

实验三（选题一）说明文档

一 实验介绍

1 实验信息

本次实验名为“现代处理器内存消歧和存取优化模拟实验”，实验内容分成两大部分，一是实现允许数据旁路的微体系结构组件，二是利用该模拟器实现 SpectreV4。在第一部分中，设计的框架以及需要实现的组件已经给出，同学们需要实现的是模拟在瞬态执行中的各组件行为。第二部分是对第一部分正确性的验证，本部分将会给出 SpectreV4 的 PoC (Proof of Concept)，同学们在自己的模拟器上实现和 PoC 的攻击效果。

本次实验的建议操作系统为 Linux，由于只涉及访存序列，没有架构的要求。

高级语言采用 C/C++。

文档接下来的章节包括：第一章的[实验目的](#)、[实验要求](#)，第二章的[背景知识](#)，第三章的[实验内容](#)，第四章的评分标准，以及第五章的参考资料。

2 实验目的

本次实验属于创新实验，在课程内容的基础上有所提高。如果同学们找到代码和文档中的纰漏和错误之处，欢迎指出，并可视程度获得一定的加分。实验的代码量较少，但需要同学们花费一定的课外时间阅读文档和代码以及查阅一些额外的资料。实验中涉及到的内存子系统的组成和内存消歧过程以及常见优化策略，本文档将会给出概要性的介绍。

本次实验的主要目的是让同学们：

- 了解现代处理器的内存子系统的构成和功能
- 了解现代处理器针对访存的预测优化

3 实验要求

实验采取线上提交实验报告的模式。

本次实验日期为**第 14 周至第 17 周**，之后算作补交。最晚**第 18 周周四**前提交实验报告，否则实验不得分。

如[第 1 小节](#)所述，本次实验分成两大部分，第一部分是内存子系统的实现，第二部分是访存数据预测的实现和利用。内存子系统主要包括 LSQ, TLB, PageTable, Cache, Physical Memory，不要求实现外存，因此本模拟器的中的虚拟地址空间和物理地址空间相等。在正确实现了内存子系统的各组

件之后就可以针对不同的访存序列进行预测优化了。该预测优化主要实现在瞬态窗口中，即指令送到了 Load store queue，但是结果尚未提交这个时间窗口中。

第一部分中，Task1 要求同学们实现出 TLB 以及页表的访问操作，以及 Cache 和 Physical Memory 的访问操作。

第二部分分成三个子任务 Task2，Task3 和 Task4。Task2 为主要任务，在本任务中需要同学们在任务一的基础上实现允许内存操作乱序执行的 LSQ 调度，在乱序执行中实现 Store-to-Load forward，并且对给定的访存序列进行测试，评估你的模拟器的 TLB miss rate 和 Cache miss rate。Task3 为针对这些激进的内存访问优化的利用。最后在 Task4 中请你从一个设计者的角度给出测试用例来验证你的实现的正确性。

二 背景知识

本节将会概要地介绍关于现代处理器的内存子系统，乱序执行，内存消歧，存取优化，以及这些优化机制所带来的各种问题。

1 乱序执行

乱序执行是现代处理器中的一种非常普遍的优化技术，它可以最大限度地理由 CPU 的所有执行单元。CPU 不会严格按照指令的到达顺序来处理指令，而是在所有的必须资源可用后立即执行它们。如果当前的执行单元被占用，其他执行单元可以继续运行。因此，只要指令的结果遵循体系结构定义，就可以继续运行。

为了实现这一点，Tomasulo[2]开发了一种算法来进行动态调度指令以支持乱序执行。Tomasulo 算法引入了保留站(Reservation Station)，它允许 CPU 使用已计算出的数据，而不是将其存储在通用寄存器上，然后并重新读取。保留站的作用为重命名寄存器，以允许在相同物理寄存器上操作的指令使用最后的逻辑寄存器来解决写后读 (RAW)，读后写 (WAR) 和写后写 (WAW) 的危险。

在英特尔架构上，流水线由前端，执行引擎（后端）和内存子系统组成。前端从内存中获取 x86 指令，并将其解码为微操作 (μOP)，然后将其发送到执行引擎。乱序执行是在执行引擎中实现的，如图 1 所示。重新排序缓冲区(ReOrder Buffer)负责寄存器分配，寄存器重命名和退出。此外，重排缓冲区直接处理其他优化。将 μOP 转发到保留站，该站将连接到执行单元的出口端口上的操作排队。每个执行单元可以执行不同的任务，例如 ALU 操作，AES 操作，地址生成单元 (AGU) 或内存加载和存储。AGU 以及加载和存储执行单元直接连接到内存子系统以处理其请求。

