浙江大学

计算机体系结构实验报告

课程名称：	计算机体系结构
姓　　名：	周楠
学　　院：	计算机科学与技术学院
专　　业：	计算机科学与技术
学　　号：	3220102535
指导教师：	常瑞

2024 年 11 月 15 日

浙江大学实验报告

课程名称：计算机体系结构　实验类型：综合

实验项目名称：L1 cache 设计

学生姓名：周楠　专业：计算机科学与技术　学号：3220102535

实验地点：玉泉实验室　实验时间：2024.10.22　指导教师：常瑞

一. 操作方法与实验步骤

1.1 CMU 模块

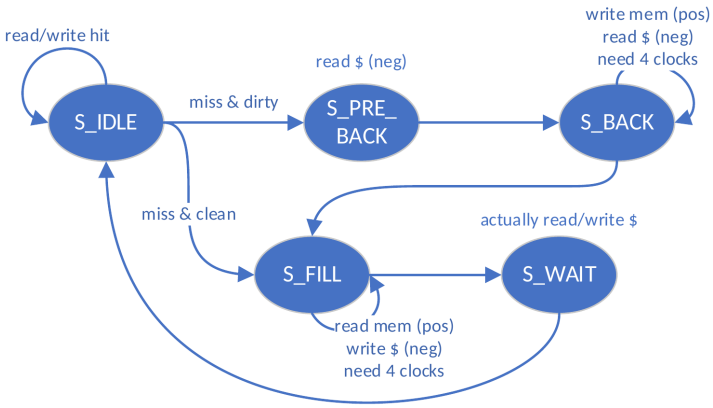

图 1: 状态机

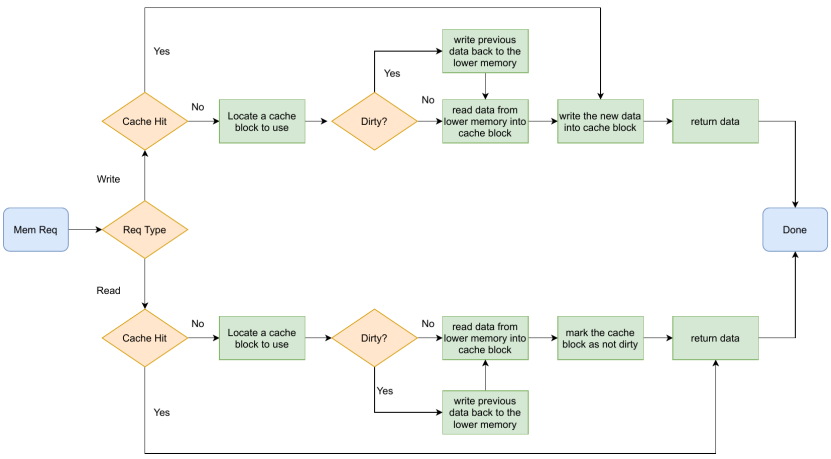

图 2: CMU 读写事务流程

1. S_IDLE: 空闲状态,cache 正常读写，即 load/store 命中或者未使用 cache。

   (a) 如果 cache 命中 (cache_hit=1), 则保持在 S_IDLE 状态。

   (b) 如果 cache 未命中，并且缓存是有效且脏（cache_valid && cache_dirty），则需要先将缓存写回内存，状态转移到 S_PRE_BACK。

   (c) 否则需要从内存中读取数据到 cache(write allocate)，状态转移到 S_PRE_BACK。

```
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
```

2. S_PRE_BACK , 准备写回状态, 在执行写回之前，需要先读取缓存（用于确定哪些数据需要写回）。

   (a) 直接转移到 S_BACK 状态。next_word_count = 2'b00，重置计数器。

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

3. S_BACK , 写回状态, 缓存中的数据将被写回内存，计数器控制写回的数据块数量，直到整个缓存块都被写回。

   (a) 如果收到内存的确认信号 (mem_ack_i)，并且计数器已经达到设定的最大值（word_count == {ELEMENT_WORDS_WIDTH{1'b1}}），则切换到 S_FILL 状态，表示写回完成，可以进行缓存填充。

