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，这一步的目的不仅是进一步验证你所编写的 cache 的正确性，也是为编写实验报告做准备。

建立 Vivado 工程

阶段二使用 Lab1 实现的 CPU，你需要在将原有的 Data Cache 替换成阶段一实现的 Cahce，并修改数据通路，来统计 Cache Miss 次数，并在流水线中加入 cache miss 发生时的流水线 stall。我们提供了两个能够运行的 benchmark：**快速排序**和**伪·矩阵乘法**，方便大家对 cache 性能在这两种情景下进行测试。

建立 Vivado 工程的步骤如下，请按照以下步骤去做:

1、 首先，建立 Vivado 工程，将你的 Lab1 代码放到 CPUSrcCode 目录下，再将 **./CPUSrcCode** 和 **./CacheSrcCode** 中的所有.v 和.sv 文件加入 vivado 工程。

2、 在"Simulation Source"中，将**./CPUSrcCode/cpu_tb.v** 设置为顶层文件。它下面包括完整的 CPU 和 Cache，对它进行仿真时，整个 CPU+cache 都会被仿真。

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。

CPU+cache 联合测试（快速排序）

生成快速排序所需要的指令

打开目录 ./ASM-Benchmark/generate_inst，使用 CMD 在其中运行命令：（注意：如果直接复制下面的指令在 CMD 执行，可能出现无法找到 asm2verilog.py 文件错误）*

```
python asm2verilog.py QuickSort.S InstructionCache.v
```

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 InstructionCache.v。使用其中的内容替换 Vivado 工程中的 InstructionCache.v。

生成快速排序所需要的数据

打开目录 ./ASM-Benchmark/generate_inst，在其中运行命令：（注意：如果直接复制下面的指令在 CMD 执行，可能出现无法找到 asm2verilog.py 文件错误）

```
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
```

2、 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。

3、 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 ./ASM-Benchmark/generate_data，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。

之所以提供修改快速排序规模的功能，是为了方便学生在写 Cache 实验报告时，能够测试不同规模的快速排序。（实际上不要求学生一定要测试不同规模的快速排序，是否有必要取决于学生写实验报告时的思路。如果仅仅使用 256 个数的排序就能说明问题，也可以不测试其它规模的快速排序）

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 ./ASM-Benchmark/generate_data，在其中运行命令：（注意：如果直接复制下面的指令在 CMD 执行，可能出现无法找到 asm2verilog.py 文件错误）

```
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：

```
// dst matrix C
ram_cell[       0] = 32'h0;  // 32'h8492d1c9;
ram_cell[       1] = 32'h0;  // 32'h0f1320b4;
ram_cell[       2] = 32'h0;  // 32'h44bb3cf0;
ram_cell[       3] = 32'h0;  // 32'h71c4df1e;
ram_cell[       4] = 32'h0;  // 32'h850892b5;
ram_cell[       5] = 32'h0;  // 32'h8655b8f1;
ram_cell[       6] = 32'h0;  // 32'h5c94fccc;
ram_cell[       7] = 32'h0;  // 32'ha03c2502;
ram_cell[       8] = 32'h0;  // 32'hbfee0a34;
```

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）。

```
// src matrix A
ram_cell[     256] = 32'h7e28c547;
ram_cell[     257] = 32'h8e8f62d9;
ram_cell[     258] = 32'he02bb62f;
ram_cell[     259] = 32'hc58904e5;
ram_cell[     260] = 32'h6e000f6d;
ram_cell[     261] = 32'h65b8308f;
ram_cell[     262] = 32'h62e720bd;
ram_cell[     263] = 32'h9cdc3666;
ram_cell[     264] = 32'ha5fab9a4;
ram_cell[     265] = 32'hf2b51502;
ram_cell[     266] = 32'h7d486690;
ram_cell[     267] = 32'hd3db5829;
ram_cell[     268] = 32'hb75986b7;
ram_cell[     269] = 32'h70c525ec;
```

```
// src matrix B
ram_cell[     512] = 32'hf74ec19a;
ram_cell[     513] = 32'h3dcfbba9;
ram_cell[     514] = 32'h4b5a459f;
ram_cell[     515] = 32'he2b69111;
ram_cell[     516] = 32'h451072d9;
ram_cell[     517] = 32'h3765de30;
ram_cell[     518] = 32'h2d5a6120;
ram_cell[     519] = 32'h2a35bde5;
ram_cell[     520] = 32'h7fb521ea;
ram_cell[     521] = 32'h5fd43b56;
ram_cell[     522] = 32'ha458526e;
ram_cell[     523] = 32'h4379e3ae;
ram_cell[     524] = 32'h35dcedd4;
ram_cell[     525] = 32'h935e73bd;
ram_cell[     526] = 32'hc2361fe6;
ram_cell[     527] = 32'hfd32bd47;
ram_cell[     528] = 32'hc55ead5e;
ram_cell[     529] = 32'hcd6e4b78;
```

图 3：两个源矩阵

生成矩阵相乘所需的指令

打开目录 ./ASM-Benchmark/generate_inst，使用 CMD 在其中运行命令：（注意：如果直接复制下面的指令在 CMD 执行，可能出现无法找到 asm2verilog.py 文件错误）*

```
python asm2verilog.py MatMul.S InstructionCache.v
```

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 InstructionCache.v。使用其中的内容替换 Vivado 工程中的 InstructionCache.v。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。

修改矩阵乘法的规模

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。

之所以提供修改矩阵规模的功能，是为了方便学生在写 Cache 实验报告时，能够测试不同规模的矩阵乘法。（实际上不要求学生一定要测试不同规模，是否有必要取决于学生写实验报告时的思路。如果仅仅使用 16*16 阶矩阵乘法就能说明问题，也可以不测试其它规模）

注：无论进行快速排序，还是矩阵乘法，最终的主存（mem.sv 模块里的 ram_cell 变量）里的数据都与正确结果整体上是相同的，但会略有差异，原因是这是写回策略的 cache，所以最终会有一些数据还在 cache 中未写入主存。属于正常现象。但如果你的 cache 写错了，那么快速排序和矩阵乘法的结果就会很离谱。在检查实验时，助教主要通过第一阶段 cache_tb.sv （即脱离 CPU 的 cache 检验）去判断你的 cache 的正确性。

对缺失率进行统计

可以在 Lab1 实现的 CPU 的 WBData.v 中加入两个 reg 变量：miss_count（缺失次数）和 hit_count（命中次数）。当进行仿真时，加入这两个变量的波形。最终当程序运行完时，在波形图中查看这两个变量就能得知缺失率等信息。