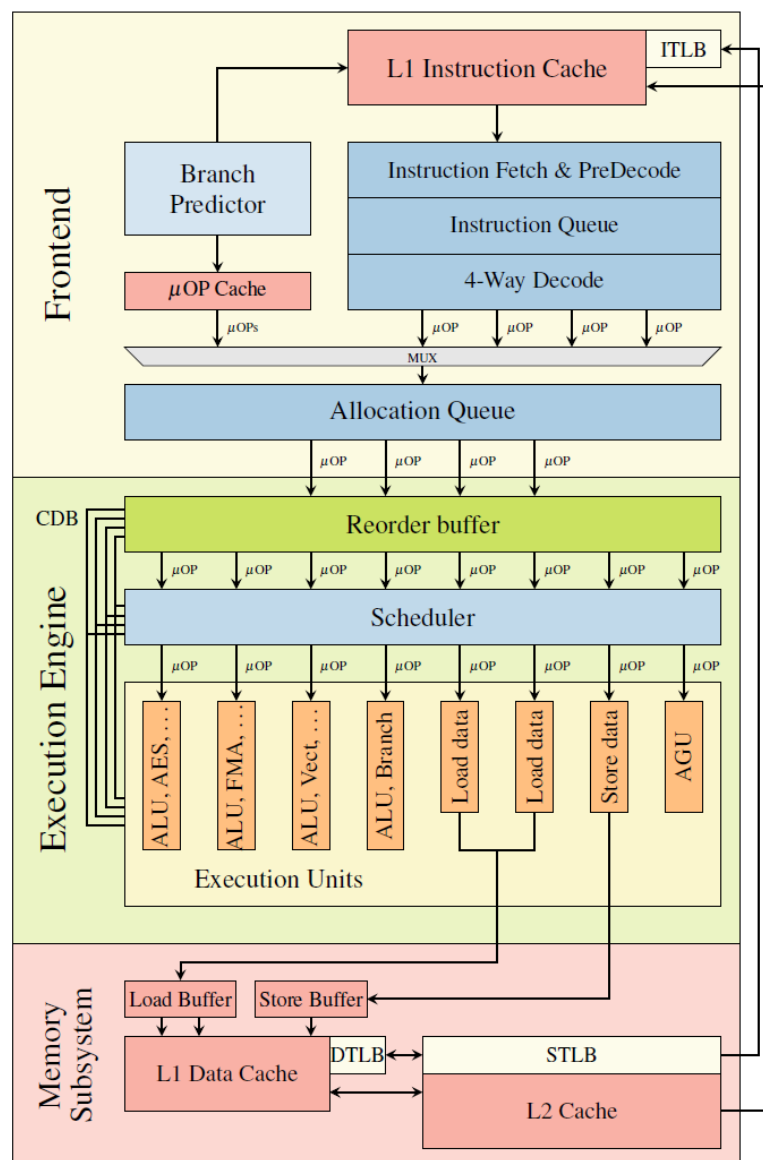


图 1 Intel Skylake 微架构的缩略图

在本实验中，我们不需要实现所有指令的乱序，我们只需要实现内存存取指令的并行执行即可。因此我们不需要实现前端，所有的指令依旧通过宏指令 (Macro OP) 来进行执行，但是我们依旧需要实现乱序执行。于是我们需要实现的组件为 ReOrder Buffer, Scheduler, AGU 以及内存子系统。为了简化实验，将焦点放在存取优化上，在本实验中我们同样弱化了保留站的存在，在给定的访存指令序列中不使用通用寄存器来进行标志，瞬态数据也不使用逻辑寄存器来进行存储，而是使用一个标志每条指令的 Control Block 来进行存储，其中各种数据的访问权限通过 Scheduler 来维护。

2 内存子系统

计算机内存存在程序员看来就是一个扁平的大存储空间。由于内存数据的存取往往是流水线中最耗时的指令，那么我们需要采取一系列的措施来进行加速。

2.1 内存层级

由于存储工艺上的提升非常有限，系统结构的涉及开始另辟蹊径，那就是利用内存数据使用的局

部性。在计算机内存系统中的局部性主要包括两个方面，一个是空间局部性，即如果程序访问了一个特定的内存地址，那么该内存地址附近的几个地址很有可能会在接下来的事件比访问；另一个是时间局部性，即访问过的内存地址和数据很有可能会在接下来一段时间被再次使用。利用该局部性原理，我们可以设计出不同访问速度和操作单元的内存层级。从架构上看，包括：Register - Cache - Main Memory - Backing Store(i.e. disk)。这些组件的访问速度逐级递减，我们编写的模拟器中也要体现出这一点；此外还有很重要的一点是这些组件的一次操作数据大小也不同。Register 和 Cache 之间的移动数据大小一般是字节或者字，这是 CPU 和 Register 之间的数据大小是相同的；而 Cache 和 Main Memory 之间的操作数据大小往往是块(Block or Line)，这个块的大小等于一个 Cache entry 所能存储的数据大小。但是由于在 Cache 中也同样存在层级的关系，这些操作单元也会跟着改变，本次实验只需要实现 L1D 即可。Main Memory 和 Backing Store 之间的操作单元往往是一个或者多个页甚至扩展到一个扇区，这取决于置换算法的实现。在本实验中我们不要求实现 Backing Store，但是在实现的细节上，我们必须注意每次数据替换的大小，以更好的实现局部性。

