皂基调研

大实验基础

在 lab5 的流水线处理器的基础上，只需要再把实验 1 中的指令实现就行，应该不太难
需要实现的指令：

> 基础版本的 Kernel 共使用了 19 条不同的指令，它们是：

```
ADD   0000000SSSSSssssss000ddddd0110011
ADDI  iiiiiiiiiiiiiissssss000ddddd0010011
AND   0000000SSSSSssssss111ddddd0110011
ANDI  iiiiiiiiiiiiiissssss111ddddd0010011
AUIPC iiiiiiiiiiiiiiiiiiiiiddddd0010111
BEQ   iiiiiiiSSSSSssssss000iiiii1100011
BNE   iiiiiiiSSSSSssssss001iiiii1100011
JAL   iiiiiiiiiiiiiiiiiiiiiddddd1101111
JALR  iiiiiiiiiiiiiissssss000ddddd1100111
LB    iiiiiiiiiiiiiissssss000ddddd0000011
LUI   iiiiiiiiiiiiiiiiiiiiiddddd0110111
LW    iiiiiiiiiiiiiissssss010ddddd0000011
OR    0000000SSSSSssssss110ddddd0110011
ORI   iiiiiiiiiiiiiissssss110ddddd0010011
SB    iiiiiiiSSSSSssssss000iiiii0100011
SLLI  0000000iiiiiissssss001ddddd0010011
SRLI  0000000iiiiiissssss101ddddd0010011
SW    iiiiiiiSSSSSssssss010iiiii0100011
XOR   0000000SSSSSssssss100ddddd0110011
```

提升

问了计一年纪的 lch 和 Fuyuki，lch 说 cache + 监控程序3 的难度 << 跑 ucore 的难度 (说相对大概快一周写完)
lch 建议的分工： csr & 中断* 1 + 页表虚存 * 1 + cache * 1
Fuyuki 当年造的是超标量流水线，也没有跑 ucore

下面是关于两者所需要实现功能的一些调研

监控程序 ver3

> ver3要实现M态和U态的切换，几个M态的csr，时钟中断，MMU -- chgg
> 看实验文档上需要实现如下功能：中断和异常，页表和虚拟地址

中断和异常实现

笔记

- CSR 指令格式

    - [31:20] csr + [19:15] rs1 + [14:12] funct3 + [11:7] rd + [6:0] opcode

   alt_text

    - CSRRS: 读取 csr 的值，零扩展到 32 位，写入 rd，原本 csr 的值按位 | rs1 的值，写回 csr
    - CSRRC: 读取 csr 的值，零扩展到 32 位，写入 rd，原本 csr 的值按位 & ~rs1 的值，写回 csr
    - CSRRW: 读取 csr 的值，零扩展到 32 位，写入 rd，rs1 的值写回 csr。当 rd 是 x0 时 直接 rs1 的值写入 csr 而不读取

- CSR listings

    - mtvec: number: 0x305 priviledge: Machine ReadWrite
    - mscratch: number: 0x340 priviledge: Machine ReadWrite
    - mepc: number: 0x341 priviledge: Machine ReadWrite
    - mcause: number: 0x342 priviledge: Machine ReadWrite
    - mstatus: number: 0x300 priviledge: Machine ReadWrite
    - mie: number: 0x304 priviledge: Machine ReadWrite
    - mip: number: 0x344 priviledge: Machine ReadWrite
    - 注：这里的 number 类似于 id，是用于 CSRRS 指令指定 csr 的

- 异常读写域的处理

    - WPRI: Write Preserve Read Ignore
        - 字面意思：写入时保留，读取时忽略
    - WLRL: Write Legal Read Legal
        - 软件只能往里面写合法值
        - 除非上一次写入了合法值，否则读取的值是不能保证合法的
        - 不要求抛异常，对于非法值的处理类似于未定义行为（个人理解）
    - WARL: Write Any Read Legal
        - 软件可以写入任何值
        - 读取时只能读取合法值，如果上次写入非法，则读取的值是未定义的合法值（但是理论上应该和那个写入的非法值相关）
        - 写入非法值不要求抛异常

- mstatus

    - constrols the hart's current operating state
    - 参考 riscv privileged spec P31 它的各个字段记录了中断的信息，如 MPP 介绍了中断前的 privilege mode，MIE 允许了 machine mode 的中断，MPIE 记录了中断前的 MIE (它写的很复杂，但是个人感觉如果只有俩特权级的话应该还是比较简单的)

