Лабораторная работа №2	М3136	2022
Моделирование схем в Verilog	Шибанов Игорь Александрович	

Цель работы: построение кэша и моделирование системы “процессор-кэш-память” на языке описания Verilog.

Инструментарий и требования к работе: весь код пишется на языке Verilog, компиляция и симуляция – Icarus Verilog 11.

Описание (формулировка задачи из условия)

Имеется следующее определение глобальных переменных и функций:

```
#define M 64
#define N 60
#define K 32
int8 a[M][K];
int16 b[K][N];
int32 c[M][N];

void mmul()
{
  int8 *pa = a;
  int32 *pc = c;
  for (int y = 0; y < M; y++)
  {
    for (int x = 0; x < N; x++)
    {
      int16 *pb = b;
      int32 s = 0;
      for (int k = 0; k < K; k++)
      {
        s += pa[k] * pb[x];
        pb += N;
      }
      pc[x] = s;
    }
```

```
    pa += K;
    pc += N;
  }
}
```

Сложение, инициализация переменных и переход на новую итерацию цикла, выход из функции занимают 1 такт. Умножение – 5 тактов. Обращение к памяти вида pc[x] считается за одну команду.

Массивы последовательно хранятся в памяти, и первый из них начинается с 0.

Все локальные переменные лежат в регистрах процессора.

По моделируемой шине происходит только обмен данными (не командами).

Определите процент попаданий (число попаданий к общему числу обращений) для кэша и общее время (в тактах), затраченное на выполнение этой функции.

Вариант

Параметры системы (общие)

<table>
<tr><td colspan="4">CPU</td></tr>
<tr><td rowspan="2">Команды</td><td>CPU → Cache</td><td>0 – C1_NOP
1 – C1_READ8
2 – C1_READ16
3 – C1_READ32
4 – C1_INVALIDATE_LINE
5 – C1_WRITE8
6 – C1_WRITE16
7 – C1_WRITE32</td><td rowspan="2">Команда 4 означает инвалидацию всей кэш-линии, содержащей указанный адрес.

Число в командах означает кол-во бит данных, запрашиваемое данной командой.

Команды, запрашивающие несколько байт, не</td></tr>
<tr><td>CPU ← Cache</td><td>0 – C1_NOP
7 – C1_RESPONSE</td></tr>
</table>

			могут пересекать кэш-линию. NOP – no operation. Response – ответ на команду.
Кэш (look-through write-back)			
Политика вытеснения		LRU	
Команды	Cache → Mem	0 – **C2_NOP** 2 – **C2_READ_LINE** 3 – **C2_WRITE_LINE**	Команды пишут и читают порциями, равными размеру кэш-линии.
	Cache ← Mem	0 – **C2_NOP** 1 – **C2_RESPONSE**	
Служебные биты		V (valid), D (dirty)	Если valid установлен в 0, то данная кэш-линия свободна и состояние остальных битов не важно. dirty означает, что кэш-линия хранит изменённые данные, которые ещё не записаны в память.

Параметры (вариант 1)

Кэш (продолжение)	
Ассоциативность	2 – **CACHE_WAY**
Размер тэга адреса	10 бит – **CACHE_TAG_SIZE**

Размер кэш-линии	16 байт – **CACHE_LINE_SIZE**	Размер полезных данных.
Кол-во кэш-линий	64 – **CACHE_LINE_COUNT**	
Память		
Размер памяти	512 Кбайт – **MEM_SIZE**	

Вычисление недостающих параметров системы

- **MEM_SIZE** = 512 Кбайт
- **CACHE_SIZE = CACHE_LINE_SIZE * CACHE_LINE_COUNT =** 1024 байта
- **CACHE_LINE_SIZE** = 16 байт
- **CACHE_LINE_COUNT =** 64
- **CACHE_WAY = 2**
- **CACHE_SETS_COUNT = CACHE_LINE_COUNT / CACHE_WAY =** 64 / 2 = 32
- **CACHE_TAG_SIZE =** 10 бит
- **CACHE_SET_SIZE =** $\boldsymbol{log_2}$**(CACHE_SETS_COUNT) =** 5 бит
- **CACHE_OFFSET_SIZE =** $\boldsymbol{log_2}$**(CACHE_LINE_SIZE) =** 4 бит
- **CACHE_ADDR_SIZE = CACHE_TAG_SIZE + CACHE_SET_SIZE + CACHE_OFFSET_SIZE =** 19 бит
- **A1 = A2 = CACHE_TAG_SIZE + CACHE_SET_SIZE =** 15
- **C1 =** $log_2 8$ = 3 бит
- **C2 =** $log_2 3$ = 2 бит
- **D1 = D2 =** 16 бит

Кэш - это меньшая по размеру и более быстрая память, расположенная ближе к ядру процессора, в которой хранятся копии данных из часто используемых ячеек основной памяти.

Принцип работы

Данные в кэше делятся на кэш линии, а кэш линии в свою очередь объединяются в сеты (блоки). От процессора к кэшу идут адрес байта в памяти, данные, команда на исполнение.

Адрес байтов в памяти состоит из tag, set, offset:

tag	set	offset
CACHE_TAG_SIZE	CACHE_SET_SIZE	CACHE_OFFSET_SIZE

Set – номер блока в кэше

Offset – байт с которого начинаются полезные данные в кэш линии

Tag – тэг, хранящийся в каждой кэш линии

Если кэш находит у себя нужную кэш линию по тэгу – это называется кэш попаданием. Тогда кэш может быстро ответить на команду. Если же кэш не находит кэш линию – это называется кэш промахом. Тогда ему нужно обратиться в память и взять кэш линию оттуда. Для в память достаточно передать в качестве адреса tag + set. Если все место в сете занято другими кэш линиями, кэш должен выбросить одну из них и записать новую кэш линию на ее место, после чего вернуть ответ на команду процессору.

Аналитическое решение задачи

Моим аналитическим решением является код, написанный на java. Он эмулирует работу системы, однако не учитывает передачу данных, а только такты. В коде написаны соответствующие комментарии.

