# Cache设计指导

王轩

#### Cache回顾

Cache结构在课本《计算机体系结构：量化研究方法》（中文第五版）的附录B：存储器层次结构回顾 中有所描述。本实验全部采用“写回+写入分派”的cache策略，这种策略在读或写命中时，直接从cache中读写数据，只需要一个时钟周期，不需要对CPU流水线进行stall；在发生缺失时，读缺失和写缺失的处理方法是相同的，都是从主存中换入缺失的line（line即块）到cache中（当然，如果要换入的line已经被使用了，并且脏，则需要在换入之前进行换出），再从cache中读写数据。总结下来，cache应该维护如下的状态机：

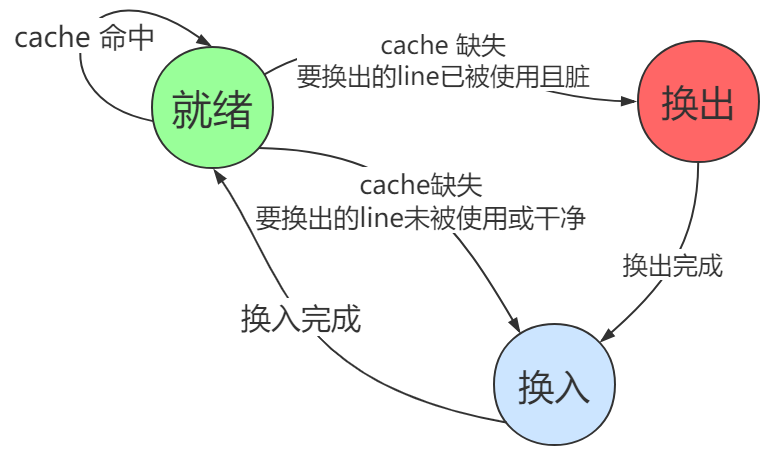
****

图1：cache状态机

我们提供的简单直接映射的cache中就有这样的状态机。当没有读/写请求时，cache保持就绪状态，当CPU发出读/写请求时，cache检查是否命中，如果命中则立刻响应读/写请求，并仍保持就绪状态。如果缺失，则进行换入（换入之前可能需要先换出），在cache进行换出换入时，cache无法响应CPU当前的读写请求，因此需要向CPU发出miss=1的信号，CPU需要使用该信号控制所有流水段进行stall。直到cache完成换出换入后重回就绪状态，此时cache就能响应这个读写请求。

在今后我们发布的CPU+cache的完整代码中，加入cache miss时对cpu流水线的stall控制，以及对cache miss率的统计。

#### Cache对外接口与时序

现在，让我们暂时把cache当作黑箱，看看它对外的接口和时序是怎样的。我们提供的简单cache的对外接口如下。当你对cache进行修改时，也要遵循这个接口和时序，否则会在连接CPU的时候遇到困难。

|  |
| --- |
| module cache #(  parameter LINE\_ADDR\_LEN = 3, //line内地址的长度，决定了每个line具有2^3=8个word  parameter SET\_ADDR\_LEN = 2, //组地址的长度，决定了一共有2^2=4个组  parameter TAG\_ADDR\_LEN = 7, //tag的长度  //parameter WAY\_CNT = 4 //组相连度，决定了每组中有多少路line，对于直接映射cache，  //该参数用不到，但组相连cache中需要大家用到这个参数  )(  input clk, rst,  output miss, // 对CPU发出的miss信号  input [31:0] addr, // 读/写请求的地址  input rd\_req, // 读请求信号  output reg [31:0] rd\_data, // 读出的数据，一次读一个word  input wr\_req, // 写请求信号  input [31:0] wr\_data // 要写入的数据，一次写一个word  ); |

当读/写命中时，时序与以往我们提供的dataRam完全一样，如图：

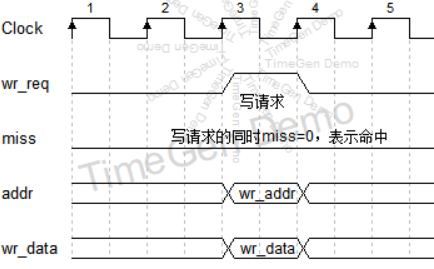
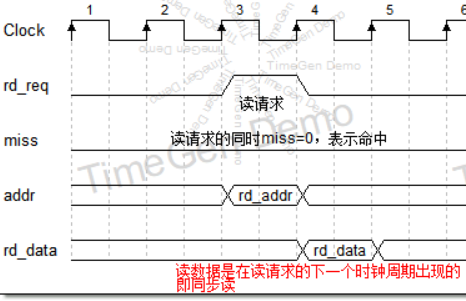
 

