一生一芯 双周分享汇报 —— NPC中 icache 的参数配置与性能优化

icache 设计的选择

1. `icache` 的硬件结构： `k` **路组相联**

- 可配置的参数：
  - `nr_way`，即一个 `cache set` 包含多少个 `cache line`；
  - `nr_set`，即多少个 `cache set`；
  - `line_sz`，即一个 `cache_line` 的大小。

代码块 1. MARKDOWN

```
+-----------------+---------+---------+
|    Tag          |  Index  | Block   |
|                 |         | Offset  |
+-----------------+---------+---------+
  (32-m-n) bits     m bits    n bits
nr_way  = k
nr_set  = 2^m
line_sz = 2^n
```

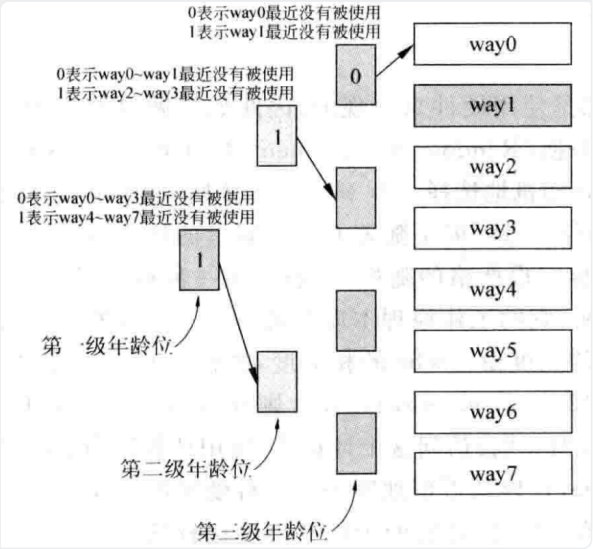
代码块 2. SCALA

```scala
// 使用寄存器阵列
val data = Reg(Vec(nr_way, Vec(nr_set, UInt((line_sz * 8).W))))
val tag  = Reg(Vec(nr_way, Vec(nr_set, UInt(tag_width.W))))
// 使用 SRAM
val data = VecInit.fill(nr_way)(SyncReadMem(nr_set, UInt((line_sz * 8).W)))
val tag  = VecInit.fill(nr_way)(SyncReadMem(nr_set, UInt(tag_width.W)))
```

2. `icache` 的替换策略——实现开销比较小的替换策略：

- FIFO(先进先出)
- Random(随机替换)
- Pseudo-LRU(伪LRU)

图 1. image-20240811154216924



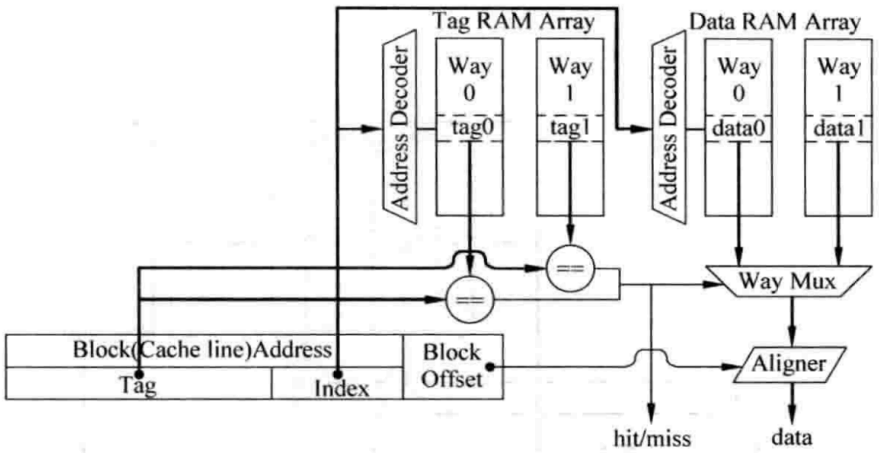
3. `icache` 的访问方式：`tag`和`data`部分**并行访问 vs 串行访问**

