# Cache Coherency Protocol

吕文龙

14210720082

Introduction

Code memory

Processor

Memory-Bus

Cache

Testbench

TODO

# 概述

两个具有指令计数器(PC), 包含五条指令, 实现取指译码(fetch and decode), 执行(exe),访存(mem)功能, 带两个register的processor

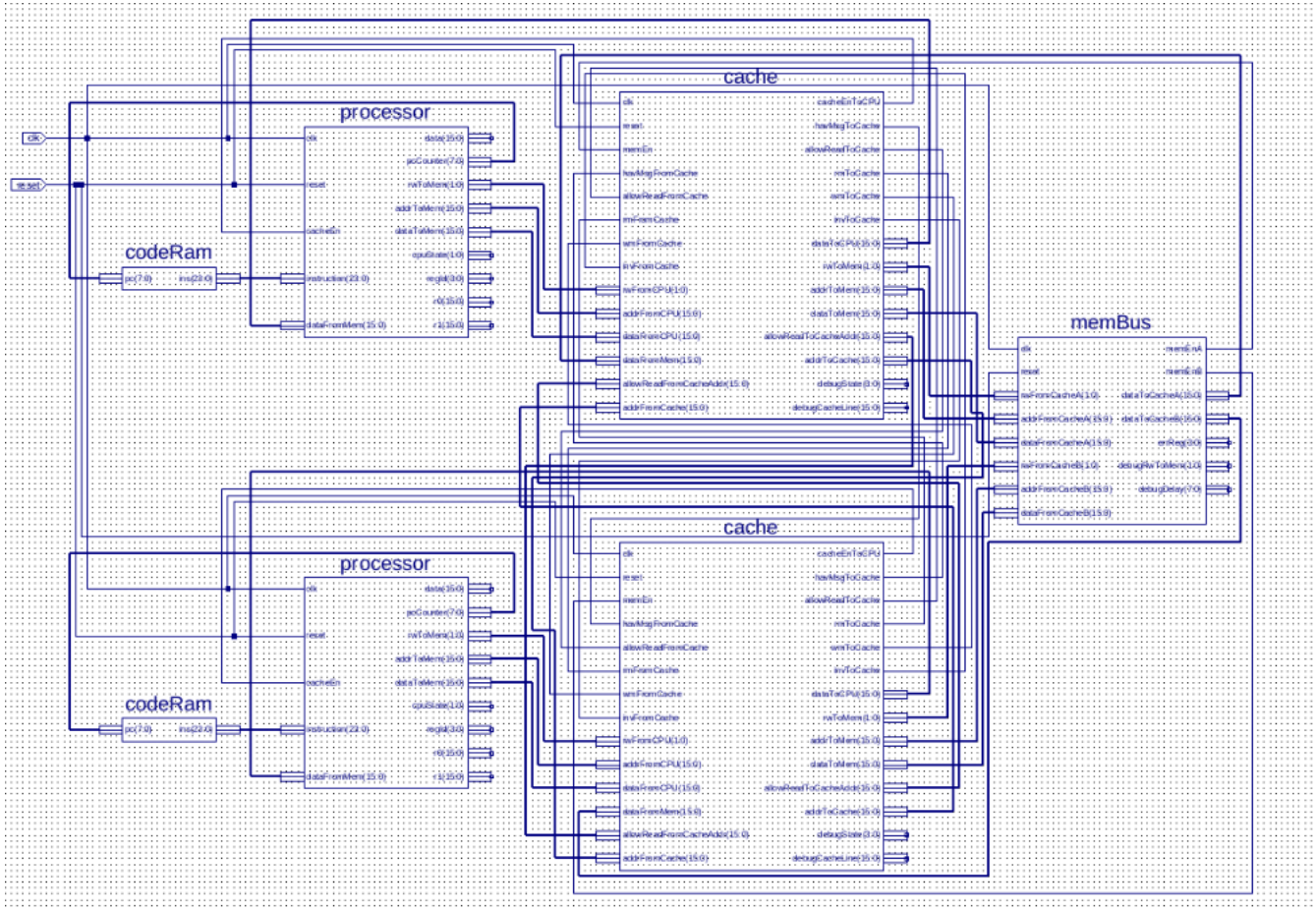
每个Processor各连接一个cache, 因为project重点是实现一致性协议, 每个cache只包含一条cacheline, 同时每个block只有一个word.

# 概述

memory bus及data memory. data memory使用verilog中的reg变量模拟, 具有5个clock cycle的访存延迟.

code memory. 每个processor的PC连接一个code memory, 其中存储该processor的指令. 测试时只要把指令存入, 就可以通过PC自动加载.