图2：写命中时序（左），读命中时序（右）

当读/写缺失时，随着请求信号的出现，miss信号同样变为1，请求信号要一直保持1，直到一个周期，miss变为0，请求信号仍为1，就完成了一次读/写。另外，在请求信号保持1的过程中，addr和wr\_data也要保持。

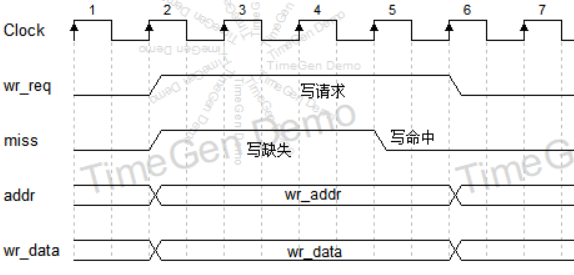


图3：写缺失时序

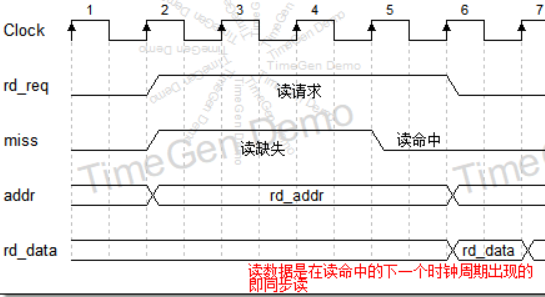


图4：读缺失时序

rd\_req与miss，wr\_req与miss，实际上构成了两对握手信号，这种握手信号时序广泛的应用于总线技术中。

注意：在以上的时序图中，缺失只持续了3个时钟周期，这只是为了方便演示。在本实验中，由于主存需要50个周期进行一次读/写，所以cache缺失会持续50多个时钟周期或100多个时钟周期。当只进行换入时，缺失持续50多个时钟周期。当先换出后换入时，缺失持续100多个时钟周期。

#### 主存对外接口与时序

主存代码由我们提供，它被我们提供的简单cache所调用，是使用BRAM模仿的DDR。包括main\_mem.sv与mem.sv两个文件，顶层文件是main\_mem.sv，它以line为读写单元，（而不是以word为读写单元），且读写周期很长，本实验设置为50个时钟周期。由于不需要学生对main\_mem做任何修改，因此也不需要读懂它的内部实现，只需要把它当作黑箱，了解其时序。

main\_mem的输入输出接口定义如下：

|  |
| --- |
| module main\_mem #( // 主存，每次读写以line 为单位，并会延时固定的50个周期  parameter LINE\_ADDR\_LEN = 3, // line内地址的长度，决定了每个line具有2^3=8个word  parameter ADDR\_LEN = 8 // 主存一共有2^8=256个line  )(  input clk, rst,  output gnt, // 读写响应信号  input [ADDR\_LEN-1:0] addr, // 读写地址  input rd\_req, // 读请求信号  output reg [31:0] rd\_line [1<<LINE\_ADDR\_LEN], // 读出的line数据，这是一个二维数组，即8个word = 8\*32bit  input wr\_req, // 写请求信号  input [31:0] wr\_line [1<<LINE\_ADDR\_LEN] // 要写入的line数据，这是一个二维数组，是8个word = 8\*32bit  ); |

main\_mem的读写时序与之前介绍的cache的读写缺失时序非常相似，也就是说，主存可以看作一个永远都会缺失，并且一缺失就缺失50个周期的cache。不同的是cache的miss信号和main\_mem的gnt信号的逻辑相反：cache的miss=0时代表命中；而main\_mem的gnt=1时代表命中。如图：

