Lab 4 : L1 Cache 设计

陈祎伟
3230102357

2025年4月16日

1 设计思路

1.1 cache模块

cache模块将可能的访存情况分为了4种：load, edit, store, invalid。

四种情况互不干涉，load是从cache中读取数据，edit是将cache中的数据进行修改，store是将数据写入cache，invalid是将cache中的数据置为无效。

为了正确实现 LRU 替换策略下的各种访存情况，使用如下信号进行控制：

```
valid <= recent1 ? valid2 : valid1;
dirty <= recent1 ? dirty2 : dirty1;
tag <= recent1 ? tag2 : tag1;

hit <= hit1 | hit2;
```

其中，前三个信号用于获取写入cache时，写入位置的相应状态信息；

而hit信号则用于判断cache是否命中。

1.2 cmu模块

cmu模块是cache的控制模块，主要用于控制cache的读写操作。

cmu分为5个状态：S_IDLE, S_PRE_BACK, S_BACK, S_FILL, S_WAIT。

各状态的相互转换如下：

- S_IDLE：空闲状态，等待访存请求。

```
S_IDLE: begin
    if (en_r || en_w) begin
        if (cache_hit)
            next_state = S_IDLE;
        else if (cache_valid && cache_dirty)
            next_state = S_PRE_BACK;
        else
            next_state = S_FILL;
    end
```

```
        next_word_count = 2'b00;
    end
```

下一个状态由访存请求的类型决定：

当 hit 时，不需要进行对内存的读写操作，直接进入空闲状态；

当 miss 时，若数据为 dirty，则需要将数据写回内存，进入 S_PRE_BACK 状态，否则进入 S_FILL 状态。

- S_PRE_BACK：写回内存准备状态，先对 cache 进行一次读操作，

```
    S_PRE_BACK: begin
        next_state = S_BACK;
        next_word_count = 2'b00;
    end
```

进入 S_BACK 状态，准备将数据写回内存。

- S_BACK：写回内存状态，进行数据的写回操作。

```
    S_BACK: begin
        if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})    // wrote back all
            words, 1 cache line = 4 words
            next_state = S_FILL;
        else
            next_state = S_BACK;

        if (mem_ack_i)
            next_word_count = word_count + 1'b1;
        else
            next_word_count = word_count;
    end
```

当收到内存的 ack 信号后，对已完成的 word 进行计数，

当所有 word 都写回后，进入 S_FILL 状态。

否则，继续保持在 S_BACK 状态。

- S_FILL：填充状态，进行数据的填充操作。

```
    S_FILL: begin
        if (mem_ack_i && word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
            next_state = S_WAIT;
        else
            next_state = S_FILL;

        if (mem_ack_i)
            next_word_count = word_count + 1'b1;
```

```
        else
            next_word_count = word_count;
    end
```

同样的，当收到内存的 ack 信号后，对已完成的 word 进行计数，

当所有 word 都填充完成后，进入 S_WAIT 状态。

否则，继续保持在 S_FILL 状态。

- S_WAIT：执行之前由于 miss 而不能进行的 Cache 操作。

```
    S_WAIT: begin
        next_state = S_IDLE;
        next_word_count = 2'b00;
    end
```

miss 消除后，可以正常进行读写操作，回到 S_IDLE 状态。

此外，由于 cache miss 等带来的延迟，在一些情况下需要对 CPU 进行 stall，实现如下：

```
assign stall = next_state != S_IDLE;
```

