cache设计文档

1. 概述

我的cache的基本思想来自wisconsin的CS 552的cache设计;

首先，cache中所有的字段都是cache\_mem的实例，

cache\_mem是这样的：

\_\_\_\_

|\_\_\_|

|\_\_\_|

|\_\_\_|

|\_\_\_|

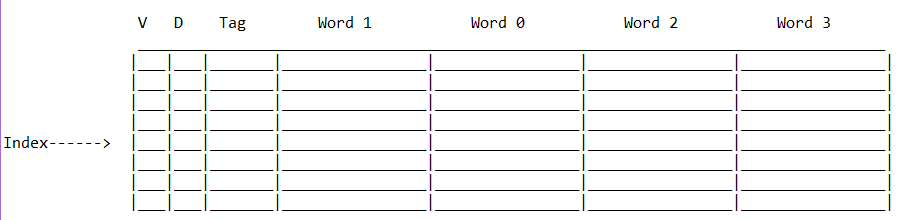
|\_\_\_|

|\_\_\_|

|\_\_\_|

|\_\_\_|

然后将它作为cache的各个列实例化为valid位，dirty位，tag字段和各个block：



比较有特点的地方是cache\_oneline中的cmp信号，这个信号使得cache在面向cpu读写和面向主存读写的时候使用同一套接口，虽然可能会带来burst read&write的速度下降，但是实现很巧妙，而且现实中的主存的速度比cache差了2个数量级，这几个周期的延迟可以忽略。

1. cache\_mem

cache\_mem的接口包括：clk,rst,write,data\_in,addr,data\_out；其中rst用于将该列cache清零，write是写使能，当write有效的时候，data\_in有意义，addr用于从cache列中选择一行，data\_out是输出数据。

1. 单路cache的实现

要完成一个简单的单路的cache，需要实例化出宽度为1的valid位，宽度为1的dirty位，宽度为20的tag位；cache每行有8个word，每个word包含32位，所以我先把cache\_mem实例化word，然后在单行cache中实例化8个word。下面所使用到的信号名与cache\_oneline模块中的信号名是统一的，建议结合模块cache\_oneline阅读。

1. cache\_oneline中的信号介绍

* input enable ：当将单行cache用于多路cache的时候，有的行就不会被使用，所以需要是能信号；
* input [INDEX\_WIDTH-1:0] index ：下标；
* input [OFFSET\_WIDTH-1:0] word\_sel ：用于选择一个block中的8个word中的某一个；
* input cmp ：当cache被cpu读写的时候，需要比较输入的tag和改行cache的tag，此时cmp有效；当执行cache写回、从主存中载入内容时，不需要比较tag，此时cmp无效；
* input write ：写使能；
* input [TAG\_WIDTH-1:0] tag\_in ：当cmp无效、write有效时，将它写入当前行的tag域；当cmp有效时，用于和当前行中的tag相比较；
* input [31:0] data\_in ：输入数据；
* input valid\_in ：要写入valid位的值；
* input [3:0] byte\_w\_en ：字节写使能；
* input clk；input rst；略
* output hit ：在当前行命中；
* output dirty ：仅读取的时候有意义，表示index所映射的行为脏；
* output [TAG\_WIDTH-1:0] tag\_out ：当前index所映射到的行的tag；
* output valid\_out ：当前index所映射到的行是否可用；
* go ：表示非reset且enable有效；
* match ：表示输入的tag与index映射到的行的tag相同；

1. word的实现

word的输入数据和输出数据为32位；

每个word中实例化4个byte，每个byte都是宽度为8位的cache\_mem，将word的32位输入输出截断为4段8字节的输入输出；每个byte的写使能都是由byte\_w\_en&&write得到。

