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：

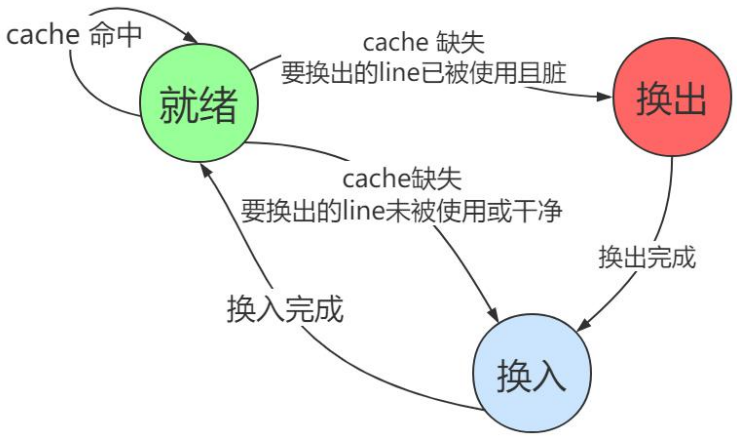


图 1：cache 状态机

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。

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：

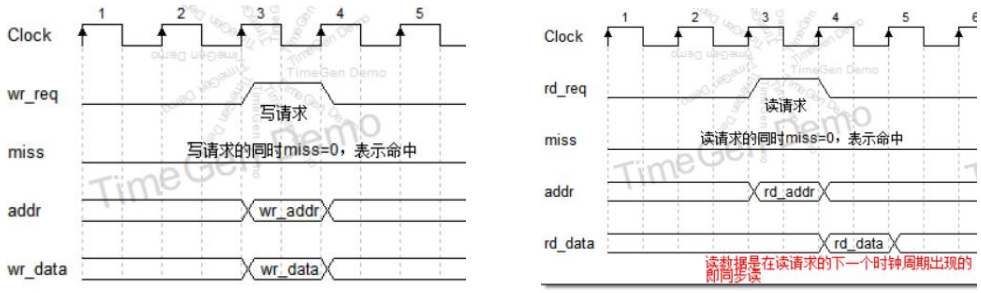


图 2：写命中时序（左），读命中时序（右）

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。

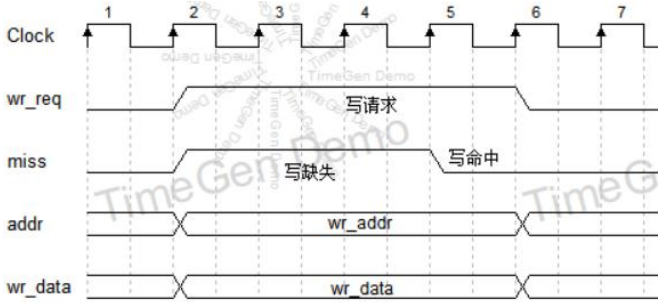


图 3：写缺失时序

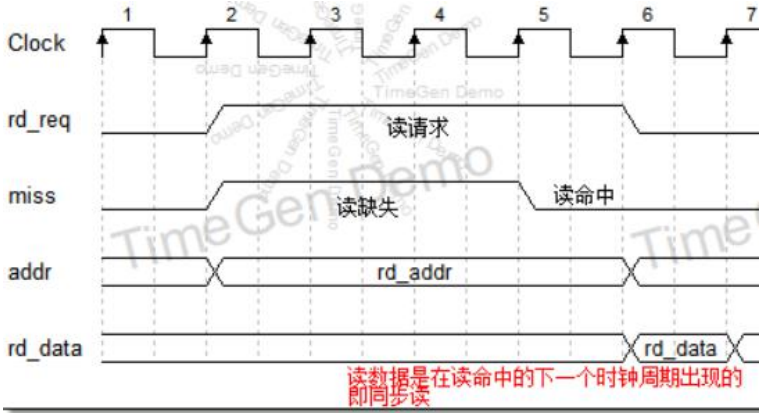


图 4：读缺失时序

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：

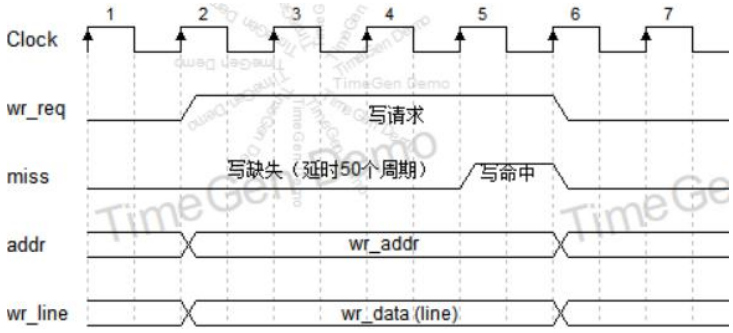


图 5：主存写时序

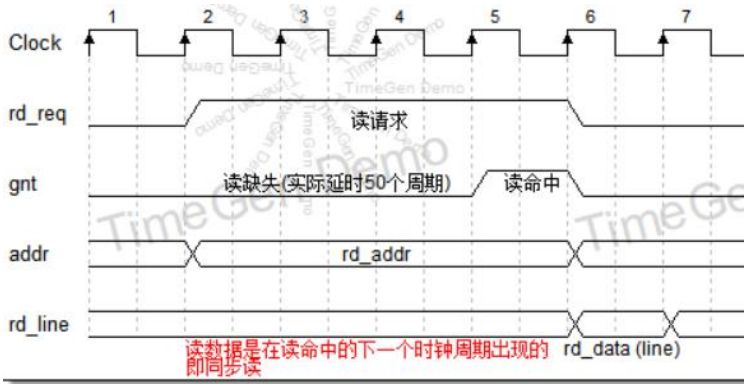


图 6：主存读时序

