

中国科学院大学

《计算机组成原理(研讨课)》实验报告

姓名 朱夏楠 学号 2022K8009929031 专业 计算机科学与技术
实验项目编号 5.2 实验名称 高速缓存设计

- 注 1: 撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下 (注意: reports 全部小写)。文件命名规则: `prjN.pdf`, 其中 `prj` 和后缀名 `pdf` 为小写, `N` 为 1 至 4 的阿拉伯数字。例如: `prj1.pdf`。PDF 文件大小应控制在 5MB 以内。此外, 实验项目 5 包含多个选做内容, 每个选做实验应提交各自的实验报告文件, 文件命名规则: `prj5-projectname.pdf`, 其中 “-” 为英文标点符号的短横线。文件命名举例: `prj5-dma.pdf`。具体要求详见实验项目 5 讲义。
- 注 2: 使用 `git add` 及 `git commit` 命令将实验报告 PDF 文件添加到本地仓库 master 分支, 并通过 `git push` 推送到实验课 SERVE GitLab 远程仓库 master 分支 (具体命令详见实验报告)。
- 注 3: 实验报告模板下列条目仅供参考, 可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明 (比如 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 L^AT_EX. 中}、相应信号的仿真波形和信号变化的说明等)

• 指令 Cache 设计。

指令 Cache 的状态转移图在讲义中给出, 如下图所示,

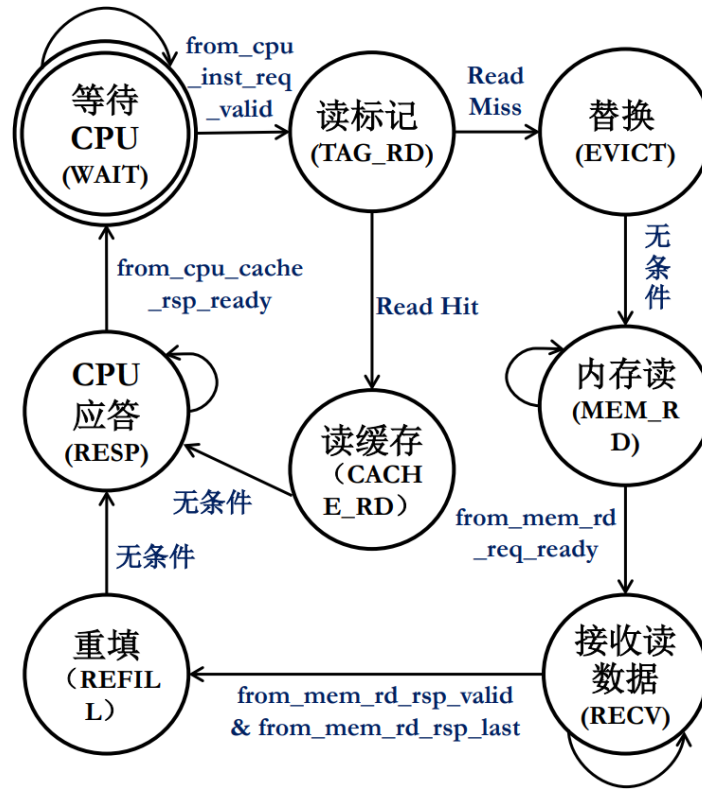


图 1: 指令 Cache 状态机

并且每个状态下需要拉高的端口信号也已经给出,因此指令 Cache 实现起来相对简单,需要考虑的部分主要有两点,一是如何判断读是否命中,二是选择怎样的 Cache 替换策略。

本次我实现的 Cache 采用的是 4 路组相联映射方式,判断读是否命中首先需要根据读地址得到相应的组号,再在该组的所有有效 Cache 块中查找和对比 Tag,若发现有 Cache 块 Tag 相同,则认为读命中,可直接从 Cache 中取出相应的指令送给 CPU,否则,需要从内存中读取指令。

```

assign tag = Address[31:8];
assign index = Address[7:5];
//各路的valid信号和tag值都是根据组地址index从Cache的四路中获取的
assign ways = {(~|(tag ^ tag_w4) & valid_w4),(~|(tag ^ tag_w3) & valid_w3),(~|(tag ^ tag_w2)
    & valid_w2),(~|(tag ^ tag_w1) & valid_w1)};
assign hit = |ways;