1. word的写使能信号

第N个word的写使能信号定义为wordN\_w\_en

assign wordN\_w\_en = go & write & (word\_sel == 3'dN) & (match | ~cmp);

go & write & (match | ~cmp) ： 该路cache的使能信号有效，写使能信号有效，且cache hit或者正在从主存载入块；

(word\_sel == 3'dN) ： 选中index映射到的行的第N个word；

1. valid位的信号

valid位的写使能型号是valid\_override，输入数据是valid\_in。当从主存中载入数据到cache中时，valid\_override信号有效，valid\_in为1；

1. dirty位的信号

dirty位的写使能信号是dirty\_override，输入数据是dirty\_in。当从主存中载入数据到cache中时，dirty\_override为1，dirty\_in为0，因为刚载入的块一定与主存中的块相同；当指令写入cache时，dirty\_in为1，因为写入后该cache块便与主存中的内容不一致。

1. 输出数据的选择：

在data\_sel这个选择器中，根据word\_sel选择输出的数据。

1. 二路组相联cache的实现

二路组相联的cache的主体是将2个oneline cache实例化得到的，但是需要处理一些细节：如何判定是否hit，miss之后如何替换。二路组相联的cache向外提供的接口以及接口的功能和单路的cache是几乎一样的（只有hit信号的意义有差别），不再赘述。下面介绍模块内部的信号，请结合模块cache\_2ways阅读：

1. reg victimway\_ff ：用于实现伪随机替换算法，表示当cache miss时，二路cache中的哪一路会被替换；
2. valid0, valid1, dirty0, dirty1 ：第一、二路cache是否有效；第一、二路cache是否为脏，他们会被用于被替换行的决策；
3. enable0, enable1 ： 第一、二路cache是否被使能，当cmp有效时，说明是在正常状态下（cpu正常运行，cache的读写是面向cpu的），此时第一、二路cache的使能由输入的enable决定；当cmp无效时，说明是在非正常状态下（cache miss，cache正在执行写回或者从主存载入），此时第一、二路的cache使能由victimway决定；
4. write0, write1 ：仅当write有效时，这两个信号才可能有效，下面的讨论建立在write有效的基础上：当cmp无效时，说明正在从主存载入块，替换victimway中的数据，所以，write0和write1由victimway决定；当cmp无效时，仅当第一或第二路命中时，它们中的一个才会被写入，我在单路cache中进行了写入与命中的判定逻辑，所以这里实际上是不需要判断是否命中的，但是在我目前的代码中在此处进行了判断，这里的判定是冗余的。
5. hit ：单路cache中的hit仅仅表明了输入的tag和index映射的tag是否相同，而cache\_2ways中的hit则还要求valid为高，注意这里对valid0和valid1的要求不是冗余的：

assign hit = (valid1 & hit1) | (valid0 & hit0);

1. dirty ：表示index映射的组中的2行都是脏的；
2. tag\_out ：目前看了这个信号也没有用
3. victimway\_sel ：victimway选择模块，建议参照该模块源码阅读以下内容：

首先，初始化时，victimway的值应该为0；

其次，当从主存中载入块或者写回的时候，victimway不能发生变化，因为enable0, enable1, wirte0, write1此时都是由他决定的，而此过程中肯定只能使用二路中的一路；

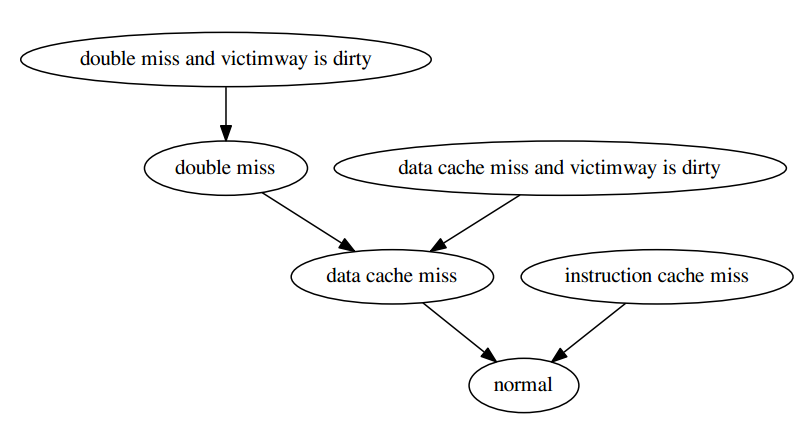
当正常模式下遇到cache miss，需要决定将主存块载入到哪一路的时候，遵从以下原则：（自上而下，优先级变低）