Передача по шинам

Рассмотрим такой случай: процессор передает по шине данные, и в моделируемой схеме кэш должен поймать эти данные в тот же такт. Однако кэш может проверить наличие данных в любой момент такта, когда они еще не пришли, из-за чего произойдет утечка. Чтобы исправить данную проблему, я сделал так, чтобы CPU, Memory работали по фронту синхронизации, а Cache по спаду. За такт принимается промежуток между

двумя сменами тактового сингала с 0 на 1. В таком случае обработка в Cache будет на полтакта позже, чем передача данных из CPU. Для удобства будем каждый раз прибавлять количество тактов, умноженное на 2, а в конце просто разделим результат на 2. Далее в описании работы схемы умножение на 2 будет упускаться.

Я начал написание аналитического решения с класса **Memory** (MemCTR + mem). По условию задачи время отклика памяти составляет 100 тактов. Далее рассмотрим 2 случая. 1) C2 == 2, тогда на передачу данных от памяти к кэшу уйдет 8 тактов (8 * CACHE_LINE_SIZE / D1 = 8 * 16 / 16). 2) C2 == 3, тогда на передачу данных от памяти к кэшу уйдет 1 такт. Передача данных от кэша к памяти включена в 100 тактов. В классе Memory я возвращаю, получившуюся задержку в методе changeMemory.

Далее я приступил к написанию класса **CacheLine**. В нем я сделал поля в соответствии с рисунком 1, а также добавил поле LRU.

valid	dirty	tag	data
1	1	CACHE_TAG_SIZE	CACHE_LINE_SIZE

Рисунок 1

После этого я написал класс **Block** (set). Т.к ассоциативность в моем случае это 2, Block будет состоять из 2 кэш линий. В данном классе я буду считать время ответа кэша в методе changeBlock. Если tag равен одному из тэгов кэш линий, значит случилось кэш попадание и нужно вернуть время задержки 6. Иначе через 4 такта кэш должен начать работать с памятью. С помощью политики вытеснения LRU найдем выкидываемую кэш линию. Проверим ее на валидность (битик valid), и, если нужно, запишем в память (битик dirty). При записи в Memory прибавим количество потраченных тактов в памяти к нашей задержке. Далее рассмотрим 2 случая.

1) 1 <= C1 <= 3 (чтение).

В этом случае считаем нужную кэш линию из памяти, запишем в кэш и вернем ее в CPU. Стоит отметить, что drity = false, т.к данная кэш линия считана из памяти без модификаций.

2) 5<=C1<=7 (запись).

В этом случае также считаем нужную кэш линию из памяти и запишем в нее данные из CPU. Полученную кэш линию запишем в кэш и

отправим ответ по C1 обратно в CPU. В данном случае dirty = true, т.к мы модифицировали кэш линию.

Политика вытеснения LRU принцип работы

LRU - это алгоритм, при котором вытесняются элементы, которые дольше всего не запрашивались. При любой модификации какой-то кэш линии в ее поле LRU я записываю 1, а в поле LRU другой кэш линии из блока (сета) я записываю 0. Соответственно, когда мне нужно выкинуть кэш линию, я ищу ту, у которой поле LRU == 0.

На следующем шагу я создал класс **Cache**. Он состоит из 32 блоков (CACHE_SETS_COUNT) или 64 кэш линий (CACHE_LINE_COUNT**)**.

Согласно рисунку 2, найдем tag и set. Отмечу, что по шине A1 в аналитическом решении передается весь адрес целиком. Далее внесем изменения в нужный блок и сохраним такты (tact += blocks[set].changeBlock(tag, C1)).

tag	set	offset
CACHE_TAG_SIZE	CACHE_SET_SIZE	CACHE_OFFSET_SIZE

Рисунок 2

Далее рассмотрим 2 случая.

1) 1<=C1<=3.

Тогда передача данных из CPU в Cache займет 2 такта (адрес передается за 2 такта), однако она уже была включена в задержку 4 или 6 (кэш промах или кэш попадание). Если C1 == 1 или C1 == 2, то передача данных из Cache в CPU займет 1 такт (по шине D1 передается 8 или 16 бит информации за 1 такт). Если C1 == 3, то передача данных из Cache в CPU займет 2 такта (по шине D1 передается 32 бита информации за 2 такта).

2) 5<=C1<=7

Тогда передача данных из CPU в Cache займет 2 такта (адрес передается за 2 такта и по шине D1 данные предадутся за 2 такта в худшем случае), но она уже была включена в задержку 4 или 6 (кэш промах или кэш попадание). Передача данных из Cache в CPU займет 1 такт (по C1 передается 7 за 1 такт).

Осталось только создать **CPU** и проэмулировать код. В соответствии с условием задачи я посчитал такты, затраченные на сложение, умножение, инициализация переменных и переход на новую итерацию цикла, выход из функции.

Как формируется адрес байта в памяти

Посчитаем сколько байтов занимают массивы a, b и c:

bytesOfA = M * K (int8)

bytesOfB = K * N * 2 (int16)

bytesOfC = M * N * 4 (int32)

Теперь найдем адреса байт куда ссылаются pa, pb, pc, исходя из того, что массивы хранятся в памяти последовательно, и первый из них начинается с 0:

pa = 0

pb = bytesOfB

pc = bytesOfB + bytesOfC

Далее найдем адрес k-ого элемента в каждом из массивов:

a: pa + k (int8)

b: pb + 2*k (int16)

c: pc + 4*k (int32)

Аналогично можно сместиться на k элементов вперед по массиву:

pa = pa + k (int8)

pb = pb + 2*k (int16)

pc = pc + 4*k (int32)

Теперь можно передавать в кэш нужный адрес в памяти. Осталось только посчитать процент кэш попаданий.

Заметим, что если кэш справился меньше чем за 100 тактов, то это было кэш попадание, иначе кэш промах.

Также, каждый раз, когда начинает работать какой-то из классов, нужно прибавлять полтакта из-за смены синхронизации.