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：

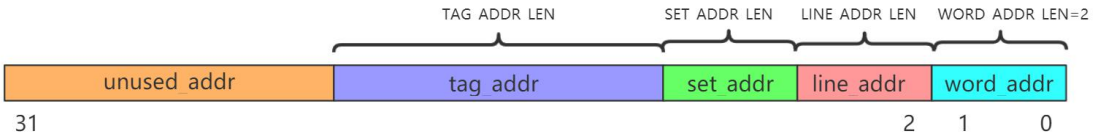


图 7：32bit 地址的分割

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：

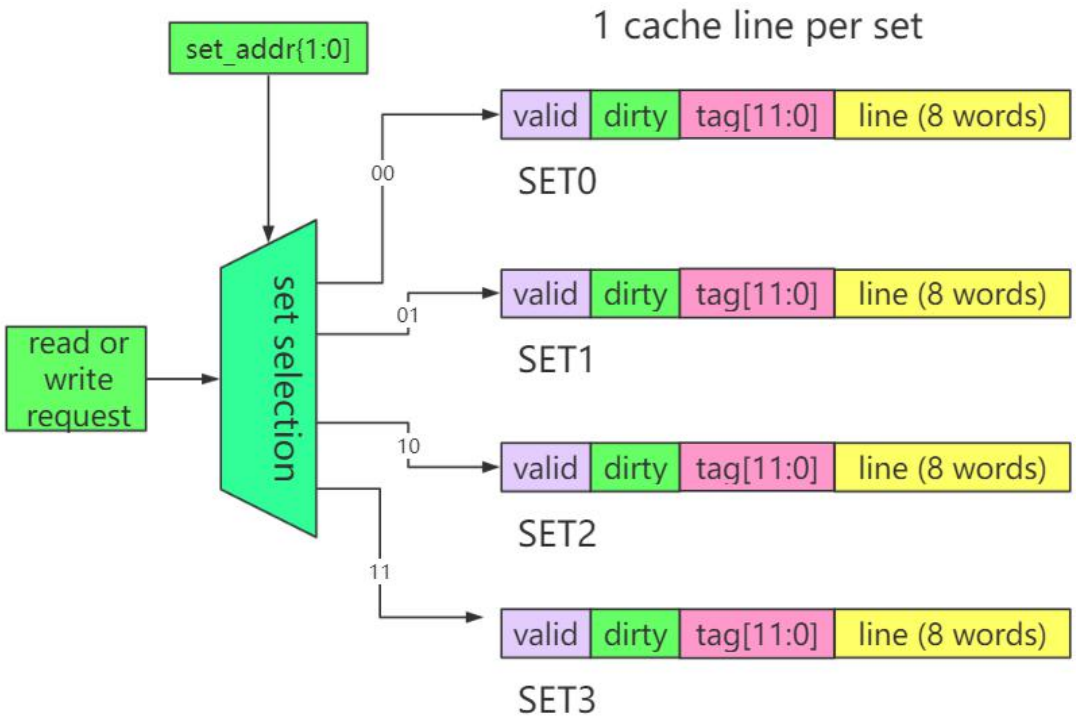


图 8：直接相连 cache 的结构

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：

```
reg [            31:0] cache_mem    [SET_SIZE][LINE_SIZE]; // SET_SIZE个line，每个line有LINE_SIZE个word
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE];            // SET_SIZE个TAG
reg                    valid        [SET_SIZE];            // SET_SIZE个valid(有效位)
reg                    dirty        [SET_SIZE];            // SET_SIZE个dirty(脏位)
```