* 优先选择无效的行，都无效时载入到第一行；
* 优先选择无脏块的行，都不为脏时选择第0行，都为脏时随机选择。

1. instruction cahe + data cache的实现 和 cache状态机

cache\_manage\_unit将两块cache封装为一个整体，提供面向主存和cpu的接口。其中，instruction cache和data cache 是将cache\_2ways实例化得到的，在本层模块中为它们提供输入信号。在不同的状态下控制信号差别很大，所以我下面分状态来谈，请结合cache\_manage\_unit和cache\_control两个模块来阅读，cache\_control模块是生成各种控制信号的组合电路。

1. miss处理过程中的状态转移图



1. 通常模式

* ic\_enable ：有效，因为每个周期都会取指令；
* ic\_cmp ：无效，因为通常模式下会比较tag是否一致；
* ic\_write ：无效，通常模式下不会写指令cache
* ic\_data\_sel ：无关；
* ic\_valid\_reg ：无关；
* ic\_word\_sel ：指令地址低3位；
* ic\_byte\_w\_en ：无关；
* dc\_enable ：当指令读写主存时有效；
* dc\_cmp ：有效，原因同上；
* dc\_write ：当指令写主存时有效；
* dc\_data\_sel ：选择来自主存的数据；
* dc\_valid ： 无关；
* dc\_word\_sel ：数据地址低3位；
* dc\_byte\_w\_en ：来自cpu的字节写使能；
* ram\_addr\_sel ：无关；
* ram\_en ：无效
* ram\_write ：无效
* status\_next ：表示下一个状态，根据ic\_hit, dc\_hit 和dc\_dirty决定进入哪一个状态，非正常状态一共有5个，如上面的状态转移图所示。
* counter\_next ：表示下一周期计数器的值，如果要进入通常状态以外的状态，那么应该将计数器的值置为0；

1. `STAT\_DOUBLE\_MISS和`STAT\_IC\_MISS

这两个状态下将指令载入到指令cache。

* ic\_enable ：有效，因为要写ic
* ic\_cmp ：无效，因为直接写入到victimway
* ic\_write　：有效；
* ic\_data\_sel　：dc命中时为１（选择来自dc的数据），否则为０；这个选择信号是用于保证ic和dc的一致性的。
* ic\_valid ： 有效；
* ic\_word\_sel ：与conuter相同；
* ic\_byte\_w\_en ：1111
* dc\_enable　：有效，因为可能需要从dc读取数据；
* dc\_cmp ：有效，因为需要确定dc是否命中；
* dc\_write ：无效；
* dc\_data\_sel ：无关；
* dc\_valid ：无关；
* dc\_word\_sel ：与conuter相同；
* dc\_byte\_w\_en ：无关；
* ram\_en ：当dc未命中时为1，否则为0；
* ram\_addr\_sel ：2’b00，选择用指令地址的高位和counter拼接得到的地址作为输入到ram的地址；
* ram\_write ：无效；