Результат аналитического решения:

```
Time: 5717362
Percentage of hits: 228080 / 249600 = 91%

Process finished with exit code 0
```

Моделирование заданной системы на Verilog

Написание кода на Verilog я начал с модуля **Memory**, который работает по фронту.

```
always @(posedge clk)
```

Согласно условию, я проинициализировал все байтики памяти.

```
for (int i = 0; i < MEM_SIZE / CACHE_LINE_SIZE; i += 1) begin
    for (int j = 0; j < CACHE_LINE_SIZE; j += 1) begin
        memory[i][j] = $random(SEED)>>16;
    end
end
```

Принцип работы шин C1, C2

Шины C1, C2 имеют модификатор inout и подключены к регистрам _C1 и _C2. В каждый момент времени модули передают по шинам сигнал 0 (NOP) силы strong0. Как только модулю нужно послать данные по C, он присваивает _C соответствующие значение силы supply1. Это значение перебьет 0 на шине. После выполнения команды модуль присваивает _C - 0.

```
assign(supply1, strong0) C1 = _C1;
```

Принцип работы шин D1, D2

Шины D1, D2 имеют модификатор inout и подключены к регистрам _D1 и _D2. В каждый момент времени модули передают по шинам сигнал z. Как только модулю нужно послать данные по шине D, он присваивает _D соответствующие значение, а после меняет его обратно на z.

Запись и чтение в Memory

Все данные в Memory для удобства разбиты на кэш линии.

```
reg[7:0] memory[0:MEM_SIZE/CACHE_LINE_SIZE-1][0:CACHE_LINE_SIZE-1];
```

Чтобы найти нужную кэш линию, достаточно обратиться по A2 в memory, т.к по A2 передается tag + set.

1) C2 == 2

Тогда сделаем задержку на 100 тактов, и будем выводить нужную кэш линию в D2 с учетом little endian.

```
for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin // little endian
    _D2 =  (memory[A2_Buffer][i] << 8) + memory[A2_Buffer][i + 1];
    #(1 * 2);
end
```

После чего пошлем по C2 1.

2) C2 == 3

Запишем данные в нужную кэш линию с учетом little endian.

```
for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin // little endian
    memory[A2_Buffer][i] = (D2 >> 8);
    memory[A2_Buffer][i + 1] = D2 % (1 << 8);
    #(1 * 2);
end
```

После чего через 92 (т.к данные мы получали за 8 тактов, а итоговое время отклика памяти = 100) такта пошлем по C1 1.

Принцип работы little endian

Предположим, что мы передаем по шине байтики 1 2 3 4 по 2 байта за такт. Тогда в первый такт передастся 3 4, а во второй 1 2. Т.е передача происходит с конца.

После Memory я приступил к написанию **Cache.** Полезная информация хранится в массиве cache, разделенном на 32 блока по 2 кэш линии в каждом.

```
reg[7:0] cache[0:CACHE_SETS_COUNT-1][0:CACHE_WAY-1][0:CACHE_LINE_SIZE-1];
```

Изначально все битики valid инициализируются 0.

```
for (int i = 0; i < CACHE_SETS_COUNT; i += 1) begin // инициализация кэша
    for (int j = 0; j < CACHE_WAY; j += 1) begin
        valid[i][j] = 0;
    end
end
```

Обработка запросов кэшом

Обработка запросов происходит в always:

```
always @(negedge clk)
```

Первым делом я нашел нужные tag, set и offset. За первый такт по A1 передаются tag + set, а за второй offset.

```
tag = A1 >> CACHE_SET_SIZE;
set = A1 % (1 << CACHE_SET_SIZE);
#(1 * 2);
offset = A1;
```

Работа с памятью (чтение и запись)

В моем коде для чтения и записи написаны соответствующие define. Разберемся с каждым из них:

1) writeToMemory

```
`define writeToMemory(cacheLine) \ //кэш линия которую записываем
    A2 = (tag << CACHE_SET_SIZE) + set; \ // адрес куда пишем
    _C2 = 3; \ // команда записи
    for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin \ // запись
в шину D2
        _D2 =  (cacheLine[i] << 8) + cacheLine[i + 1]; \
        #(1 * 2); \
    end \
    _C2 = 0; \
    _D2 = 'z; \
    while(C2 != 1) begin \ // ожидание ответа от памяти
        #(1 * 2); \
    end \
    #(1 * 2);
```

В A2 посылаем адрес нужной кэш линии, а по C2 посылаем команду на запись. Далее по D2 передаем кэш линию в формате little endian и ждем ответ от памяти.

2) readFromMemory

```
`define readFromMemory(cacheLine) \ // кэш линия, куда нужно читать
    A2 = (tag << CACHE_SET_SIZE) + set; \ // адрес куда пишем
    _C2 = 2; \ // команда чтения
    #(1 * 2); \
    _C2 = 0; \
    while (C2 != 1) begin \ // ожидание ответа от памяти
        #(1 * 2); \
    end \
```

```
        for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin \ //чтение в
кэш линию
            cacheLine[i] = (D2 >> 8); \
            cacheLine[i + 1] = D2 % (1 << 8); \
            #(1 * 2); \
        end
```

В A2 посылаем адрес нужной кэш линии, а по C2 посылаем команды на чтение. Ждем ответ от памяти, а далее записываем кэш линию из шины D2.

Чтение и запись в кэше

Общий код для 1<=C1<=3 && 5<=C1<=7:

cacheHitIndex – индекс кэш линии (в блоке) с которой будет происходит модификация (изначально = -1).

Аналогично аналитическому решению найдем нужную кэш линию в кэше (кэш попадание).

```
    for (int i = 0; i < CACHE_WAY; i += 1) begin
        if (cacheTag[set][i] == tag && valid[set][i] == 1) begin// кэш
попадание
            cacheHitIndex = i;
        end
    end
```

Если такой не оказалось найдем кэш линию (кэш промах), которую можно выкинуть с помощью LRU вытеснения. Индекс полученной кэш линии поместим в cacheHitIndex.

```
    if (cacheHitIndex == -1) begin // кэш промах
        cacheHitIndex = 0;
        for (int i = 0; i < CACHE_WAY; i += 1) begin
            if (valid[set][i] == 0 || LRU[set][i] == 0) begin // LRU
                cacheHitIndex = i;
            end
        end
    end
```

При необходимости запишем выкидываемую кэш линию обратно в память.

```
    if (valid[set][cacheHitIndex] == 1 && dirty[set][cacheHitIndex] == 1)
begin
        `writeToMemory(cache[set][cacheHitIndex])
    End
```

Далее считаем нужную кэш линию из Memory.

```
    `readFromMemory(cache[set][cacheHitIndex])
```

Если 5<=C1<=7 в кэш линию нужно внести нужные значения из D1.