# 系统框图



Introduction

Code memory

Processor

Memory-Bus

Cache

Testbench

TODO

# Code memory

```
`include "./def.v"
module codeRam(
    input[`PCWIDTH-1:0] pc,
    output reg[`INSWIDTH-1:0] ins
);
parameter CODESIZE = 8; //a program could have at most 8 ins
reg[`INSWIDTH-1:0] codes[0:CODESIZE-1];
reg[3:0] codeSize;
always @(pc,codeSize) begin
    if(pc >= codeSize) begin
        ins = 0; //return nop
    end
    else begin
        ins = codes[pc];
    end
end
endmodule
```

# load instructions

```
initial begin
    clk             = 1'b0;
    reset           = 1'b0;
    code1.codeSize = 3;
    code2.codeSize = 3;
    code1.codes[0] = {`SET,  `R0,    `WORDWIDTH'd3}; //p1.r0 = 3
    code1.codes[1] = {`ST,   `R0,    `ADDRWIDTH'd0}; //mem[0] = p1.r0, write miss
    code1.codes[2] = {`NOP,  `R0,    `WORDWIDTH'd0}; //nop;
    code2.codes[0] = {`SET,  `R0,    `WORDWIDTH'd4}; //p1.r0 = 3
    code2.codes[1] = {`NOP,  `R0,    `WORDWIDTH'd0}; //nop
    code2.codes[2] = {`ST,   `R0,    `ADDRWIDTH'd0}; //mem[0] = p1.r0, write miss
    // rest of test bench code
end
```

Introduction

Code memory

Processor

Cache

Memory-Bus

Testbench

TODO

# Processor

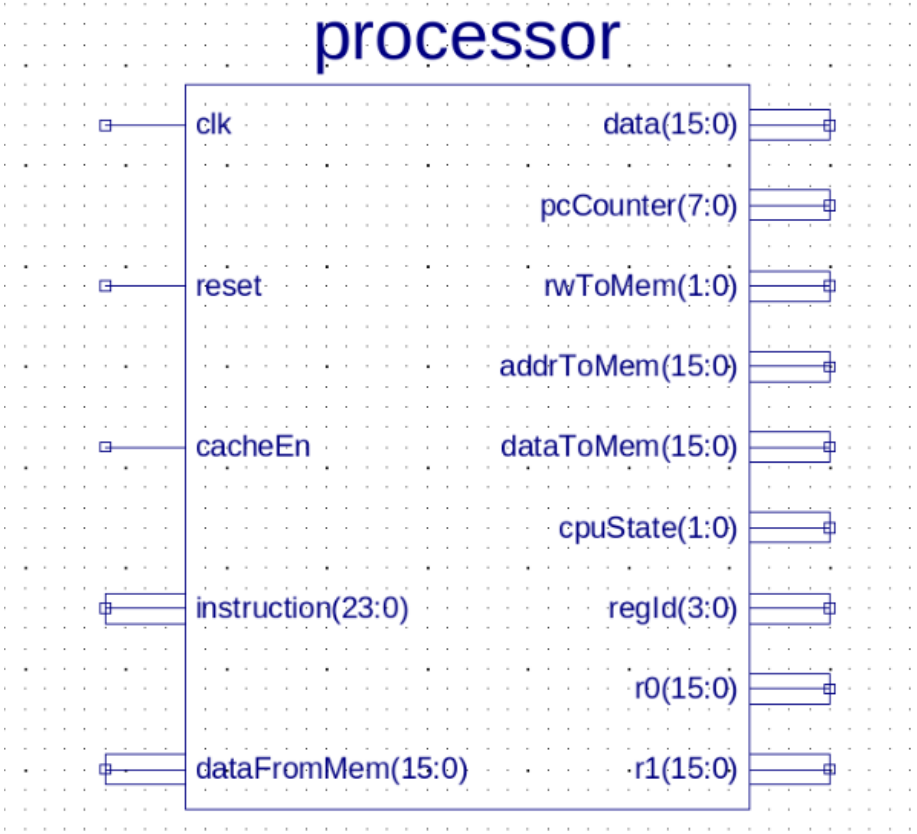
Processor具有五条指令, 能够完成内存读写, 寄存器读写功能.

Cache对Processor透明, hit与miss对CPU而言, 只是访存时间变化而已.