- mtvec

    - 组成：base[31:2] + mode [1:0]
    - 记录 trap vector configuration
    - base 记录 exception handler（或一定偏移）的地址，mode 是 0 时，中断将 pc 置为 base，否则 mode 是 1 时，中断将 pc 置为 base + 4* cause

- mip & mie

    - mip 记录 pending 的中断，mie 记录中断使能

    - mie 中的 MSIE，SSIE，USIE 分别是 machine,supervisor,user mode 的使能
    - mip 中的 pending bits 为 1 时，代表软件发出了信号需要处理器处理
        - 分 （user mode,supervisor mode,machine mode）(Software Interrupt Pending Bits,Timer Interrupt Pending Bits,External Interrupt Pending Bits(比如外设啥的))
    - 交互方式：
        - MTIP 只有在写 memory-mapped machine-mode timer compare register 时会被写，UTIP 会被 M mode 软件写去给 usermode 发时钟中断
        - MSIP 感觉在我们这里没啥应用场景（ ）SSIP 可以被 machine mode 的 csr 设置访问

- mscratch
    - 一个用于上下文切换时给 trap handler 保存临时值的地方，惯例是存一个 machine mode 上下文空间的指针
- mepc
    - 由中断进入 machine mode 的时候，保存被中断或者遇到异常的指令的虚拟地址，是 WARL 的
- mcause
    - [31:31] interrupt + [30:0] Exception Code (WLRL)
    - interrupt 域表示进入 M-mode 是否由中断产生，Exception Code 和上述 mtvec 用于 handler 地址计算
- mtime & mtimecmp
    - 当 mtime 的值大于 mtimecmp 的时候会产生 timer 中断，当写 mtimecmp 的时候中断取消
- ecall & ebreak
    - ecall 只产生一个 exception (感觉像是 syscall 一类的)，ebreak 是给 debugger 用的中断（类似于 Windows int 3）
    - 在 ecall/ebreak 时，mepc 会被设置为 ecall/ebreak 指令的地址
- trap return instructions
    - pc 设置为 mepc 中的地址，此外，还要更新 mstatus 中的 bits
- satp
    - 保存当前指令指针

问题理解

1. 监控程序涉及 M 态和 U 态，监控程序工作在 M 态上，像是 shell.S 中的第 252 行，通过 mret 指令进入用户态
2. CSR 寄存器变化：
    1. 在 init.S 中，将 EXCEPTION_HANDLER/VECTORED_EXCEPTION_HANDLER 地址赋给 mtvec
    2. init.S MIE_MTIE 置位
    3. init.S 设置 mscratch 为当前线程 TCB 地址

4. 执行 G 命令执行用户程序时，是如何切换内核态，并跳转到对应位置的?
    1. 用户程序入口写入EPC
    2. 对 mstatus 设置 previous priviledged mode
    3. 设置 mtimecmp，但是在监控程序中直接用一个 `CLINT_MTIMECMP` 地址表示 mtimecmp
    4. mret 时， pc 会被赋值为 mepc 的值，即是用户态程序入口，从用户态回到内核态的时候，`jr ra` 跳转到 `USERRET_USER`

3. 用户程序执行 ecall 指令之后会发生什么

    1. "初始化时设置 mtvec = EXCEPTION_HANDLER，使用正常中断模式（MODE = DIRECT）",
    2. **看上述 mtvec 的功能， ecall 的时候会把 pc 指向 mtvec 中存放的语句，于是就这样做就行了**

4. 如何在异常处理函数中判断当前处理的异常发生在什么态？来自 M 态的时钟中断会发生吗？

    1. 利用 mstatus 里面的 MPP 域，看是否被置位推测是 kernel 还是 user mode
    2. 来自 M 态的时钟中断不会发生，因为直接到 context_switch 然后返回了