Задержка при кэш попадании и кэш промахе:

При кэш попадании я делаю задержку 5, т.к мы уже сделали задержку в 1 такт при чтении адреса, а в итоге время отклика кэша при кэш попадании должно быть 6. При кэш промахе я делаю задержку 3 по той же самой причине.

Отдельный случай C1 == 4:

В данном случае я ищу нужную кэш линию в кэше и при нахождении меняю бит valid на false. В случае кэш попадания - задержка 6 тактов. В случае кэш промаха – 4.

Код для C1 == 4:

```
cacheHitIndex = -1;
for (int i = 0; i < CACHE_WAY; i += 1) begin
    if (cacheTag[set][i] == tag && valid[set][i] == 1) begin
        cacheHitIndex = i;
    end
end

if (cacheHitIndex != -1) begin
    valid[set][cacheHitIndex] = 0;
    LRU[set][1 - cacheHitIndex] = 1;
end
```

На следующем шаге я создал модуль **CPU**, который работает по фронту.

```
always @(posedge clk)
```

Работа с кэшом (чтение и запись)

Для чтения и записи я написал define.

1) get (readFromCache)

```
`define get(adr, value, bitSize) \
    if (bitSize == 8) \
        _C1 = 1; \
    if (bitSize == 16) \
        _C1 = 2; \
    if (bitSize == 32) \
        _C1 = 3; \
    A1 = adr >> CACHE_OFSET_SIZE; \
    #(1 * 2); \
```

```
    A1 = adr % (1 << CACHE_OFSET_SIZE); \
    #(1 * 2); \
    _C1 = 0; \
    while(C1 != 7) begin \
        #(1 * 2); \
    end \
    value = D1; \
    #(1 * 2); \
    if(bitSize == 32) begin \
        value = (D1 << 16) + value; \
        #(1 * 2); \
    end \
    _C1 = 0;
```

adr – адресс в памяти
value – место, куда записывается результат
bitSize – битность переменной

Первым шагом в зависимости от битности посылаем нужные данные по C1. Далее ждем ответ от кэша и записываем в value D1. Если битность переменной == 32, на следующем такте запишем оставшуюся часть данных.

2) set (writeToCache)

```
`define set(adr, value, bitSize) \
    if (bitSize == 8) \
        _C1 = 5; \
    if (bitSize == 16) \
        _C1 = 6; \
    if (bitSize == 32) \
        _C1 = 7; \
    _D1 = value % (1<<16); \
    A1 = adr >> CACHE_OFSET_SIZE; \
    #(1 * 2); \
    A1 = adr % (1 << CACHE_OFSET_SIZE); \
    _D1 = (value >> 16); \
    #(1 * 2); \
    _C1 = 0; \
    while(C1 != 7) begin \
        #(1 * 2); \
    end \
    #(1 * 2); \
    _D1 = 'z; \
    _C1 = 0;
```

value – значение, которое записывается в кэш

Первым шагом в зависимости от битности посылаем нужные данные по C1. Далее по D1 посылаем первую часть байтиков, а через такт вторую (если bitSize != 32 по D1 пойдут 0). После этого ждем ответ от кэша.

Основной цикл в CPU идентичен циклы CPU в аналитическом решении.

Для подсчета кэш попаданий я завел переменную t и присваивал ей значение $time, до обращения в кэш. После обращения я вычитал из $time – t, тем самым получая, сколько затратил кэш на обращение.

```
 t = $time;
`get(pb + x * 2, getPb, 16) //pb[x] чтение из памяти
if ($time - t < 2 * 100)

    cacheHit += 1;
cacheCount += 1;
```

В **testbench** все модули соединяются проводами. Также здесь происходит изменение синхронизации.

```
count = 0;
while(count <= 15000000) begin // изменение синхронизации
    count += 1;
    clk = 1 - clk;
    #1;
end
```

Также в модулях Cache и Memory есть блоки always для RESET (сброс к начальным значениям) и _DUMP (C_DUMP, M_DUMP – вывод полезных данных в консоль). Данные блоки always работают по фронту.

Результат моделирования Verilog

```
PS C:\Users\Бобыч\Desktop\End> iverilog  -g2012 -o a.out .\testbench.sv
PS C:\Users\Бобыч\Desktop\End> vvp .\a.out
Time:              5717362
Percentage of hits:      228080/     249600          91%
PS C:\Users\Бобыч\Desktop\End>
```

Результат аналитического решения и результат моделирования на Verilog совпали.

Источники:

