Лабораторная работа №3	10	2023
Моделирование системы “процессор-кэш-память”	Лихтар Анна Викторовна	

Инструментарий и требования к работе: работа выполняется на C/C++ (C11 и новее / C++20), Python (3.11.5). Используемый язык программирования – C++, Версия – C++17.

Ссылка на репозиторий: https://github.com/skkv-mkn/mkn-comp-arch-2023-cache-likhhtar

Результат работы написанной программы

```
LRU:    hit perc. 96.6571%      time: 4081739
pLRU:   hit perc. 96.6406%      time: 4086454
```

Результат расчета параметров системы

MEM_SIZE	512 Кбайт
ADDR_LEN	19 бит
CACHE_WAY	4
CACHE_TAG_LEN	10 бит
CACHE_IDX_LEN	4 бит
CACHE_OFFSET_LEN	5 бит
CACHE_SIZE	2048 байт
CACHE_LINE_SIZE	32 байт
CACHE_LINE_COUNT	64
CACHE_SETS_COUNT	16
ADDR1_BUS_LEN	19
ADDR2_BUS_LEN	14
CTR1_BUS_LEN	3
CTR2_BUS_LEN	2

$$ADDR\_LEN = log_2 MEMSIZE = 19 \text{ бит}$$

$$CACHE\_OFFSET\_LEN = log_2 CACHE\_LINE\_SIZE = 5 \text{ бит}$$

$$CACHE\_IDX\_LEN = ADDR\_LEN - CACHE\_TAG\_LEN - CACHE\_OFFSET\_LEN = 19 - 10 - 5 = 4 \text{ бит}$$

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

$$CACHE\_WAY = \frac{CACHE\_LINE\_COUNT}{2^{CACHE\_IND\_LEN}} = \frac{64}{16} = 4$$

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

По шине A1 передается адрес за 1 такт. Таким образом, передается 19 бит (*ADDR_LEN*). По шине A2 не нужно передавать смещение, поэтому передается *ADDR_LEN – CACHE_OFFSET_LEN = 19 – 5 = 14 бит*.

Команды для C1: C1_READ8, C1_READ16, C1_READ32, C1_WRITE8, C1_WRITE16, C1_WRITE32, C1_RESPONSE.

Команды для C2: C2_READ_LINE, C2_WRITE_LINE, C2_RESPONSE.

Размер шин команд равен $\log_2$ (количество команд). Таким образом, размер шины C1 равен 3, а размер C2 – 2.

Описание работы написанного кода

Конфигурация кэша – look-through и write-back. Что это значит?

Write-back означает, что данные сначала записываются в кэш, а потом при необходимости записываются в оперативную память. Таким образом избегаются лишние запросы к оперативной памяти.

Look-through означает, что данные сначала ищутся в кэше, а потом, если не нашлись, то и в оперативной памяти.

Была смоделирована работа кэш-процессора с обоими предложенными вариантами политики вытеснения (LRU и bit-pLRU). Разберемся, что это значит.

LRU. Если расшифровать аббревиатуру, получится *Least Recently Used*. Храним время последнего обращения к кэш-строке. При взаимодействии со строкой её «возраст» становится самым маленьким. Если нужно заместить, убираем самую старшую строку.

Bit-pLRU. Работает достаточно похоже на LRU (но наоборот :), потому что ищем строку с наименьшим «возрастом»), при этом «возраст» строк в нем может быть только 0 и 1. Тогда после нахождения подходящей строки с возрастом 0 делаем ее возраст равным 1. Если в это время возраст всех строк, кроме этой, был равным 1, то сделаем их 0.

Чтобы смоделировать работу кэша, был создан class Cache. Его поля выглядят следующим образом.

```
int tags[CACHE_SETS_COUNT][CACHE_WAY]{}, lru[CACHE_SETS_COUNT][CACHE_WAY]{};
bool mas[CACHE_SETS_COUNT][CACHE_WAY]{}, mod[CACHE_SETS_COUNT][CACHE_WAY]{};
int requestCount, hitCount, time;
bool is_lru; // 1 - LRU, 0 - bit_pLRU
```

В массивах всего *CACHE_LINE_COUNT* элементов. В массиве *tags* хранятся тэги, в *lru* – значения, которые хранят «возраст» строк. *Mas* отвечает за то, хранится ли уже хорошая строка в кэше, *mod* – за то, нужно ли будет отправить эти данные в оперативную память при записи на это место другого элемента.