5. 中断发生的具体条件是什么？监控程序中发生的中断类型是？

    1. 给一个笼统的说法就是，可以触发异常的事件，就像 syscall，除0，还有调试器中断之类的
    2. 具体来说，在咱们的监控程序中，有如下条件： TIMER, INT, ECALL, BREAK
    3. timer 的话，因为它写了 mtimecmp，所以在 mtime >= mtimecmp 的时候触发
    4. ecall 就是正常的 ecall 指令调用，
    5. ebreak 是在 .USERRET_USER 里面调用的，会跳到 `USERRET_MACHINE` 完成栈的恢复之类的再回到 shell 的交互
    6. 好的吧，这里把中断和异常放到一起说了，简单来说，异常是可以找得到具体指令对应的 但是中断不行，我们需要考虑的异常和中断只有 **ecall ebreak timer 缺页异常** 四类，其中前两者为异常后两者为中断

6. 用户程序在执行完成后（执行 jr ra 指令回到监控程序后）会发生什么

    1. `.USERRET_USER:` ->`EXCEPTION_HANDLER` -> `HANDLE_BREAK` -> `USERRET_MACHINE` -> 保存用户态寄存器，恢复当前栈 -> 回到 shell 的下一轮交互

7. 如何判断 CPU 当前运行在什么内核态？这个信息有暴露给软件吗？

    1. 有点没明白这个想问啥
    2. 如果我们没实现 supervisor mode 是不是就没有这个问题了。。

8. 如何读写 mtime 和 mtimecmp 寄存器

    1. 在 common.h 中定义了这俩寄存器的地址，直接读写对应地址上的数即可
    2. 更类似于读串口的状态吧

9. 监控程序在处理 mtvec 寄存器时使用了复杂的逻辑，解释其原因

    1. 首先是因为它有两个 mode，如果 direct mode 设置失败的话还可以设置 vectored mode
    2. 虽然是 WARL 但是一般还是会保证如果是非法值的话 读出还是会和正常值不一样，所以 `beq t0, s0, mtvec_done` 可以正常跳转

csr 和读写指令

不是很明白流水线的冲突处理（逃），到时候可以一起研究一下
在文档里描述的处理冲突的位置中，问了 chgg，他说是在 mem 做的，这样的话需要参考 load-relate 类数据冲突进行额外处理（虽然对流水线不是很了解，但是但看文档也感觉在 mem 段做会简单吧）

基础异常处理

chgg 也是在 mem 段做的
看到监控程序3中，我们需要关注的异常主要是 ecall, ebreak 这两条指令，然后我们需要做这几件事：pc <- mtvec; mstatus， mip 置位; mepc <- 当前地址
所有寄存器的赋值一定在同一个时钟上升沿完成，即是文档中所述之原子化

中断处理

需要考虑的中断是 timeout 的中断，此外，之后可能涉及到缺页的处理
文档中流程：

- IF 段取指令时 / ID 段译码时，检查当前是否满足中断发生的条件，并在发生中断时给指令带上异常标记。（这一步对于 timeout 来说，之前都有每个固定时间 mtime <- mtime + 1 的步骤，如果发现 mtime == mtimecmp 的时候给 cpu 设置一个标志时钟中断的 flag 在这里读取该 flag 是否正常就行）
- 进入 EX 阶段后，跳过该指令在 EX 段的执行
- 在异常处理阶段接受中断，修改对应的 CSR 寄存器

页表实现

笔记

satp

存储了根页表的 PPN (physical page number), address space identifier, MODE 域
对 RV 32 的话，我们 mode 是 Sv32，如果有 RV64 的话 mode 是 SV39

pmpcfg0,pmpaddr0

PMP unit 可以为每个物理内存区域单独指定访问权限（rwx）
省流：PMP 可以给 S mode,U mode 权限或者限制 M mode 权限
pmpcfg0 是由 [pmp3cfg, pmp2cfg, pmp1cfg, pmp0cfg] 拼接成的
pmpaddr0 存下了 rv32 物理地址的 [33:2] 位
组成：L[7:7] + 0[6:5] + A[4:3] + X[2:2] + W[1:1] + R[0:0]
A 域和对应的 pmpaddr register 一起定义了保护内存的范围
L 如果被置位，则下一次 reset 才可复位，期间对 pmpicfg pmpiaddr 的写都失效
对于我们的监控程序，可以看到对 pmpaddr 设置为 0xffffffff，对 pmp0cfg 设置为 0b00001111 它 addr matching 是 1，是 TOR 的 matching 方式，所以该权限生效的物理地址区域是全部的，代表直接给所有物理地址赋了 rwx 权限（这完全不考虑内存安全了这 哈人）
所以纯从正确性来说，我们可以就实现一个可以被赋值的 dummy register 然后不管他（逃

sv32

S-mode 和 U-mode 的虚拟地址都通过一个 page table 转化为 S-mode 的物理地址，但是 S-mode 除了 satp 外的别的都不需要实现

