

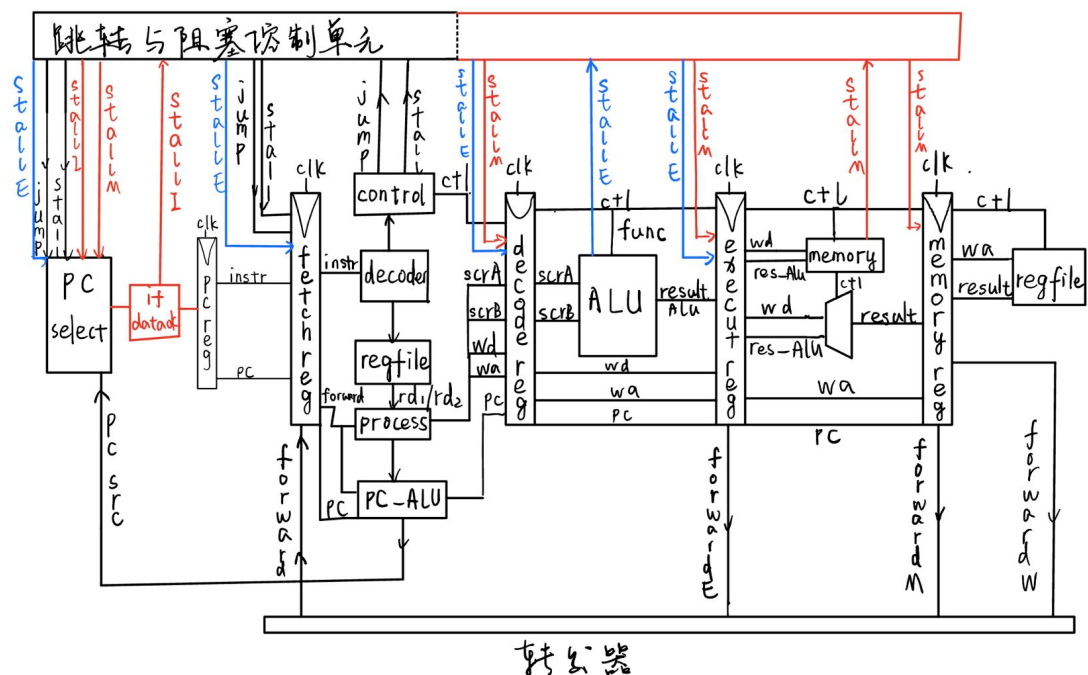
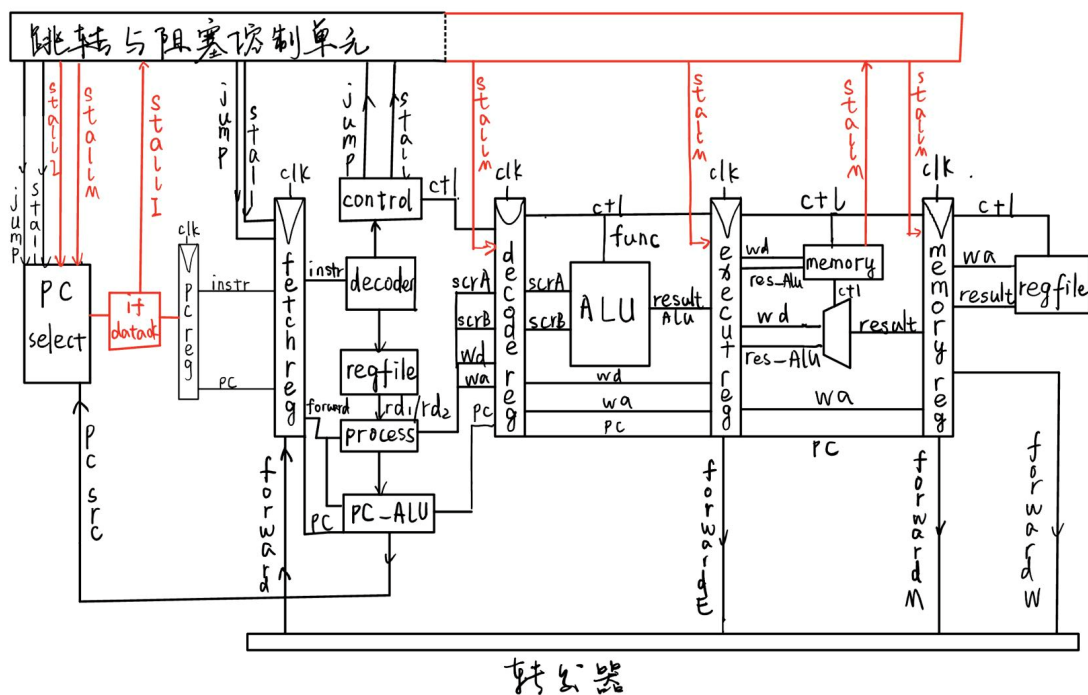
缓存 实验报告

