Лабораторная работа 3	Группа 05	2023
Кэш	Ратьков Андрей Игоревич	

1 Инструментарий, описание, ссылки

1.1 Инструментарий

1. VS Code
2. Язык C++
3. Typst для написания отчёта ♡

1.2 Описание

Необходимо программно смоделировать работу кэша процессора в двух вариантах: с политиками вытеснения LRU и bit-pLRU. Реализованную модель необходимо использовать для определения процента попаданий (число попаданий к общему числу обращений) и общего времени (в тактах), затраченного на выполнение задачи.

1.3 Ссылка на репозиторий и краткое изложение того, что я сделал

https://github.com/skkv-mkn/mkn-comp-arch-2023-cache-AndrewRatkov

Были реализованы обе политики вытеснения LRU и bit-pLRU и получены следующие результаты работы программы:

```
LRU:  hit perc. 96.6571%  time: 3878864
pLRU: hit perc. 96.6406%  time: 3883781
```

2 Реализация кэша

2.1 Подчёт параметров

Далее везде, где написано выражение $\log_2 x$ если x — не степень 2, подразумевается, что логарифм округляется вверх до целого числа.

Для подсчётов параметров системы используются следующие формулы:

1. $\text{MEM\_SIZE} = 2^{\text{ADDR\_LEN}}$
2. $\text{CACHE\_SIZE} = \text{CACHE\_LINE\_SIZE} \cdot \text{CACHE\_LINE\_COUNT}$
3. $\text{CACHE\_LINE\_SIZE} = 2^{\text{CACHE\_OFFSET\_LEN}}$
4. $\text{ADDR\_LEN} = \text{CACHE\_TAG\_LEN} + \text{CACHE\_IDX\_LEN} + \text{CACHE\_OFFSET\_LEN}$
5. $\text{CACHE\_LINE\_COUNT} = \text{CACHE\_SETS\_COUNT} \cdot \text{CACHE\_WAY}$
6. $\text{CACHE\_SETS\_COUNT} = 2^{\text{CACHE\_IDX\_LEN}}$

Параметр	Обозначение	Формула вычисления	Значение
Объём оперативной памяти	MEM_SIZE	дано в описани	512 Кбайт = 2^{19} байт
Конфигурация кэша	—	дано в описани	look-through write-back
Политика вытеснения кэша	—	дано в описании	LRU и bit-pLRU
Длина адреса тега	CACHE_TAG_LEN	дано в описании	10 бит
Объём данных в кэш-линии	CACHE_LINE_-SIZE	дано в описании	32 байт

Количество кэш-линий в кэше	CACHE_LINE_COUNT	дано в описании	$64 = 2^6$
Объём полезных данных в кэше	CACHE_SIZE	CACHE_LINE_COUNT · CACHE_LINE_SIZE	$32 \cdot 2^6 = 2^{11}$ байт
Размер адреса	ADDR_LEN	$\log_2(\text{MEM\_SIZE})$	19 бит
Длина смещения внутри кэш-линии	CACHE_OFFSET_LEN	$\log_2(\text{CACHE\_LINE\_SIZE})$	5 бит
Длина индекса блока кэш-линий	CACHE_IDX_LEN	ADDR_LEN – CACHE_TAG_LEN – CACHE_OFFSET_LEN	$19 - 10 - 5 = 4$ бита
Количество блоков кэш-линий	CACHE_SETS_COUNT	$2^{\text{CACHE\_IDX\_LEN}}$	$2^4 = 16$
Ассоциативность	CACHE_WAY	$\frac{\text{CACHE\_LINE\_COUNT}}{\text{CACHE\_SETS\_COUNT}}$	$\frac{64}{16} = 4$
Битность шины A1	ADDR1_BUS_LEN	ADDR_LEN	19
Битность шины A2	ADDR2_BUS_LEN	CACHE_TAG_LEN + CACHE_IDX_LEN	14
Битность шины D1	DATA1_BUS_LEN	дано в описании	2 байта (16 бит)
Битность шины D2	DATA2_BUS_LEN	дано в описании	2 байта (16 бит)
Битность шины C1	CTR1_BUS_LEN	log(количество команд между процессором и кэшем)	$\log_2 7 = 3$ бита
Битность шины C2	CTR2_BUS_LEN	log(количество команд между MEM и кэшем)	$\log_2 3 = 2$ бита

2.2 Как вообще работает это чудо

2.2.1 Простенькие структурки

- Структурка для хранения типа политики вытеснения кэша

```
enum struct Policy{
      LRU=0,
      bit_pLRU=1
};
```

- Структура для хранения типа команды C1.

```
enum struct COMMAND1{
    READ8=0, // read 8 bytes
    READ16=1, // read 16 bytes
    READ32=2, // read 32 bytes
    WRITE8=3, // write 8 bytes
    WRITE16=4, // write 16 bytes
    WRITE32=5, // write 32 bytes
//    RESPONSE=6
};
```