指令分为fetch and decode, exe, mem三个阶段, 没有流水线, 当访存时, CPU要stall

两个16bit寄存器

# Processor——Schematic Symbol



# Processor——输入输出

```
module processor(
    input clk,
    input reset,

    //interact with real world
    input [`INSWIDTH-1:0]  instruction,
    output reg[`WORDWIDTH-1:0] data,
    output[`PCWIDTH-1:0]          pcCounter,
    //interact with memory, cache is transparent to CPU
    output reg[`IOSTATEWIDTH-1:0] rwToMem,
    output reg[`ADDRWIDTH-1:0] addrToMem,
    output reg[`WORDWIDTH-1:0] dataToMem,
    input wire cacheEn,
    input wire[`WORDWIDTH-1:0] dataFromMem,

    //for debug purpose
    output [`CPUSTATENUMWIDTH-1:0] cpuState,
    output [`REGWIDTH-1:0] regId,
    output [`WORDWIDTH-1:0] r0,
    output [`WORDWIDTH-1:0] r1
);
```

# Processor——ISA

```
`define NOP   4'd0
`define LD    4'd1
`define ST    4'd2
`define SET   4'd3
`define GET   4'd4
```

为CPU设计了5条简单的指令,
指令为24位,
其中4位表示指令类型,
4位为寄存器ID,
16位为内存地址或者立即数,
指令格式均为OP  REGID  ADR_DATA.

# Processor——状态转换

如果当前状态为FETCH, 则说明取指与译码已经完成, 转入exe阶段. 代码如下:

如果当前状态为EXE, 对于非访存操, 执行操作, 并转入FETCH, 对访存操作, 转入MEM

如果当前状态为MEM, 则等待memory访问完成后, 转入FETCH

Introduction

Code memory

Processor

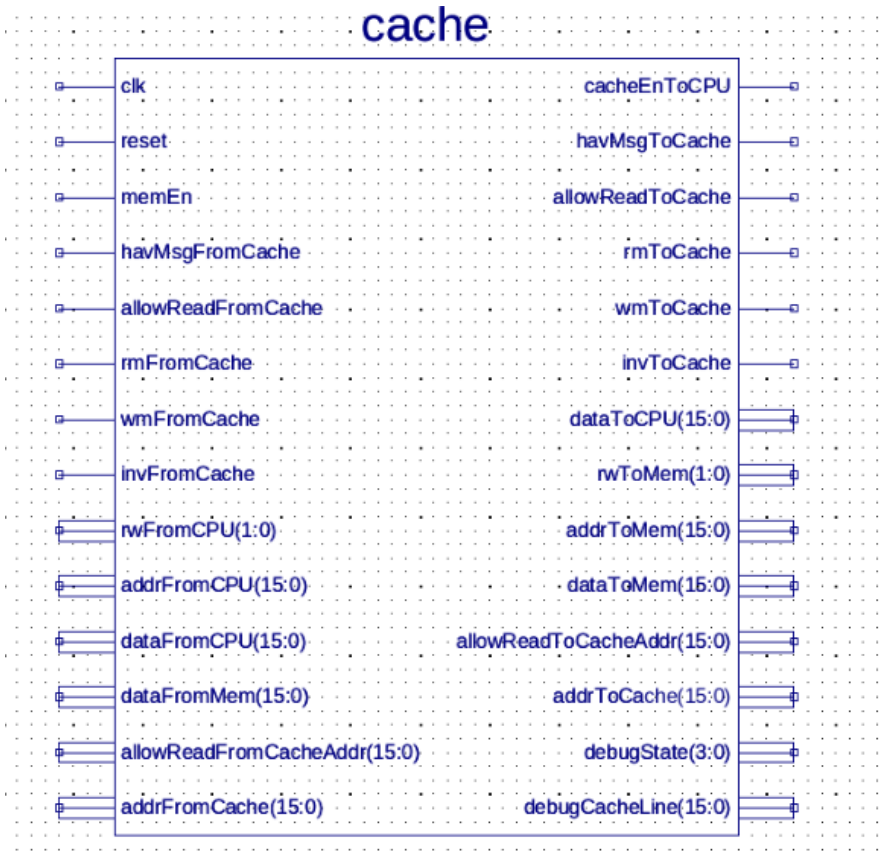
Cache

Memory-Bus

Testbench

TODO

# Cache


cache
clk
reset
memEn
havMsgFromCache
allowReadFromCache
rmFromCache
wmFromCache
invFromCache
rwFromCPU(1:0)
addrFromCPU(15:0)
dataFromCPU(15:0)
dataFromMem(15:0)
allowReadFromCacheAddr(15:0)
addrFromCache(15:0)
cacheEnToCPU
havMsgToCache
allowReadToCache
rmToCache
wmToCache
invToCache
dataToCPU(15:0)
rwToMem(1:0)
addrToMem(15:0)
dataToMem(15:0)
allowReadToCacheAddr(15:0)
addrToCache(15:0)
debugState(3:0)
debugCacheLine(15:0)

# Cache——设计难点

某个cycle同时监听到总线信息和CPU访存请求时的处理

课程课件上的状态转换图只有MODFIED, SHARED, INVALID三种状态, 但这个状态图并非是一个cycle to cycle的状态转换图. 即, 状态的转换尚需要许多中间动作, 这些中间动作本质上是新的状态.