   (b) 否则，保持在 S_BACK 状态，继续写回。

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

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

4. S_FILL, 填充状态, 从内存中读取数据填充到 cache 中。

   (a) 如果收到内存的确认信号（mem_ack_i），并且计数器已经完成填充（word_count == {ELEMENT_WORDS_WIDTH{1'b1}}），则切换到 S_WAIT 状态，表示填充完成，准备进入等待阶段。

   (b) 否则，保持在 S_FILL 状态，继续填充。

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

5. S_WAIT, 等待状态, 等待内存完成写回操作。

   (a) 直接转移到 S_IDLE 状态。next_word_count = 2'b00，重置计数器。

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

如果 next_state = S_IDLE, 则不需要进行 stall，否则需要进行 stall。

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

1.2 cache 模块

1.2.1 addr 解析

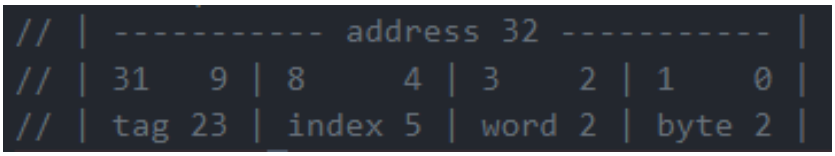

图 3: addr 解析

1. tag 占 23 位，对应的是 addr[31:9]。tag 用于匹配 cache 中的数据。
2. index 占 5 位，对应的是 addr[8:4]。index 用于确定 cache 对应的 set。
3. word 占 2 位，对应的是 addr[3:2]。一个 cache line 中有 4 个 word。
4. byte 占 2 位，对应的是 addr[1:0]。一个 word 中有 4 个 byte。

1.2.2 cache 设置

- addr_tag, addr_index, 根据上述 addr 解析得到 tag 对应地址的 31:9，index 对应地址的 8:4。

```
assign addr_tag = addr[31:9];
assign addr_index = addr[8:4];
```

- addr_element。由于采用两路组相联，一个 set 中有两个 cache line，也就是说有两个 element。先确定 index，找到对应的 set，在末尾添加 1'b0，表示 element0，添加 1'b1，表示 element1。

```
assign addr_element1 = {addr_index, 1'b0};
assign addr_element2 = {addr_index, 1'b1}; //need to fill in
```

- addr_word, word address = element address + word offset, 计算 word 的地址，首先需要确定 element 的地址，然后根据 word offset 确定 word 的地址。

```
assign addr_word1 = {addr_element1, addr[ELEMENT_WORDS_WIDTH+
    WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]};
assign addr_word2 = {addr_element2, addr[ELEMENT_WORDS_WIDTH+
    WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]}; //need to fill in
```

- word, half_word, byte。address 的后两位用于确定 word 中的 byte。

  1. 先取出 word，再根据 addr[1] 取出 half_word,addr[1]=1 表示高半字，addr[1]=0 表示低半字。
  2. 取出 half_word 后，再根据 addr[0] 取出 byte，addr[0]=1 表示高字节，addr[0]=0 表示低字节。

```
assign word1 = inner_data[addr_word1];
assign word2 = inner_data[addr_word2];
assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];
assign byte1 = addr[1] ?
               addr[0] ? word1[31:24] : word1[23:16] :
               addr[0] ? word1[15:8] : word1[7:0] ;
assign byte2 = addr[1] ?
               addr[0] ? word2[31:24] : word2[23:16] :
               addr[0] ? word2[15:8] : word2[7:0];
```

