缓存

本文档以 DCache 为例，讲解如何实现 Cache。

0 前置知识

缓存的架构：直接映射、全相联、组相联

缓存形式：写回、直写等

缓存运作流程：命中时做什么，miss 时做什么

1 实现思路

我们将在 `vsrc/cache/DCache.sv` 中实现 DCache。以下代码均位于 DCache 内部。

端口：Cache 与 Memory 之间采用 CBus，Cache 与 Pipeline(Core) 之间采用 DBus。

参数：

- 8 Byte Word，支持 Byte 写
- 1 Cache Line = 16 Words = 128 Bytes，每次 CBus Transaction 传输一个 Cache Line
- 2/4 Cache Ways
- 8/4 Cache Sets
- Total 2KB

1.1 画电路图

理清缓存行为与架构

1.2 准备工作

参数化、写取位函数。

```
localparam WORDS_PER_LINE = 16;
localparam ASSOCIATIVITY = 2;
localparam SET_NUM = 8;

localparam OFFSET_BITS = $clog2(WORDS_PER_LINE);
localparam INDEX_BITS = $clog2(SET_NUM);
localparam TAG_BITS = 64 - SET_BITS - OFFSET_BITS - 3; /* Maybe 32, or
smaller */

localparam type offset_t = logic [OFFSET_BITS-1:0];
localparam type index_t = logic [INDEX_BITS-1:0];
localparam type tag_t = logic [TAG_BITS-1:0];

function offset_t get_offset(addr_t addr)
    return addr[3+OFFSET_BITS-1:3];
endfunction

function index_t get_index(addr_t addr)
    return addr[3+INDEX_BITS+OFFSET_BITS-1:OFFSET_BITS+3];
endfunction

```

```
function tag_t get_tag(addr_t addr)
    return addr[3+INDEX_BITS+OFFSET_BITS+TAG_BITS-1:3+INDEX_BITS+OFFSET_BITS];
endfunction

localparam type state_t = enum logic[2:0] {
    INIT, FETCH, WRITEBACK
};
```

1.3 状态机

状态机采用 `always_comb + unique_case` 的形式实现。`always_comb` 输入 hit、dirty、总线控制信号，仅驱动 state，其他信号基本上都可以独立地用 `assign` 驱动。

1.4 Cache Set

每个 Cache Set 包含 `ASSOCIATIVITY` 个 Cache Line，包括：

- Data
- Meta
  - Tag
  - Valid
  - Dirty

```
typedef struct packed {
    u1 valid;
    u1 dirty;
    tag_t tag;
} meta_t;
```

- 替换策略：若干个计数器，每次访存选择对应的加1（建议在 INIT 状态下加一）

例化多个 Cache Set：

```
for (genvar i = 0; i < SET_NUM; i++) begin : cache_sets
    // ...

end : cache_sets
```

在不同状态下，creq，dresp，set 内部信号都会有对应的值。

1.5 存储结构

在 Cache 里，如果存储结构与逻辑结构一致，会导致资源浪费。

```
for (genvar i = 0; i < SET_NUM; i++) begin : cache_sets
    u64 data_array [WORDS_PER_LINE-1:0];

    assign data_from_sets[i] = data_array[get_offset(addr)];
end : cache_sets

assign data = data_from_sets[get_index(addr)];
```

`data_array` 是存储元件，`data_from_sets` 是 `wire`，后者的选择电路（64bits）会消耗大量电路资源。

Cache 内主要的存储元件：

- Data
- Meta
- Age
- State, Counter

后两项几乎不占空间，主要考虑前两项。

1.5.1 Meta

一个周期内，我们需要读取一个 Set 里的所有 Meta，最多写入一个 Set 里的一个 Meta。

所以，从全局来看，只需要读一个 Set 的 Meta，把这个 Meta 交给所有 Set。

1.5.2 Data

一个周期内，我们需要读取一个 Word，最多写入一个 Word。

所以，从全局来看，只需要读一个 Word 的 Data，把这个 Data 交给所有 Set（Set 内部似乎不用获取 Data 了）。

1.5.3 RAM 模型

`ram/template.sv` 提供了若干 RAM 以下几种 RAM 模型：

- 单端口 `RAM_SinglePort`
- 一读一写 `RAM_SimpleDualPort`
- 双端口 `RAM_TrueDualPort`
- 一读，一读/写 `LUTRAM_DualPort`

