浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	王若鹏
学　　院：	信息与电子工程学院
专　　业：	电子科学与技术
学　　号：	3170105582
指导教师：	刘　鹏
上课时间：	周四 3, 4, 5 节

2019 年 12 月 9 日

专业：电子科学与技术
姓名：王若鹏
学号：3170105582
日期：2019.12.9
地点：教 7-202

浙江大学实验报告

课程名称：计算机组成与设计　指导老师：刘 鹏　成绩：_____

实验名称：cache 控制器的设计　实验类型：设计型

一、实验目的与任务

1、理解并熟悉有关 cache 的架构和运行机制

2、编写 Verilog 代码，实现第一级 cache 控制器 (L1D)

二、实验主要仪器与设备

装有 VSCode、ModelSim 的电脑

三、实验原理与内容

3.1 Cache 性能参数

TMS320C621x/C671x DSP 拥有两级存储架构，包括第一级程序 cache (L1P)、第一级数据 cache (L1D)、第二级主存(L2)。下表列出了该 DSP 的一些参数：

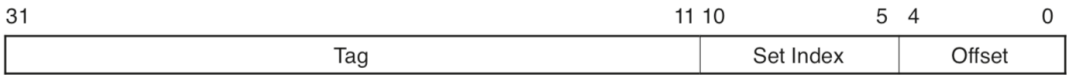
Table 1

A Simplified Model of TMS320C621x/C671x DSP	
Internal memory structure	Two Level
L1D size	4Kbytes
L1D organization	2-way set associative
L1D line size	32 bytes
L1D replacement strategy	Least Recently Used (LRU)
L1D read miss action	1 line allocates in L1D
L1D read hit action	Data read from L1D
L1D write miss action	No allocation in L1D, data sent to L2
L1D write hit action	Data updated in L1D, line marked dirty
L2 line size	128 bytes
L2 → L1D read path width	128-bit
L1D → L2 write path width	32-bit

替换脏块时，必须将整个 L1D 的 cache line 写回，但 L1D 到 L2 的写入路径宽度仅为 32

位，即需要重复 8 次才能完成。类似地，一次从二级 cache 加载会返回 128 字节的数据，也需要重复 8 次才能完成。为了简单起见，可以假设有一个无限的写缓冲区 write buffer，而忽略 TLB。此外，假设一次只允许处理一个 ld/st。

3.2 Data cache (L1D) 参数与机制



L1D 是 4K 字节 cache。它是一个双向组关联 cache，具有 32 字节的行和 64 个组。它还具有 L1D 和 L2 内存之间的 32 位 4 项写入缓冲区 write buffer。物理地址直接映射到 cache。物理地址分为三个字段，如图所示。地址的位 4－0 指定行内的偏移量 offset。地址的位 10－5 选择 cache 中 64 个组中的一个 index。地址的位 31－11 用作行的标记 tag。

当传入的两组 tag 中有一组与 address[31:11]相同时，即为 hit，否则为 miss。

当发生 write hit 时，给 L1D 发出控制信号，允许数据写入 L1D，同时将对应的 dirty 位变为 1。当发生 read hit 时，给 L1D 发出控制信号，使 L1D 中对应数据写出，并交给处理器处理。

当发生 write miss 时，由于 cache 的 write miss 机制是 no allocation in L1D, sent data to L2。因此直接将对应的数据写进 write buffer。同时，由于数据只有 32bits。无需 stall。

当发生 read miss 时，分两种情况：

当对应的 dirty 位为 0 时，无需将 L1D 中的数据进行 write back 操作，可以直接将 L2 中对应的数据写进 L1D 中，但是 L2 的 line size 为 128 bytes，L2 到 L1D 的带宽为 128 bits，因此需要分 8 次传回。这里需要一个计数器，当 8 个时钟周期后才完成传输。这里的 8 个时钟周期将基于计数器模块实现。

当对应的 dirty 位为 1 时，需要将 L1D 中的数据 write back。因为 L1D 中的 line size 为 32 bytes，而 L1D 到 L2 的数据通路为 32 bits，所以需要 8 个时钟周期传输数据。解决方法同样采用计数器。