```

在读未命中的情况下,需要从内存的相应位置取出指令交给 CPU,同时也要依照一定的策略更新 Cache 的内容,更进一步,在组相联映射的情况下,仅有 index 所对应的组需要进行更新。若该组未被填满,则可以直接选择一个未被填充的 Cache 块将从内存中取出的 32 字节数据存入,若该组已经被填满,需要根据一定的策略选出一个 Cache 块进行替换。我实现的替换策略是 LRU(Least Recent Used),在替换时,从四个 Cache 块中选出最早使用的一个将其内容替换为新读到的数据块。

具体实现方式为,对于每一组,使用一个 $4 \times 4\text{bit}$ 寄存器堆进行存储,当有一块 Cache 被访问或者被更新时,将对应号码的行和列进行处理,如下图所示。

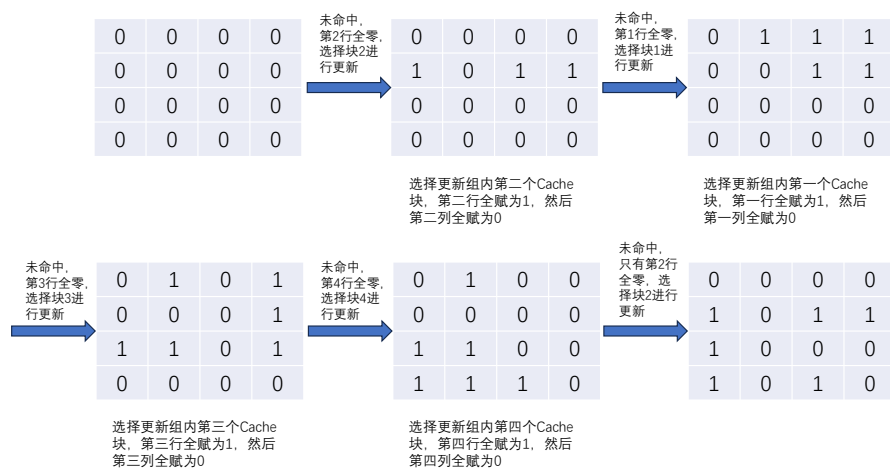


图 2: LRU 替换策略示意图

可以看到，随着不断访问 Cache，最先被存入数据的第二个 Cache 块成为了四个块中最近最久未被使用的块，因此在更新该组时，硬件逻辑会优先选择这一块。

解决了这两个问题之后，基本上按照讲义中给出的状态转移图描述状态机，并在不同状态对控制信号进行赋值即可，相应的实现如下

```

localparam WAIT = 8'b00000001, //0
TAG_RD = 8'b00000010, //1
CACHE_RD = 8'b00000100, //2
EVICT = 8'b00001000, //3
MEM_RD = 8'b00010000, //4
RECV = 8'b00100000, //5
REFILL = 8'b01000000, //6
RESP = 8'b10000000; //7

//I-Cache的状态机
always @(posedge clk)
begin
    if(rst == 1'b1)
        current_state <= WAIT;
    else
        current_state <= next_state;
end
reg [7:0]next_state;
always @(*)
begin
    case (current_state)
        WAIT:
            begin
                if(from_cpu_inst_req_valid & to_cpu_inst_req_ready)
                    next_state = TAG_RD;
                else