2.2 寻址方式

寻址方式是现代处理器的指令集架构的一部分。不同的寻址方式定义了指令的机器码应该如何确定指令的操作数。寻址方式指明了应该如何从使用保存在寄存器中的信息，或包含在机器指令或其他地方的常量来计算操作数的有效地址。寻址方式在不同的指令集中会有不同的表现，比如 ARM 是一个 Load-Store 存取模型，也就是只有 ldr 和 str 指令才能访问内存，而 Intel 则可以直接使用 mov 指令来进行访存。通常情况下，寻址方式分为直接寻址（绝对寻址）和相对寻址，其中直接寻址为在指令中直接给操作数在内存中的真实地址，而相对寻址又分为基址寻址（通过指定的基址寄存器来进行偏移计算），寄存器寻址（真实地址保存在寄存器中），一次间接以及多级间接（指令操作数是数据真实地址的地址）。

在本实验中，限定的存取模型为 Load-Store 模型，算术和 mov 等指令不允许访存。需要模拟的寻址方式只有两种，直接寻址和一次间接寻址。具体模拟要求将会在三 实验内容中展开描述。

2.3 虚拟内存和地址翻译

虚拟内存的作用是扩大程序所能使用的空间，并且使得主内存可以在不同的进程之间共享。原理是每个活动进程的代码和数据区域的实际是哟个部分保留在主内存中，而在当前执行阶段未访问的哪些部分保留在 Backing Store 上。

由于主内存只能通过物理地址来进行寻址，因此我们需要实现虚拟地址到物理地址的转换。这个转换的实现是通过页表(Page Table)来实现的，每一个页表项都包含了一个虚拟页号到物理页号的转换。页表本质上也是内存的某些页中的内容，因此需要考虑页表项的内容是否会超出一个页的存储能力，如果超出了，将需要用到多级页表。

对比内存，我们也同样可以使用一样的方法来进行翻译加速。也即内存中的页表也可以通过一定方式缓存到访问更快的组件 TLB 中。在本实验中，同学们需要实现虚拟地址翻译到物理地址的过程。

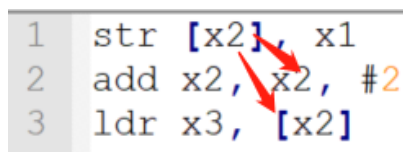
3 内存消歧和存取优化

3.1 内存依赖

在乱序执行处理器中，编译器、系统、架构必须保证程序语义的正确性，对内存访问尤其如此。如果不能保证内存位置的独立性，也就是存在内存操作对同一个位置的读写，则必须对这些位置执行严格的访问顺序。这被称为内存的抗别名 (Memory anti-aliasing) 或者是内存消歧 (Memory disambiguation) 问题。

在流水线处理器中，指令以推测方式获取，解码和执行 (Speculative Execution)，并且直到指令提交后才允许修改系统状态。对于修改寄存器的指令，通常使用寄存器重命名来实现。对于存储 (Store) 到内存中的情况，推测性存储在执行时写入存储队列 (Store Queue/Buffer)，并且仅在提交存储指令后才写入高速缓存。

但是，存储队列 (Store Queue/Buffer) 会带来新的问题。如果 Load 与更早的 Store 的数据相关，则 Load 要么必须等待 Store 成功提交，然后才能从高速缓存加载值，否则存储队列 (Store Queue) 必须能够将推测 Load 所需的值转发到加载 (Store-to-Load forward)。这要求处理器知道给定的 Load 是否取决于较早尚未提交的 Store，但这比找出寄存器依赖性要困难得多。如图 2 不同指令在内存上的依赖很难发现所示：



```
1 str [x2], x1
2 add x2, x2, #2
3 ldr x3, [x2]
```

图 2 不同指令在内存上的依赖很难发现

在指令 1 中，寄存器 x2 有两个依赖关系，一是 2 和 3 之间有一个 RAW 冒险，另一个是 1 和 3 之间也同样存在一个 RAW 冒险。这种通过寄存器携带的依赖关系很容易提前知道，因为源和目标寄存器可以通过指令显式地知道。但是如果依赖是在于内存地址，只有在地址生成单元将地址完全计算出来才能知道内存地址，从而进行依赖消除。在本例中，指令 1 和 3 就包含了内存依赖，如果我们想要实现乱序执行，比如 ldr 指令在 str 指令之前执行，那么我们必须 Know（非预测执行状态下）或者是 predict(推测执行)该 Load 指令是否和之前的 Store 指令有相关 (Memory dependency speculation)，然后再进行预测的验证 (Memory disambiguation)。

3.2 Store-to-Load Forward 预测优化

当 Load 指令依赖于较早的 Store 指令时，需要通过存储队列 (Store Queue) 从 Store 指令中进行旁路转发。这可以通过以下方式进行：在存储队列 (Store Queue) 中搜索地址与 Load 匹配的 Store 指令，如果存在匹配项，则直接转发，否则再向高速缓存 (Cache) 中进行检索。在实际的现代处理器，比如 Intel 的很多架构，都允许大小不等的未对齐内存访问，因此有许多的组合需要考虑。比如在图 2 不同指令在内存上的依赖很难发现代码片段中，如果 Load 指令和 Store 指令只是低位发生了依赖，