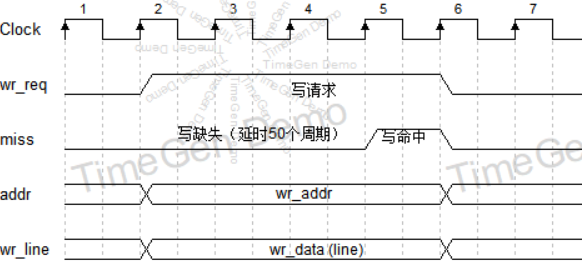


图5：主存写时序

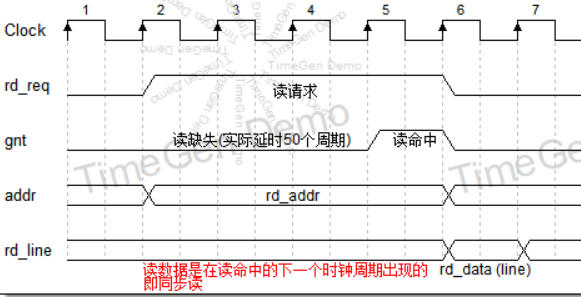


图6：主存读时序

#### 直接映射cache的实现

本节的内容务必结合我们提供的cache代码去阅读。要理解cache首先要看32bit addr是如何分割的，如下图：

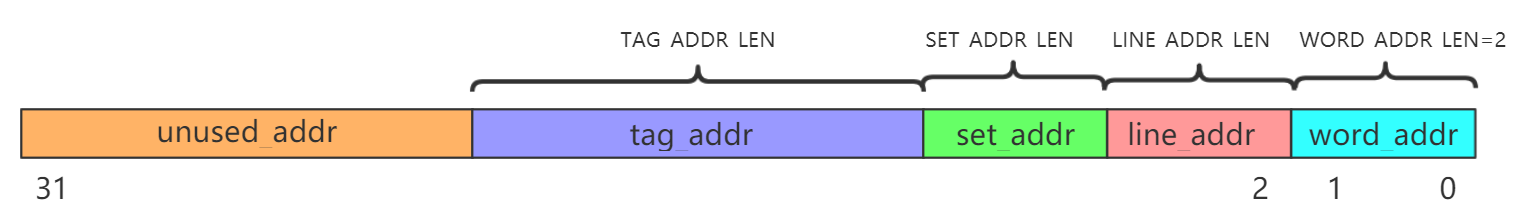


图7：32bit 地址的分割

word\_addr[1:0] : 字节地址，即指定字节是word(字）中的第几个。固定为2bit。在做cache实验是，为了代码简便，方便大家抓住重点，不要求处理独热码，因此word\_addr不需要处理。同时我们提供的cache相关的汇编代码中不出现半子和字节的读写指令，只使用lw和sw指令做内存读写。

line\_addr: line内地址，其长度由参数LINE\_ADDR\_LEN决定。例如，如果希望每个line中有16个word，则LINE\_ADDR\_LEN应设为4，因为2^4=16。在cache读写过程中，line\_addr用于指示要读写的word是line中的哪一个word。

set\_addr: line地址，其长度由参数SET\_ADDR\_LEN决定。例如，如果希望cache中有4个cache组，则SET\_ADDR\_LEN应该设置为2，因为2^2=4。在cache读写过程中，set\_addr负责将读写请求路由到正确的组。

tag\_addr: 是该32位地址的TAG。当发生读写请求时，cache应该把32位地址中的tag\_addr取出，与cache中的TAG比较，如果相等则命中。如果不等则缺失。

unused\_addr: 32位地址中的高位，直接丢弃。

在我们提供的代码中，使用一句assign完成32bit地址的分割：

|  |
| --- |
| assign {unused\_addr, tag\_addr, set\_addr, line\_addr, word\_addr} = addr; |

在我们提供的简单cache中，line size可以通过调节LINE\_ADDR\_LEN去改变，组数可以通过调节SET\_ADDR\_LEN去改变。这里，以LINE\_ADDR\_LEN=3，SET\_ADDR\_LEN=2, TAG\_ADDR=12为例，给出直接相连cache的结构图：