四、实验步骤

1、分析并设计 Cache 的有限状态机

（1）绘制状态图，为每个状态分配一个唯一的二进制代码来指示，并标记转换条件信号。

（2）根据状态图，列出一个状态转换表，根据所有可能的状态信号给出每个当前状态的下一个状态。必须确保在实际电路中显示的所有状态代码都列在表中（包括在状态图中未显示的状态代码）。

（3）绘制缓存控制器的流程图。

2、通过 Verliog 语言编写 Cache 有限状态机 cache_controller.v

3、编写测试文件 cache_controller_tb.v

4、在 ModelSim 上进行仿真，并用 Vivado 实现电路图

五、实验记录与结果分析

5.1 状态机分析与设计

5.1.1 状态转移表

根据设计要求，确定有限状态机为 7 个状态，列出状态转移表：

状态	二进制	名称	功能
s0	000	Initialize	初始化，将所有控制信号置 0
s1	001	Compare_tag	比较 tag，产生 hit 与 miss 的信号，并将所有控制信号清零
s2	010	Idle_ld	写命中 write hit，将 load_ready 信号置为 1
s3	011	Idle_st	读命中 read hit，将 write_l1 信号置为 1，并将对应的 dirty 位置为 1
s4	100	Allocate	读缺失且 dirty 为 0；读命中且 valid 为 0。 当 l2_ack 为 1 时，将 read_l2 信号置为 1，等待 8 个时钟周期后，将对应的 valid 信号置为 1
s5	101	Idle_write	写缺失 write miss，将 write_l2 信号置为 1
s6	110	Write_back	读命中且 dirty 为 1，将 write_l2 信号置为 1，等待 8 个时钟后跳转到 s4 状态

由于 s2、s3、s5 状态在处理控制信号后，有可能会接收新的 ld 或者 st 信号，因此不能从这三个状态先跳回 s0 后再接收 ld/st 信号，否则会使 cache 的处理速度下降。因此，这三个状态需要悬挂 Idle，即没有 ld/st 信号时，下一个状态仍热停在自己的状态。当有 ld/st 信号输入时，跳到 s1。

进而列出状态转移的真值表：

current_state	input							next_state
	ld	st	valid	dirty	l2_ack	hit	miss	
s0	0	0	x	x	x	x	x	s0
	1	0	x	x	x	x	x	s1
	0	1	x	x	x	x	x	
s1	1	0	1	0	x	1	0	s2
	0	1	x	x	x	1	0	s3
	1	0	0	x	x	1	0	s4
	1	0	x	0	x	0	1	
	0	1	x	x	x	0	1	s5
	1	0	x	1	x	0	1	s6
s2	1	1	x	x	x	x	x	s0
	1	0	x	x	x	x	x	s1
	0	1	x	x	x	x	x	
	0	0	x	x	x	x	x	s2
s3	1	1	x	x	x	x	x	s0
	1	0	x	x	x	x	x	s1
	0	1	x	x	x	x	x	
	0	0	x	x	x	x	x	s3
s4	x	x	x	x	x	x	x	s2*
	x	x	x	x	1	x	x	s4
s5	1	1	x	x	x	x	x	s0
	1	0	x	x	x	x	x	s1
	0	1	x	x	x	x	x	
	0	0	x	x	x	x	x	s5
s6	x	x	x	x	x	x	x	s4*
	x	x	x	x	x	x	x	s6

* 表示等待8个时钟周期后跳转

5.1.2 状态转移图

下面对 7 个状态间的转移情况进行分析：

s0：只要有 ld 或者 st 信号输入，就跳转到 s1 状态。

s1：当 hit && ld && valid = 1 时，跳转到 s2；当 hit && st = 1 时，跳转到 s3；当(miss && ld &&!dirty)||(hit && ld &&!valid) = 1 时，跳转到 s4；当 miss && st = 1 时，跳转到 s5；当 ld && dirty && miss = 1 时，跳转到 s6。

s2：当!ld && !st = 1 时，跳转回本状态 s2；当 ld && st = 1 时，跳转回 s0；当 ld || st = 1 时，跳转到 s1 状态。