虚存地址用 VPN (virtual page number) + page offset 表示

- 两级的页表查询： 20 bit VPN -> 22 bit PPN; PMP 检查权限; supervisor-level physical addresses -> machine-level physical addresses
- 虚拟地址表示：VPN[1] [31:22] + VPN[0] [21:12] + page_offset [11:0]
- 物理地址表示: PPN[1] [33:22] + PPN[0] [21:12] + page_offset [11:0]
- page table entry: PPN[1] [31:20] + PPN[0] [19:10] + RSW[9:8] + D + A + G + U + X + W + R + V
- RWX 代表了 read write execute 的权限，DAG，RSW 都不用实现， V 代表 page table entry 是 valid 的，如果不是的话，其他位都可以被软件自由使用。U 代表用户态的 software 可以 access 该页表
- 地址转换：
    - STAGE_0: a := satp.ppn×PAGESIZE; i := 1 (PAGESIZE = 4096)
    - STAGE_1: pte := a + va.vpn[i]×PTESIZE (PTESIZE = 4)
    - STAGE_2: If pte.v = 0, or if pte.r = 0 and pte.w = 1, stop and raise a page-fault exception
    - STAGE_3: if pte.r = 0 or pte.x = 1 goto STAGE_4 else i = i-1 ; if i<0 raise a page-fault exception; else a = pte.ppn×PAGESIZE goto STAGE_1
    - STAGE_4: check is legal access, if not, raise a page fault exception
    - STAGE_5: if i > 0, pte.ppn[i−1 : 0] != 0, raise page fault
    - STAGE_6: (calc): pa.pgoff = va.pgoff (偏移量相等)； if i > 0 pa.ppn[i − 1 : 0] = va.vpn[i − 1 : 0]; pa.ppn[LEVELS−1 : i] = pte.ppn[LEVELS−1 : i]
    - 拼出来的 34 位物理地址可以直接去掉最高的两位当作 32 位地址进行使用

问题回答

0. 我们什么时候需要做页表查询：**在 U-mode 下**需要访问地址的时候就要做，就像是在判断当前是 U-mode 的时候 在与 sram 交互的时候套一层虚拟-物理地址转化

1. 监控程序使用的是动态页表映射还是静态的？
    - 监控程序使用的是静态页表映射

2. 监控程序本身使用的地址是虚拟地址吗？
    - 监控程序使用的是物理地址（对于监控程序的代码段，虚拟地址和物理地址相同，对于串口，应该访问的是物理地址）

3. G 指令指定的用户程序地址应为虚拟地址还是物理地址？A 指令呢？
    - **我们考虑取址的位置 在用户态读取的地址都是虚拟地址 在内核态读取的地址只能是物理地址**
    - G 指定虚拟地址，因为我们在g指令填写的地址是会被放到 mepc 上，mret 之后进入 U-mode， if 段从mepc开始取指，而且在u态，所以是在 U-mode 读取指令地址
    - A 指定物理地址，因为内核态没有页表，只能看到物理地址

实现思路

- 首先我们需要维护当前是用户态还是内核态，只有在用户态才会查询地址

- 然后我们在判断当前为 U-mode 且需要访问 sram 的时候加一层地址转换如上图算法所示，应该不会很复杂感觉（ ）
- 冲突处理：按照上面的说法，我们如果在 MEM 段处理异常，则需要在检查完页表之前，阻拦潜在的 EX 段跳转和 CSR 写入

cache

需要实现两部分 指令缓存（ICache）数据缓存（DCache）

ppt 笔记

Cache 是高速静态存储器，缓存了 cpu 频繁访问的信息
有四个点需要注意：

- 如何根据主存地址得到Cache中的数据
- Cache中的内容是否已经是主存对应地址的内容
- Cache中的内容与主存内容以多大的粒度交换
- 如何提高 Cache 命中率

defs

- 块：数据交换的最小单位（典型大小是 4 ~ 128 bytes）
- 命中（hit）：cache 上找到了对应的块 (典型命中率是 80% ~ 99%)
- 缺失（miss）: 要在较低层次访问块（指通过总线访问 sram 之类的）
- 命中时间 << 缺失访问的时间
- cache 容量的典型数据是 1 ~ 256 KB