   并行访问：如果命中，可以当拍读出命中的数据！但是需要根据`tag`的比较结果对读出的`data`进行选择，时序开销比较大；

   串行访问：如果命中，第一拍读`tag`，第二拍读`data`。但是可以有较高的时钟频率。

   将读`tag`和读`data`放在两级流水线中完成，仍然可以实现每拍读取指令的效果。

图 2. image-20240811154102601



4. 不同结构配置下的性能测试结果（`RISCV32IM` 指令集，使用PLRU替换策略，阻塞 `icache`，运行 `microbench` 的 `train` 测试）

   a. `cache set` 数量和 `cache_line` 大小变化的影响

表 1. nr_wanr_se...周期数 AMATCP

nr_way	nr_set	line_sz	icache(data)大小	icache 命中率	取指平均周期数 AMAT	CPI
4	8	8	256B	0.96	1.65	2.88
4	16	8	512B	0.98	1.42	2.69
4	8	16	512B	0.99	1.41	2.70
4	16	16	1KB	0.99	1.34	2.64
4	32	16	2KB	1.00	1.29	2.59
4	16	32	2KB	1.00	1.29	2.59
4	32	32	4KB	1.00	1.26	2.56

   b. `icache` 路数变化的影响

表 2. nr_wanr_se...周期数 AMATCP

nr_way	nr_set	line_sz	icache(data)大小	icache 命中率	取指平均周期数 AMAT	CPI
2	16	16	512B	0.98	1.56	2.86
4	16	16	1KB	0.99	1.34	2.64
8	16	16	2KB	1.00	1.30	2.60

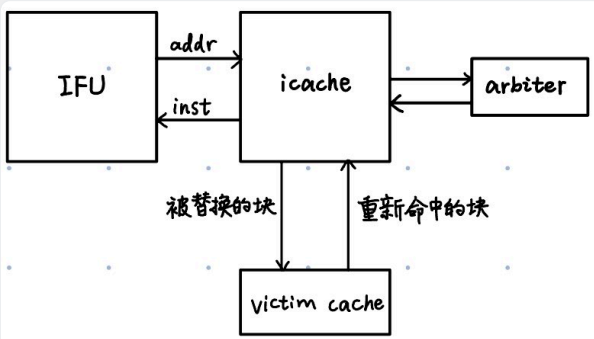
> 随着 `icache` 容量的增加，CPI 减小的幅度在变小；
>
> 需要平衡好性能和面积。

icache 性能的优化

1. 什么时候会发生 cache miss? `3C` **模型**: `Compulsory`, `Capcity`, `Conflict`

- 针对 `Compulsory`，即数据第一次访问的 cache miss，采用预取(`prefetch`)；
- 针对 `Capcity`，即因为 `cache set` 数量限制导致的 cache miss，增大 cache 容量，调整 cache 结构，以及优化替换策略...
- 针对 `Conflict`，即因为 `cache way` 数量限制导致的 cache miss，使用 `victim cache`。

图 3. 5D0285B1BCCC3A0E048698DB95CD9331(20240811-115046)



- `victim cache` 加入的收益: (`nr_way`=4, `nr_set`=16, `line_sz`=16)

表 3. victim cac...周期数 AMATCP

`victim cache` 的项数	`icache` 命中率	取指平均周期数 AMAT	CPI
0	0.99	1.34	2.64
4	0.99	1.33	2.63
8	0.99	1.32	2.63

2. 取指通路非阻塞的设计

- AXI 接口的两个重要性质：
  - **支持`outstanding`操作，即`master`不用等待上一个事务传输完成就可发送下一个事务的地址；**
  - **支持`out of order`操作，即`slave`返回数据时可不必按照`master`发送的地址顺序。**
- 相应地，非阻塞取指通路能够实现：
  - **不需要等待上一条指令取回，就可以继续发送下一条指令的取指请求；**
  - **后发的取指请求可能先返回指令。**
- `IFU` => `icache` => `xbar` => **SoC 上的设备**，其中的每一个部分都要支持 **AXI 的非阻塞传输**；
  - `master`端的`IFU`：