有以下几个参数：

- `ADDR_WIDTH`：地址位数。这里的地址对应于数组下标，和内存地址不同。
- `DATA_WIDTH`：一个字的位数，是读取最小单位与写入最小单位的最小公倍数。
- `BYTE_WIDTH`：一个字节的位数。这里的字节表示写入的最小单位。在 Meta 里，可以这么设置：`.DATA_WIDTH($bits(meta_t) * ASSOCIATIVITY), .BYTE_WIDTH($bits(meta_t))`
- `MEM_TYPE`：在基础部分，设置为0。
- `READ_LATENCY`：读延迟，在基础部分，设置为0。
- 如果 `DATA_WIDTH != BYTE_WIDTH`，则写入需要写使能，`wdata` 的设置规则与 memory 中的一致。

用这种存储结构，只需对 RAM 的地址做选择（地址位数`log2(CACHE_SIZE/DATA_WIDTH)`，本例中为16，远小于数据的64）。

1.6 支持 Uncached Request

`dreq.addr[31] == 0` 的请求不经过 Cache，不应将状态切换至 `FETCH` 或 `Writeback`，不应对 Meta、Data、Age 进行修改。

可以添加一个状态，直接把这个请求发给 creq。burst 设置为 `FIXED`，len 设置为 `MLEN1`，size 设置为 `dreq.size`。

1.7 支持 Reset

每个测试开始时会有 10k 个 reset 时钟周期。你需要遍历所有 Cache set，将所有 Valid 位置 0。这只需要一个不停递增的计数器即可实现。

2 进阶

流水线 Cache

非阻塞 Cache （burst = wrap）

支持 Read_Latency > 0 的 RAM（两级 Memory 流水段）

3 测试

执行 `make test-cache`。

基础测试在 `verilate/vsrc/VCacheTop/tests.inl` 里，你可以在 `verilate/vsrc/VCacheTop/tests.cpp` 内添加更多测试。

每个测试的结构为：

`WITH [SKIP] [DEBUG] [TRACE] [CMP_TO(ref)]{ [test code] } AS("name");`

`SKIP` 表示允许失败，`DEBUG` 表示打印调试信息，`TRACE` 表示输出波形图，`CMP_TO(ref)` 表示进行对照。

对照模型在 `verilate/vsrc/VCacheTop/cache_ref.cpp` 内里实现，可以在每一次读写操作里检查 Cache 内部信号是否符合预期。

获取内部信号的方式：

- 在 `vsrc/VCacheTop.sv` 里声明信号，用 `top.xxx.yyy` 来 assign 过去。

```
for (genvar i = 0; i < SET_NUM; i++) begin : cache_sets
    u64 data_array [WORDS_PER_LINE-1:0];

    assign data_from_sets[i] = data_array[get_offset(addr)];
end : cache_sets

// cache_sets[i].data_array[1]
```

- 在 `verilate/vsrc/VCacheTop/cache_ref.cpp` 里使用 `scope->signal_name` 来获取信号内容。

参考模型如需对内存进行操作，可使用 `mem.load(addr)` 或 `mem.store(addr, data, mask)`，详细用法可参考 `REFERENCE_CACHE`，对应执行 `make test-refcache`。

可在 `verilate/vsrc/VCacheTop/cache_ref.h` 里添加成员函数或成员变量。

执行 `make test-cache-gdb` 用 gdb 调试仿真程序（推荐 `pwndbg`）。

4 要求

实现组相联，替换策略任意

Read_Latency 为 0 的 RAM，总大小小于等于 5KB（≤ 2KB I + ≤ 2KB D + ≤ 1 KB Meta）

Read_Latency 大于 0 的 RAM，总大小小于等于 32KB

执行 `make test-cache` 通过