```

```

        next_state = WAIT;
    end
TAG_RD:
begin
    if(hit)
        next_state = CACHE_RD;
    else
        next_state = EVICT;
    end
end
CACHE_RD:
begin
    next_state = RESP;
end
EVICT:
begin
    next_state = MEM_RD;
end
MEM_RD:
begin
    if(from_mem_rd_req_ready)
        next_state = RECV;
    else
        next_state = MEM_RD;
    end
end
RECV:
begin
    if(from_mem_rd_rsp_valid & from_mem_rd_rsp_last)
        next_state = REFILL;
    else
        next_state = RECV;
    end
end
REFILL:
begin
    next_state = RESP;
end
RESP:
begin
    if(from_cpu_cache_rsp_ready)
        next_state = WAIT;
    else
        next_state = RESP;
    end
end
default:
begin
    next_state = WAIT;
end
endcase
end
//根据不同状态对信号赋值
assign to_cpu_inst_req_ready = current_state[0] & ~rst;

```

```

assign to_mem_rd_req_valid = current_state[4];
assign to_mem_rd_rsp_ready = current_state[5];
assign to_cpu_cache_rsp_valid = current_state[7];
assign to_mem_rd_req_addr = {Address[31:5],5'b0};
assign to_mem_rd_req_valid = current_state[4];
assign to_mem_rd_rsp_ready = current_state[5];
assign to_cpu_cache_rsp_valid = current_state[7];

```

另需注意的是,若从内存中读取数据,指令 Cache 一次要读 32 字节的数据,而 CPU 只需要其中的 4 字节。可以在突发传输过程中 256 位的移位寄存器保存从内存中传来的数据,然后再利用块内偏移地址选出需要的 4 字节传送给 CPU,同时将这 32 字节存入 Cache 中。

• 数据 Cache 设计。

相较于指令 Cache,数据 Cache 除了读数据之外还增加了写数据的需要,因此,需要选择并实现相应的写策略,本次实验中选择的写命中策略为 Write-back,即 CPU 写回数据时,若 Cache 命中,则将 Cache 中对应的数据更新,并在对应的块将要被替换时,将数据保存到内存中;选择的写缺失策略为 Write allocate,若 Cache 未命中,从内存中读出写地址对应的数据块,选择一个 Cache 块进行更新,并在该块再次要被替换时,将数据保存到内存中。此外,一部分的地址空间在数据 Cache 中被设置为不可缓存,需要另加旁路对其进行处理。得到的状态转移图如下。

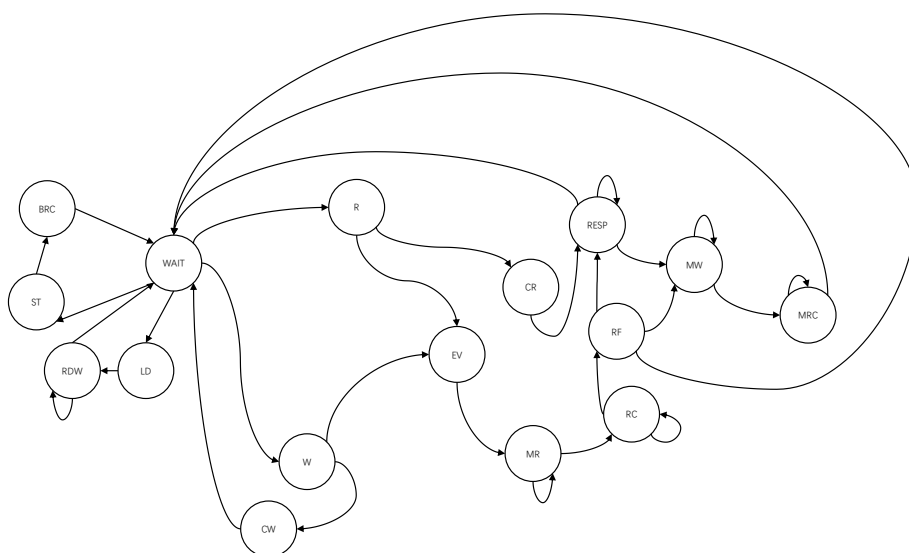


图 3: 数据 Cache 状态转移图

状态机共有 16 个状态,其中 LD,RDW,ST,BRC 四个状态为地址不可缓存时的情况,实际实现时可以将其并入地址可缓存时的情况,但此处为了直观展现将其分离。对于地址可缓存的情况,要分为读写两种情况考虑,由于数据 Cache 要支持写缓存,与指令 Cache 相比,还需要额外增加 4 个脏位数组,用于标记每个 Cache 块是否被写请求更新过。