2 思考题

2.1 在实验报告分别展示 Cache hit、Cache miss+dirty 两种情况，分析两种情况下的状态机状态变化以及需要的时钟周期。

1. Cache hit：

图 1: Cache hit 仿真结果

当 cache 命中时，状态机的状态变化一直保持在 S_IDLE 状态，只需要一个时钟周期即可完成读写操作。

2. Cache miss + dirty：

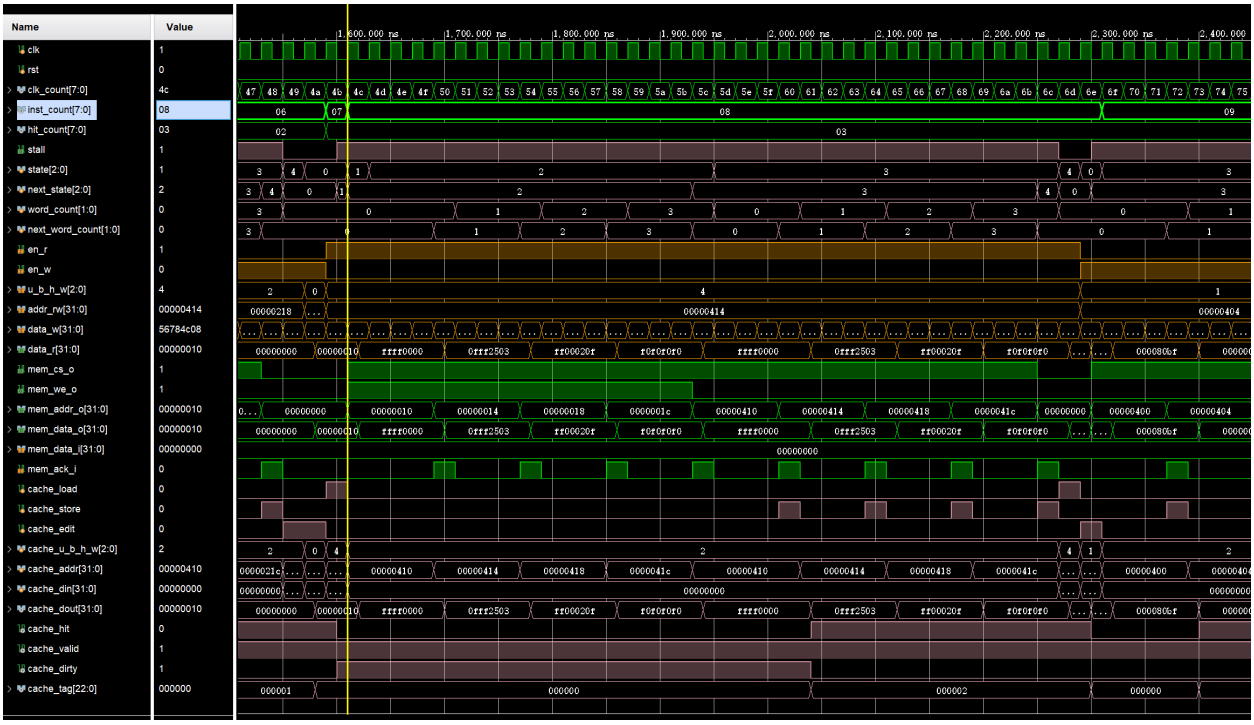

图 2: Cache miss + dirty 仿真结果

当 cache miss 且 dirty 时，状态机的状态变化如下：

初始状态为 S_IDLE，接收到访存请求后（1个周期），进入 S_PRE_BACK 状态（1个周期），

接着进入 S_BACK 状态，依次完成对 4 个 word 的写回操作（16个周期），

然后进入 S_FILL 状态，依次完成对 4 个 word 的填充操作（16个周期），

最后进入 S_WAIT 状态，完成之前由于 miss 而不能进行的 Cache 操作（1个周期），

然后回到 S_IDLE 状态，完成读写操作。

一共需要 35 个时钟周期。

2.2 在本次实验中，Cache 采取的是 2 路组相联，在实现 LRU 替换的时候，每一个 set 需要用多少 bit 来用于真正的 LRU 替换实现？

在 cache 模块的实现中，LRU 相关定义如下：

```
1    reg [ELEMENT_NUM-1:0] inner_recent = 0; // for LRU
2
3    assign recent1 = inner_recent[addr_element1];
4    assign recent2 = inner_recent[addr_element2];
5
6    ...
7    valid <= recent1 ? valid2 : valid1;
8    dirty <= recent1 ? dirty2 : dirty1;
9    tag <= recent1 ? tag2 : tag1;
```

可见，在 2 路组相联的情况下，每一个 set 只需要 1 bit 来表示 LRU 替换策略。