而高位则仍需要从 Cache 中获取。这种情况比较少，在这种情况下不会发生转发。在本实验中指令有限，不考虑寄存器，所以保证在测试样例中不存在这种情况。

通过搜索存储队列（Store Queue）仅仅是一种转发的办法，也是本实验需要实现的算法。在不同的架构中会有非常多不同的实现，转发的成功和失败的情况细节将会有非常多的不同，比如在 Intel 的很多处理器实现中，转发成功的条件将会激进到只需要地址的低 12 位（页内偏移）相等即可进行转发。这种激进的优化策略配合其 SMT（同步多线程）技术有较明显的性能提升，但是也同样带来了一些安全上的问题，比如 Fallout[3]攻击就是利用了 SMT 和 Intel 激进的 Store-to-Load Forward。有兴趣的同学可以课后进行了解。

4 Store-to-Load Forwarding 的利用

4.1 Load Store Queue/Buffer

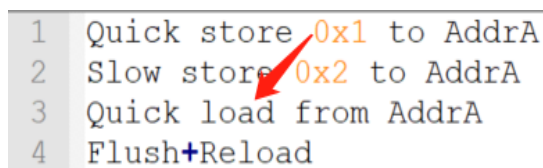
当执行单元需要将数据写入内存时，它不必等待 Store 指令完成，而只是将请求放入 Store Queue 中。这样，CPU 可以继续执行当前执行流中的指令，而不必等待写入完成。这种优化之所以行之有效，因为在许多情况下，写操作不会影响后续指令，即，仅影响相同地址的 Load 指令。同时，Store Queue 异步处理写操作，确保将结果写入内存。因此，Store Queue 可防止在等待内存子系统完成写操作时 CPU 停顿。同时，即使执行顺序混乱，它也可以确保写入顺序地到达内存子系统。

对于每个发射的写操作，CPU 都会在 Store Queue 中分配一个条目。该条目需要目标的虚拟地址和物理地址。该组件一般不会像 Cache Line 一样条目有非常明确的格式化和大小，它是通过条目来进行统计的。在 Intel CPU 上，存储缓冲区最多具有 56 个条目，从而允许同时处理多达 56 个存储。仅当存储缓冲区已满时，前端才会停止，直到一个空插槽再次可用。尽管存储缓冲区隐藏了存储的延迟，但它也增加了加载的复杂性。这种复杂性已经在 3.2 Store-to-Load Forward 预测优化一节中讨论过了。

在本实验中，为了减少两个队列之间的交互，Load Queue 和 Store Queue 合并起来共同处理访存请求，其中每个条目的格式已经给出，同学们需要在此基础上实现内存消歧。

4.2 SpectreV4

SpectreV4[4]也称为 SpectreSSB(Speculative Store Bypass)，顾名思义，它的攻击效果就是恶意操作被推测性地绕过了 Store。它的攻击原理如图 3 SpectreV4 Gadget 所示：



```
1 Quick store 0x1 to AddrA
2 Slow store 0x2 to AddrA
3 Quick load from AddrA
4 Flush+Reload
```

图 3 SpectreV4 Gadget

在上述代码片段中存在两处 RAW 冒险，一处是 1 和 3 之间，一处是 2 和 3 之间。在旁路转发原理

中我们提到，只有当 LSQ 中的 Store 指令的虚拟地址被解析出来之后才能被后续的 Load 指令利用，进行数据转发。在本例中，Slow Store 和 Quick Store 的区别就在于地址解析的快慢，如果 Slow Store 需要进行复杂的寻址，地址生成单元的结果返回较慢。而 Quick Store 是直接寻址，那么 Load 指令在 Store Queue 中进行检索的时候，因为 Slow Store 尚未准备好，并不能从 Slow Store 中进行旁路，而是从一个更早 Store 旁路，甚至是从高速缓存中获得了一个值，这个值称为 Stale Value。它是一个陈旧的，不应该被再次利用的值。旁路完成以后 Load 指令就已经执行完毕，等待提交，不会再进行检索和访存。然而当 Slow Store 执行完成，在提交阶段检查后续的所有相关的指令，查看是否有 Load 指令和本 Store 发生了冒险，并且该 load 指令的值却是从更早的 Store 或者是内存获取的，如果有，该 Load 指令以及后续的所有指令都必须重发。否则将会造成执行出错。

SpectreV4 就是利用了这种错误的 Store-to-Load Forwarding，在 Load 指令旁路完成，尚未提交的这个瞬态窗口中利用 Cache 侧信道把 Stale Value 泄露出来。

在本实验中，同学们要求在实现了旁路转发的基础上实现 SpectreV4，测试方法是给定一个 SpectreV4 Gadget 的输入，要求能够恢复出 Stale Value。由于在本实验中攻击者是在泄露本进程内的数据，该威胁模型比较弱，所以同进程的 SpectreV4 也作为验证同学们实现的正确性。

4.3 Cache Side Channel

Cache 侧信道的利用原理是 Cache 命中和缺失的时间差很明显，这个时间差可以作为侧信道被利用。下面将介绍 Flush + Reload [5] 侧信道攻击。