两个cache之间的通信以及互动

# Cache——设计思路

cacheLine在每个clock cycle首先检查总线信息, 如果总线上有信息, 则优先处理总线信息.

设置一些中间状态, 即除了MSI三个状态之外, 另设计一些中间状态表示状态转换的进程.

因为是双核系统, 因此cache之间可以直接互联

# Cache——输入输出

```
input clk,  //时钟信号
input reset,//复位信号

input[`IOSTATEWIDTH-1:0] rwFromCPU,          //表示CPU访存动作，允许状态有read/write/idel
input[`ADDRWIDTH-1:0]    addrFromCPU,        //由CPU传入的访存地址
input[`WORDWIDTH-1:0]    dataFromCPU,        //由CPU传入的数据
input[`WORDWIDTH-1:0]    dataFromMem,        //从memory取回的数据
input                    memEn,              //memory bus向cache发送的表示访存完成的信号
input                    havMsgFromCache,    //表示收到其他cache的广播
input                    rmFromCache,        //收到广播的read miss信号
input                    wmFromCache,        //收到广播的write miss信号
input                    invFromCache,       //收到广播的invalidate信号
input                    allowReadFromCache, //表示这个cache不必等待其他cache的访存完成
input[`ADDRWIDTH-1:0] allowReadFromCacheAddr,    //需要等待其他cache访存完成的地址
input[`ADDRWIDTH-1:0] addrFromCache,         //其他cache广播信号的内存地址
output reg                       cacheEnToCPU, //告诉CPU写入完成或者可以放心读取数据
output reg[`WORDWIDTH-1:0]       dataToCPU,    //响应CPU read发送的数据
output reg[`IOSTATEWIDTH-1:0]    rwToMem,      //cache访存的动作，可以为write/read/idel
output reg[`ADDRWIDTH-1:0]       addrToMem,    //传入memory的地址
output reg[`WORDWIDTH-1:0]       dataToMem,    //要写入memory的数据
output reg havMsgToCache,                      //表示有需要广播的信号
output reg allowReadToCache,                   //表示其他cache是否需要等待本cache完成访存才能读取特定地址数据
output reg[`ADDRWIDTH-1:0] allowReadToCacheAddr, //本cache正在访存因而其他cache不能读取的内存地址
output reg[`ADDRWIDTH-1:0] addrToCache,      //bus广播中的内存地址
output reg rmToCache,                          //广播read miss信号
output reg wmToCache,                          //广播write miss信号
output reg invToCache,                         //广播invalidate信号
```

# Cache——状态

```
`define ERROR      4'h0 //表示cacheline遇到了硬件错误.
`define MODIFIED   4'h1 //表示cacheline为MODIFIED.
`define M_SRM_WB   4'h2 //表示cacheline为MODIFIED, 并且监听到了总线上的readmiss 信号, 正在执行写回操作.
`define M_SWM_WB   4'h3 //表示cacheline为MODIFIED, 并且监听到了总线上的writemiss信号, 正在执行写回操作.
`define M_WM_WB    4'h4 //表示cacheline为MODIFIED, 并且遇到了CPU writemiss, 正在执行写回操作.
`define M_RM_WB    4'h5 //表示cacheline为MODIFIED, 并且遇到了CPU readmiss,  正在执行写回操作.
`define M_WM_RD    4'h6 //表示cacheline为MODIFIED, 并且遇到了CPU writemiss, 已经执行完写回操作, 正在执行读
`define M_RM_RD    4'h7 //表示cacheline为MODIFIED, 并且遇到了CPU readmiss,  已经执行完写回操作, 正在执行读
`define SHARED     4'h8 //表示cacheline为SHARED.
`define S_RM_RD    4'h9 //表示cacheline为SHARED, 并且遇到了CPU readmiss, 正在执行读内存操作.
`define S_WM_RD    4'ha //表示cacheline为SHARED, 并且遇到了CPU writemiss, 正在执行读内存操作.
`define INVALID    4'hb //表示cacheline为INVALID.
`define I_RM_RD    4'hc //表示cacheline为INVALID, 并且遇到了read miss,正在执行读内存操作.
`define I_WM_RD    4'hd //表示cacheline为INVALID, 并且遇到了writemiss, 正在执行读内存操作.
```

# Cache——MODIFIED

如果检测到总线上的readmiss或者writemiss信号, 则执行写回操作, 并将状态置为 M_SRM_WB 或 M_SWM_WB. 否则, 检查CPU动作,

如果readhit/writehit, 则状态仍为 MODIFIED.

