# Cache实验报告

**实验报告：**

* 实验目标：
  + 第一阶段

1.权衡cache size增大带来的命中率提升收益和存储资源电路面积的开销

2.权衡选择合适的组相连度（相连度增大cache size也会增大，但是冲突miss会减低）

3.体会使用复杂电路实现复杂替换策略带来的收益和简单替换策略的优势（有时候简单策略比复杂策略效果不差很多甚至可能更好）

4.理解写回法的优劣

* + 第二阶段：

Cache 与Lab2的四级流水线CPU组合，并对benchmark性能进行测试，进一步验证编写的cache的正确性

* 实验环境和工具：Vivado
* 实验内容和过程（使用我们提供的快速排序和矩阵乘法的benchmark进行实验，鼓励自己编写更多的汇编benchmark进行测试，体会cache size、组相连度、替换策略针对不同程序的优化效果，以及策略改变带来的电路面积的变化。针对不同程序，权衡性能和电路面积给出一个较优的cache参数和策略。其中“性能”参数使用运行仿真时的时钟周期数量进行评估。“资源占用”参数使用vivado或其它综合工具给出的综合报告进行评估。进行这一步时需要用阶段二的结果进行一些实验，不能仅仅进行理论分析，实验报告中需要给出实验结果（例如仿真波形的截图、vivado综合报告等）。）：

**阶段1：**

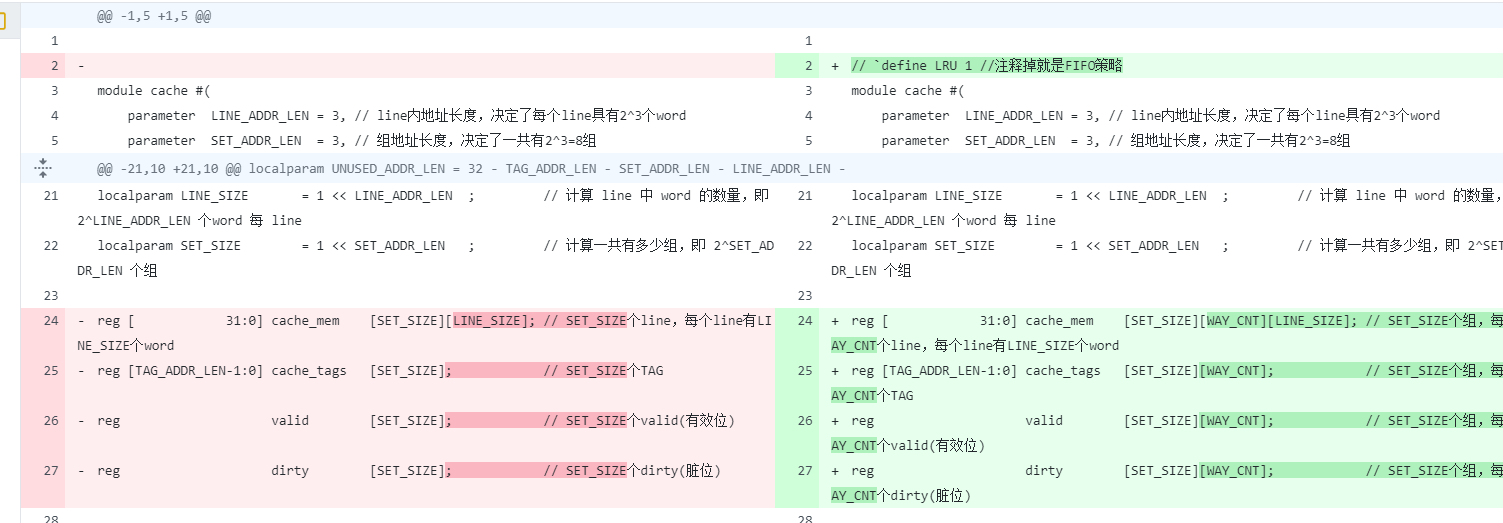
理解提供的直接映射策略的cache，将它修改为N路组相连的cache，并通过我们提供的cache读写测试，FIFO、LRU两种替换策略。

使用提供的python脚本生成一个新的testbench，并对自己的cache进行验证（要求FIFO和LRU策略都要验证，并修改组相连度等参数进行多次验证），在实验报告中解释自己所编写的代码。

