Лабораторная работа №3: Кэш

Цыганов Пётр Игоревич M3139

December 2023

Ссылка на репозиторий

1 Инструментарий:

openjdk 17.0.9 2023-10-17
OpenJDK Runtime Environment Temurin-17.0.9+9 (build 17.0.9+9)
OpenJDK 64-Bit Server VM Temurin-17.0.9+9 (build 17.0.9+9, mixed mode, sharing)

2 Результат работы (Вариант 1)

```
LRU:    hit perc. 96.6571%      time: 4136644
pLRU:   hit perc. 96.6406%      time: 4141563
```

3 Результат расчёта параметров системы (Вариант 1)

3.1 Исходные данные системы

MEM_SIZE = 512 Кбайт
Конфигурация кэша - look-through write-back
Политика вытеснения кэша - LRU и bit-pLRU
CACHE_TAG_LEN = 10 бит
CACHE_LINE_SIZE = 32 байт
CACHE_LINE_COUNT = 64
DATA1_BUS_LEN, DATA2_BUS_LEN = 16 бит (размер шин данных)

3.2 Расчёт остального

$$ADDR\_LEN = \log_2(MEM\_SIZE) = \log_2(512 * 1024) = 19$$

$$CACHE\_OFFSET\_LEN = \log_2(CACHE\_LINE\_SIZE) = \log_2(32) = 5$$

$$CACHE\_IDX\_LEN = ADDR\_LEN - CACHE\_OFFSET\_LEN - CACHE\_TAG\_LEN = 19 - 10 - 5 = 4$$

$$CACHE\_SETS\_COUNT = 2^{CACHE\_IDX\_LEN} = 2^4 = 16$$

$$CACHE\_WAY = \frac{CACHE\_LINE\_COUNT}{CACHE\_SETS\_COUNT} = \frac{64}{16} = 4$$

$$CACHE\_SIZE = CACHE\_LINE\_COUNT * CACHE\_LINE\_SIZE = 64 * 32 = 2048 \text{ байт}$$

$ADDR1\_BUS\_LEN = ADDR\_LEN = 19$ бит, т.к процессор должен передать весь адрес

Так как кэш считывает и записывает всю кэш-строку в оперативную память, то можно отбросить offset, ибо все данные в кэш-линии имеют одинаковый индекс и тэг:

$$ADDR2\_BUS\_LEN = ADDR\_LEN - CACHE\_OFFSET\_LEN = 19 - 5 = 14 \text{ бит}$$

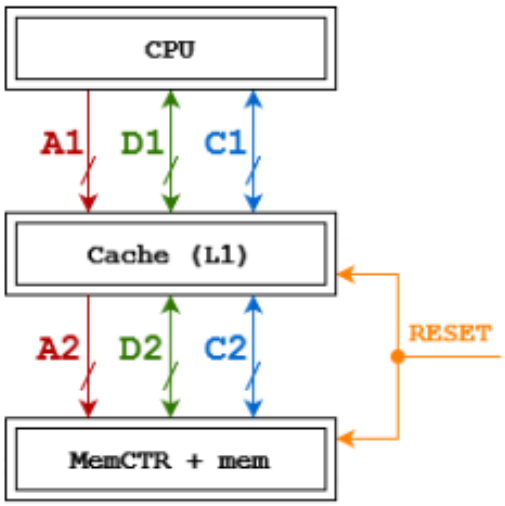
Всего 7 комманд, передающихся по шине C1, и 3 - по C2 =>
$CTR1\_BUS\_LEN = \lceil \log_2 7 \rceil = 3$ бит,
$CTR2\_BUS\_LEN = \lceil \log_2 3 \rceil = 2$ бит

4 Описание работы кода

4.1 Что было реализовано?

Реализованы обе политики вытеснения - LRU и bit-pLRU.

4.2 Структура проекта



Было реализовано 3 основных класса: Cpu, LruCache(реализует интерфейс Cache и в зависимости от параметра pseudo, переданного в конструктор, реализует pLru или Lru) и Mem.

Как и в конфигурации look-through Cpu общается с Cache, а Cache - с Mem. Кэш состоит из кэш-блоков, а те из кэш-линий.

4.3 Consts - константы

Конфигурация системы задаётся с помощью констант, которые были описаны выше. Все эти константы лежат в классе Consts.

4.4 Вспомогательные классы

4.4.1 Address

Address хранит в себе валидный адрес на ячейку в памяти, естестественно адресс - неизменяемый, но может возвращать новый адрес, образованный из этого сдвигом на shift байт. Реализованы getTag(), getIndex(), getOffset() и mainAddress() - адресс с занулённым offset.

```
private final int addr;

public Address(int addr) {
    if (addr < 0 || addr >= (1 << (Consts.ADDR_LEN))) {
        throw new IllegalArgumentException("Invalid addr len");
    }
    this.addr = addr;
}
```

4.4.2 Counter

Counter - простая оболочка для счётчика, которая нужна, чтобы считать такты. Преимущество Counter над простым интом - то, что его можно передавать в другие функции по ссылке. Реализует методы int get() - получить значение и void add(int x) - добавить x к счётчику.