两路组相联

全相联和直接映射都有明显的硬伤，而这个方案在 ppt 上的说法是常用且折中，所以就先研究它了呀

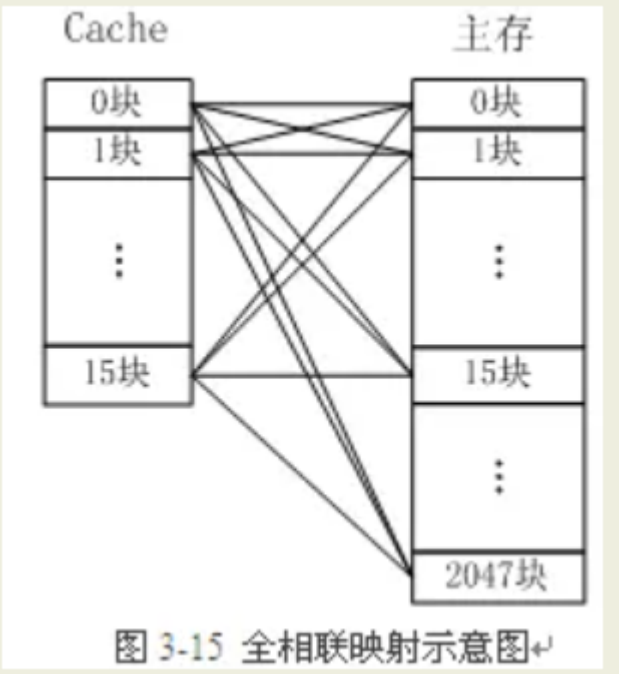
- 全相联
  - 

图 3-15 全相联映射示意图

    - 就是主存中的任意一个块可以装到 cache 中的任意一个块中去，问题是标志位较长，比较电路的成本太高
    - 每一个 entry 的结构是 有效位 + 标记 + 数值，其中每个字段的大小计算如下：
        - 有效位标志该 cache 中的数据是否有效，就一个字节
        - 标记和 **块 index** 位数相同，就像如果有 2**32 个块，则位数为 32
        - 数值和块大小相同，如如果一个块是4字节 则数值也是4字节