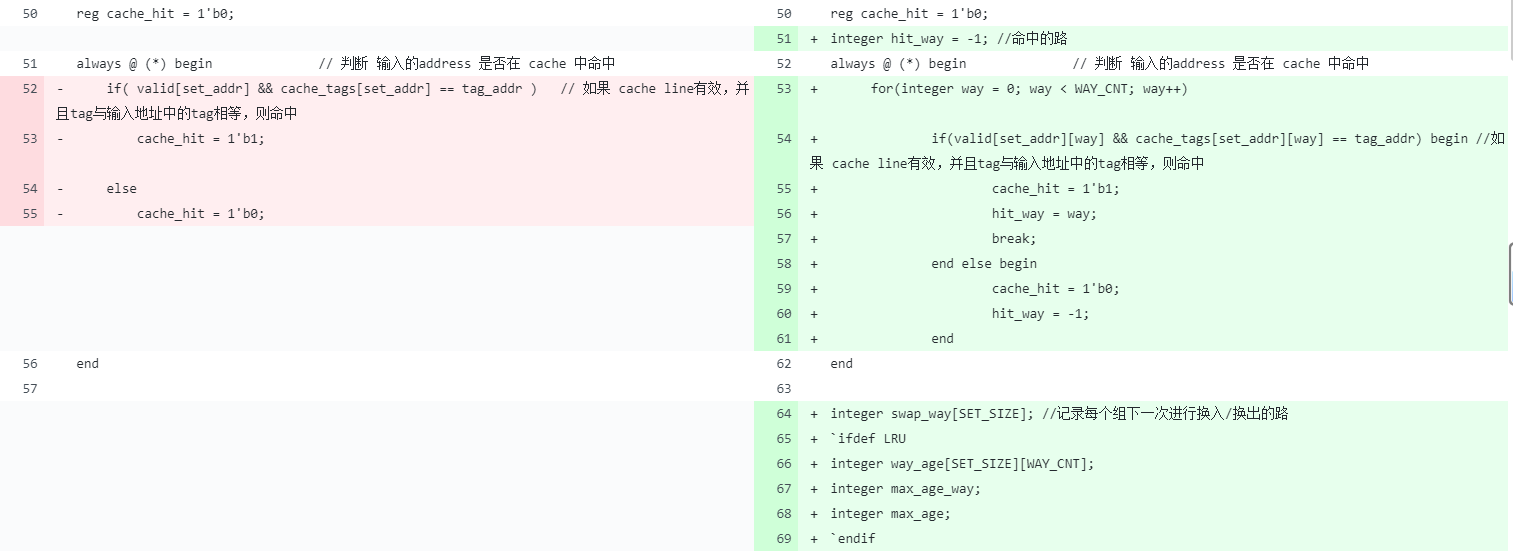
1. Cache实现代码编写

在理解原cache代码的基础上，将其修改为组相联的cache，并实现LRU和FIFO两种调度策略：（为方便起见，使用github的diff工具进行代码修改说明）

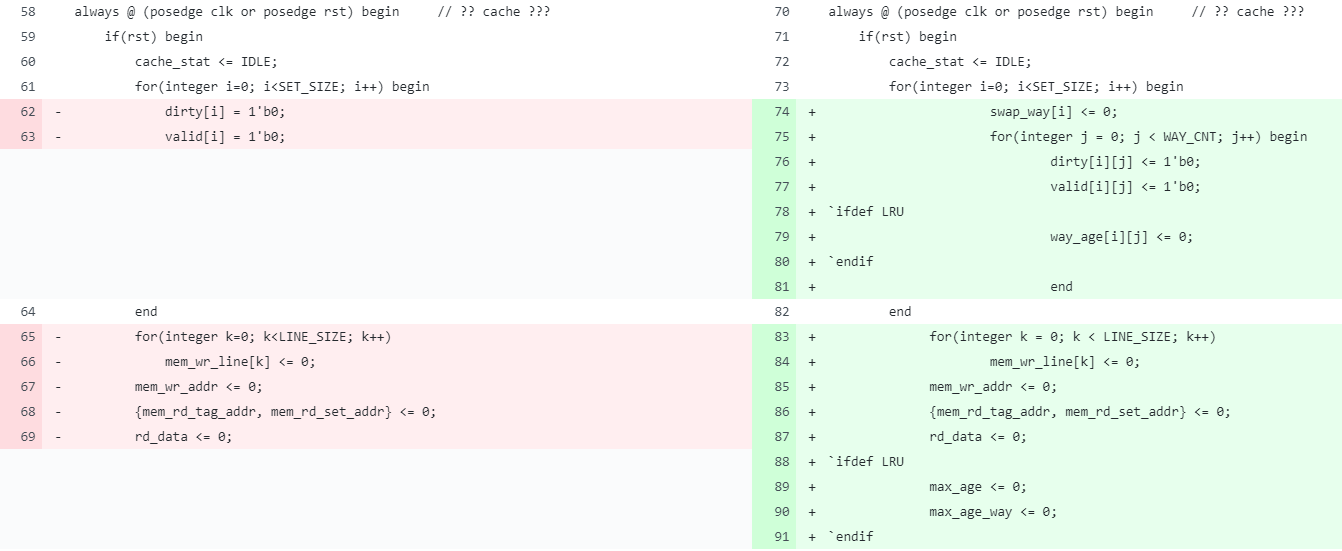
1. 首先，采用宏定义切换LRU和FIFO：



1. 其次，在reg中增加数组的维度（增加了[WAY\_CNT]，图如上）
2. 然后修改cache命中部分，增加变量记录命中的是哪一路，为实现置换策略，增加数组记录下一次换入换出的路（LRU需要额外的存储空间，记录每个路的年龄和最大年龄的路）：