Значение *requestCount* хранит количество обращений к кэшу, *hitCount* – количество кэш-попаданий, *time* – такты.

Is_lru показывает, какой политикой вытеснения сейчас пользуется кэш. Если *is_lru = true*, то *LRU*, иначе *bit_pLRU*.

При обращении к кэшу вызывается функция ***find***, которая выглядит следующим образом.

```
bool Cache::find (int addr, bool W_R) {
    int set = Cache::getSet(addr);
    int tag = Cache::getTag(addr);
    this->requestCount++;

    if (W_R) return this -> checkWrite(set, tag);
    else return this -> checkRead(set, tag);
}
```

По адресу функция находит *set* и *tag*. Увеличивается общее количество обращений к памяти. Если обращение было сделано для записи, то вызывается функция *checkWrite*, если для чтения – *checkRead*.

Номер сета получается посредством вызова следующей функции.

```
int Cache::getSet (int addr) {
    return (addr >> CACHE_OFFSET_LEN) % CACHE_SETS_COUNT;
}
```

Этот метод принимает адрес и возвращает индекс набора кэша, в который должен быть помещен этот адрес. Он делает это путем сдвига адреса на длину смещения кэша (*CACHE_OFFSET_LEN*) и затем берет остаток от деления на количество наборов кэша (*CACHE_SETS_COUNT*).

Тег получается посредством вызова следующей функции.

```
int Cache::getTag (int addr) {
    return (addr >> (CACHE_OFFSET_LEN + CACHE_IDX_LEN));
}
```

Этот метод возвращает тег из адреса памяти. Он сдвигает биты адреса вправо на количество бит, равное сумме длин смещения кэша и длины индекса кэша, чтобы получить только теговую часть адреса.

Рассмотрим работу ***checkWrite***.

```
bool Cache::checkWrite(int set, int tag) {
    for(int i = 0; i < CACHE_WAY; i++) {
        if (tag == this->tags[set][i]) {
            this->hit();
            this->mod[set][i] = true;

            if (this -> is_lru) this->lru[set][i] = 0;
            else this->bit_pLRUupd(set, i);

            return true;
        }
    }

    this->missed(set, tag, true);
    return false;
}
```

Проходимся по кэш-строкам нашего набора и проверяем, лежит ли необходимая строка уже там. Если лежит (т. е. теги равны), то это кэш-попадание. Вызывается функция *hit*, которая увеличит значение тактов на 6 и значение кэш-попаданий на 1. Мы затронули эту кэш-линию после ее прочтения из оперативной памяти, поэтому *mod[set][i]* теперь равен *true*. Далее идет логика для разных политик вытеснения. Если это LRU, то мы делаем ее самой молодой, т.е. равной 0. Если это bit_pLRU, то вызывается следующая функция.

```
void Cache::bit_pLRUupd(int set, int idx) {
    bool all = true;
    for (int i = 0; i < CACHE_WAY; i++) {
        if (i == idx) this->lru[set][i] = 1;
        else if (this->lru[set][i] == 0) all = false;
    }
    if(all) {
        for (int i = 0; i < CACHE_WAY; i++) {
            if (i != idx) this->lru[set][i] = 0;
        }
    }
}
```

Ее логику я уже описала выше (там, где описан *Bit-pLRU*).