首先数据 Cache 需要判断 CPU 的请求是写还是读,若是写请求,用与指令 Cache 相同的方式进行写命中的判断。若命中,则更新对应的 Cache 块,同时将该块的脏位置一;若未命中,则需要从内存中读出对应的数据,替换 Cache 中的一个数据块,数据 Cache 选择的替换策略同样为 LRU,故不再赘述,同时,若被替换的数据块脏位为一,则要将其写回内存的对应位置。对于读请求,基本与指令 Cache 相同,不过同样在被替换的数据块脏位为一的时候,要将其写入内存对应的位置。

我大致浏览了一下 AXI 总线协议,发现数据 Cache 和内存之间应该可以存在一读写两个数据传输通道,但我阅读本实验的代码框架之后没有发现相应的支持,因此我选择将写回数据放在从内存读数据之前。并且无论是 CPU 的读请求还是写请求,在 Cache 未命中之后的处理流程基本相同,区别在于读请求需要将数据返回给 CPU,写请求需要根据 CPU 给出的数据额外改变从内存读到的数据,因此我选择将读写请求在未命中后替换和写回状态合并。

数据 Cache 与指令 Cache 还有一点不同是,Cache 与内存、外设等之间进行数据传输的方式是突发传输,指令 Cache 由于只有读请求,并且默认其发出的所有地址都是可缓存的,因此会默认其突发传输的长度为 8。而数据 Cache 考虑到旁路一次请求只需要进行一次 4 字节的数据传输,突发传输长度为 1,需要硬件逻辑根据请求的类型指定传输长度,同时,对于写请求,需要 Cache 主动向数据接收方发起 last 信号,来指明数据传输的结束。

```
always @(posedge clk)
begin
    if(rst)
        len <= 8'b0;
    else if(current_state[8]) //正常Cache读写请求,指定传输长度为8
        len <= 8'b111;
    else if(current_state[1] | current_state[3]) //旁路请求,指定传输长度为1
        len <= 8'b0;
end

/*计数器,根据不同的数据传输要求置不同的初值,在值减到1时发出last信号*/
always @(posedge clk)
begin
    if(rst)
    begin
        cnt1 <= 8'b0;
    end
    else if(current_state[8])
    begin
        cnt1 <= 8'b1000;
    end
    else if(current_state[3])
    begin
        cnt1 <= 8'b1;
    end
    else if((current_state[15] | current_state[14]) & from_mem_wr_data_ready)
    begin
        cnt1 <= cnt1 - 1;
    end
end
end
```

二、实验过程中遇到的问题、对问题的思考过程及解决方法(比如 RTL 代码中出现的逻辑 bug,逻辑仿真和 FPGA 调试过程中的难点等)

● LRU 实现错误。

在实现采用 LRU 替换策略的 Cache 之后,我经过测试,发现性能的提升符合我的预期,因此认为替换策略

部分的实现不存在问题,但在写报告的时候,我又检查了一遍代码,发现替换策略的实现存在问题。以下为之前的替换策略的一部分,描述了第一组的四个数据块如何进行最近最少使用的排序。

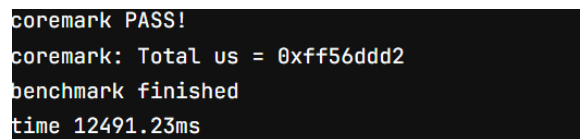
```
integer j;
always @(posedge clk)
begin
    if(rst)
    begin
        for(j=0;j<4;j=j+1)
        begin
            Freq_array_s0[j] <= 4'b0;
        end
    end
    else if(current_state[5] | current_state[7] | current_state[8])
    begin
        if(index==3'b0)
        begin
            for(j=0;j<4;j=j+1)
            begin
                Freq_array_s0[rank][j] <= 1'b0;
            end
            Freq_array_s0[rank] <= 4'b1111;
        end
    end
end
end
```