1. 修改begin逻辑，初始化我们增加的变量：

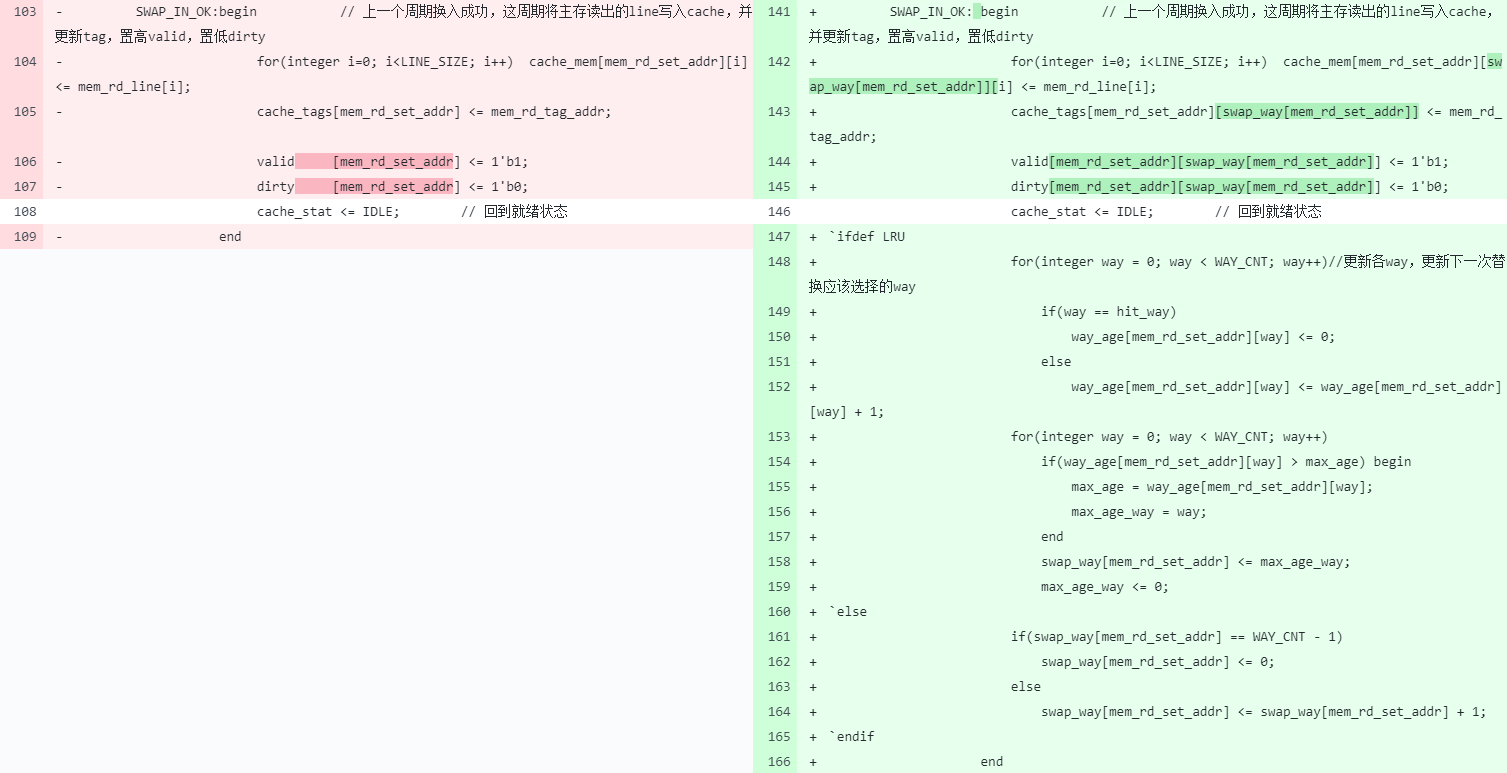


1. 修改idle逻辑，主要是各数组增加维度后要做的修改，以及处理LRU的年龄更新逻辑（LRU年龄就是未命中就加1，命中了就归0，然后再找出max\_age）：