https://bimlibik.github.io/posts/cache-algorithms/

https://en.wikipedia.org/wiki/CPU_cache

https://www.youtube.com/watch?v=7n_8cOBpQrg&ab_channel=AlekOS

Листинг кода

```
module Memory #(
    parameter _SEED = 225526,
    parameter MEM_SIZE = 512 * 1024,
    parameter CACHE_LINE_SIZE = 16,
    parameter CACHE_TAZ_SIZE = 10,
    parameter CACHE_SET_SIZE = 5
)
(
    input wire clk,
    input wire[CACHE_TAZ_SIZE + CACHE_SET_SIZE - 1:0] A2,
    inout wire[15:0] D2,
    inout wire[1:0] C2,

    input wire M_DUMP,
    input wire RESET
);

    integer SEED = _SEED;

    reg[CACHE_TAZ_SIZE + CACHE_SET_SIZE - 1:0] A2_Buffer;// буффер для
сохранения A2
    reg[15:0] _D2;
    reg[1:0] _C2;

    assign D2 = _D2;
    assign(supply1, strong0) C2 = _C2;

    reg[7:0] memory[0:MEM_SIZE/CACHE_LINE_SIZE-1][0:CACHE_LINE_SIZE-1]; //
data

    initial begin
        _C2 = 0;// по C2 отправляем NOP
        _D2 = 'z;
        for (int i = 0; i < MEM_SIZE / CACHE_LINE_SIZE; i += 1) begin
//инициализация
            for (int j = 0; j < CACHE_LINE_SIZE; j += 1) begin
                memory[i][j] = $random(SEED)>>16;
            end
        end

    end


    always @(posedge clk) begin
        A2_Buffer = A2; // сохранение значения A2
        if(C2 == 2) begin // чтение
            #(100 * 2); // задержка памяти
            _C2 = 1;// ответ
```

```
            for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin // little
endian
                _D2 =  (memory[A2_Buffer][i] << 8) + memory[A2_Buffer][i + 1];
                #(1 * 2);
            end
            _C2 = 0;
            _D2 = 'z;
        end
        if(C2 == 3) begin // запись
            for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin // little
endian
                memory[A2_Buffer][i] = (D2 >> 8);
                memory[A2_Buffer][i + 1] = D2 % (1 << 8);
                #(1 * 2);
            end
            #(92 * 2);// задержка 92 т.к в for тратит 8 тактов 100 - 8 = 92
            _C2 = 1;// ответ
            #(1 * 2);
            _C2 = 0;
        end
    end


    always @(posedge M_DUMP) begin
        for (int i = 0; i < MEM_SIZE / CACHE_LINE_SIZE; i += 1) begin //
выгрузка памяти
            for (int j = 0; j < CACHE_LINE_SIZE; j += 1) begin
                $write("%d ", memory[i][j]);
            end
            $display();
        end
    end

    always @(posedge RESET) begin
        for (int i = 0; i < MEM_SIZE / CACHE_LINE_SIZE; i += 1) begin // сброс
к начальным значениям
            for (int j = 0; j < CACHE_LINE_SIZE; j += 1) begin
                memory[i][j] = $random(SEED)>>16;
            end
        end
    end

endmodule

//Запись кэш линии в память
`define writeToMemory(cacheLine) \ //кэш линия которую записываем
    A2 = (tag << CACHE_SET_SIZE) + set; \ // адрес куда пишем
    _C2 = 3; \ // команда записи
```

```
    for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin \ // запись в шину
D2
        _D2 =  (cacheLine[i] << 8) + cacheLine[i + 1]; \
        #(1 * 2); \
    end \
    _C2 = 0; \
    _D2 = 'z; \
    while(C2 != 1) begin \ // ожидание ответа от памяти
        #(1 * 2); \
    end \
    #(1 * 2);


//Чтение кэш линии из памяти
`define readFromMemory(cacheLine) \ // кэш линия, куда нужно читать
    A2 = (tag << CACHE_SET_SIZE) + set; \ // адрес куда пишем
    _C2 = 2; \ // команда чтения
    #(1 * 2); \
    _C2 = 0; \
    while (C2 != 1) begin \ // ожидание ответа от памяти
        #(1 * 2); \
    end \
    for (int i = CACHE_LINE_SIZE - 2; i >= 0; i -= 2) begin \ //чтение в кэш
линию
        cacheLine[i] = (D2 >> 8); \
        cacheLine[i + 1] = D2 % (1 << 8); \
        #(1 * 2); \
    end

module Cache #(
    parameter CACHE_LINE_SIZE = 16,
    parameter CACHE_TAZ_SIZE = 10,
    parameter CACHE_SET_SIZE = 5,
    parameter CACHE_OFSET_SIZE = 4,
    parameter CACHE_WAY = 2,
    parameter CACHE_SETS_COUNT = 32,
    parameter CACHE_LINE_COUNT = 64
)
(
    input wire clk,

    input wire[CACHE_TAZ_SIZE+CACHE_SET_SIZE-1:0] A1,
    inout wire[15:0] D1,
    inout wire[3:0] C1,

    output reg[CACHE_TAZ_SIZE + CACHE_SET_SIZE - 1:0] A2,
    inout wire[15:0] D2,
    inout wire[1:0] C2,
```

```
    input wire C_DUMP,
    input wire RESET
);

    reg[15:0] _D1;
    reg[3:0] _C1;
    reg[15:0] _D2;
    reg[1:0] _C2;

    reg[3:0] C1_buffer;// буффер для сохранения C1

    assign D1 = _D1;
    assign(supply1, strong0) C1 = _C1;
    assign D2 = _D2;
    assign(supply1, strong0) C2 = _C2;

    reg valid[0:CACHE_SETS_COUNT-1][0:CACHE_WAY-1]; // битики valid
    reg dirty[0:CACHE_SETS_COUNT-1][0:CACHE_WAY-1];// битики dirty
    reg[CACHE_TAZ_SIZE-1:0] cacheTag[0:CACHE_SETS_COUNT-1][0:CACHE_WAY-1];//
тэги кэш линий
    reg[7:0] cache[0:CACHE_SETS_COUNT-1][0:CACHE_WAY-1][0:CACHE_LINE_SIZE-1];
// полезная информация

    reg LRU[0:CACHE_SETS_COUNT-1][0:CACHE_WAY-1]; //счетчик LRU

    //разделние адреса A1
    reg[CACHE_TAZ_SIZE-1:0] tag;// tag из адреса
    reg[CACHE_SET_SIZE-1:0] set;// set из адреса
    reg[CACHE_OFSET_SIZE-1:0] offset;// ofset из адреса

    int cacheHitIndex;//индекс кэш попадания
    reg[7:0] cpuData[0:3];//данные, пришедшие по D1 из CPU

    initial begin
        _D1 = 'z;
        _C1 = 0;// NOP
        _D2 = 'z;
        _C2 = 0;// NOP
        for (int i = 0; i < CACHE_SETS_COUNT; i += 1) begin // инициализация
кэша
            for (int j = 0; j < CACHE_WAY; j += 1) begin
                valid[i][j] = 0;
            end
        end
    end

    always @(negedge clk) begin
        if (1 <= C1 && C1 <= 3) begin //чтение
```

```
            cacheHitIndex = -1;
            C1_buffer = C1;// сохранение C1

            tag = A1 >> CACHE_SET_SIZE;
            set = A1 % (1 << CACHE_SET_SIZE);
            #(1 * 2);
            offset = A1;

            for (int i = 0; i < CACHE_WAY; i += 1) begin
                if (cacheTag[set][i] == tag && valid[set][i] == 1) begin// кэш
попадание
                    cacheHitIndex = i;
                end