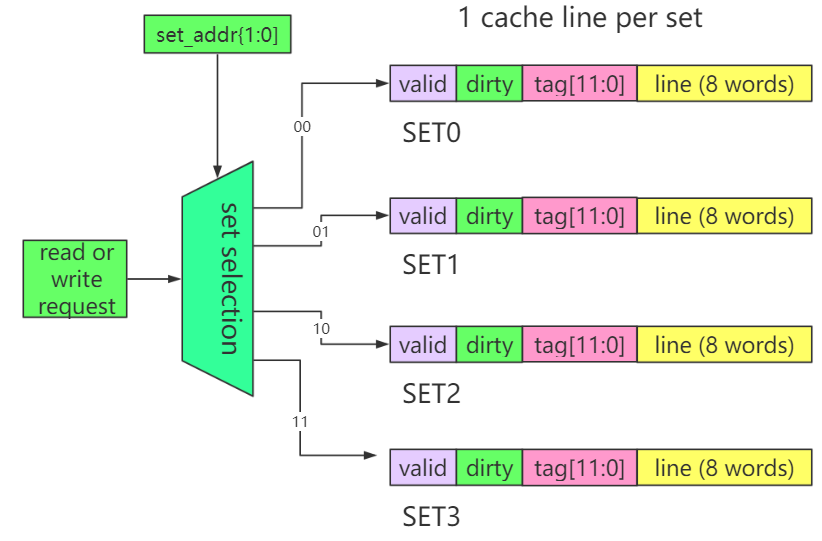


图8：直接相连cache的结构

实际上，直接相连是组相连的特殊情况，相当于1路组相连，因此每个set中只有1个line。每个 line是8个word，除此之外，每个line还需要1个TAG，一个dirty（脏位），一个valid（有效位）。这些在systemverilog代码里如下：

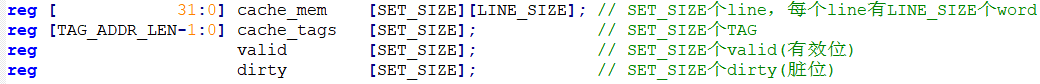


图9：直接相连cache中的一些变量

当有读写请求时，根据地址中的set\_addr字段，决定要到哪个line中读写数据。然后，查看该line是否valid，如果valid=0则一定是缺失，如果valid=1，说明这个line是有效的，需要比较这个line的tag和地址中的tag是否相同，相同则命中，不同则缺失。如果命中，则立即响应读写请求。当然，如果是写请求，要把dirty置1。

如果cache缺失，要从主存中换入该块到这个cache line中。在换入前，也需要考虑是否需要先换出。如果valid=1且dirty=1，说明该cache块是有效的并且已经被修改过，则需要先进行换出。此时需要控制cache状态机的状态转移。cache状态机的状态如下，请结合图1理解这个状态机：

|  |
| --- |
| enum {IDLE, SWAP\_OUT, SWAP\_IN, SWAP\_IN\_OK} cache\_stat = IDLE; |

相比图1，这里多出一个状态SWAP\_IN\_OK，该状态一定出现在SWAP\_IN状态之后，只占用一个时钟周期，负责把主存中读出的数据写入cache line。

#### 组相连cache的实现

Cache实验的主要任务是理解我们提供的直接相连cache的代码，并修改成WAY\_CNT路组相连的cache代码（强烈建议WAY\_CNT作为参数可调，这是为了方便在写cache实验报告时快速的修改组相连度进行实验）。