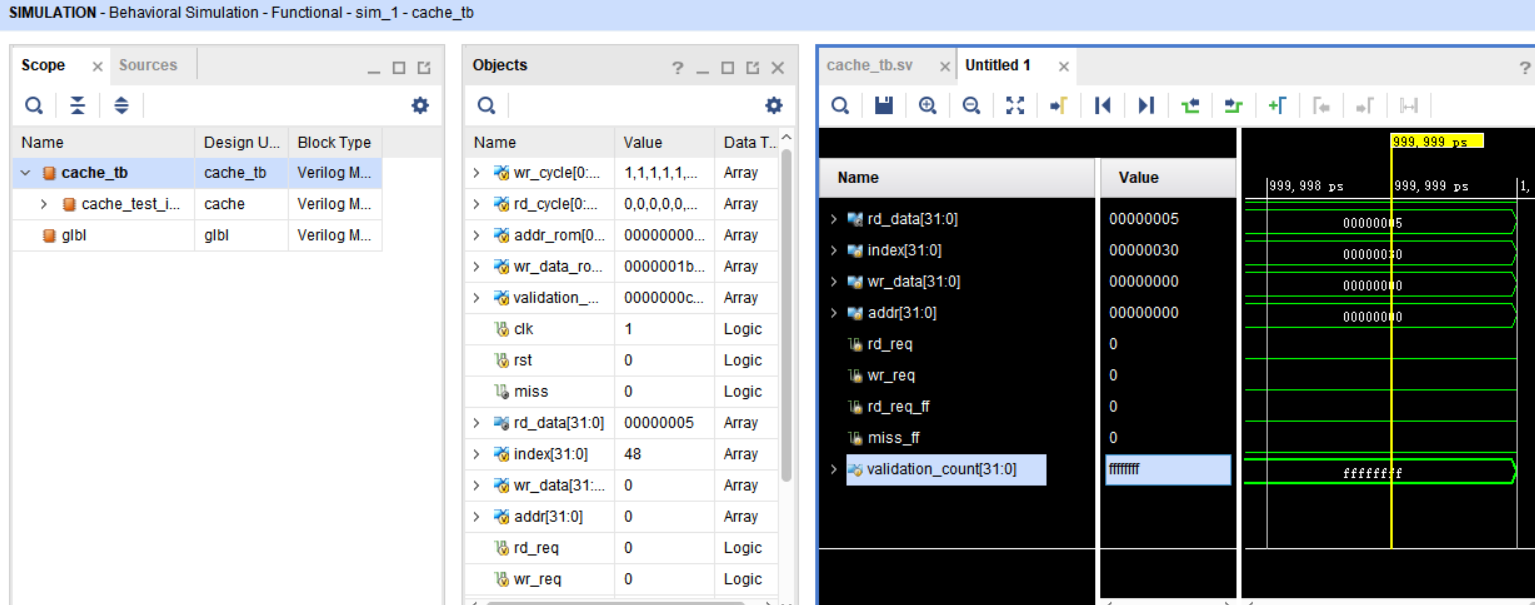


1. 接下来的很多中间状态并不需要处理，直接修改SWAP\_IN\_OK中的逻辑即可，这里由于LRU和FIFO的逻辑不一致（FIFO只需要数组值在0和WAY\_CNT-1之间就行了），因此采用了ifelse的写法：

