# Cache实验指导

王轩

#### 回顾

这里是cache实验的第二阶段指导文档。Cache实验分为两个阶段：

1. **Cache 的实现和独立测试**。也就是第一阶段。独立测试即脱离CPU的测试。具体请见文档《Lab3-王轩-cache编写指导.docx》
2. **Cache 与CPU组合，并对benchmark性能进行测试。**也就是第二阶段，这一步完成后，加深对Cache策略在不同场合下对性能和电路面积的影响。

#### CPU+cache 联合测试（快速排序）

阶段二我们提供了完整CPU的代码，并在流水线中假如了cache miss发生时的流水线stall动作。并提供了两个能够运行的benchmark：快速排序和伪·矩阵乘法，方便学生对cache性能在这两种情景下进行测试。

首先，建立Vivado工程，将 ./3\_CacheLab/CPUSrcCode 和 ./3\_CacheLab/CacheSrcCode 中的所有 .v和.sv文件加入vivado工程。此时Vivado工程的文件结构应该如图1 。注意，./3\_CacheLab/CPUSrcCode/Testbench/cpu\_tb.v 应该作为仿真的顶层文件而存在。

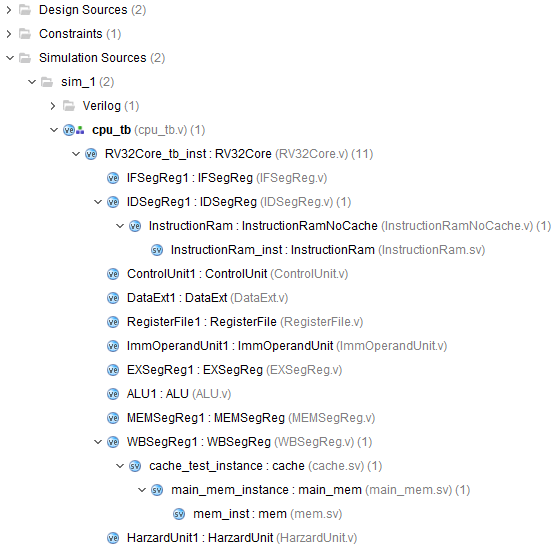


图1：CPU+Cache 工程的文件结构图

此时还不能仿真，因为我们还没将要运行的程序的指令和数据放入指令RAM和数据RAM中。我们首先测试一下快速排序。

###### 生成快速排序所需要的指令

打开目录 ./3\_CacheLab/ASM-Benchmark/generate\_inst，在其中运行命令：

|  |
| --- |
| python asm2verilog.py QuickSort.S InstructionRAM.sv |

代表汇编QuickSort.S文件，得到一个保存了指令流的指令存储器文件InstructionRam.sv。使用其中的内容替换Vivado工程中的InstructionRam.sv。

###### 生成快速排序所需要的数据

打开目录 ./3\_CacheLab/ASM-Benchmark/generate\_inst，在其中运行命令:

|  |
| --- |
| python .\generate\_mem\_for\_quicksort.py 256 > mem.sv |

表示生成256个被打乱的数，保存在数据存储器文件mem.sv中，使用其中的内容替换vivado工程中的mem.sv文件。

###### 进行仿真

在vivado工程中开始仿真。波形运行一段时间后，会发现mem.sv中原本乱序的数组变有序了（从小到大排列），说明快速排序运行成功，当然，前提是你所编写的cache.sv是正确的。

注意：我们提供的cache.sv虽然可以正确运行，但它是直接映射策略的，你需要保证你所编写的FIFO和LRU策略的组相连cache也能成功运行快速排序。

###### 修改快速排序规模

1. QuickSort.S 中固定的对256个数进行排序，假如我们想把排序的规模改成512个，需要在QuickSort.S中，修改第一个指令为：

|  |
| --- |
| xor a3, zero, 0x200 |

1. 然后重新运行asm2verilog.py脚本进行汇编。注意，规模不要太大，否则被排序的数组会占用栈的空间（快速排序涉及递归，需要用到栈），不过，你可以在QuickSort.S中修改第二条指令， 把栈的起始地址改大一些，以避免地址冲突。