可以看出若按照此硬件逻辑,若数据块全被填充,则每一个数据块的计数器的值都将都为 4'b1111,无法起到排序的作用。在改正之后,我在行为仿真阶段对比了 memcpy 这个测试程序的仿真时间(此时只启用了指令 Cache),发现两者相同,观察了波形后发现直到程序结束,指令 Cache 多数组仍然处于未填充状态,因此无法反应出替换策略对程序的影响。再在启用全部 Cache 的情况下将改正之后的结果推上云平台进行 FPGA 仿真,观察了 coremark 和 dhrystone 的测试结果,发现改正之后有了明显的变化。



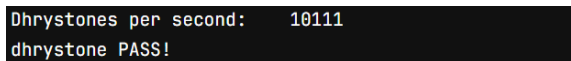
```
Dhrystones per second: 13119
dhrystone PASS!
```

(a) 改正后 dhrystone



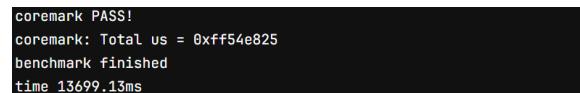
```
coremark PASS!
coremark: Total us = 0xff56ddd2
benchmark finished
time 12491.23ms
```

(b) 改正后 coremark



```
Dhrystones per second: 10111
dhrystone PASS!
```

(c) 改正前 dhrystone



```
coremark PASS!
coremark: Total us = 0xff54e825
benchmark finished
time 13699.13ms
```

(d) 改正前 coremark

另外感谢叶从容助教,在我验收时询问我如何验证替换策略实现正确,我在重新审阅代码的时候对这方面投入了更多的注意力,检查出了这个错误。



图 4: memcpy 的仿真结果

• 数据 Cache 出现大量 FPGA 仿真错误。

在完成数据 Cache,并通过本地行为仿真进行 debug 之后,我将代码推送到云平台上进行 FPGA 仿真,发现出现了大面积的仿真错误情况,参考之前完成流水线时遇到的同样的问题,我认为可能是在发出复位信号的时候,某些控制信号设置错误导致的。为此,我先针对 basic 程序进行了一次 FPGA 仿真加速,发现波形不存在问题,为了进一步确认,我将硬件逻辑中控制数据 Cache 转入旁路请求的信号持续拉高,以此消除 Cache 本身的影响,发现同样出现大量的 FPGA 仿真错误。

到了这一步,我选择初步选定可能出错的信号,并进行尝试,最终发现 $to_{mem_rd_spr_ready}$ 这个信号在开始时未被拉高,修改之后,FPGA 仿真便可以正常通过了,猜测这是因为在 FPGA 上进行仿真的时候,由于数据未被清除干净,需要在复位阶段拉高该信号将其他设备向 Cache 的数据传输队列清空。

• 同时读指令和读数据冲突。

这个问题只出现在预测策略为 Always Taken 的五级流水线处理器启用数据 Cache 而不启用指令 Cache 时的行为仿真中,可以看到,在红线标记的位置 CPU 向内存发起了读数据的指令的请求,与此同时,数据 Cache 向内存发出了读数据的请求,此后虽然指令传输了过来,但相应的握手信号并没有拉高,PC 就不再更新了。