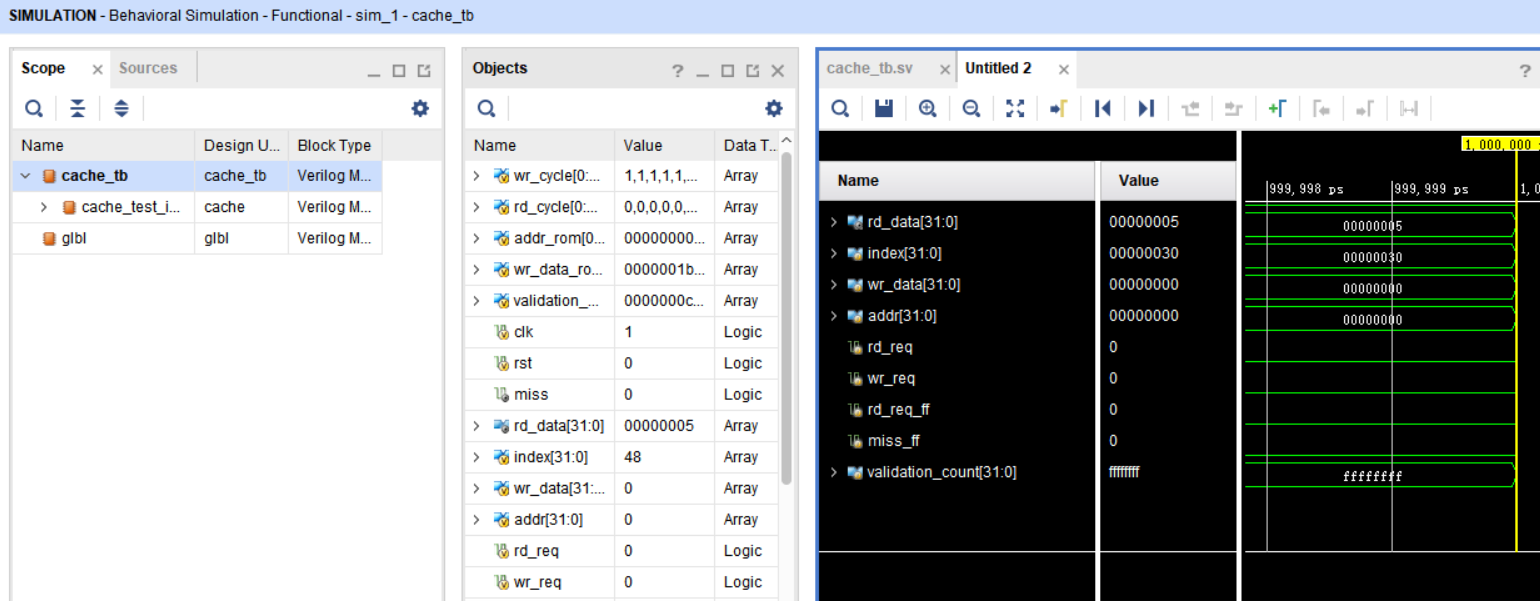


1. 其余逻辑不需要做修改，可以直接沿用。
2. Cache测试
3. 使用原来的cache读写测试：

LRU：

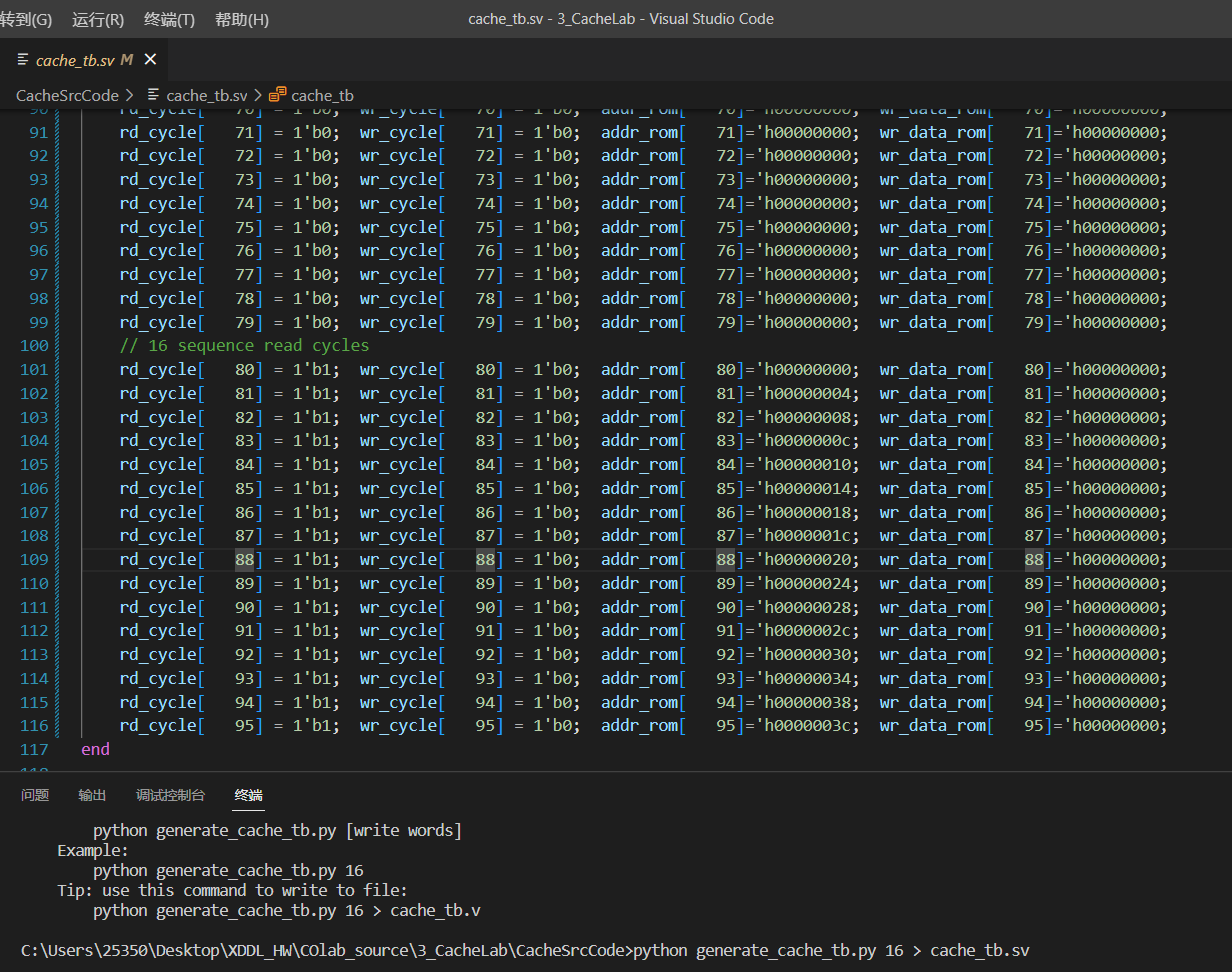


FIFO：



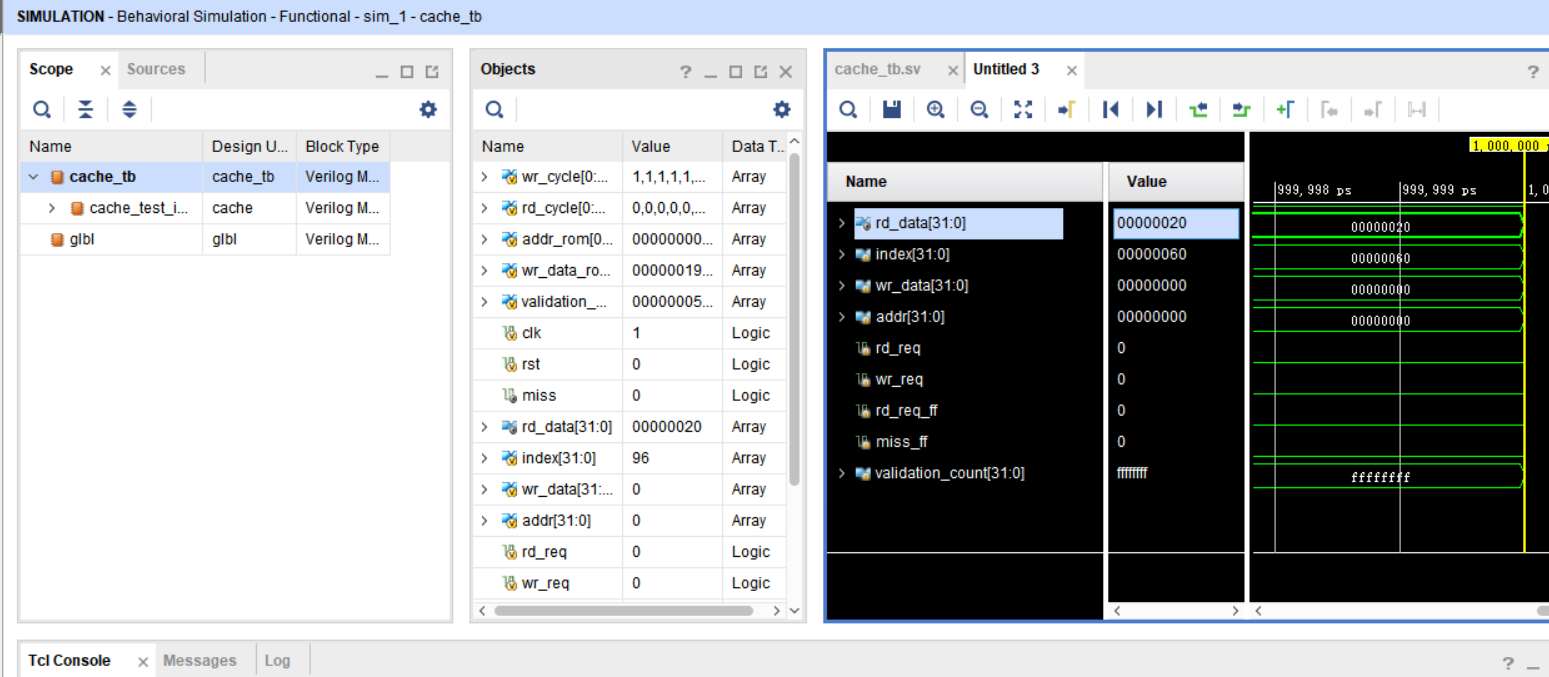
validation\_count值为-1，测试全部通过。

1. 使用generate\_cache\_tb.py生成新的测试文件：