如果发现readmiss/writemiss, 则执行写回操作, 发送总线广播信号, 并将状态置为 M_RM_WB或 M_WM_WB . 表示正在执行写回操作, 写回完成之后需要进行读取.

# Cache——MODIFIED, 中间状态

M_SRM_WB: 判断写回是否已经完成, 如果已经完成, 则状态转为SHARED.

M_SWM_WB: 判断写回是否完成, 如果已经完成, 状态转为INVALID.

M_WM_WB/M_RM_WB: 判断写回是否完成, 如果完成, 状态转为M_WM_RD/M_RM_RD

M_WM_RD/M_RM_RD: 判断读取是否完成, 如果已经完成, 则读取数据线上数据, 状态变为MODIFIED/SHARED

# Cache——SHARED

当检测到总线readmiss信号时, 不必响应,

如果检测到总线writemiss或者invalidate信号, 将相应地址的cacheline状态置为INVALID.

如果检测到CPU的readmiss或者writemiss, 此时需要读内存, 先征得其他cache的同意, 然后将状态置为 S_RM_RD 或者S_WM_RD. 并向memory发送访存请求.

# Cache——SHARED, 中间状态

对于总线请求, 没有中间状态, 因为只需要

对于 S_RM_RD或者S_WM_RD`状态, 需要查询memEn判断是否访存完毕, 如果完毕, 则将状态置为SHARED或者MODIFIED

# Cache——INVALID

对于 INVALID状态, 不必响应总线请求, 所有的CPU请求均为miss.

对于readmiss,总线广播, 如果其他cache允许对该地址进行读取, 则将状态置为 I_RM_RD. 对于writemiss, 动作相同, 不过状态要置为 I_WM_RD;

对于 I_WM_RD或者I_RM_RD, 查询memEn判断访存是否成功, 如果成功, 读取内存数据, 进行修改(对于writemiss), 并将状态置为 MODIFIED或者 `SHARED`.

Introduction

Code memory

Processor

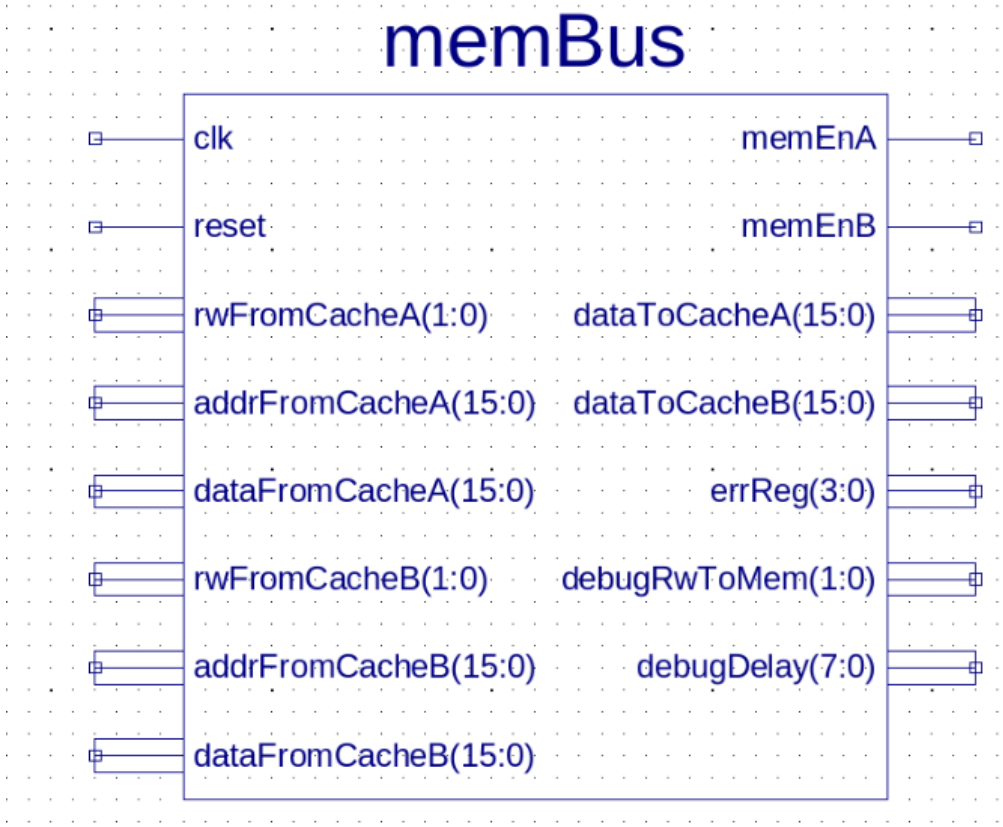
Cache

Memory-Bus

Testbench

TODO

# Memory


memBus
clk
reset
rwFromCacheA(1:0)
addrFromCacheA(15:0)
dataFromCacheA(15:0)
rwFromCacheB(1:0)
addrFromCacheB(15:0)
dataFromCacheB(15:0)
memEnA
memEnB
dataToCacheA(15:0)
dataToCacheB(15:0)
errReg(3:0)
debugRwToMem(1:0)
debugDelay(7:0)