            end
            if (cacheHitIndex == -1) begin // кэш промах
                #(3 * 2);// задержка при кэш промахе
                for (int i = 0; i < CACHE_WAY; i += 1) begin
                    if (valid[set][i] == 0 || LRU[set][i] == 0) begin // LRU
вытеснение
                        cacheHitIndex = i;
                    end
                end

                if (valid[set][cacheHitIndex] == 1 &&
dirty[set][cacheHitIndex] == 1) begin // линия занята, ее нужно записать в
память
                    `writeToMemory(cache[set][cacheHitIndex])
                end

                dirty[set][cacheHitIndex] = 0;
                valid[set][cacheHitIndex] = 1;
                `readFromMemory(cache[set][cacheHitIndex])//чтение кэш линии
из памяти
            end
            else begin
                #(5 * 2);// задержка при кэш попадании
            end

            LRU[set][cacheHitIndex] = 1;
            LRU[set][1 - cacheHitIndex] = 0;
            cacheTag[set][cacheHitIndex] = tag;


            // Отправка в CPU
            if (C1_buffer == 1) begin
                _C1 = 7;//ответ
                _D1 = cache[set][cacheHitIndex][offset];
                #(1 * 2);
```

```
                _C1 = 0;
                _D1 = 'z;
            end

            if (C1_buffer == 2) begin
                _C1 = 7;//ответ
                _D1 = (cache[set][cacheHitIndex][offset] << 8) +
cache[set][cacheHitIndex][offset + 1];
                #(1 * 2);
                _C1 = 0;
                _D1 = 'z;
            end

            if (C1_buffer == 3) begin
                _C1 = 7;//ответ
                _D1 = (cache[set][cacheHitIndex][offset + 2] << 8) +
cache[set][cacheHitIndex][offset + 3];
                #(1 * 2);// передаем 32 бита за 2 такта
                _D1 = (cache[set][cacheHitIndex][offset] << 8) +
cache[set][cacheHitIndex][offset + 1];
                #(1 * 2);
                _C1 = 0;
                _D1 = 'z;
            end
        end

        else if (5 <= C1 && C1 <= 7) begin//запись
            C1_buffer = C1;
            cacheHitIndex = -1;
            //чтение data
            if (C1_buffer == 5) begin
                cpuData[0] = D1;
            end

            if (C1_buffer == 6) begin
                cpuData[0] = (D1 >> 8);
                cpuData[1] = D1 % (1 << 8);
            end

            if (C1_buffer == 7) begin
                cpuData[2] = (D1 >> 8);
                cpuData[3] = D1 % (1 << 8);
            end

            tag = A1 >> CACHE_SET_SIZE;
            set = A1 % (1 << CACHE_SET_SIZE);
            #(1 * 2);
            offset = A1;
```

```
            if (C1_buffer == 7) begin// чтение оставшихся 16 бит
                cpuData[0] = (D1 >> 8);
                cpuData[1] = D1 % (1 << 8);
            end

            for (int i = 0; i < CACHE_WAY; i += 1) begin
                if (cacheTag[set][i] == tag && valid[set][i] == 1) begin //
кэш попадание
                    cacheHitIndex = i;
                end
            end
            if (cacheHitIndex == -1) begin // кэш промах
                #(3 * 2);// задержка при кэш промахе
                for (int i = 0; i < CACHE_WAY; i += 1) begin
                    if (valid[set][i] == 0 || LRU[set][i] == 0) begin // LRU
вытеснение
                        cacheHitIndex = i;
                    end
                end

                if (valid[set][cacheHitIndex] == 1 &&
dirty[set][cacheHitIndex] == 1) begin // линия занята, ее нужно записать в
память
                    `writeToMemory(cache[set][cacheHitIndex])
                end

                dirty[set][cacheHitIndex] = 1;
                valid[set][cacheHitIndex] = 1;
                `readFromMemory(cache[set][cacheHitIndex])// чтение кэш линии
из памяти
            end
            else begin
                #(5 * 2);//зарержка при кэш попадании
            end
            LRU[set][cacheHitIndex] = 1;
            LRU[set][1 - cacheHitIndex] = 0;
            cacheTag[set][cacheHitIndex] = tag;

            //внесение изменений в кэш линию
            if (C1_buffer == 5) begin
                cache[set][cacheHitIndex][offset] = cpuData[0];
            end
            if (C1_buffer == 6) begin
                cache[set][cacheHitIndex][offset] = cpuData[0];
                cache[set][cacheHitIndex][offset + 1] = cpuData[1];
            end
            if (C1_buffer == 7) begin
                cache[set][cacheHitIndex][offset] = cpuData[0];
```

```
                cache[set][cacheHitIndex][offset + 1] = cpuData[1];
                cache[set][cacheHitIndex][offset + 2] = cpuData[2];
                cache[set][cacheHitIndex][offset + 3] = cpuData[3];
            end

            _C1 = 7;// ответ
            #(1 * 2);
            _C1 = 0;
        end
        else if (C1 == 4) begin // инвалидация
            cacheHitIndex = -1;

            tag = A1 >> CACHE_SET_SIZE;
            set = A1 % (1 << CACHE_SET_SIZE);
            #(1 * 2);
            offset = A1;

            for (int i = 0; i < CACHE_WAY; i += 1) begin //ищем нужную кэш
линию
                if (cacheTag[set][i] == tag && valid[set][i] == 1) begin
                    cacheHitIndex = i;
                end
            end

            if (cacheHitIndex != -1) begin//если нашли - то делаем ее
невалидной
                valid[set][cacheHitIndex] = 0;
                LRU[set][1 - cacheHitIndex] = 1;
                #(5 * 2);//задержка при кэш попадании
            end
            else begin
                #(3 * 2);//задержка при кэш промахе
            end
            _C1 = 7;
            #(1 * 2);
            _C1 = 0;
        end
    end


    always @(posedge C_DUMP) begin //вывод полезных данных
        for (int i = 0; i < CACHE_LINE_COUNT; i += 1) begin
            $display("Line = %d:", i);
            for (int j = 0; j < CACHE_LINE_SIZE; j += 1) begin
                $write("%d ", cache[i/2][i%2][j]);
            end