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。

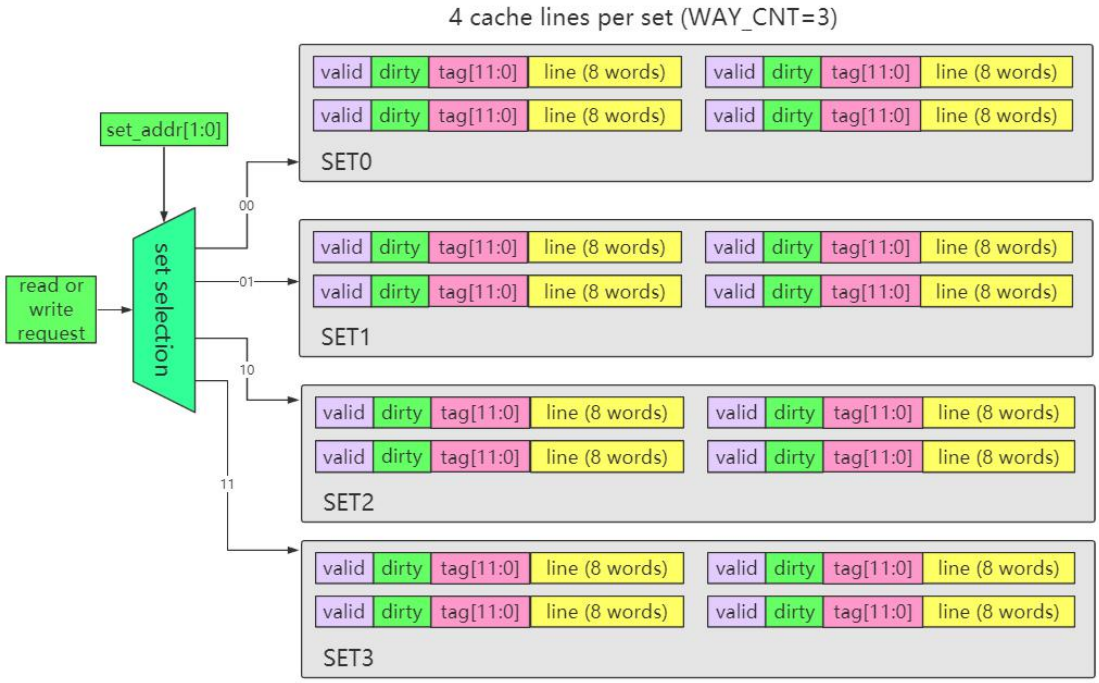


图 10：4 路组相连 cache （WAY_CNT=4）

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。

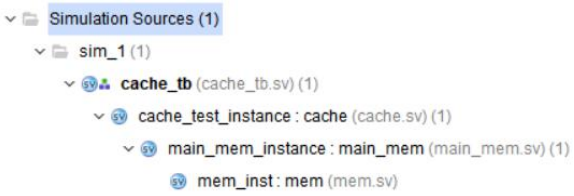


图 11：使用 Vivado 添加 cache 相关文件后呈现的层次结构

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：

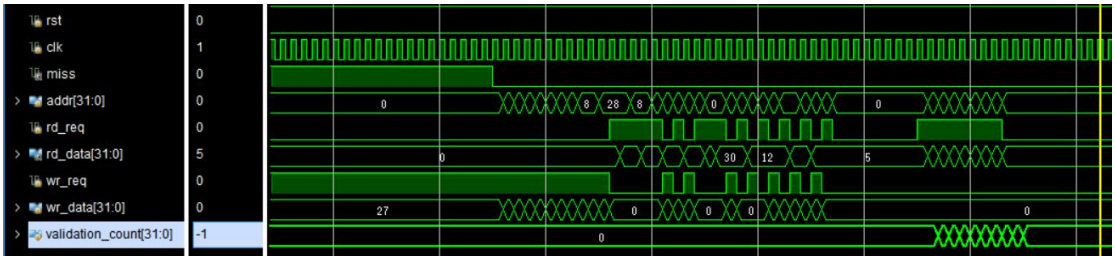
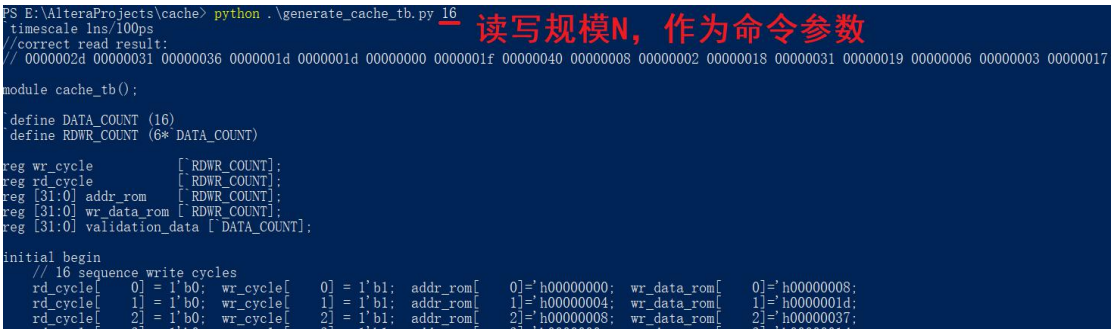


图 12：cache 读写测试波形

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。

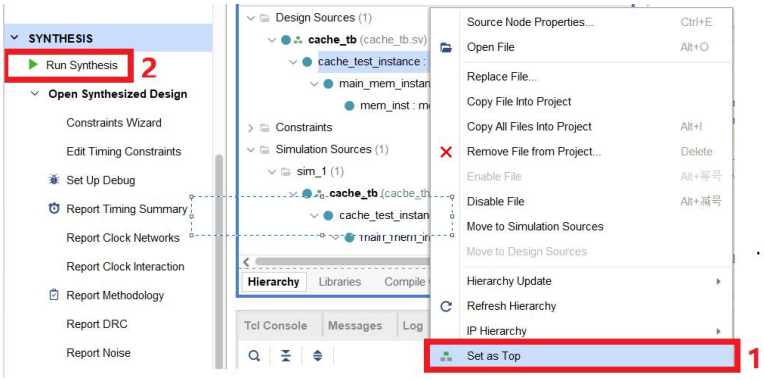


图 14：设置 cache.sv 为顶层，并开始综合

综合大概需要几十秒到几分钟，完成后，点击 Open Synthesized Design，选择 Report Utilization，生成 Utilization 文件后，可以看到一个框，即资源占用报告。点击 Summary 可以看到这些资源占用的绝对数值（以 Vivado 2019.2 为例）。如图 15.

Resource	Utilization	Available	Utilization %
LUT	1140	303600	0.38
FF	3060	607200	0.50
BRAM	4	1030	0.39
IO	81	600	13.50

图 15：资源占用报告