目录

- CPU电路图对比
- 测试通过结果截图
- 为了支持随机延迟，流水线的改动
- 缓存的实现

内容

CPU电路图对比



两次实验CPU电路图对比。红色是lab2新添部分，蓝色是lab3新添部分。

测试通过结果截图

test-cache

```
[OK] void (8ms)
[--] reset (skipped)
[OK] fake load (77ms)
[OK] fake store (95ms)
[OK] naive (8ms)
[--] akarin~ (skipped)
[OK] strobe (11ms)
[OK] ad hoc (10ms)
[OK] pipelined (9ms)
[OK] memory cell (9ms)
[OK] memory cell array (8ms)
"build/cmp--word.fst": stop @168800
[OK] cmp: word (62ms)
[OK] cmp: halfword (16ms)
[OK] cmp: byte (19ms)
"build/cmp--random.fst": stop @24165800
[OK] cmp: random (4096ms)
[OK] memset (92ms)
[OK] memcpy (105ms)
[OK] load/store repeat (74ms)
[OK] backward memset (238ms)
[OK] backward load/store (309ms)
[OK] random step (278ms)
[OK] random load/store (3554ms)
[OK] random block load/store (1564ms)
"std::sort": bingo!
[OK] std::sort (1712ms)
"std::stable_sort": bingo!
[OK] std::stable_sort (3565ms)
"heap sort": bingo!
[OK] heap sort (7234ms)
"binary search tree": bingo!
[OK] binary search tree (5373ms)
(info) 27 tests passed.
```

上板的test-lab3

```
WAACTVAAAADDERPUTAJBBPTLZGPIAAIBAAANIBAAAAALALIBLAAJGUHLEZTAAAAAIAA
Run coremark
Running CoreMark for 10 iterations
2K performance run parameters for coremark.
CoreMark Size : 666
Total time (ms) : 242
Iterations : 10
Compiler version : GCC9.4.0
seedcrc : 0xe9f5
[0]crc1list : 0xe714
[0]crcmatrix : 0x1fd7
[0]crcstate : 0x8e3a
[0]crcfinal : 0xfcfa
Finished in 242 ms.
=====
CoreMark Iterations/Sec 41
Run dhrystone
Dhrystone Benchmark, Version C, Version 2.2
Trying 10000 runs through Dhrystone.
Finished in 673 ms
=====
Dhrystone PASS 26 Marks
vs. 100000 Marks (17-7700K @ 4.20GHz)
Run stream
=====
STREAM version $Revision: 5.10 $
=====
This system uses 8 bytes per array element.
=====
Array size = 2048 (elements), Offset = 0 (elements)
Memory per array = 0.0 MiB (= 0.0 GiB).
Total memory required = 0.0 MiB (= 0.0 GiB).
Each kernel will be executed 10 times.
The "best" time for each kernel (excluding the first iteration)
will be used to compute the reported bandwidth.
```

copy的速度为 $22MB/s$

为了支持随机延迟，流水线的改动

为了支持乘除法器，流水线的改动主要聚集在阻塞控制单元上。

阻塞控制单元的改动

进行乘除法时，由于采用多周期的方式，在进行计算时，流水线应该相应阻塞。

本实验的乘法器使用了*符号，用了助教参考文档里的算法，两个周期完成。第一个周期将64位拆为四个32位，分别相乘，在第二个周期将相乘结果。由于我们只有结果只有64位，所以高32乘高32是可以直接舍去的。因为利用了DSP资源，所以性能不错。

本实验的除法器则用了移位相减，每个时钟周期执行2次移位相减，一共耗时32个时钟周期(若每个时钟周期执行4次移位相减，则最大延迟将从24ns增加至28ns，所以只执行2次)。因为除法指令出现较少，所以对性能影响不大。