    1）`ar`通道和`r`通道独立控制；

    2）对乱序返回的指令进行缓存，然后顺序地传给`IDU`。

- `icache`既是`slave`，又是`master`！

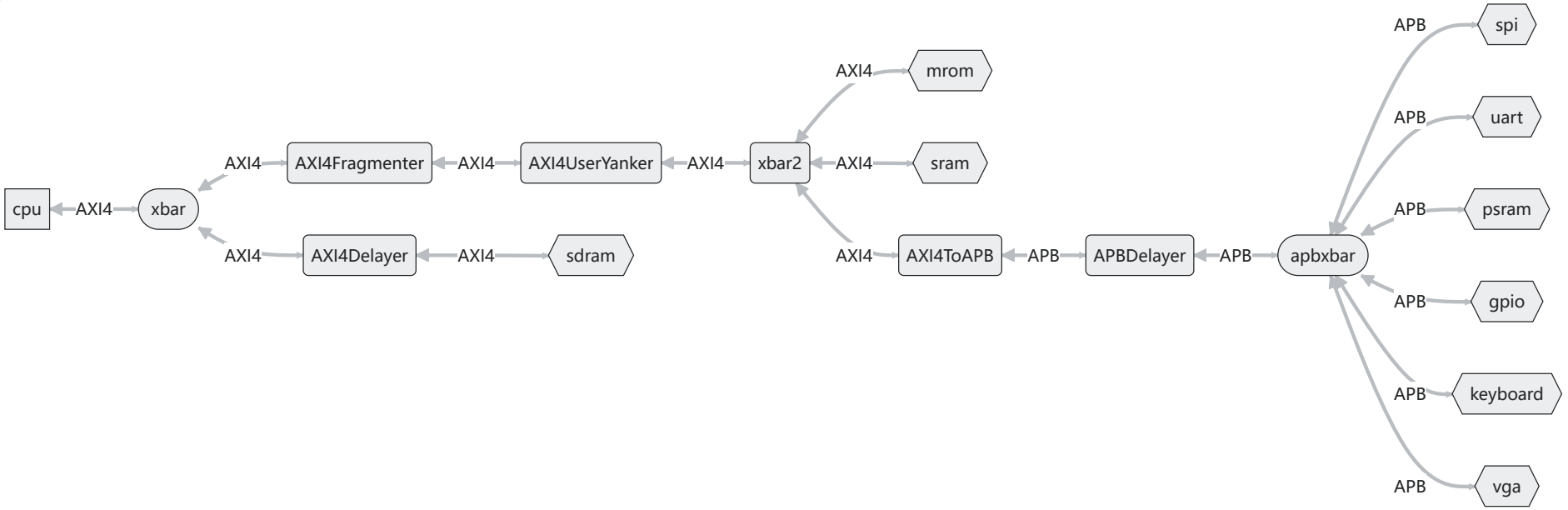
  1）对 cache hit 的请求，响应的时间是固定的，不需要乱序返回；
  2）对 cache miss 的请求，通过 **MSHR** 寄存器保存多个未命中的请求（还需要对落在相同 cache line 的未命中请求合并），并且对乱序返回的数据进行接收；
  3）需要考虑`data`和`tag`重填带来的读写冲突.

- `xbar`

  1）AXI的5个通道独立地控制：对于`ar`，`aw`&`w`通道，根据 master 的请求连接到对应的 slave；对于`r`，`b`通道，根据 slave 的响应连接到对应的 master.
  2）一个小问题：master 的`rready`，`bready`信号如果依赖于 slave 的`rvalid`，`bready`信号，有时候会出现组合环路；
  我的方法是让`rready`，`bready`始终拉高（仿照类sram总线，认为master只要发出请求，就随时能够接收响应）