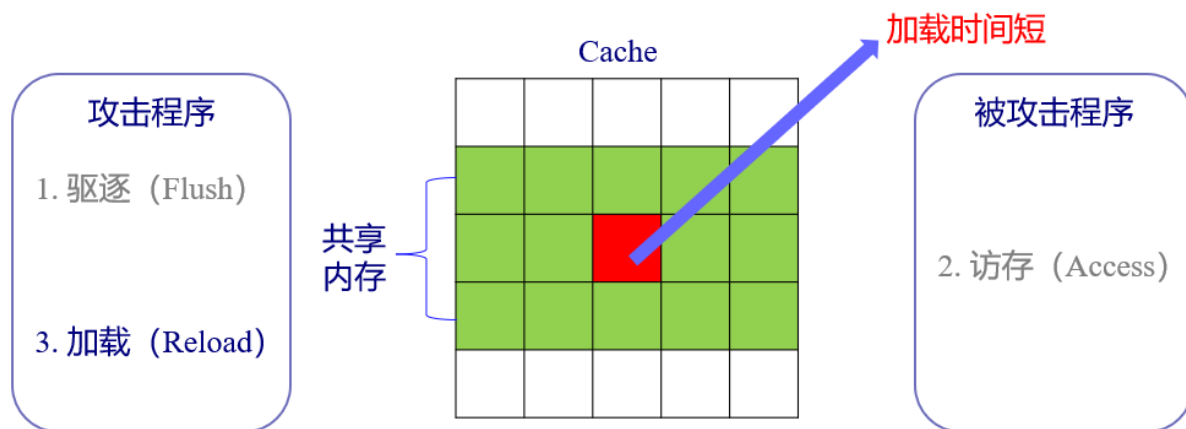


图 4 Flush + Reload 攻击

如图 4 Flush + Reload 攻击所示，攻击程序和受害者程序共享了一段连续的内存，攻击者首先将该段内存从 Cache 中驱逐出去，这样任何对连续内存的访问都会发生缺失，从而返回时间将会变得非常慢。驱逐完成后，如果受害者程序访问了连续内存中的某一块，那么该块将会被加载进入 Cache。攻击者如果在这个时候做一个重加载，逐个访问连续内存的块，将会得到其中某一个块的访问速度会比其他的块明显快。这样我们就可以知道攻击者在这段时间访问了哪段内存。这种利用方式可以加强到泄露出受害者在非共享区的访问内容：

```
1 ldr x1, [secret_addr]
2 ldr x2, [probe_array + (x1 * BLOCK_SIZE)]
```

图 5 Flush+Reload 的利用

如图 5 Flush+Reload 的利用所示，在步骤 1 中，受害者先访问自己独享的私密数据进入了 x1 寄存器，随后以 x1 乘以 Cache 块大小的结果作为偏移，访问共享内存的指定块。如果此前攻击者做了 Flush，随后做 Reload，就可以得到共享内存的第几个块被访问了，从而逆向出 x1 寄存器中的值。

在本实验中，同学们需要结合 Flush + Reload 侧信道和 Store-to-Load Forward 将 SpectreV4 复现出来。

三 实验内容

整个实验的文件组成如下：

```
.
|-- agu.h
|-- controller.h
|-- define.h
|-- logs
|   |-- debug.log
|-- lsq.cpp
|-- lsq.h
|-- main.cpp
|-- Makefile
|-- memory.cpp
|-- memory.h
|-- test.c
|-- tlb.cpp
|-- tlb.h
|-- trace
|   |-- spectrev4.trace
|   |-- trace0.out
|   |-- trace1.out
|   |-- trace2.out
|   |-- trace3.out
```

图 6 代码整体结构

1 实现内存访问过程中各组件的功能.