- recent2, valid2, dirty2, tag2, hit2 分别表示第二路的最近访问位、有效位、脏位、标记位、命中位，直接照搬第一路的写法即可。

- valid, dirty, tag, hit。valid, dirty, tag, hit 分别表示 cache 的有效位、脏位、标记位、命中位。如果 recent1=1，则表示第一路最近被使用，此时需要使用第二路的数据，反之亦然。

  1. 因此判断数据是否有效就是看 valid2，判断第二路的数据是否有效。
  2. 判断数据是否为脏数据也是看 dirty2，判断第二路的数据是否为脏数据。
  3. 判断数据是否对应就是看 tag2，判断第二路的 tag 是否与 addr_tag 对应。

```
assign valid = recent1 ? valid2 : valid1;
assign dirty = recent1 ? dirty2 : dirty1;
assign tag = recent1 ? tag2 : tag1;
assign hit = recent1 ? hit2 | hit1;
```

- load 操作。以 hit 第一路为例。

  1. 选择读取的数据类型：u_b_h_w 是一个用于控制读取数据宽度的信号。

     u_b_h_w[2] 判断是否需要填充 0 或符号扩展，如果 u_b_h_w[2]=1，则需要填充 24 个 0 位。

     u_b_h_w[1] 判断读取的长度是否为 word，如果 u_b_h_w[1]=1, 则加载 word。

     u_b_h_w[0] 判断读取的长度是否为 half_word，如果 u_b_h_w[0]=1, 则加载 half_word，否则加载 byte。

  2. 更新 inner_recent。由于第一路缓存命中，inner_recent 中 addr_element1 被设置为 1，表示第一路最近被使用，第二路则被标记为 0。

```
if (load) begin
    if (hit1) begin
        dout <=
            u_b_h_w[1] ? word1 :
            u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word1
                [15]}}, half_word1} :
            {u_b_h_w[2] ? 24'b0 : {24{byte1[7]}}, byte1};

        // inner_recent will be refreshed only on r/w hit
        // (including the r/w hit after miss and replacement)
        inner_recent[addr_element1] <= 1'b1;
```

```
11          inner_recent[addr_element2] <= 1'b0;
12      end
13      else if (hit2) begin
14              //need to fill in
15          dout <=
16              u_b_h_w[1] ? word2 :
17              u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word2
                    [15]}}, half_word2} :
18              {u_b_h_w[2] ? 24'b0 : {24{byte2[7]}}, byte2};
19          inner_recent[addr_element1] <= 1'b0;
20          inner_recent[addr_element2] <= 1'b1;
21      end
22  end
23  else dout <= inner_data[ recent1 ? addr_word2 : addr_word1 ];
```

- edit 操作。以 hit 第一路为例。

  1. u_b_h_w[1] 为真：表示写入的是一个完整的字（word）。此时，直接将 din（数据输入）写入 inner_data[addr_word1]。
  2. u_b_h_w[0] 为真：表示写入的是半字（half word）。根据 addr[1] 的值决定写入是位于该半字的高 16 位还是低 16 位：
     - addr[1] 为 1：写入 din 的低 16 位到 word1 的高 16 位，高 16 位保持不变。
     - addr[1] 为 0：写入 din 的低 16 位到 word1 的低 16 位，高 16 位保持不变。
  3. 字节（byte）写入：根据 addr[1] 和 addr[0] 的值决定写入 din 到 word1 的具体位置：
     - addr[1] = 1 且 addr[0] = 1：将 din[7:0] 写入 word1 的最高字节，剩下的部分为零。
     - addr[1] = 1 且 addr[0] = 0：将 din[7:0] 写入 word1 的第二个字节，剩下的部分保持不变。
     - addr[1] = 0 且 addr[0] = 1：将 din[7:0] 写入 word1 的第三个字节，剩下的部分保持不变。
     - addr[1] = 0 且 addr[0] = 0：将 din[7:0] 写入 word1 的最低字节，剩下的部分保持不变。