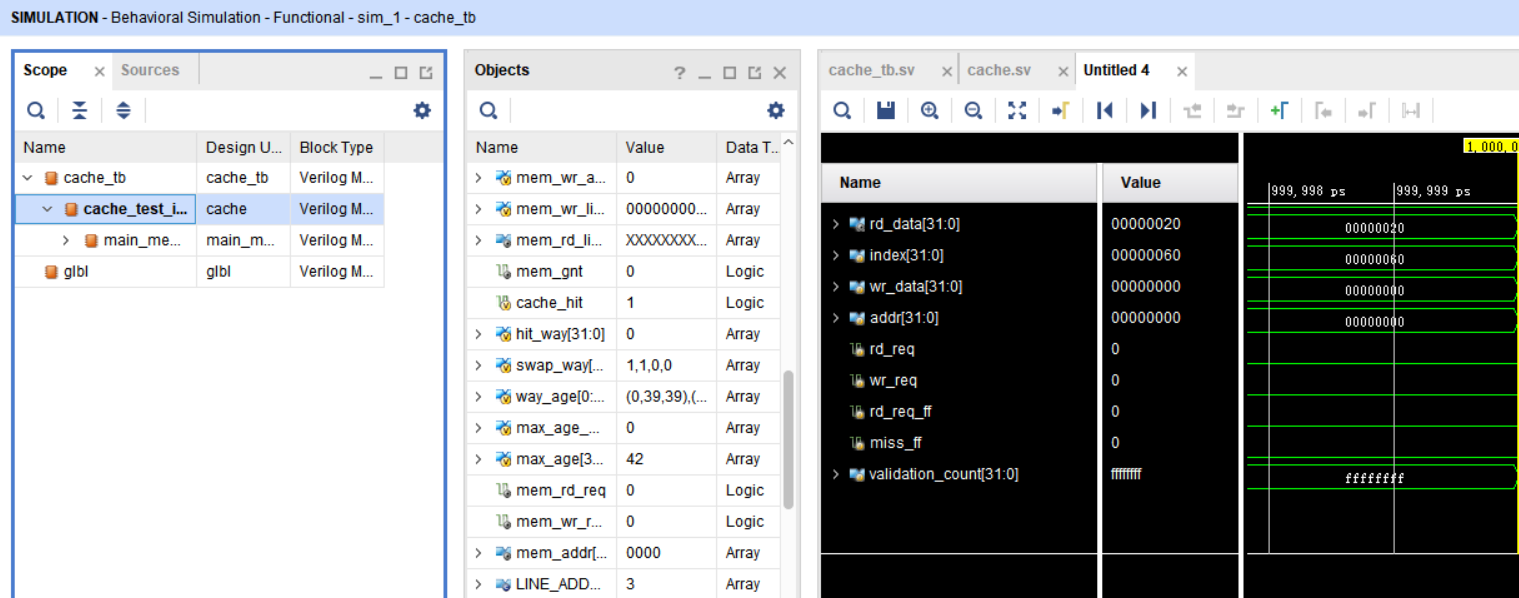


1. 使用新的testbench对cache进行验证：

FIFO：



LRU:



validation\_count值为-1，测试全部通过。更改组相联的测试不在此次写明了，在寻找更好的cache参数时会做测试，也均通过

1. Cache性能评估

1.资源占用方面

体会cache size、组相连度、替换策略针对不同程序的优化效果，以及策略改变带来的电路面积的变化。

保持主存大小为 2^13，TAG\_ADDR\_LEN + SET\_ADDR\_LEN + LINE\_ADDR\_LEN = 13

首先对直接映射的cache进行资源占用测试，更改数据后得到结果如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 直接映射（LUT/FF） | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 517/1123 | 738/1671 | 1233/2759 |
| 3 | 1154/2012 | 1354/3068 | 1860/5172 |
| 4 | 2151/3197 | 2332/5873 | 3477/10017 |