下图是组相连cache的结构图。取LINE\_ADDR\_LEN=3，SET\_ADDR\_LEN=2, TAG\_ADDR=12，WAY\_CNT=4为例。

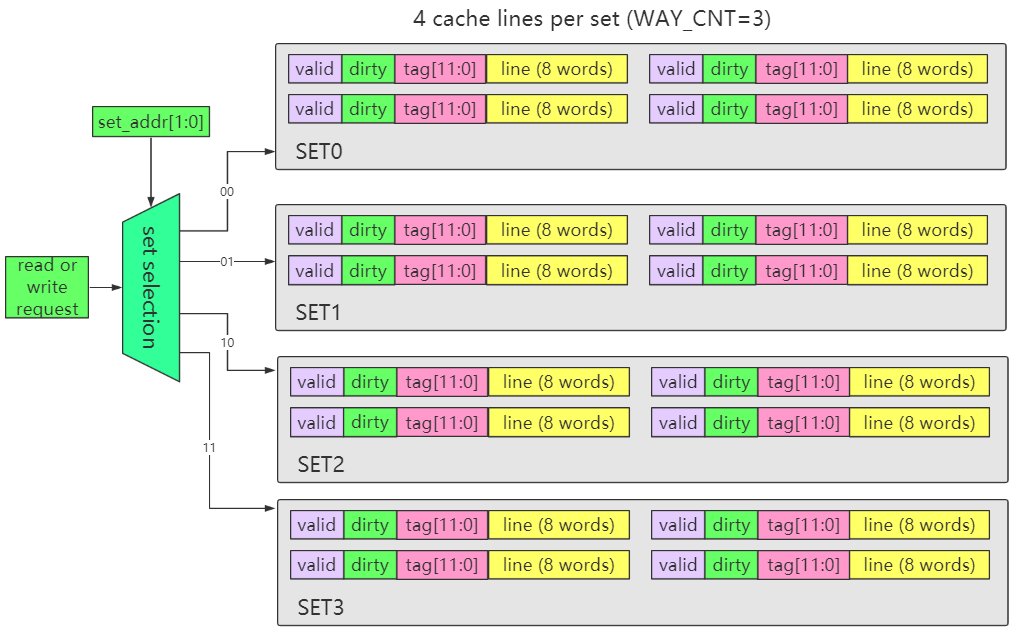


图10：4路组相连cache （WAY\_CNT=4）

相比直接相连cache，组相连cache需要加入的有：

1. 将图9中所示的数组添加一个维度，该维度的大小为 WAY\_CNT
2. 实现并行命中判断：为了判断是否命中，直接相连cache每次只需要判断一个valid，一个dirty，一个TAG是否命中，但组相连cache则需要在组内并行的判断每路line是否命中
3. 实现替换策略：当cache需要换出时，直接相连cache没有选择，因为每个组中只有1个line，只能换出换入这唯一的line。但组相连cache需要决策换出哪个line。本实验要求实现FIFO换出策略与LRU换出策略（请见《计算机体系结构：量化研究方法》（中文第五版）附录B）。为了实现FIFO策略和LRU策略，还需要加入一些辅助的wire和reg变量。

#### Cache testbench的生成和使用

我们提供的python脚本(generate\_cache\_tb.py)用于生成针对cache进行正确性测试的testbench。并且提供一个已经生成好的testbench (cache\_tb.sv)

请建立Vivado工程，将 cache.sv、main\_mem.sv、mem.sv添加进Vivado工程的Design Sources，将cache\_tb.sv添加进Vivado工程的Simulation Sources。添加后vivado工程应该呈现如下的层次结构。

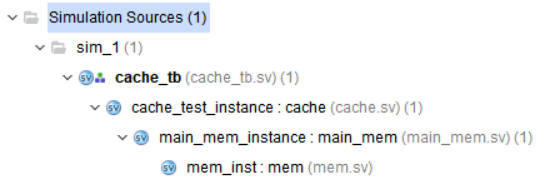


图11：使用Vivado添加cache相关文件后呈现的层次结构

添加后，点击Run Simulation->Run Behavioral Simulation 进行行为仿真，然后可以看到如下的波形：

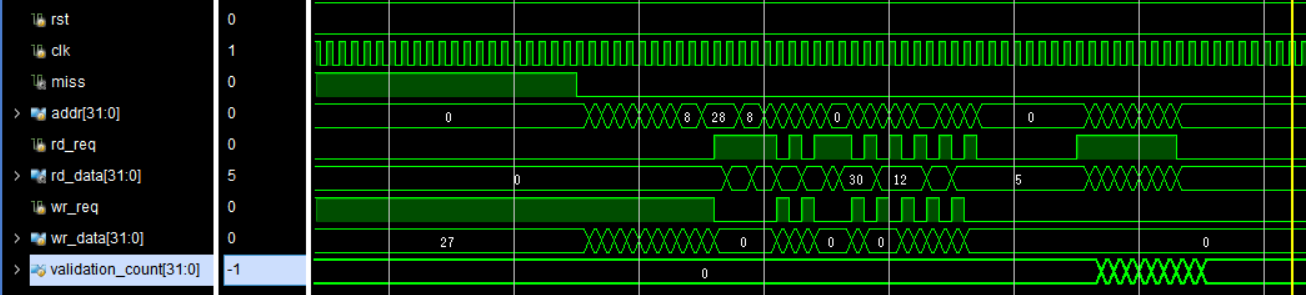


图12：cache读写测试波形

该testbench对cache进行N次顺序写入，再进行3N次随机读写，最后进行N次顺序读出，并验证读出的数据是否符合预期，每当读取的数据符合预期，validation\_count这个变量就+1，直到完成所有读出数据的验证后，validation\_count变成-1，即0xffffffff。说明cache读写验证通过。

