的替换策略，使得这个dirty块不被替换出，避免数据竞争引起的stall。

- 心得体会

  在设计cache的过程中，我重温了状态机的写法，对时序电路和组合电路的配合使用有了更深刻的认识。在这个实验中，我们仅仅要求以模拟的形式给出cpu以及memory到cache的数据，这样一定程度上规避了同步问题的解决。而在现实应用中，解决cpu、cache、memory三者的同步问题往往需要花费更大心思。