- SoC 上的设备

图 4. AXI4AXI4AX...boardvgavg

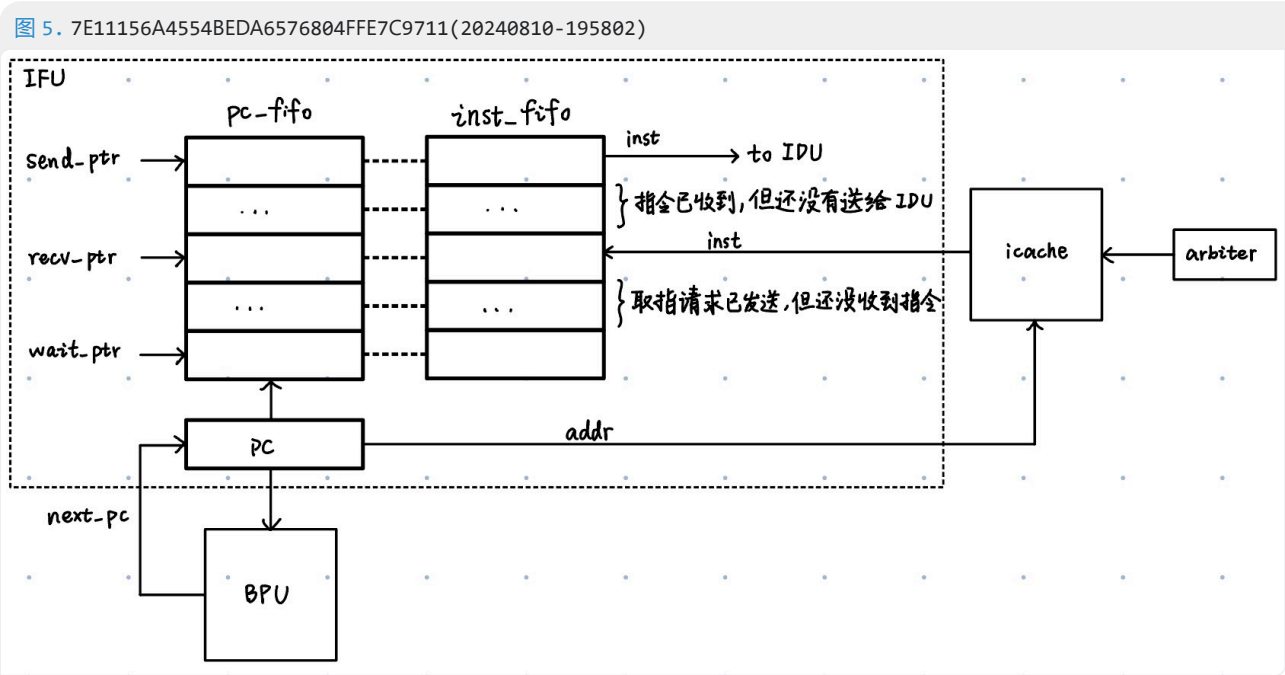


> flash 控制器的对外接口是APB接口，不支持非阻塞传输；
>
> sdram 控制器的对外接口是AXI接口，支持非阻塞传输；（*Config.sdramUseAXI = true*）

3. 实现多条指令的同时取指

- 实现非阻塞的取指通道后，我们可以方便地在 `IFU` 中实现多条指令的同时取指。
- 通过一个 `FIFO` 实现对乱序返回指令的顺序发送，通过3个指针 `wait_ptr`，`recv_ptr`，`send_ptr` 控制对 FIFO 的写入和读出；最多 `fifo_sz` 条指令同时在 `IFU` 中取指。

图 5. 7E11156A4554BEDA6576804FFE7C9711(20240810-195802)



- 多条指令同时取指的收益是什么？

> **可以掩盖 cache miss 带来的延迟。**当某一条指令取指未命中而需要等待时，由于在它前面的指令已经取回保存在FIFO中，`IFU`仍然可以每拍向`IDU`传递一条指令；当后面的流水线阻塞时，可以让后面的指令先进入流水线开始取指。

- 多条指令同时取指的代价是什么？

> 实际上增加了流水线的级数，当一条分支指令出现分支预测错误时，有更多的指令已经进入流水线；

- `fifo_sz` 对性能的影响：（`icache` 参数：`nr_way`=4，`nr_set`=16，`line_sz`=16）

表 4. fifo_szIFU...周期数 AMATCP

`fifo_sz`	`IFU` 发送的指令/`IFU` 取到的指令	取指平均周期数 AMAT	CPI
2	97.70%	2.10	3.14
3	95.53%	1.35	2.64
4	93.60%	1.34	2.64
5	91.76%	1.34	2.65
6	89.99%	1.31	2.63
7	88.30%	1.31	2.63
8	86.68%	1.30	2.63

> `fifo_sz`=3之后再增加，CPI没有明显的变化；
>
> 可能的原因：1) 分支预测准确率比较低，导致分支预测失败带来的惩罚比较严重；2) 指令的数据依赖，即后续指令必须等待前面指令的执行结果，预取更多的指令无法减少这些等待时间。