1.1 整体结构

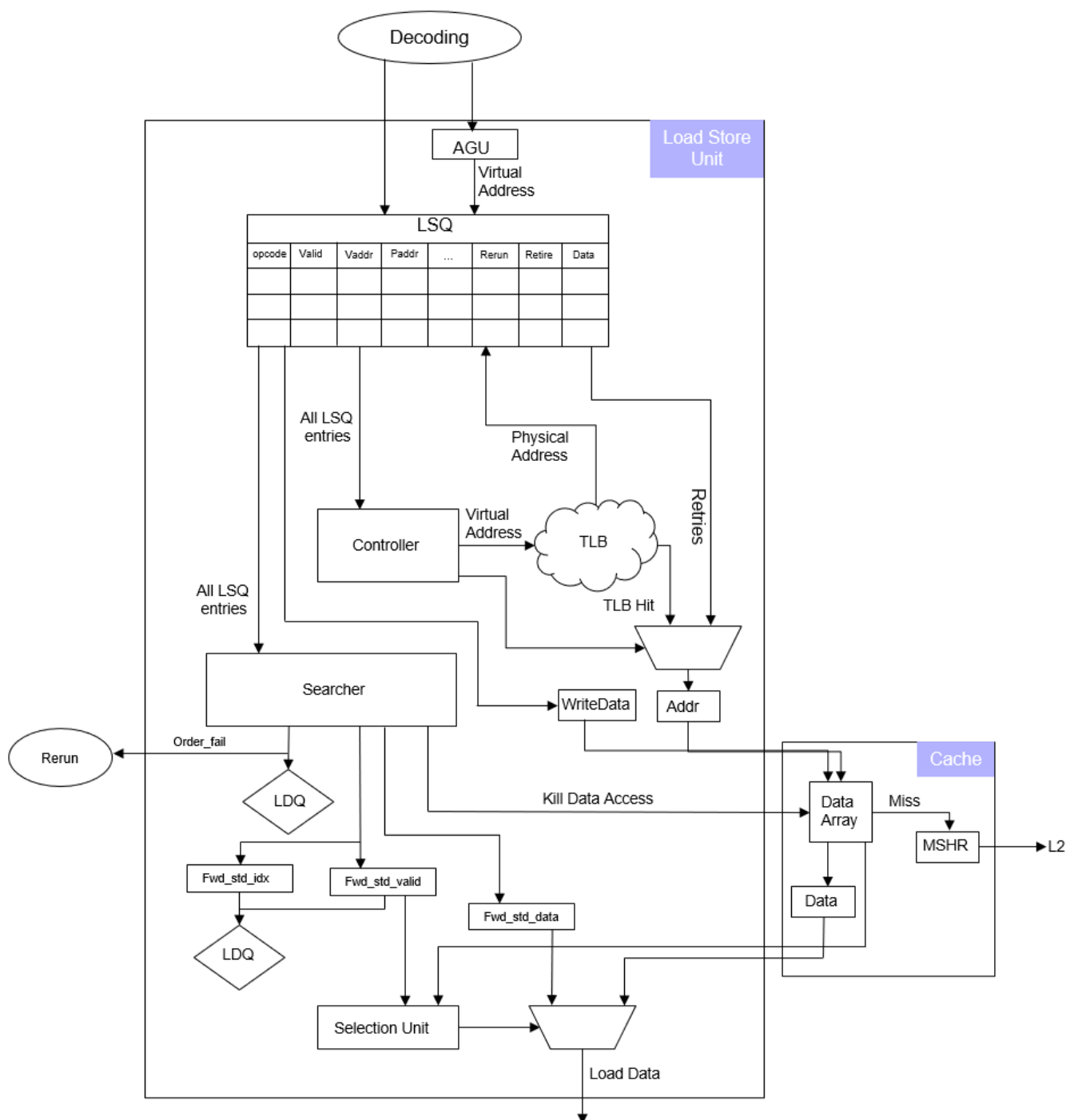


图 7 模拟器的数据和控制流

整体的流程为，获取输入指令，将其解码，然后将解码结果放入 LSQ (Load Store Queue)，并且通过地址生成单元 AGU 获得虚拟地址。在本实验中，由于只有两种模拟的寻址方式，AGU 的作用是根据这两种寻址方式来返回寻址时间。寻址完成之后，Controller 通过每一条 LSQ entry 来进行地址翻译，在本模拟器中，由于是事件驱动的，所以除了获取 TLB 的翻译内容，还需要 TLB 和 PageTable 返回所用掉的时钟周期数，以确定本指令的下一事件应该何时触发。

通过 TLB 获取到物理地址和时延之后，Controller 可以根据请求来进行内存层次的访问，和 TLB 同理，除了需要返回相应的数据，还需要获取时延，更新 LSQ 的相应表项。

对于 Load 指令，除了要进行访存，还需要消除 RAW 乱序的歧义。因此需要使用 Searcher 检索 LSQ，获取可能的旁路。如果旁路成功，那么将访存申请杀掉，不再往内存中检索。

其中虚拟地址长度和物理地址长度都为 14bit，页大小为 256B，LSQ 大小为 6 个 entry。Cache Block 大小是 64B。在 define.h 以及各头文件中定义了各组件的格式。更详细内容请参照源码进行阅读。

1.2 Cache

其中 Cache 的格式如下：

```
/******Cache line format*****  
*+-----+-----+  
*|      TAG      |RD|RA|V|D|  
*|      2        | 1| 1|1|1|  
*+-----+-----+  
*@TAG: Two most significant bits in the physical address  
*@RD: We use the round robin replacement algorithm for the cache,  
|    RD is set when this entry is recently write  
*@RA: set when this entry is recently accessed  
*@V: valid bit  
*@D: dirty bit  
*/  
#define CACHE_TAG_MASK          0X30  
#define CACHE_REP_ACCESSED_MASK 0X4  
#define CACHE_REP_DIRTY_MASK    0X8  
#define CACHE_VALID             0X2  
#define CACHE_INVALIDATE        0XFD  
#define CACHE_DIRTY             0X1
```

图 8 Cache Line Format

物理地址到 Cache Line 的映射为 4 路组相联。

Cache 的替换策略为 Round Robin, 使用两位来辅助判断, 低位为 rep_accessed, 高位为 rep_dirty。在进行置换的时候, 轮询所有的表项, 如果没有空项, 检索是否有表项的置换位是否为 00, 如果有将其返回。如果没有, 将所有 rep_accessed 置位 0, 再次检索是否有第一轮中的情况。如果这个时候还没有, 将 rep_dirty 位置零, 同时将相应数据写回内存。需要完成的函数在 memory.cpp 中, 如下:

```

/*****TODO*****/
/*actual execution on cache, find the data, and update related entries
*@paddr: the physical address
*@data: put the data load from memory to this reference
*return 1 if hit, 0 if miss
*/

int loadFromCache(Paddr_t paddr, uint32_t &data){

    uint32_t block_offset = paddr & PADDR_BLOCK_OFFSET_MASK;
    uint32_t set = (paddr & PADDR_SET_MASK) >> 6;
    uint32_t input_tag = (paddr & PADDR_TAG_MASK) >> 12;

    int idx = access_cache(paddr);
    //if cache hit
    if(idx != -1){
        cache_hits++;
        /*
        *TODO: what should we do if cache hit
        */
        return 1;
    }

    /*
    *TODO: what about miss?
    * load from the memory, update cache
    */

    return 0;
}