当你需要修改 Cache 的参数（组数、组相连度、line 大小等）时，直接在 cache.sv 中进行修改，如图：

```
module cache #(
    parameter  LINE_ADDR_LEN = 3, // line内地址长度，决定了每个line具有2^3个word
    parameter  SET_ADDR_LEN  = 3, // 组地址长度，决定了一共有2^3=8组
    parameter  TAG_ADDR_LEN  = 6, // tag长度
    parameter  WAY_CNT       = 3  // 组相连度，决定了每组中有多少路line，这里是直接映射型cache，因此该参数没用到
)(
    input  clk, rst,
    output miss,               // 对CPU发出的miss信号
    input  [31:0] addr,        // 读写请求地址
    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
);
```

修改这些参数后，重新进行综合，则综合报告中消耗的资源数量会改变。由此可以看出 cache

规模对资源数量的影响。

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 等：这些与 FPGA 管脚相关，完全不需要在意。

注意：当修改这些参数时，cache 规模会发生变化，主存也会。在进行实验时，为了排除主存大小对资源占用的影响，可能需要固定主存的大小。主存大小是 2^(LINE_ADDR_LEN+SET_ADDR_LEN+TAG_ADDR_LEN) 个字。当你将 SET_ADDR_LEN 或 LINE_ADDR_LEN 改大时，TAG_ADDR_LEN 就要改小，这样就能保证主存的大小不变。