除了汇编语言要改以外，数据存储器中初始化的数字个数也要改成512个，打开目录 ./3\_CacheLab/ASM-Benchmark/generate\_inst，在其中运行命令:

|  |
| --- |
| python .\generate\_mem\_for\_quicksort.py 512 > mem.sv |

表示生成512个被打乱的数。

#### CPU+Cache联合测试（矩阵乘法）

###### 生成矩阵相乘所需的数据

打开目录 ./3\_CacheLab/ASM-Benchmark/generate\_inst，在其中运行命令:

|  |
| --- |
| python .\generate\_mem\_for\_matmul.py 16 > mem.sv |

表示生成两个初始的方阵（源矩阵）放在数据RAM内，这两个RAM的大小为16\*16，同时为矩阵乘法的结果（目的矩阵）准备一块内存空间。打开mem.sv，我们发现，RAM的首地址开始是目的矩阵，全部初始化赋值为0，但python脚本已经帮你算好了它在完成矩阵乘法后正确的值是什么，这个结果被放在注释里，如图2：

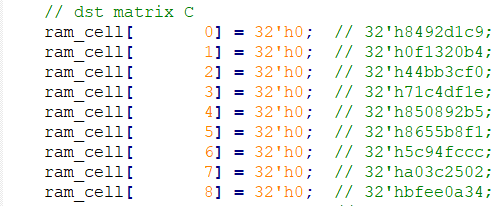


图2：目的矩阵的初始化（右边注释是算完矩阵乘法后的正确值）

在mem.sv中，再往后是两个源矩阵，如图3。矩阵乘法程序做的事情就是把两个源矩阵相乘后，结果放在目的矩阵的位置上，其结果应该和注释相同（前提是你的cache写的是对的）。

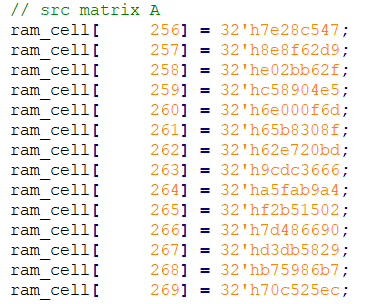
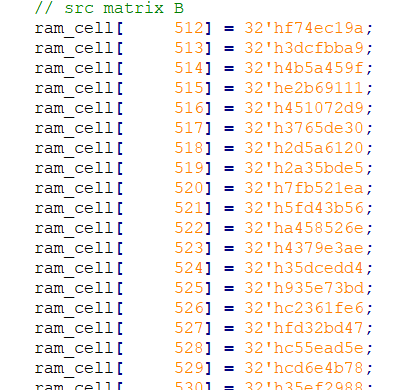
 

图3：两个源矩阵

###### 生成矩阵相乘所需的指令

打开目录 ./3\_CacheLab/ASM-Benchmark/generate\_inst，在其中运行命令：

|  |
| --- |
| python asm2verilog.py MatMul.S InstructionRAM.sv |

代表汇编MatMul.S文件，得到一个保存了指令流的指令存储器文件InstructionRam.sv。使用其中的内容替换Vivado工程中的InstructionRam.sv。然后进行仿真即可。仿真后请查看波形图中，mem.sv中的ram\_cell变量，是否与注释中相同。如果相同说明运行正确。

因为我们的 RV32I CPU 没有实现乘法指令，所以这里的MatMul.S实际上是伪矩阵乘法，它使用按位或代替加法，用加法代替乘法，完成矩阵运算。虽然不是真的矩阵乘法，但能够模仿矩阵乘法对RAM的访问过程，对cache的性能研究起到作用。

###### 修改问题规模

要修改矩阵相乘中矩阵的规模，首先，我们修改MatMul.S中的第一条指令：

|  |
| --- |
| xori a4, zero, 4 |

a4寄存器决定了计算的规模，矩阵规模=N\*N，N=2^a4。例如a4=4，则矩阵为 2^4=16阶方阵。该值可以修改。例如修改成3，则矩阵就是2^3=8阶方阵。

然后，我们在运行generate\_mem\_for\_matmul.py时修改命令行参数：

|  |
| --- |
| python .\generate\_mem\_for\_matmul.py 8 > mem.sv |

参数8代表生成的矩阵的规模为8\*8，即8阶方阵。

使用新生成的8阶方阵的指令和数据去进行仿真即可。