由于第一路缓存命中，inner_recent 中 addr_element1 被设置为 1，表示第一路最近被使用，第二路则被标记为 0。inner_dirty 被设置为 1，表示缓存中的数据被修改。

```
else if (hit2) begin
        //need to fill in
    inner_data[addr_word2] <=
        u_b_h_w[1] ?
            din
        :
            u_b_h_w[0] ?
                addr[1] ?
                    {din[15:0], word2[15:0]}
                :
                    {word2[31:16], din[15:0]}
            :
                addr[1] ?
                    addr[0] ?
                        {din[7:0], word2[23:0]}
                    :
                        {word2[31:24], din[7:0], word2[15:0]}
                :
                    addr[0] ?
                        {word2[31:16], din[7:0], word2[7:0]}
                    :
                        {word2[31:8], din[7:0]}
    ;
    inner_dirty[addr_element2] <= 1'b1;
    inner_recent[addr_element1] <= 1'b0;
    inner_recent[addr_element2] <= 1'b1;
end
```

- store 操作。如果 recent1=0，则表示第一路最近未被使用，不管 recent2 取值如何，都更改第一路的数据

  1. recent2=1, 表示第二路最近被使用，此时需要更改第一路的数据。
  2. recent2=0，表示该 set 中的数据都未被使用，此时需要更改第一路的数据

  store 操作之后，inner_data[addr_word1]=din, inner_valid[addr_element1]=1, inner_tag[addr_element1]=addr_tag。

```
else begin
    // recent2 == 1 => replace 1
    // recent2 == 0 => no data in this set, place to 1
    //need to fill in
    inner_data[addr_word1] <= din;
    inner_valid[addr_element1] <= 1'b1;
    inner_dirty[addr_element1] <= 1'b0;
    inner_tag[addr_element1] <= addr_tag;
end
```

二. 思考题

2.1 在实验报告中分别展示 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 第三条和第四条指令都是 cache hit，状态机状态都是 state=0，对应的是 S_IDLE 状态。需要 1 个时钟周期。

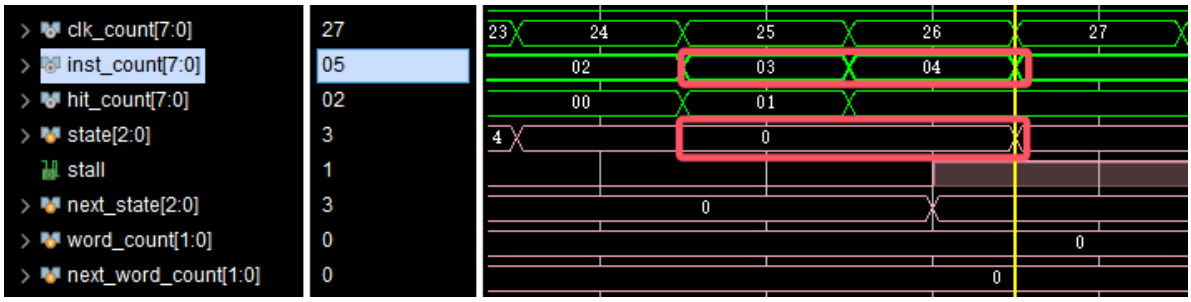

图 4: cache hit

2. cache miss+dirty 第八条指令发生 read miss，同时需要将脏数据写回内存，经历的状态为 S_IDLE->S_PRE_BACK->S_BACK->S_FILL->S_WAIT,

   需要 $1 + 1 + 4 * 4 + 4 * 4 + 1 = 35$ 个时钟周期。

图 5: cache miss+dirty

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？

1 个 bit。假设有两个 cache line A 和 B，

- 如果 LRU = 0，表示 cache line A 是最近最少访问的，cache line B 是最近访问的。
- 如果 LRU = 1，则表示 cache line B 是最近最少访问的，cache line A 是最近访问的。
- 如果最近访问的是 cache line A，就设置 LRU = 1，否则设置 LRU = 0。

三. 讨论与心得