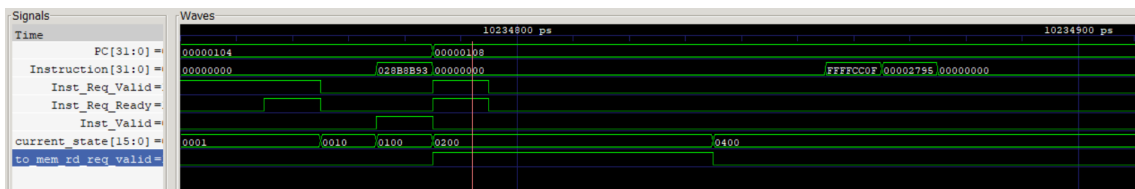


图 5: 错误的行为仿真波形

这是因为本次实验的框架代码会导致内存在拉高读指令准备信号的同时将传输队列更改为数据传输。

```
if (~resetn)
    arid_r <= {ID_WIDTH{1'b0}};
else if (~arbusy & cpu_mem_arvalid)
    arid_r <= DATAID;
```



```

else if (~arbusy & cpu_inst_arvalid)
    arid_r <= INSTID;
//.....
assign s_axi_arid    = arid_r;
assign cpu_mem_arready = s_axi_arready & s_axi_arid == DATAID;
assign cpu_inst_arready = s_axi_arready & s_axi_arid == INSTID;

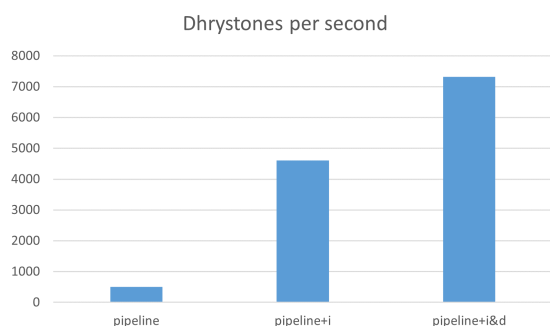
```

当 arid 为 INSTID 时, 回向 CPU 发出 Inst_Req_Ready 信号, 但由于 cpu_mem_arvalid 拉高, 在该硬件逻辑下, 在下一拍, arid 改变为 DATAID, 因此读取的指令进行传输时, 无法生成相应的握手信号, CPU 不能接收到下一条指令。

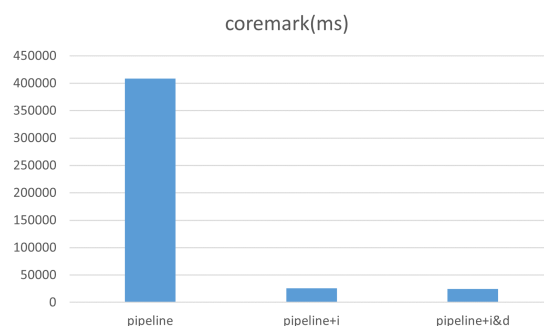
由于该问题只出现在特定的情况的行为仿真中, 与仿真时序有关, 且目前我没有发现在不修改框架的情况下规避该方法, 因此我未对此问题进行处理。

三、 思考和总结

- **性能对比。** 比较了在主频设置为 100MHz 的情况下, 流水线处理器在全部 Cache 不启用、只启用指令 Cache、启用全部 Cache 三种情况下 dhrystones 和 coremark 的测试结果



(a) dhrystone



(b) coremark

可以看到启用了 Cache, 尤其是指令 Cache 之后, 程序的运行速度有了一个很大的提升, 这主要是由于未启用 Cache 时, 从内存中取指操作在整个程序运行过程中所占用的时钟周期数相当的多。

- **未来改进方向。**

针对 Cache, 尤其是针对数据 Cache 需要考虑继续优化其状态机设计。同时在 CPU 发出写请求时, 需要将新的数据写入 Cache 块, 但一个数据块是 32 字节, 而一次写入的数据为 4 字节, 因此需要加入较为复杂的硬件选择逻辑, 而我又选择全部使用组合逻辑实现, 因此代码既不美观, 且导致延时过大, 可能需要将其拆分成多个状态。

对于 Cache 可选择的替换策略, 我也只测试了 LRU 一种, 而其他策略表现如何也需要实际的测试。本次实验 Cache 我只实现了 4 路组相联, 根据目前的情况看, 在硬件允许的情况下继续增加 Cache 的容量应该还是可以对处理器的性能有可观的提升的。

四、 实验所耗时间

在课后, 你花费了大约 30 小时完成此次实验。