Если наша строка не лежит в кэше, вызывается функция ***missed***.

```
void Cache::missed(int set, int tag, bool W_R) {
    int lru_row;
    if (this -> is_lru) lru_row = this->LRU(set);
    else lru_row = this->bit_pLRU(set);
    this->miss();

    if(W_R && mod[set][lru_row]) {
        addTime(CACHE_LINE_SIZE * 8 / 16);
        addTime(100); //memory response
    }

    addTime(1); // c2 - отправляет команду, чтобы прочитать значение из памяти
    addTime(100); //memory response
    addTime(CACHE_LINE_SIZE * 8 / 16);

    this->mas[set][lru_row] = true;
    this->tags[set][lru_row] = tag;
    this->mod[set][lru_row] = W_R;
    if (this -> is_lru) this->lru[set][lru_row] = 0;
    else this->bit_pLRUupd(set, lru_row);
}
```

Первым делом здесь вызываются функции, которые ищут строку, в которую мы будем записывать данные. Для разных политик вытеснения вызываются разные функции.

Для *LRU*:

```
int Cache::LRU(int set) {
    int maxLRU = -1, idx = -1;
    for (int i = 0; i < CACHE_WAY; i++) {
        if (this->lru[set][i] > maxLRU) {
            maxLRU = this->lru[set][i];
            idx = i;
        }
    }
    return idx;
}
```

Просто ищется самая старая строка.

Для *bit-pLRU*:

```
int Cache::bit_pLRU(int set) {
    for (int i = 0; i < CACHE_WAY; i++) {
        if (this->lru[set][i] == 0) {
            return i;
        }
    }
    this->bit_pLRUupd(set, 0);
    return 0;
}
```

Если есть строка с нулевым возрастом, то возвращаем её. Если все значения 1, то вызывается функция bit_pLRUupd, после выполнения которой все значения, кроме нулевой строки будут равны 0.

Вернемся к разбору функции *missed*.

После нахождения строк для замещения вызывается функция *miss*, которая увеличит счетчик тактов на 1.

Происходит проверка. Если программа сейчас пытается записать что-то в эту строку, но при этом она была изменена после ее последнего чтения из оперативной памяти (*mod*), то сначала необходимо отправить эту измененную строку в оперативную память. По шине D2 за один такт передается 16 бит, тогда отправление кэш-линии займет *(CACHE_LINE_SIZE * 8 / 16)* тактов, т.к. ее размер хранится в байтах. Параллельно по шинам A2 и C2 передаются адрес и команда соответственно. Далее память будет отвечать 100 тактов.

Теперь можем спокойно делать запрос к оперативной памяти. По A2 и C2 за один такт отправляем адрес и команду в оперативную память для чтения. Далее 100 тактов ждем ответ от памяти. И затем получаем кэш-линию по шине D2.

Далее делаем массивы с валидными значениями (добавляем тег, true для mas, mod=true для read). Также делаем новую кэш-линию самой молодой.

Теперь поймем, как работает ***checkRead***.

```
bool Cache::checkRead(int set, int tag) {
    if (this -> is_lru) this->LRUupd(set);

    for (int i = 0; i < CACHE_WAY; i++) {
        if(tag == this->tags[set][i] && this->mas[set][i]) {
            this->hit();

            if (this -> is_lru) this->lru[set][i] = 0;
            else this->bit_pLRUupd(set, i);
            return true;
        }
    }

    this->missed(set, tag, false);
    return false;
}
```

Логика похожа на write. Сначала сделаем все строки этого сета старше на 1 (если LRU). Затем ищем необходимую кэш-строку. Если нашлась, то вызываем *hit* и изменяем значения *lru*.

Если строка не нашлась, вызываем *missed*.

Теперь опишу такты для самой задачи. При инициализации добавлялся один такт, например:

```
int pc = cSt;
cache->addTime(1); //init
```

На каждой итерации цикла добавляется два такта (сама итерация плюс сравнение).

При каждом обращении к кэшу параллельно отправляются адрес и команда за один такт, затем вызывается функция *find*, а потом по D1 возвращается значение. Пример:

```
cache->addTime(1); // a1 c1
cache -> find(pa + k * aSZ, false);
cache->addTime(1); // 8 бит, значит, 8 / 16, 1 такт
```

Сложение занимает 1 такт, умножение 5 тактов.

Выход из функции – 1 такт.