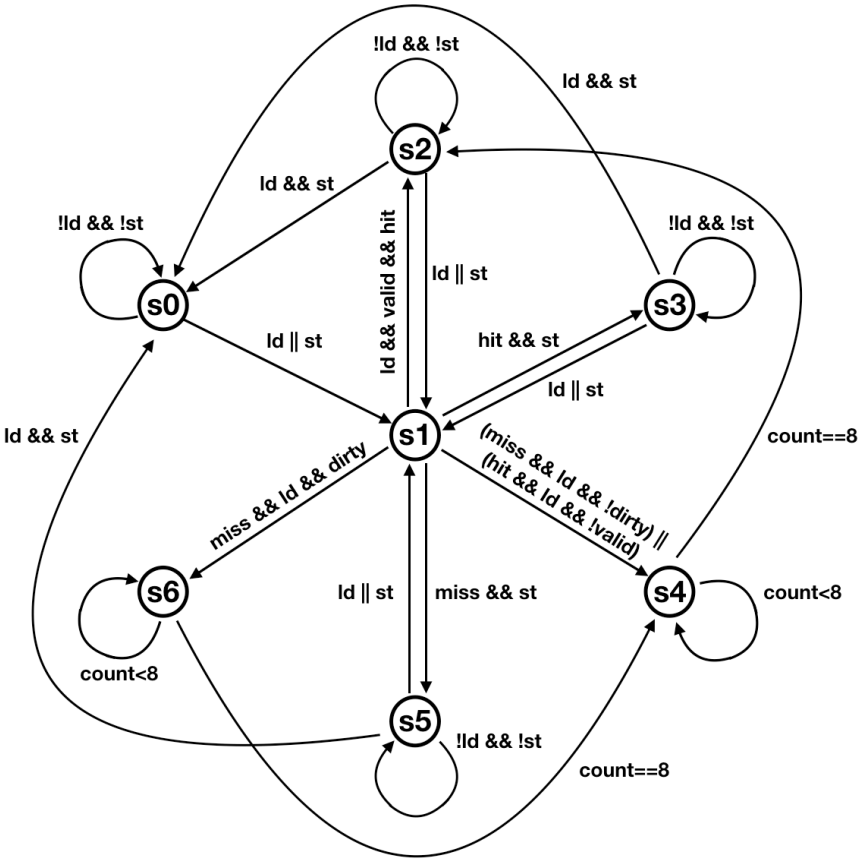
s3：当!ld && !st = 1 时，跳转回本状态 s3；当 ld && st = 1 时，跳转回 s0；当 ld || st = 1 时，跳转到 s1 状态。

s4：完全 8 次写入后，count=8，跳转到 s2 状态；当 count<8 时，跳转回本状态。

s5：当!ld && !st = 1 时，跳转回本状态 s5；当 ld && st = 1 时，跳转回 s0；当 ld || st = 1 时，跳转到 s1 状态。

s6：完全 8 次写出后，count=8，跳转到 s4 状态；当 count<8 时，跳转回本状态 s6。

综上所述，结合状态转移表，可以绘制出状态转移图：



5.2 Verilog 实现

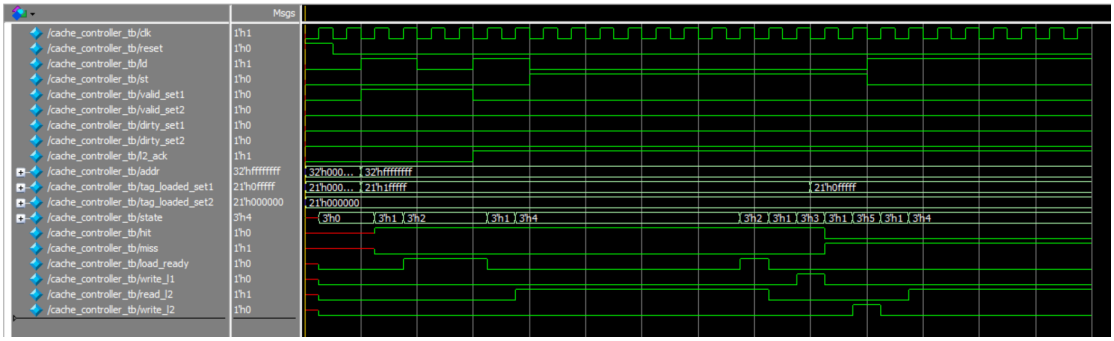
在 pdf 给出输入输出信号的基础上，我又添加了一些新的信号，以便更详细的描述 cache 中的信息，下面对输入输出信号作出解释：