# Memory——输入输出

```
input clk,
input reset,

//interact with cache A
input wire[`IOSTATEWIDTH-1:0] rwFromCacheA,
input wire[`ADDRWIDTH-1:0]    addrFromCacheA,
input wire[`WORDWIDTH-1:0]    dataFromCacheA,
output reg[`ADDRWIDTH-1:0]    dataToCacheA,
output reg memEnA,

//interact with cache B
input wire[`IOSTATEWIDTH-1:0] rwFromCacheB,
input wire[`ADDRWIDTH-1:0]    addrFromCacheB,
input wire[`WORDWIDTH-1:0]    dataFromCacheB,
output reg[`ADDRWIDTH-1:0]    dataToCacheB,
output reg memEnB,
output reg[`ERRWIDTH-1:0] errReg,

//debug output
output[`IOSTATEWIDTH-1:0] debugRwToMem,
output[7:0] debugDelay
```

# Memory——行为模型

通过内部寄存器chipSelect表示当前占据内存的cache, 为了防止某个cache一直占据内存, 设置了一个prefer寄存器, 当cacheA释放对内存的占有后, 将prefer置为CB, 则下一个cycle, 若A和B同时发来访存请求, 优先查询B的访存请求. prefer只对同时发来的访存请求有效, 如果A先发来请求而B后发, 即使prefer为B, 仍然会先响应A的请求.

# Memory——行为模型

在memBus内部, 有内部寄存器rwToMem/addrToMem/dataToMem, 表示占据内存的cache的读写请求, 当rwToMem为idel时, 表示当前没有cache占据内存. 每个时钟上升沿, 查询rwToMem状态, 如果为idel, 则对两个cache, 按照prefer建议的查询顺序进行查询, 当查询到某个cache有访存请求时, 开始执行该cache的访存请求. 此时延迟计数器开始减一计数.

当延迟计数器为0时, 完成访存, 将memEn置一, 并将延迟计数器重新置位.

Introduction

Code memory

Processor

Cache

Memory-Bus

Testbench

# Test Case1——WAW

```
clk            = 1'b0;
reset          = 1'b0;
code1.codeSize = 3;
code2.codeSize = 3;
code1.codes[0] = {`SET, `R0,    `WORDWIDTH'd3}; //p1.r0 = 3
code1.codes[1] = {`ST,  `R0,    `ADDRWIDTH'd0};  //mem[0] = p1.r0, write miss
code1.codes[2] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[0] = {`SET, `R0,    `WORDWIDTH'd4}; //p1.r0 = 3
code2.codes[1] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[2] = {`ST,  `R0,    `ADDRWIDTH'd0};  //mem[0] = p1.r0, write miss
#5 reset  = 1'b1;
#17 reset = 1'b0;
#1000;
$display("Write After Write");
$display("time:%d,P1.r0:%d,C1.cachLine:%d,C1.state:%h", $time,P1.regFile[0],C1.cacheLine, C1.state);
$display("time:%d,P2.r0:%d,C2.cachLine:%d,C2.state:%h", $time,P2.regFile[0],C2.cacheLine, C2.state);
$display("mem[0]:%d", mb.mem[0]);
$display("_____________________________________________________");
```