- `struct Address` — для хранения 19-битного адреса. У него есть методы `tag()`, `idx()` и `offset()`, которые выводят целые числа, бинарные записи которых соответствуют подзаписям адреса (тегу, индексу, оффсету).
- `struct CacheLine` — иммитатор кэш-линии. Имеет слелующие поля:

- `tag` (тег)
- `valid` (валидная / инвалидная она: кэш-линии, которые в начале ещё ни разу не заполнялись, называются инвалидными)
- `dirty` (“грязность кэш-линии”: кэш-линия называется грязной, если после того, как её прочитали из MEM-а, её поменяли в кэше. Тогда при удалении из кэша её буде необходимо перезаписать в MEM)
- `f` (в LRU — число от 0 до CACHE_WAY - 1 (порядок кэш-линии в смысле давности её последнего использования), в bit-pLRU — 0 или 1).
  Как меняется параметр `f` в политике LRU: в начале у всех инвалидных кэш-линий (у которых `valid = false`) он равен 0:
  - При чтении/записи кэш-линии, когда эта кэш-линия есть в кэше (в состоянии `valid = true`). Пусть t — значение её параметра f. Тогда у всех остальных валидных кэш-линий, у которых этот параметр `f` меньше чем t, он увеличивается на 1. Параметр `f` у кэш-линии, которую читаем/пишем делается равным 0.
  - Когда этой кэш-линии нет в кэше, и при этом не все кэш-линии валидные, выбирается первая инвалидная кэш-линия, делается валидной, в ней `f` становится 0. У остальных валидных кэш-линий этот параметр увеличивается на 1).
  - Когда этой кэш-линии нет в кэше, и при этом все кэш-линии валидные (когда все валидные, поддерживается условие, что они все пронумерованы параметром `f` от 0 до CACHE_WAY - 1), выбирается единственная кэш-линия, у которой этот параметр равен CACHE_WAY - 1, меняется на новую, после чего её значение `f` делается равным 0. У остальных кэш-линий значение параметра `f` увеличивается на 1.

  Как меняется параметр `f` в политике bit-pLRU: в начале у всех инвалидных кэш-линий (у которых `valid = false`) он равен 0:
  Когда все кэш-линии становятся валидными, будем поддерживать, что всегда есть та, у которой значение параметра `f` равно 0 и та, у которой этот параметр 1.
  - Когда в кэше есть инвалидные кэш-линии, чтение/запись той, которая уже есть, оставляет её флаг `f` равным 1. А появление новой кэш-линии — найти первую по порядку инвалидную, сделать её валидной и записать в неё текущую кэш-линию. После чего сделать её флаг `f` равным 1. Если случается такое, что у всех кэш-линий `f = 1`, то он делается равным 0 у всех, кроме последней кэш-линии (последней в том смысле, что в последний раз работали именно с ней).
  - Когда все кэш-линии валидные: при работе с кэш-линией, которая уже есть в кэше, её параметр `f` делается равным 1. При этом, если у всех кэш-линий `f = 1`, то он делается равным 0 у всех, кроме этой кэш-линии. А при работе с кэш-линией, которой в кэше нет, выбирается первая по порядку кэш-линия с параметром `f = 0`, меняется, после чего её параметр `f` делается равным 1. И если окажется, что у всех кэш-линий этот параметр 1, то он делается равным 0 у всех остальных.

  В общем, мораль в том, что у самых недавно использованных кэш-линий этот параметр равен 1, у тех, кто давнее всего использовался, он 0.

  А также методы `tacts_to_remove_from_stack()` — сколько тактов надо для того, чтобы удалить кэш-линию из стека (если она “грязная”, то не 0, иначе 0) и `print()` для дебага.
- `struct info_op` — маленькая структурка, которая потом понадобится, когда я буду считать для каждого запроса к кэшу:
  - `HITS` — случилось ли попадание (true / false).
  - `TACTS` — сколько тактов понадобилось на этот запрос
- `struct CacheSet` — вектор из CACHE_WAY кэш-линий. Содержит следующие методы:

- `void init()` — для инициализации
- `void print()` – для дебага
- `bool is_not_filled()` — отвечает, если ли инвалидные кэш-линии
- `bool first_not_filled()` — если есть инвалидные, то какой номер первой
- `void renumerate(int i, Policy pol)` — переставляет флаги f у кэш-линий, когда при запросе у нас была обработана i-ая из них по политике pol.
- `info_op compute(Address addr, COMMAND1 cmd, Policy pol)` — если запрос с адресом addr и командой cmd поступил в текущий кэш-сет (политика pol), то оно обрабатывает запрос, меняет какую-то кэш-линию (если надо) и выдаёт объект типа info_op — как раз информацию, случилось ли попадание, и сколько тактов потребовалось.