I/O	信号	功能
input	clk	时钟信号
	reset	复位信号
	ld	载入请求信号
	st	存储请求信号
	addr	CPU 当前请求的物理地址
	l2_ack	从 L2 载入的数据已经到达
	tag_loaded_set1	当前 index 的第一组 block 的 tag
	tag_loaded_set2	当前 index 的第二组 block 的 tag
	valid_set1	cache 第一组的有效位

	valid_set2	cache 第二组的有效位
	dirty_set1	cache 第一组的脏位
	dirty_set2	cache 第二组的脏位
output	hit	命中信号
	miss	缺失信号
	load_ready	表示数据已经成功载入寄存器，等待 CPU 处理
	write_l1	当发生 write hit 时，控制数据写进 L1 cache
	read_l2	当发生 read miss 且脏位为 0 时，控制 L2 将数据写进 L1D
	write_l2	当发生 write miss 或者发生 read miss 时，且脏位为 1，控制数据写进 write buffer
	state	当前状态机所处状态

5.3 仿真

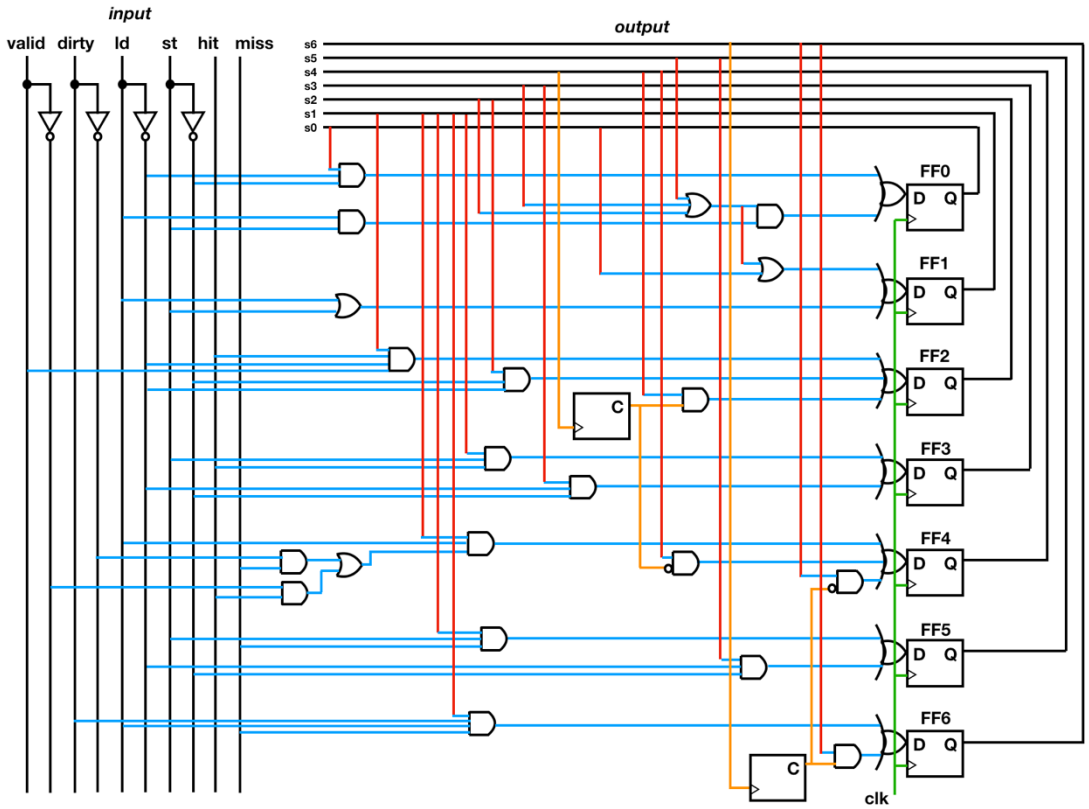
编写 cache_controller_tb.v 测试文件进行测试，依次仿真了读命中、悬挂、读命中但是 valid 为 0、写命中、写缺失、读缺失且 dirty 为 0 这几种情形。从 state 可以得到状态的转移信息与 5.1.3 中分析的相同，结合实验原理得知对应的控制信号正确，可见 cache 控制器设计成功。