# TestCase1——Instructions

PROCESSOR1

```
set r0 3; // r0 = 3
store mem[0] r0;
nop;
```

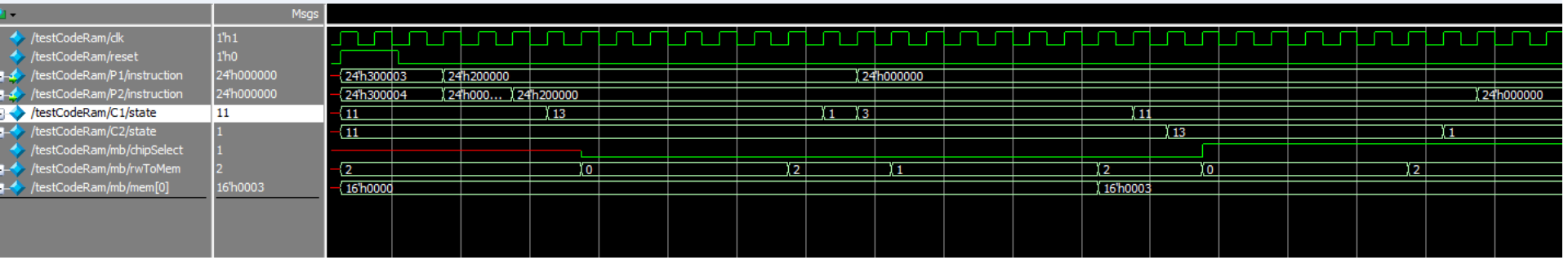
PROCESSOR2

```
set r0 4;
nop;
store mem[0] r0;
```

# TestCase1——结果

```
Write After Write
P1.r0:      3, C1.cachLine:     3,C1.state:b
P2.r0:      4, C2.cachLine:     4,C2.state:1
mem[0]:     3
```

# TestCase1——波形




可以看到, Processor1首先执行 store操作,
而processor2发生write miss后就等待processor1完成.
cache1的状态经历了 INVALID->I_WM_RD->MODIFIED->M_SWM_WB->INVALID,

而cache2在 INVALID状态发生writemiss后就一直等待cache1的写回完成信息,
当写回完成后, 状态变为 I_WM_RD进行读内存, 读入后状态变为 MODIFIED.

# TestCase2——RAW

```
code1.codeSize = 3;
code2.codeSize = 3;
code1.codes[0] = {`SET, `R0,    `WORDWIDTH'd3}; //p1.r0 = 3
code1.codes[1] = {`ST,  `R0,    `ADDRWIDTH'd0};  //mem[0] = p1.r0, write miss
code1.codes[2] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[0] = {`SET, `R0,    `WORDWIDTH'd4}; //p1.r0 = 3
code2.codes[1] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[2] = {`LD,  `R0,    `ADDRWIDTH'd0};  //mem[0] = p1.r0, write miss
reset          = 1'b1;
#20 reset      = 1'b0;
#1000;
$display("Read After Write");
$display("time:%d,P1.r0:%d,C1.cachLine:%d,C1.state:%h",$time,P1.regFile[0],C1.cacheLine, C1.state);
$display("time:%d,P2.r0:%d,C2.cachLine:%d,C2.state:%h",$time,P2.regFile[0],C2.cacheLine, C2.state);
$display("mem[0]:%d", mb.mem[0]);
$display("_____________________________________________________");
```

# TestCase2——Instructions

PROCESSOR1

```
set r0 3

store mem[0] r0

nop
```

PROCESSOR2

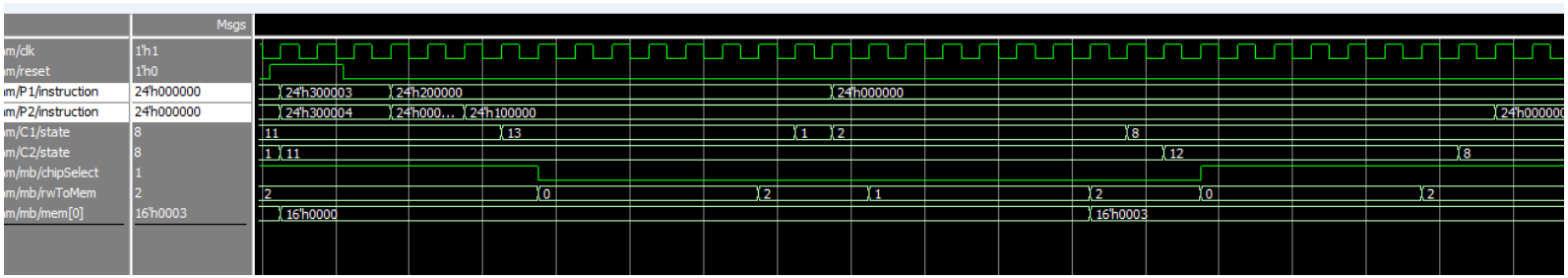
```
set r0 4

nop

load r0 mem[0]
```

# TestCase2——结果

```
Read After Write
time:                2042,P1.r0:    3,C1.cachLine:    3,C1.state:8
time:                2042,P2.r0:    3,C2.cachLine:    3,C2.state:8
mem[0]:    3
```

由结果可知, processor2的load指令执行完成后,
memory,register,cachline相关数据均为3,
且cacheline状态为 SHARED.