Generate\_cache\_tb.py 能够生成不同N值的testbench，N值代表读写测试的规模。运行方法是在命令行中运行命令，附带参数N（需要安装python2或python3）。图13展示了如何生成读写规模N=16的testbench代码：



图：使用工具

你可以使用管道命令，将打印结果写入.sv文件中：

|  |
| --- |
| python .\generate\_cache\_tb.py 16 > cache\_tb.sv |

#### 关于SystemVerilog

注意到我们提供的cache代码实际上是SystemVerilog，以.sv为文件后缀。SystemVerilog与Verilog兼容性极强，模块之间可以互相调用（类似于C和C++的关系）。这里我们使用SystemVerilog是因为它更能方便的操作多维数组，cache实验中很多地方使用多维数组非常方便。不需要学生去系统的学习SystemVerilog语法，只需要了解它的少量feature即可。

#### 关于Vivado的综合(Synthesis)

因为cache实验最终要求学生对不同参数的cache进行资源消耗评估，所以必须学会使用使用Vivado将systemverilog代码综合成电路，并查看综合报告。

首先，为了提升综合速度，我们仅仅将cache模块作为顶层进行综合。我们要修改cache.sv中的各个cache参数为你想要综合的参数。然后在Vivado中，设置cache.sv为顶层文件，然后点击Run Synthesis进行综合，如图14。

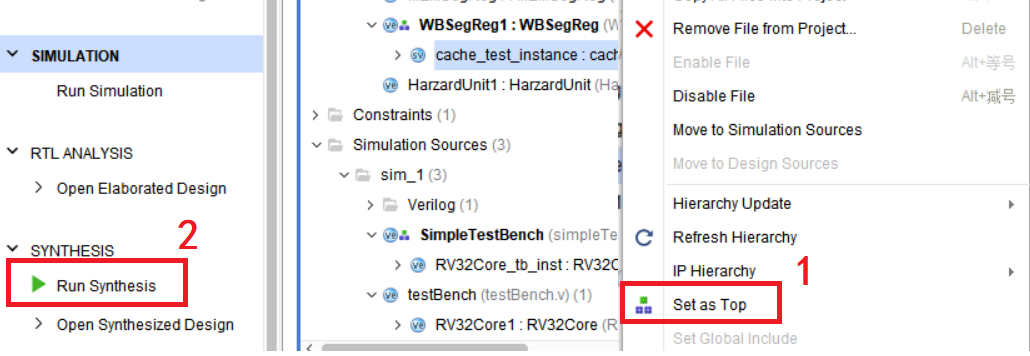


图14：设置cache.sv为顶层，并开始综合

综合大概需要几十秒到几分钟，完成后，点击Open Synthesized Design，在Project Summary中下拉，可以看到一个Utilization框，即资源占用报告。点击Table可以看到这些资源占用的绝对数值。如图15.

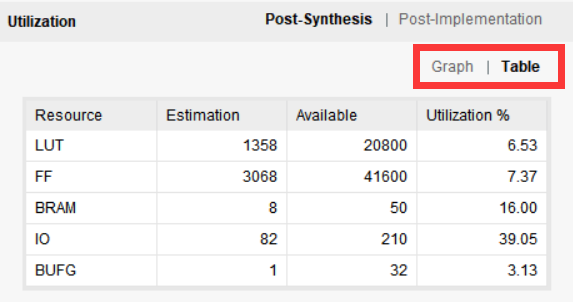


图15：资源占用报告

解读：

LUT、FF：是我们最在意的资源，因为cache的逻辑均使用LUT和FF实现。这两个参数的使用量就代表了你的cache所占用电路的资源量。

BRAM：主存main\_mem被综合成了BRAM。由于我们不对主存进行修改，所以这一项不需要在意。

IO、BUFG等：这些与FPGA管脚相关，完全不需要在意。