5.4 电路图

将状态机的逻辑模块绘制成电路图，如下图所示。其中最左边为输入信号，右上方位输出信号总线，中间是相应的逻辑门运算，右边一列为 7 个状态寄存器，分别对应 s0~s6。布线过程中，为了避免混淆，输入端引出的信号为蓝线，输出端返回的信号为红线，绿线是时

钟信号，橙线是计数器信号。计数器模块为简化图例，计满 8 次后 C 输出高电平，下一轮时钟周期置零重新计数。



附录：Verilog 代码

(1) cache_controller.v

```
module cache_controller(clk, reset, ld, st, addr, l2_ack,
                            tag_loaded_set1,   tag_loaded_set2,   valid_set1,   valid_set2,
dirty_set1, dirty_set2,
                            //input
                            hit, miss, load_ready, write_l1, read_l2, write_l2, state
                            //output
                            );
     input ld, st, reset, clk;
     input [31:0] addr;
     input [20:0] tag_loaded_set1, tag_loaded_set2;
     input valid_set1, valid_set2, dirty_set1, dirty_set2, l2_ack;
     output reg hit, miss, load_ready, write_l1, read_l2, write_l2;
     output reg [2:0] state;

     parameter s0=3'b000, s1=3'b001, s2=3'b010, s3=3'b011, s4=3'b100, s5=3'b101, s6=3'b110;

     reg [2:0] nextstate;
     reg valid_set1_active, valid_set2_active, dirty_set1_active, dirty_set2_active;
     wire valid, dirty, hit_block1;
     wire [2:0] count;

     assign hit_block1 = (addr[31:11] == tag_loaded_set1);
     assign valid = hit_block1 ? valid_set1 : valid_set2;
     assign dirty = hit_block1 ? dirty_set1 : dirty_set2;

     counter_n #(.n(8),.counter_bits(3)) counter_8(
          .clk(clk),
          .en(write_l2||read_l2),
          .r((miss && ld && !dirty)||(hit && ld && !valid)||(miss && ld && dirty)),
          .q(count),
          .co());



     always @(posedge clk)
          begin
               if(reset) state = s0;
               else state = nextstate;
          end
```

```
always @(*)
     case(state)
          s0:  //Initialize
          begin
               load_ready=0;
               write_l1=0;
               read_l2=0;
               write_l2=0;
               dirty_set1_active=0;
               dirty_set2_active=0;
               valid_set1_active=0;
               valid_set2_active=0;
               if(!ld && !st) nextstate = s0;
               else if(ld||st) nextstate = s1;
          end

          s1:  //Compare_tag
          begin
               hit = (addr[31:11]==tag_loaded_set1)||(addr[31:11]==tag_loaded_set2);
               miss = (addr[31:11]!=tag_loaded_set1)&&(addr[31:11]!=tag_loaded_set2);
               load_ready=0;
               write_l1=0;
               read_l2=0;
               write_l2=0;
               dirty_set1_active=0;
               dirty_set2_active=0;
               valid_set1_active=0;
               valid_set2_active=0;
               if(hit && ld && valid) nextstate = s2;
               else if(hit && st) nextstate = s3;
               else if((miss && ld && !dirty)||(hit && ld && !valid)) nextstate = s4;
               else if(miss && st) nextstate = s5;
               else if(miss && ld && dirty) nextstate = s6;
          end

          s2:  //Idle_ld
          begin
               load_ready=1;
               if(!ld && !st) nextstate = s2;
               else if(ld || st) nextstate = s1;
               else if(ld && st) nextstate = s0;
          end

          s3:  //Idle_st
```

```
            begin
                write_l1=1;
                if(hit_block1) dirty_set1_active=1;
                else dirty_set2_active=1;
                if(!ld && !st) nextstate = s3;
                else if(ld || st) nextstate = s1;
                else if(ld && st) nextstate = s0;
            end

            s4: //Allocate
            begin
                read_l2=l2_ack;
                if(count!=3'b111)
                    nextstate = s4;
                else
                    begin
                    nextstate = s2;
                    if(hit_block1) valid_set1_active=1;
                    else valid_set2_active=1;
                    end
            end

            s5:  //Idle_write
            begin
                write_l2=1;
                if(!ld && !st) nextstate = s5;
                else if(ld || st) nextstate = s1;
                else if(ld && st) nextstate = s0;
            end

            s6:  //Write_back
            begin
                write_l2=1;
                if(count!=3'b111) nextstate = s6;
                else begin nextstate = s4; write_l2=0; end
            end


        endcase

endmodule


module counter_n(clk, en, r, q, co);
```

```
    parameter n=1;
    parameter counter_bits=1;
    input clk, en, r;
    output co;
    output [counter_bits-1:0] q;
    reg [counter_bits-1:0] q=0;
    assign co=(q==(n-1))&&en;
    always @(posedge clk)
        begin
        if(r)
            q=0;
        else if(en)
            begin
            if(q==(n-1)) q=0;
            else q=q+1;
            end
        end
endmodule
```