# TestCase2——波形




由波形图, cache1的状态变换为:
INVALID, I_WM_RD, MODIFIED, M_SRM_WB, SHARED

而cache2在发生readmiss后, 就一直等待cache1的allowRead信号,
当cache1完成写回操作之后, 开始读内存, 状态变为 SHARED.

# TestCase3——Special RAW

```
code1.codeSize = 4;
code2.codeSize = 5;
code1.codes[0] = {`SET, `R0,    `WORDWIDTH'd3}; //p1.r0 = 3
code1.codes[1] = {`ST,  `R0,    `ADDRWIDTH'd0};  //mem[0] = p1.r0, write miss
code1.codes[2] = {`NOP, `R0,    `WORDWIDTH'd0};
code1.codes[3] = {`LD,  `R0,    `WORDWIDTH'd0};
code2.codes[0] = {`SET, `R0,    `WORDWIDTH'd4}; //p1.r0 = 3
code2.codes[1] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[2] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[3] = {`NOP, `R0,    `WORDWIDTH'd0};
code2.codes[4] = {`ST,  `R0,    `ADDRWIDTH'd0};
reset          = 1'b1;
#20 reset      = 1'b0;
#1500;
$display("Special Read After Write");
$display("time:%d,P1.r0:%d,C1.cachLine:%d,C1.state:%h",$time,P1.regFile[0],C1.cacheLine, C1.state);
$display("time:%d,P2.r0:%d,C2.cachLine:%d,C2.state:%h",$time,P2.regFile[0],C2.cacheLine, C2.state);
$display("mem[0]:%d", mb.mem[0]);
$display("_____________________________________________________");
```

# TestCase3——Instructions

PROCESSOR1

```
set r0 3

store mem[0] r0

nop

load r0 mem[0]
```

PROCESSOR2

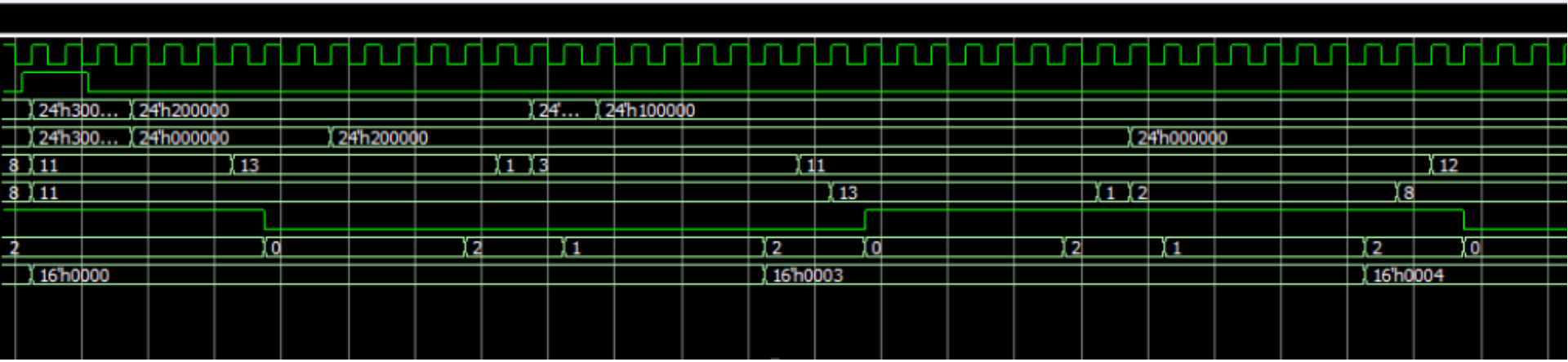
```
set r0 4;

nop;

nop;

nop;

store mem[0] r0;
```

是否需要进行同步?

# TestCase3——结果

```
Special Read After Write
time:                8082,P1.r0:    4,C1.cachLine:    4,C1.state:8
time:                8082,P2.r0:    4,C2.cachLine:    4,C2.state:8
mem[0]:    4
```

# TestCase3——波形




从波形图上可以看出, processor1的 `load`操作是在 Processor2的`store`操作之后完成的.

processor1的状态转换为 `INVALID`-> `I_WM_RD`->`MODIFIED`->`M_SWM_WB`->`INVALID`->`I_RM_RD`->`SHARED`, 说明它首先经历了一个 `INVALID`遭遇write miss而变成 `MODIFIED`的状态转换, 随后立即监听到另一个cache广播的write miss信号,

而看processor2的状态转换图, 初始为`INVALID`状态, 遭遇writemiss后等待cache1写回, 等待写回完成后,读取内存并修改, 状态称为`MODIFIED`, 随后立即检测到cache1广播的readmiss状态, 将数据写回, 并状态变为`SHARED`.