1. `STAT\_DC\_MISS\_D和`STAT\_DOUBLE\_MISS\_D

这两个状态进行脏块写回操作。

* ic\_enable ：无效；
* ic\_cmp ：无效；
* ic\_write　：无效；
* ic\_data\_sel　：无关；
* ic\_valid ： 无关；
* ic\_word\_sel ：无关；
* ic\_byte\_w\_en ：无关
* dc\_enable　：有效，因为需要读取的内容dc；
* dc\_cmp ：无效，因为此时写回的行由victimway指定，不需要比较；
* dc\_write ：无效；
* dc\_data\_sel ：无关；
* dc\_valid ：无关；
* dc\_word\_sel ：与conuter相同，因为需要写回整个块；
* dc\_byte\_w\_en ：无关；
* ram\_en ：有效，因为要写回；
* ram\_addr\_sel ：2’b11，选择（dc中index映射的组中被victimway选中的行的tag和index）和（counter）拼接起来的地址作为写入到ram的地址。
* ram\_write ：有效；

1. `STAT\_DC\_MISS

此状态下要做的事是将数据载入到数据cahe

* ic\_enable ：无效；
* ic\_cmp ：无效；
* ic\_write　：无效；
* ic\_data\_sel　：无关；
* ic\_valid ： 无关；
* ic\_word\_sel ：无关；
* ic\_byte\_w\_en ：无关
* dc\_enable　：有效，因为需要写入dc；
* dc\_cmp ：无效，因为需要覆盖的行由victimway指定，不需要比较；
* dc\_write ：有效；
* dc\_data\_sel ：1，选择来自主存的数据；
* dc\_valid ：有效，因为载入完成后该行cache便变为有效；
* dc\_word\_sel ：与conuter相同，因为需要载入整个块；
* dc\_byte\_w\_en ：4’b1111；
* ram\_en ：有效，因为要读取主存内容；
* ram\_addr\_sel ：2’b01，选择（当前cpu请求的数据地址的高位）和（counter）拼接起来的地址作为读取ram的地址。
* ram\_write ：无效；

1. 数据一致性

关于数据一致性的一些信号在这里进行解释。

1. ic与dc的最常见的一致性问题是在使用loader的时候发生的：当loader将程序加载到内存中时，由于dc采用的是写回策略，所以程序很可能还存在于cache中，并没有修改相应的主存，如果这时候执行载入的程序，那么很可能执行的是主存中的旧的指令。为了解决这种不一致性，我在cache\_manage\_unit模块中增加了一个信号：

wire loading\_ic = status ==`STAT\_IC\_MISS || status == `STAT\_DOUBLE\_MISS;

这个信号有效时，送入dc的tag和index是指令地址的相应字段，而不是数据地址的相应字段：

assign ic\_tag = ic\_addr[29:29-TAG\_WIDTH+1];

assign dc\_tag = (~loading\_ic) ? dc\_addr[29:29-TAG\_WIDTH+1] : ic\_tag;

assign ic\_index = ic\_addr[29-TAG\_WIDTH:OFFSET\_WIDTH];

assign dc\_index = (~loading\_ic) ? dc\_addr[29-TAG\_WIDTH:OFFSET\_WIDTH] : ic\_index;

如果发现dc hit，那么就从dc中载入块到ic中，避免了上述的不一致性。

1. 还有一种比较特殊的情况是执行了某一段代码之后，它还存在于ic中，紧接着对应地址被写入了新的数据，这样ic中的数据仍然是旧值。目前我对这种情况采用的是鸵鸟策略，因为我认为这种情况发生的概率太低。如果这种情况发生了，应该这样处理：当dc miss时，如果cpu送入的写使能信号有效，当从主存中载入块到dc时，将数据地址的高位送给ic，如果hit则将index映射到的行的valid位写为0。这样处理之后，如果某个块先被写入，然后被执行，那么1中的策略会使他执行正确的值；如果它先被执行再被写入，那么ic中的块会被置为无效，下次执行时会重新载入新的指令。
2. 但是在2中仍然没有解决这种情况：某个块先被执行，再被写入，再被执行，此时ic和dc中相应的行都被变为valid，如果再对他进行写入，再执行，那么此时会执行到旧的指令。我认为目前的cache结构难以解决这个问题，不过我认为在我们的实验中他不会发生。