- `struct Cache` — иммитатор кэша. Просто CACHE_SETS_COUNT кэш-линий. Имеет методы
  - `void init()` — инициализация
  - `void print()` — для дебага
  - `info_op compute(Address addr, COMMAND1 cmd, Policy pol)` — для обработки запроса (нужно просто переслать этот запрос `addr.idx()`-ой ячейке)

2.3 main

1. Введём следующие константы количеств тактов: сколько тактов что занимает (16 тактов занимает переписывание кэш-линии из кэша м MEM и обратно, 100 тактов отвечает MEM, 4 такта нужно кэшу, чтобы в результате промаха отправить запрос к памяти, 6 тактов нужно кэшу, чтобы начать отвечать)

```
  // constants
  const int TIME_DELETE_UNSAVED = 101; // sending cache line from cache to MEM and saving it in MEM
  const int TIME_READ_WHEN_FOUND = 6; // reading cacheline when it is in cache
  const int TIME_READ_WHEN_NOT_FOUND = 121; // reading cacheline when it is not in cache:
  const int TIME_WRITE_WHEN_FOUND = 6; // writing to a cacheline when it is in cache
  const int TIME_WRITE_WHEN_NOT_FOUND = 121; // writing to a cacheline when it is not in cache
```

   Константы выбраны такими, потому что
   - при перезаписи кэш-линии из кэша в MEM: MEM начнёт отвечать через 100 тактов и ему понадобится 1 такт, чтоб передать ответ в кэш по C2_RESPONSE.
   - когда кэш ищет то, что в нём есть, ему надо 6 тактов.
   - когда кэш ищет то, чего в нём нет, ему надо 4 такта перед тем, как отправить запрос в MEM, потом 100 тактов, пока MEM отвечает, 1 такт — MEM передаёт в кэш команду C2_RESPONSE (то, что он будет сейчас писать кэш-линию в кэш), и 16 тактов, пока MEM пересылает в кэш кэш-линию по D2.

     Пересылка одной кэш-линии из кэша в MEM занимает 16 тактов, так как размер кэш-линии — 32 байта, а по шине D2 за такт передаётся 2 байта (16 бит). Поэтому на пересылку тратится $32 : 2 = 16$ тактов.

2. хеш-таблицы
   `std::unordered_map<int, int> a_i_j_to_pointer, b_i_j_to_pointer, c_i_j_to_pointer`
   понадобятся, чтобы по ячейке матрицы быстро понимать, какой адрес первого байта её информации. Например, a_i_j_to_pointer[$i \cdot K + j$] — это указатель на первый байт, в котором записана ячейка a[i][j].

3. Создаём иммитаторы обоих кэшей

```
Cache lru_cache;
lru_cache.init();
Cache bit_plru_cache;
bit_plru_cache.init();
```

4. Мы будем считать такты работы процессора без работы с кэшем отдельно. Для этого вводим следующие переменные
   - `int add_assign=0;` — количество сложений с присваиванием типа `s += ...;`
   - `int assign=0;` — количеств присваиваний типа `s = ...;`
   - `int mul=0;` — количество умножений, которые делает процессор
   - `int inc=0;` — количество инкрементаций в циклах
   - `int compares=0;` — количество переходов в циклах (GOTO) (то есть количество сравнений в циклах)

   Тогда количество тактов, которые делает процессор, не считая операций с кэшем и MEM-ом будет вычисляться по формуле
   `int tacts = add_assign + assign + mul * 7 + inc + compares + 1;`
   тут `mul` умножаем на 7, так как по условию каждое умножение стоит 7 тактов (5 + 2 — ещё 2 на чтение данных (каждого из множителей) из кэш-линии), а прибавление 1 в конце — это такт, потраченный на выход из функции.

5. Для каждой политики кэша будем считать, сколько тактов ушло на обращения к этому кэшу, сколько было попаданий и сколько промахов:

   ```
   int lru_tacts = 0;
   int bit_plru_tacts = 0;

   int lru_hits = 0;
   int bit_plru_hits = 0;

   int lru_misses = 0;
   int bit_plru_misses = 0;
   ```

6. Далее написан немного видоизменённый код задачи, внутри него происходят обращения к кэшам (где мы что-то читаем или пишем в матрицы a, b, c) (соответственно увеличиваются переменные из пункта 4), а также увеличиваются счётчики из пункта 3.

7. В итоге остаётся просто вывести ответ (вывод дебага я убрал):

   ```
   printf("LRU:\thit perc. %3.4f%%\ttime: %u\npLRU:\thit perc. %3.4f%%\ttime: %u\n",
   100 * (double)(lru_hits) / (lru_hits + lru_misses), tacts + lru_tacts, 100 * (double)
   (bit_plru_hits) / (bit_plru_hits + bit_plru_misses), tacts + bit_plru_tacts);
   ```

Кажется, на этом всё. Я благодарен читателю за то, что он добрался до этого места.

Figure 1: Когда узнал, что лабу назвали в честь тебя