```

```

/*****TODO*****/
/*actual execution on cache, find the data, and update related entries
*@paddr: the physical address
*@value: store the value to specified location
*return 1 if hit, 0 if miss
*/

int store2Cache(Paddr_t paddr, uint32_t value){
    uint32_t block_offset = paddr & PADDR_BLOCK_OFFSET_MASK;
    uint32_t set = (paddr & PADDR_SET_MASK) >> 6;
    uint32_t input_tag = (paddr & PADDR_TAG_MASK) >> 12;

    int idx = access_cache(paddr);
    if(idx != -1){
        cache_hits++;
        /*
        *TODO: what should we do if cache hit
        */
        return 1;
    }

    /*
    *TODO: what about miss?
    * load from the memory, update cache
    */

    return 0;
}

```

其中写策略采用写回法搭配写分配法，详情请参考源代码。

需要注意的是，我们说的需要计算访问时延来更新 LSQ entry，但是这个时延的计算需要 Controller 提前完成以便注册事件。所以在实现的时候要注意计算时延并没有实际修改组件状态和内容，真正修改应该到相应的 Clock 进行。

1.2 TLB

TLB 和页表的格式如下：

```
/******TLB entry format*****  
*+-----+  
*|V|D|      PNO      |      FNO      |  
*|1|1|      6        |      6        |  
*+-----+  
*  
*@V: valid bit  
*@D: dirty bit  
*@PNO: virtual page number  
*@FNO: physical frame number  
* We use direct map between virtual page number and TLB entry.  
*/  
#define TLB_VALID_MASK          (1<<13)  
#define TLB_DIRTY_MASK          (1<<12)  
#define TLB_VADDR_MASK          0XFC0  
#define TLB_PADDR_MASK          0X3F  
#define TLB_INDEX_MASK          0XF00  
#define TLB_VALID_SHIFT          13  
#define TLB_VADDR_SHIFT          6  
#define TLB_INDEX_SHIFT          8  
  
/******Page Table entry format*****  
*+-----+  
*|V|      PNO      |  
*|1|      6        |  
*+-----+  
*  
*@V: valid bit, use for page replacement between disk and memory  
*@PNO: page frame number  
*Since we did not implement the Device, and there is no page fault,  
*therefore, the page table entry is always valid.  
*/  
#define PAGETABLE_VALID_MASK      (1<<6)  
#define PAGETABLE_FRAME_NUM_MASK 0x3F
```

虚拟地址到 TLB entry 的映射为直接映射。需要完成的函数在 tlb.cpp 中，如下：

```

/*****TODO*****/
/*get physical frame from input virtual address
*this function is just query function, did not change the function unit's state an
*return frame number if TLB hit, else return NULL(0).
*/

FrameNo_t getFrameNoFromTLB(Vaddr_t vaddr){
    int valid;
    FrameNo_t frame;
    PageNo_t page;
    PageNo_t input_page = getPageNo(vaddr);
    register int i = (vaddr & TLB_INDEX_MASK) >> TLB_INDEX_SHIFT;

    /*
    *TODO
    */

    valid =
    page =
    frame =

    if(DEBUG) printf("input page %x, tlb entry page: %x\n",input_page, page);

    if(valid && input_page == page){
        tlb_hits++;
        return frame;
    }

    // the vaddr of 0 refers to NULL, as well as the first virtual page,
    // dereference the NULL address will cause an exception
    // the initial address input does not contain any of them
    return 0;
}

```

```

/*****TODO*****/
/*@vaddr: virtual address
*@page: virtual page
*@frame: physical frame
*/
void updateTLB(Vaddr_t vaddr, PageNo_t page, FrameNo_t frame){
    /*
    *TODO: you need to finish the virtual address map to the TLB index
    *      and then use the direct map strategy to update TLB entry
    */

    int valid = (tlb[i]&TLB_VALID_MASK)>>TLB_VALID_SHIFT;

    if(DEBUG) printf("updated tlb entry: 0x%x, valid: %d\n", tmp, valid);
}

```

TLB 和 Cache 类似，需要将查询和修改接口解耦，以便更新 LSQ。其他详情请参考源码注释。

2 实现 Store-to-Load forwarding.