            $display();
        end
    end
```

```
    always @(posedge RESET) begin //сброс данных к начальному состоянию
        for (int i = 0; i < CACHE_SETS_COUNT; i += 1) begin
            for (int j = 0; j < CACHE_WAY; j += 1) begin
                valid[i][j] = 0;
            end
        end
    end

endmodule
`define get(adr, value, bitSize) \ // считывание из кэша
    if (bitSize == 8) \ //отправка нужной команды
        _C1 = 1; \
    if (bitSize == 16) \
        _C1 = 2; \
    if (bitSize == 32) \
        _C1 = 3; \
    A1 = adr >> CACHE_OFSET_SIZE; \
    #(1 * 2); \
    A1 = adr % (1 << CACHE_OFSET_SIZE); \
    #(1 * 2); \
    _C1 = 0; \
    while(C1 != 7) begin \ // ожидание ответа от кэша
        #(1 * 2); \
    end \
    value = D1; \ // запись data
    #(1 * 2); \
    if(bitSize == 32) begin \ // продожение записи для 32 бит
        value = (D1 << 16) + value; \
        #(1 * 2); \
    end \
    _C1 = 0;


`define set(adr, value, bitSize) \ //запись в кэш
    if (bitSize == 8) \ //отправка нужной команды
        _C1 = 5; \
    if (bitSize == 16) \
        _C1 = 6; \
    if (bitSize == 32) \
        _C1 = 7; \
    _D1 = value % (1<<16); \ // посылаем данные по шине D1
    A1 = adr >> CACHE_OFSET_SIZE; \
    #(1 * 2); \
    A1 = adr % (1 << CACHE_OFSET_SIZE); \
    _D1 = (value >> 16); \ // посылаем оставшиеся данные
    #(1 * 2); \
    _C1 = 0; \
    while(C1 != 7) begin \ // ожидание ответа от кэша
```

```
        #(1 * 2); \
    end \
    #(1 * 2); \
    _D1 = 'z; \
    _C1 = 0;


`define M 64
`define N 60
`define K 32

module CPU #(
    parameter CACHE_LINE_SIZE = 16,
    parameter CACHE_TAZ_SIZE = 10,
    parameter CACHE_SET_SIZE = 5,
    parameter CACHE_OFSET_SIZE = 4,
    parameter CACHE_WAY = 2,
    parameter CACHE_SETS_COUNT = 32,
    parameter CACHE_LINE_COUNT = 64
)
(
    input wire clk,

    output reg[CACHE_TAZ_SIZE+CACHE_SET_SIZE-1:0] A1,
    inout wire[15:0] D1,
    inout wire[3:0] C1,

    output reg C_DUMP,
    output reg M_DUMP,
    output reg RESET
);
    reg[15:0] _D1;
    reg[3:0] _C1;


    assign D1 = _D1;
    assign(supply1, strong0) C1 = _C1;

    //массивы из условия
    reg[7:0] a[0:`M-1][0:`K-1];
    reg[15:0] b[0:`K-1][0:`N-1];
    reg[31:0] c[0:`M-1][0:`N-1];

    int bytesOfA = `M * `K; //размер массива a
    int bytesOfB = `K * `N * 2;//размер массива b
    int bytesOfC = `M * `N * 4;//размер массива C

    int pa;
    int pb;
```

```
int pc;

int count = 0;

reg [31:0] s;
reg [7:0] getPa;//считывание pa из памяти
reg [15:0] getPb;//считывание pb из памяти
int t;//время

int cacheHit;//число кэш попаданий
int cacheCount;//число кэш обращений


initial begin
    _D1 = 'z;
    _C1 = 0;
end

always @(posedge clk) begin
    cacheHit = 0;
    cacheCount = 0;

    #(3 * 2); // инициализация 3 массивов
    pa = 0;
    #(1 * 2);// int8 *pa = a; инициализация

    pc = bytesOfA + bytesOfB;
    #(1 * 2);// int32 *pc = c; инициализация

    for (int y = 0; y < `M; y += 1) begin
        for (int x = 0; x < `N; x += 1) begin
            pb = bytesOfA;
            #(1 * 2);// int16 *pb = b; инициализация

            s = 0;
            #(1 * 2);// int32 s = 0; инициализация
            for (int k = 0; k < `K; k += 1) begin
                t = $time;//запоминаем на каком такте начинается чтение
                `get(pa + k, getPa, 8) //pa[k] чтение из памяти
                if ($time - t < 2 * 100)
                    cacheHit += 1;
                cacheCount += 1;
                t = $time;//запоминаем на каком такте начинается чтение
                `get(pb + x * 2, getPb, 16) //pb[x] чтение из памяти
                if ($time - t < 2 * 100)
                    cacheHit += 1;
                cacheCount += 1;

                s += getPa * getPb;
```

```
                    #(5 * 2);//pa[k] * pb[x] умножение
                    #(1 *2);//s + pa[k] * pb[x]; сложение
                    #(1 * 2);//s = s + pa[k] * pb[x]; инициализация

                    pb += `N * 2;
                    #(1 * 2);//pb + N; сложение
                    #(1 * 2);//pb = pb + N; инициализация

                    #2;//переход на новую итерацию цикла
                end

                t = $time;//запоминаем на каком такте начинается запись
                `set(pc + x * 4, s, 32)//pc[x] = s; запись в память
                if ($time - t < 100)
                    cacheHit += 1;
                cacheCount += 1;

                #(1 * 2);//переход на новую итерацию цикла
            end
            pa += `K;
            #(1 * 2);//pa + K сложение
            #(1 * 2);//pa = pa + K инициализация

            pc += `N * 4;
            #(1 * 2);//pc + N сложение
            #(1 * 2);//pc = pc + N инициализация

            #(1 * 2);//переход на новую итерацию цикла
        end
        #(1 * 2);//выход из функции
        $display("Time: %d",$time / 2);
        $display("Percentage of hits: %d/%d %d\%",cacheHit, cacheCount,
cacheHit * 100 / cacheCount);
        $finish;
    end

endmodule

`include "Cache.sv"
`include "CPU.sv"
`include "Memory.sv"
module testbench #(
    parameter CACHE_LINE_SIZE = 16,
    parameter CACHE_TAZ_SIZE = 10,
    parameter CACHE_SET_SIZE = 5,
    parameter CACHE_OFSET_SIZE = 4,
    parameter CACHE_WAY = 2,
```

```
    parameter CACHE_SETS_COUNT = 32,
    parameter CACHE_LINE_COUNT = 64
);
    reg clk = 0;
    reg M_DUMP;
    reg C_DUMP;
    reg RESET;

    wire[CACHE_TAZ_SIZE+CACHE_SET_SIZE-1:0] A1;
    wire[15:0] D1;
    wire[3:0] C1;

    reg[CACHE_TAZ_SIZE + CACHE_SET_SIZE - 1:0] A2;
    wire[15:0] D2;
    wire[1:0] C2;
    CPU cpu(clk, A1, D1, C1, C_DUMP, M_DUMP, RESET);
    Cache cache(clk, A1, D1, C1, A2, D2, C2, C_DUMP, RESET);
    Memory memory(clk, A2, D2, C2, M_DUMP, RESET);
    int count;

    initial begin
        count = 0;
        while(count <= 15000000) begin // изменение синхронизации
            count += 1;
            clk = 1 - clk;
            #1;
        end
    end
endmodule
```