在进行乘除运算时，流水线应该进行阻塞，阻塞逻辑与先前的相似，即：在此之前的阶段，寄存器存值，在此之后的阶段，寄存器刷新。

阻塞信号的驱动为：

```
assign stallE = (valid_mul && ~done_mul) || (valid_div && ~done_div);
```

*valid*和*done*分别表示乘除法器有效和完成计算。

*stallE*需要阻塞*fetch_reg*和*decode_reg*，同时刷新*execute_reg*里的指令。这里以*fetch_reg*为例表示阻塞：

```
always_ff @(posedge clk)
begin
    if(reset)
        begin
            end
        else
            begin
                if(stallE || stallM)
                    begin
                        dataF_nxt.raw_instr <= dataF_nxt.raw_instr;
                        dataF_nxt.pc <= dataF_nxt.pc;
                    end
                else if(stallI || jump || stallI)
                    begin
                        dataF_nxt.raw_instr <= '0;
                        dataF_nxt.pc <= dataF_nxt.pc;
                    end
                else
                    begin
                        dataF_nxt <= dataF;
                    end
            end
        end
end
```

注意到阻塞的优先级应该比清空更高，因为，例如，*stallE*和*stallI*同时生效时，如果信息被刷新，则这条指令会被读为NOP而导致错误；而若信息被保留，因为后续流水线被阻塞，故信息也不会无限地传送下去导致读到多次相同的有效指令。

清空的逻辑很简单，将 *ctl.nopsignal* 设为1即可，这样读到这条指令时会自动认为是NOP而跳过。

这里有一点值得注意，在前面的实验中，*jump*的判断被提前至*decode*阶段，但由于存在流水线后续阶段阻塞时，转发信号还没有正确计算出的问题，故*jump*的判断应该在后续流水线没有阻塞时执行：

```
if(ctl.branch && ~stallE && ~stallM)
```

其他改动

本实验没有再进行其他模块逻辑或设计上的改动。

缓存的实现

本实验实现了八组两路的组相联cache，写操作的处理采用了写回机制，替换策略为LRU，整体架构使用了有限状态机的书写方法。

组相联

本实验采用了data和meta分开存储的方式，利用了两个SinglePortRam。

```
//data
RAM_SinglePort #(
    .ADDR_WIDTH(8),
    .DATA_WIDTH(64),
    .BYTE_WIDTH(8),
    .READ_LATENCY(0)
) ram_for_data (
    .clk(clk), .en(ram_data.en),
    .addr(addr_data),
    .strobe(ram_data.strobe),
    .wdata(ram_data.wdata),
    .rdata(ram_rdata)
);

//meta
RAM_SinglePort #(
    .ADDR_WIDTH(3),
    .DATA_WIDTH($bits(meta_t) * ASSOCIATIVITY),
    .BYTE_WIDTH($bits(meta_t)),
    .READ_LATENCY(0)
) ram_for_meta (
    .clk(clk), .en(ram_meta.en),
    .addr(index),
    .strobe(ram_meta.strobe),
    .wdata(ram_meta.wmeta),
    .rdata(ram_rmeta)
);
```

并且分开维护二者的更新。

写回机制

本实验在meta里设置了dirty位，当需要替换且被替换块中脏位为1时，状态转入WRITEBACK状态，将被替换块中的数据写回主存。WRITEBACK阶段结束后转入FETCH状态，从替换块的地址找到存储数据取回。

替换策略

替换策略采用LRU，由于一组只有两个块，所以LRU可以很简单的通过记录最近使用块实现。记录只需要8个bit: `u8 last_use;`

然后在状态机里进行记录与保持，只有在INIT状态下，`dreq.valid`为真时，`last_use[index] <= index_line`; 其中index为当前的组号索引，index_line为选择的块，因为一组只有两块，所以只有0和1。在其他情况，都是 `last_use <= last_use`;

在替换时，只需要替换对应的 `~last_use[index]`，因为 `last_use[index]` 记录的是最近的一次访问，所以 `last_use[index]` 的年龄一定比 `~last_use[index]` 的年龄小。

总结

本次实验通过扩展阻塞控制单元，新增一个阻塞信号`stallE`，解决了握手总线的延迟问题；同时，本次实验还新实现了一系列指令，扩展了CPU功能。

本实验报告提供了两次实验的CPU对比图，对为支持随机延迟而进行的流水线改动进行了解释，并简单介绍了实现的乘除法器 and 缓存。