- 直接映射

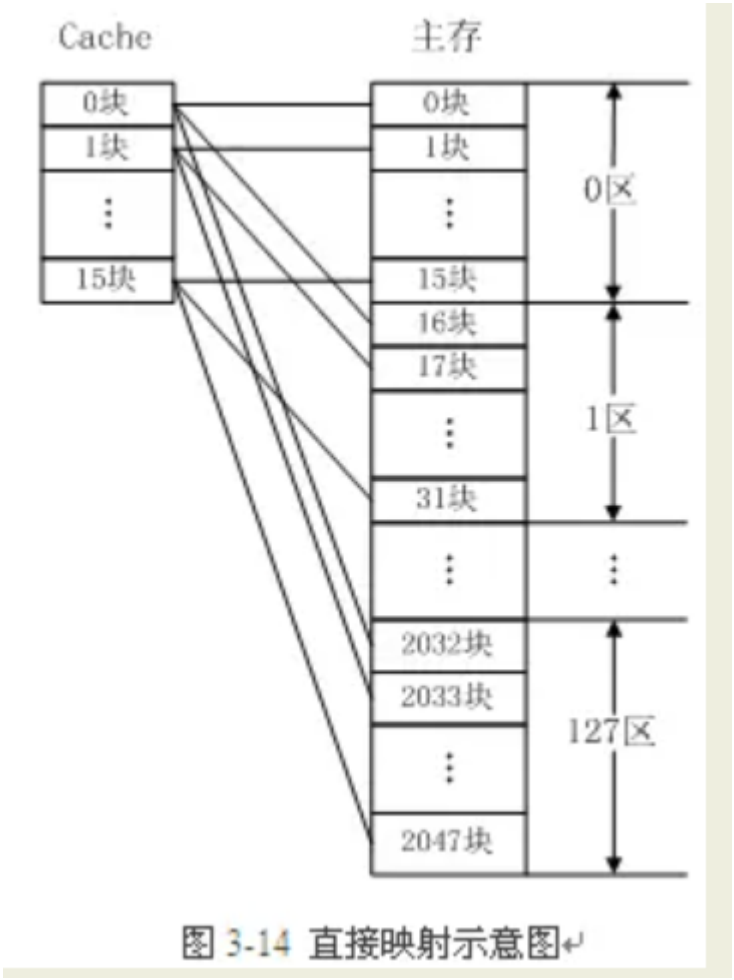
    - 

图 3-14 直接映射示意图
    - 每一个区中的块只能装到主存中 index 相同的块中（*内存中每个单元在 cache 中只会有唯一的位置与其对应*），问题是利用率和效率低
    - 每一个结构的标志是 valid + tag + 数值，但是和之前不同，将 sram 的地址拆成了 tag + index 的形式，比如如果 cache 是有 2**10 个块，则 tag 是 2**
    - 对于 tag 的大小，比如说 一个块是 4 字节的话，那么一个地址的表示可以拆成 [31:12] tag + [11:2] idx + [1:0]offset，这样的话 tag 只用 20 位就够了

- 组相联

    - **主存和Cache都分组，主存中一个组内的块数与Cache中的分组数相同，组间采用直接映射，组内采用全相联映射**
    - 将Cache分成u组，每组v块，对于一个主存块，它存放到哪个组是固定的，至于存到该组哪一块则是灵活的。
    - 给个参数 from chgg：4路组相连，总大小1KiB，一行是 32B
    - cache 中的数据是以**缓存线**（line）为单位组织的，一条缓存线对应于一段连续内存，这些线被保存在**路**中，每一路都有一个专门的目录（directory）用来保存一些登记信息

- 如果把每一路连同它的目录视为电子表格中的一列，则表的一行构成了cache的一**组**。列中的每一个单元（cell）都含有一条缓存线，由与之对应的目录单元跟踪管理
- cache 认为 物理内存按页分配，这个 page 和页表的 page 不同，每一页的缓存线数 == 4k/行的大小
- 在组相联中，内存中一条给定的缓存线只能被保存在一个特定的组（或行）中，就如任意物理内存页的第0条缓存线必须存储到第0组，以此类推，如果是 n 路组相联，则每组有 n 个单元可以插入缓存线
- 查找地址时，如按 chgg 的参数，则缓存线编号 == addr [11:5]，而我们还需要知道是一组中哪个单元含有所需要的数据，这时候我们考虑每一路的目录，每一个缓存线都被其对应的目录单元做了标记（tag）；这个标记就是一个简单的内存页编号，指出缓存线来自于哪一页，我们对每一路的目录单元进行查询，找到了对应的 tag 即可

失效处理

从主存中取出新块，替换出一个Cache行然后插进去

一致性保证（Q2）

因为一致性主要出现在写之后 cache 和 sram 数据不一样的情况，所以对于指令缓存只有读的情况就不用考虑了

- 写直达（write through）
  - if hit: 主存和cache同时写入（这个应该比较直观）
  - else 写分配/非写分配，其中，写分配是指在 cache miss 的时候，在写内存的同时在 cache 上分配空间然后写进去，非写内存则在该步直接写入主存，不管 cache
  - 个人认为写分配更合理

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate Hit	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

  - 看图上 fetch on miss 代表当遇到 cache miss 的时候，从内存中取出完整的块，更新数据后写入 cache，之后写回 mem，而 no fetch on miss 则直接更新部分 cache 然后写入 mem（问题，如果

做 no fetch on miss 的话 是不是需要维护 cache 中的块哪些字节是有效的？）

- 拖后写（write back）
  - 弱一致性保证，替换时再写主存 (主动/被动替换)
  - 通过监听总线上的访问操作来实现，较为复杂但是效率高（是在 DCache 的进阶功能里面）

cache 缺失

- 原因：必然缺失（开机/切进程/首次访问数据块），容量缺失（active 的数据总大小 > cache 大小），冲突缺失（多个内存块映射到同一Cache块，使得有行满了），无效缺失（其他进程改了数据，这里不用考虑qaq）
- 替换策略：感觉 FIFO 不错

（TODO）cache 接入系统的体系结构

是不是可以把 cache 实现为一个 sram_controller wrapper？ 杰哥没回消息qaq