再对组相联进行测试：

|  |  |  |  |
| --- | --- | --- | --- |
| 组相连FIFO（LUT/FF） | | | |
| WAYCNT = 3 | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 1668/2357 | 1987/4129 | 3345/7652 |
| 3 | 3397/4262 | 3403/7558 | 5720/14128 |
| 4 | 5434/8087 | 6202/14443 | 10571/27133 |

|  |  |  |  |
| --- | --- | --- | --- |
| 组相连FIFO（LUT/FF） | | | |
| WAYCNT = 4 | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 2303/2913 | 2685/5233 | 4337/9843 |
| 3 | 4217/5326 | 4747/9678 | 7730/18352 |
| 4 | 8264/10171 | 8514/18603 | 13815/35437 |

|  |  |  |  |
| --- | --- | --- | --- |
| 组相连FIFO（LUT/FF） | | | |
| WAYCNT = 5 | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 2615/3469 | 3052/6337 | 5094/12035 |
| 3 | 4490/6390 | 5123/11798 | 9888/22595 |
| 4 | 8777/12255 | 10241/22768 | 16378/43739 |

|  |  |  |  |
| --- | --- | --- | --- |
| 组相连LRU（LUT/FF） | | | |
| WAYCNT = 3 | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 2460/2809 | 2968/4969 | 4618/9264 |
| 3 | 3897/4714 | 4367/8398 | 6659/15739 |
| 4 | 6616/8541 | 6924/15279 | 11462/28744 |

|  |  |  |  |
| --- | --- | --- | --- |
| 组相连LRU（LUT/FF） | | | |
| WAYCNT = 4 | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 3624/3489 | 3952/6329 | 6148/11973 |
| 3 | 5409/5902 | 6052/10774 | 9083/20483 |
| 4 | 9028/10747 | 9459/19704 | 15237/37565 |

|  |  |  |  |
| --- | --- | --- | --- |
| 组相连LRU（LUT/FF） | | | |
| WAYCNT = 5 | | | |
| LINE\_ADDR\_LEN\SET\_ADDR\_LEN | 2 | 3 | 4 |
| 2 | 4049/4173 | 4648/7692 | 7077/14676 |
| 3 | 6156/7098 | 6519/13151 | 11130/25221 |
| 4 | 12248/12973 | 10187/24114 | 17919/46386 |

我们可以看到，LUT 和FF资源的占用和以下几个因素呈正相关：每个 LINE 中 WORD 的数量（2^LINE\_ADDR\_LEN）、SET 的数量（2^SET\_ADDR\_LEN）、每个 SET 中的 WAY 的数量（WAY\_CNT，如果是组相连的话）

注意到，SET\_ADDR\_LEN 为 2 和 3 的时候，LUT 占用相差不大，这很可能和 LUT 的实现方式有关。

查找表（Look-Up-Table）简称为 LUT，本质上就是一个 RAM。目前FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成一个有 4 位地址线的的 RAM。PLD/FPGA 开发软件会自动计算逻辑电路的所有可能结果，并把真值表（即结果）事先写入RAM，这样，每输入一个信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可。由于一个 LUT 对应 4 个输入，如果电路的规模不是很理想，比如，并不是正好为 4 的倍数，可能带来一些不必要的浪费，上面的 SET\_ADDR\_LEN 为 2 和 3时，资源使用差不多，说明 SET\_ADDR\_LEN 为 2 时，占用了不必要的资源。

从上面的表中可知，直接映射及组相联的两种策略在 LUT 和 FF 的占用方面对于几个参数的变化是一致的（随着 LINE\_ADDR\_LEN，SET\_ADDR\_LEN，WAY\_CNT 的增大而增大，即Cache的 size 变大，所需的资源变多）。

组相联的 FIFO 策略/LRU 策略资源占用数量上较为接近，其他参数相同的情况下，使用 LRU 策略的资源占用比 FIFO 高出约 10%-20%（也就是要记录的变量多几个）。

在实现 Cache 的时候，资源占用方面主要考虑的是几个参数的大小，也就是 Cache 的大小，Cache 越大，资源占用越多。

2.

**阶段2：**

使用阶段一编写的N路组相连cache，正确运行我们提供的几个程序。

要求在cpu中加入你所编写的N路组相连的cache，并要求能成功运行这个排序算法（所谓成功运行，是指运行后的结果符合预期）

想要成功运行带cache的cpu,首先由于指令数据读取机制在这个lab与前两个lab的不同，运行数据和指令都是提前通过python生成内存指令内容并手动输入：

因此也需要对cpu的IDSegReg,Hazard，以及主要是封装cache模块的MWSegReg进行修改适配。

* **实验结果分析（在这一实验报告中，需要对比所实现的三个CPU：单周期CPU、四级流水线CPU以及带缓存的四级流水线CPU的性能、功耗资源分析、时钟频率以及CPI（Cycle Per Instruction）等指标，进行评估与分析）**
* 实验总结（说说踩的坑，总结收获，分析下自己花了多少时间，都用来做什么事情）：

许珂钒：

项奕玮：

* 提出改进实验的意见：

**分数分配：**