4.5 LruCache

В конструктур LruCache передаётся ссылка на память source, с которой будет общаться Cache, и флажок boolean pseudo. Если pseudo = true, то в данном кэше реализована политика pLru, иначе - Lru.

Отличие первого варианта от второго отличается лишь в реализации кэш-блоков, из которых состоит кэш. Вытеснение происходит в них, поэтому более подробно про то, как вытесняются кэш-линии описано в разделе 4.6 про кэш-блоки.

Методы getData и writeData в начале проверяют есть ли кэш-линия в кэше или нет. Если её нет, то они её выгружаются из памяти:

```
if (blocks.get(index).find(tag) < 0) {
    time.add(Consts.CACHE_MISS_TIME);
    byte[] data = source.getData(addr.mainAddress(), Consts.CACHE_LINE_SIZE, time);
    int writeTime = blocks.get(index).writeData(addr, data);
    time.add(writeTime);
    cacheMissCount += 1;
} else {
    time.add(Consts.CACHE_HIT_TIME);
}
```

P.S: Counter time используется для подсчёта тактов, он передаётся cpu в кэш, чтобы кэш туда записывал все свои использованные такты. Source - это память системы.

4.5.1 getData(Address addr, int bytes, Counter time)

Кэшу процессор может послать комманды C1_READ8, C1_READ16, C1_READ32. Эти комманды реализует функция getData(Address addr, int bytes, Counter time) возвращает bytes байт по адресу addr. Bytes может принимать значения 1, 2 или 4, остальные варианты - невозможны.

getData проверяет есть ли в блоке Index кэш-строка с тегом tag. Если есть, то кэш обрабатывает эту ситуацию за 6 тактов и начинает возвращать ответ, иначе - за 4 такта он это понимает и идёт в блок вытеснять строку и записывать новую, количество тактов, которое он на это потратит определяется ситуацией.

4.5.2 writeData(Address addr, byte[] newData, Counter time)

Кэшу процессор может послать аналогичные комманды на запись. Эти функции реализует writeData(Address addr, byte[] newData, Counter time). Если кэш-строка уже есть в кэше, то запись произойдёт за 6 тактов, иначе - за 4 + вытеснение кэш-линии из блока.

4.6 CacheBlock

Реализован класс AbstractCacheBlock, реализующий интерфейс CacheBlock. Есть два вида кэш-блоков - LruCacheBlock и pLruCacheBlock. Они отличаются друг от друга политикой вытеснения. AbstractCacheBlock просит реализовать наследников метод used(index) и push(Address addr, byte[] data).

used вызывается тогда, когда была использована кэш-линия под номером index(в Lru перемещает в начало массива, а в pLru записывает во флажок 1), а push записывает новую кэш-линию, при этом вытесняя старую.

4.6.1 LruCacheBlock

В LruCacheBlock поддерживается инвариант, что кэш-линии лежат в массиве в порядке, в котором они использовались, то есть 1ая строка использовалась совсем недавно, вторая позже и т.д. Тогда push может просто вытеснить последнюю строку массива. Если эта строка модифицировалась когда-либо, то её необходимо записать в память. На место этой строки записывается новая и перемещается в начало массива.

Когда используется кэш-строка вызывается used() и строка перемещается в начало массива

4.6.2 pLruCacheBlock

Вытесняется самая левая строка с нулевым флажком. Когда используется кэш-строка, вызывается метод used он записывает во флажок 1. При этом если все флажки равны 1, то он зануляет все остальные.

4.6.3 find(int tag)

```
public int find(int tag) {
    for (int i = 0; i < Consts.CACHE_WAY; i++) {
        if (cacheLines.get(i).getTag() == tag) {
            return i;
        }
    }
    return -1;
}
```

Функция перебирает все кэш-линии и возвращает номер той, у которой тэг равен заданному

4.7 CacheLine

Хранит массив байтиков размера CACHE_LINE_SIZE, тэг и флаг dirtyFlag, который говорит изменялись ли в ней данные. CacheLine имеет метод changeData(int offset, byte[] data), который изменяет данные, начиная с offset. После вызова этого метода флаг становится равным 1.

boolean isDirty() возвращает dirtyFlag.

4.8 Mem

Хранит в себе данные, возвращает данные и записывает данные по адресу.

4.9 Подсчёт тактов

Время отклика – расстояние в тактах от первого такта команды до первого такта ответа. То есть в начале происходит команда за 1 такт(параллельно идут другие данные), а потом обработка.

1) Процессор отсылает запрос на запись/чтение данных - 1 такт.

2) Кэш-попадание - 6 + запись в память, если что-то выталкивается

Кэш-промах - 4 + получение данных из памяти + запись в память, если что-то вытесняется.

3) Выгружаются данные из памяти за 1 на команду + 100 на обработку + 16 на передачу в кэш.

Запись происходит за 1 на команду + 100 на обработку + 1 на ответ

4) Кэш посылает ответ процессору за 1 такт. Причём если был запрос на получение данных, то $\lceil data\_size \div 2 \rceil$ тактов тратиться , ответ в данном случае бесплатный, потому что данные передаются параллельно как минимум также по тактам.

Количество тактов, которое тратит функция mmul() считается строго по условию.

Все факторы учитываются. Мало-по-малу собирается много тактов, потому что вместе они сила!