(2) cache_controller_tb.v

```
`timescale 1ns / 1ps

module cache_controller_tb;

    parameter delay=100;

    reg clk, reset, ld, st, valid_set1, valid_set2, dirty_set1, dirty_set2, l2_ack;
    reg [31:0] addr;
    reg [20:0] tag_loaded_set1, tag_loaded_set2;
    wire [2:0] state;
    wire hit, miss, load_ready, write_l1, read_l2, write_l2;

    cache_controller L1D(
        .clk(clk),
        .reset(reset),
        .ld(ld),
        .st(st),
        .addr(addr),
        .tag_loaded_set1(tag_loaded_set1),
        .tag_loaded_set2(tag_loaded_set2),
        .valid_set1(valid_set1),
        .valid_set2(valid_set2),
```

```
          .dirty_set1(dirty_set1),
          .dirty_set2(dirty_set2),
          .l2_ack(l2_ack),
          .hit(hit),
          .miss(miss),
          .load_ready(load_ready),
          .write_l1(write_l1),
          .read_l2(read_l2),
          .write_l2(write_l2),
          .state(state));


     initial
     begin
          // Initialize Inputs
          clk = 0;
          reset = 1;
          ld = 0;
          st = 0;
          valid_set1 = 0;
          valid_set2 = 0;
          dirty_set1 = 0;
          dirty_set2 = 0;
          l2_ack = 0;
          addr = 32'b0;
          tag_loaded_set1 = 21'b0;
          tag_loaded_set2 = 21'b0;

          #(delay)  reset=0; //initialize
          #(delay) ld = 1; addr = 32'hffffffff; tag_loaded_set1 = 21'h1fffff; valid_set1 = 1;
//read hit and valid
          #(delay*2) ld = 0; //idle
          #(delay*2)  ld = 1; valid_set1 = 0; l2_ack = 1; //read hit but invalid
          #(delay*2) ld = 0; st = 1; //write hit and valid
          #(delay*10) st = 1; tag_loaded_set1 = 21'h0fffff; //write miss
          #(delay*2) st = 0; ld = 1; dirty_set1 = 0; //read miss and the dirty bit is 0

          #(delay*8)  $stop;

     end

     always   #(delay/2) clk=~clk;

endmodule
```