Load 旁路的数据来源有三种，一种是正确的旁路, 另一种是旁路了旧数据, 最后一种是从内存获取了数据。后两种被称为 Ordering Failure。该 Failure 的检查是在每个 Store Commit 的时候。

完善源码 controller.h 中所有 TODO 部分。并且回答 controller.h 中的 THINKING 问题。

3 实现 SpectreV4 的模拟实验

完善 main.cpp 和 agu.h 中的所有 TODO 部分。

4 测试用例的格式和设计

本实验包含两种测试用例，一种是以.out 为扩展的文件，这四个文件作用是为了验证你的 TLB 和 Cache 的 miss rate。其中只包含两种指令，一种是 ldrb，另一种为 strb，指令第二个操作数表明存储的数据，最后一个操作数表明是操作的虚拟地址。两种寻址方式，却别在于是否在虚拟地址上加入了[]，[]表示间接寻址，寻址时间增加 10 个 Clock。如图所示：

```
ldrb [0x2378]
ldrb 0x1692
strb 0x22 0x27a2
strb 0xed 0x22ff
strb 0xce 0x1cce
```

图 9 输入指令格式样例

除了.out 文件外，还有一种测试样例为.trace 结尾的文件，这种文件用作测试你的实验的各种单元功能和瞬态行为的正确性，一般不会超过 10 条指令，这种测试用例也就是需要同学们进行辅助设计的样例。这种样例的正确性评估是根据 DEBUG trace 来进行评估。其中指令也是只有 ldrb 和 strb 两种。

除此之外，在 SpectreV4 测试中还加入了两种指令如所示，flush 和 pldr。Flush 指令比较简单，只需简单的清空 cache。Pldr 本质上是一条 load 指令，它的操作数 idx 操作数指明了它的虚拟地址的生成依赖于 lsq 中位置为 idx 的指令执行结果。一旦该指令结果所依赖的值计算完成，那么 AGU 将会给该 load 指令生成真正的虚拟地址。

```
1  flush
2  str 0x5 0x1234
3  str 0x6 [0x1234]
4  ldr 0x1234
5  pldr 3
```

图 10 SpectreV4 Gadget

如图中的 pldr 3 指明了本 load 指令依赖于 lsq 中 index 为 3 的 load 指令的执行结果。保证除了 SpectreV4 的测试 trace 需要该指令，其他情况不会出现。需要注意的是，指令重发的时候是占据该指令原本在 LSQ 中的位置，因此 pldr 指令重发的依赖也不应该修改。

本部分为开放性实验，在本实验中，请设计一个测试用例，验证 load 指令所取的数据是否正确，或者是 store 的数据是否合法写到内存，或者是瞬态窗口中的操作是否合法。并且给出你的评价方法。

四 评价标准

本次实验的基础分共 20 分。

1 评分细则

表 4-1 评分细则

任务	子项	评分/pts
Task1	TLB 以及页表的访问	1
	Cache 以及物理内存的访问	1
Task2	在 LSQ 中的调度 ^[a]	5
	访存序列的 Profile ^[a]	2
	实现 Store-to-Load forwarding	2
Task3	Cache Encoding + Flush + Reload	2
Task4	设计测试用例	1
report	模拟器和调度的设计说明	1
	测试用例的说明	1
	回答问题 ^[b]	3
	对实验的意见与建议	1

[a]见第 2 小节，[b]见第 3 小节

2 正确性评价

访存序列的 Profile 正确性评价，评价指标为在给定访存序列中，根据你的统计数据进行给分，最后还会综合全部同学的实验结果进行加权，最高不超过 2 分。

调度正确性的评价标准主要针对两个方面，一个是指令执行的正确性，表现为 load 指令的输出结果正确与否。另一个是微架构上的行为正确性，表现为各个操作的开始和完成时间。评价标准将会以输入指令序列的执行结果以及 DEBUG 中的 trace 来进行评判, 每个指令序列不超过 10 条指令。分值将会以通过的测试样例个数折算，总测试样例个数会根据同学们提供的 Hacking 数据进行调整。其中如果有同学提供的数据将别人的模拟 Hacked 掉，将会获得 0-2 分的加分。

3 回答问题

请从以下几个问题中选择三个感兴趣的问题回答。

(1) 在本实验中仅仅针对 RAW hazard 进行了内存消歧，我们是否需要针对 RAW，WAW，WAR 进行消

歧？如果需要请给出设计思路，并且使用简单样例来验证你的加速效果；否则给出不需要的理由。

- (2) 在瞬态窗口中的操作原则是不能影响宏观结构上的组件，避免瞬态数据的泄露。但是在实验二中我们依旧使用侧信道将数据提取出来了，请分析原因。
- (3) 请结合你的模拟器的实现，分析要保证瞬态执行结果不被系统程序员获取应该做出怎样的限制。
- (4) 除了实验二中的 SpectreV4，我们是否有其他的利用行为？可以结合通用寄存器，内存，外设等组件进行联想。
- (5) 给定一个平台（或处理器），应该如何**挖掘**平台上的瞬态执行漏洞？给出你的设计。
- (6) 给定一个平台（或处理器），应该如何**验证**平台上是否存在，以及存在何种瞬态执行漏洞？给出你的设计。
- (7) 如何从一个设计者的视角验证你的设计的正确性？给出你的验证点以及评价指标。

五 参考资料

- [1]. <https://developer.arm.com/documentation/102105/latest/>
- [2]. VINTAN, L. N., AND IRIDON, M. Towards a high performance neural branch predictor. In Neural Networks, 1999. IJCNN'99. International Joint Conference on (1999), vol. 2, IEEE, pp. 868–873.
- [3]. Canella C, Genkin D, Giner L, et al. Fallout: Leaking data on meltdown-resistant cpus[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 769-784.
- [4]. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639>
- [5]. Yarom Y, Falkner K. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack[C]//23rd {USENIX} Security Symposium ({USENIX} Security 14). 2014: 719-732.