Аналитика

```
public class Memory {

    int changeMemory(int C2) {
        int tact = 0; // время отклика
        tact++;//смена синхронизации
        tact += (100 * 2); // временная задержка памяти
        if (C2 == 2) { // читаем
            tact += (8 * 2); // передача от Memory к Cache
        }
        if (C2 == 3) { // пишем
            tact += (1 * 2);// передача данных от Memory к Cache
        }
        return tact;
    }
}
```

```
public class CacheLine {
    public boolean valid; // пуста ли строка
    public boolean dirty; // выгрузили ли в память
    public int tag;// tag для памяти
    public int lru; // порядок вытеснения

    public CacheLine(){
        valid = false;
    }
    public CacheLine(int tag, boolean valid, boolean dirty, int lru){
        this.tag = tag;
        this.lru = lru;
        this.valid = valid;
        this.dirty = dirty;
    }

}

public class Block {
    final static int CACHE_WAY = 2;
    CacheLine lines[] = new CacheLine[CACHE_WAY]; // блок кэш линий
    Memory memory = new Memory(); // наша память (т.к там одна функция можем
делать сколько угодно экземпляров)

    Block() {
        for (int i = 0; i < CACHE_WAY; i++) {
            lines[i] = new CacheLine(); // наши линии
        }
    }

    int changeBlock(int tag, int C1) {
        int tact = 0; // количество тактов
        if (lines[0].valid == true && lines[0].tag == tag) {// попали в кэш
линию
            lines[0].lru = 1;
            lines[1].lru = 0;
            return 6 * 2; // 6 тактов на попадание
        }
        if (lines[1].valid == true && lines[1].tag == tag) {// попали в кэш
линию
            lines[0].lru = 0;
            lines[1].lru = 1;
            return 6 * 2;// 6 тактов на попадание
        }
        tact += (4 * 2);// кэш промах
        int ind;
        if (lines[1].lru == 0) {// политики вытеснения LRU
```

```
            ind = 1;
            lines[0].lru = 0;
        } else {
            ind = 0;
            lines[1].lru = 0;
        }
        if (lines[ind].valid == true && lines[ind].dirty == true) { //
проверка нужно ли записать строчку в память
            tact += memory.changeMemory(3);
            tact++;//смена синхронизации
        }

        if (1 <= C1 && C1 <= 3) { // чтение
            lines[ind] = new CacheLine(tag, true, false, 1);
            tact += memory.changeMemory(2);
            tact++;//смена синхронизации
        }
        if (5 <= C1 && C1 <= 7){
            lines[ind] = new CacheLine(tag, true, true, 1);
            tact += memory.changeMemory(2);
            tact++;//смена синхронизации
        }
        return tact;
    }
}

public class Cache {
    final static int CACHE_SETS_COUNT = 32;
    Block blocks[] = new Block[CACHE_SETS_COUNT];

    Cache() {
        for (int i = 0; i < CACHE_SETS_COUNT; i++) {
            blocks[i] = new Block();
        }
    }

    int changeCache(int A1, int C1) {
        int tact = 0;
        tact++;//смена синхронизации
        int tag = (A1 >> 9); // формирование tag
        int set = (A1 >> 4) % (1 << 5);// формирование set
        tact += blocks[set].changeBlock(tag, C1);// внесение изменений в блок
        if (1 <= C1 && C1 <= 3) { //чтение
            if (C1 == 1 || C1 == 2) {
                tact += (1 * 2);//передача данных из Cache в CPU
            }
            if (C1 == 3) {
                tact += (2 * 2);//передача данных из Cache в CPU
```

```
            }
        }
        if (5 <= C1 && C1 <= 7) {//запись
            tact += (1 * 2);//передача данных из Cache в CPU
        }
        return tact;
    }
}

public class CPU {

    static final int M = 64;
    static final int N = 60;
    static final int K = 32;

    public static void main(String args[]) {
        Cache cache = new Cache();
        int cacheTact;//сколько тактов потратил кэш
        int cacheCount = 0;//количество обращение к кэшу
        int cacheHit = 0;//количество кэш попаданий

        int bytesOfA = M * K;
        int bytesOfB = K * N * 2;
        int bytesOfC = M * N * 4;

        int pa;
        int pb;
        int pc;

        int tact = 0; // количество тактов
        tact += (3 * 2); // инициализация 3 массивов

        pa = 0;
        tact += (1 * 2); // int8 *pa = a; инициализация

        pc = bytesOfA + bytesOfB;
        tact += (1 * 2);// int32 *pc = c; инициализация

        for (int y = 0; y < M; y++) {
            for (int x = 0; x < N; x++) {
                pb = bytesOfA;
                tact += (1 * 2);// int16 *pb = b; инициализация

                tact += 2;// int32 s = 0; инициализация
                for (int k = 0; k < K; k++) {
                    cacheTact = cache.changeCache(pa + k, 1);
                    tact++;//смена синхронизации
                    if (cacheTact < 2 * 100) {
```

```
                cacheHit += 1;
            }
            cacheCount += 1;
            tact += cacheTact;//pa[k] чтение из памяти

            cacheTact = cache.changeCache(pb + x * 2, 2);
            tact++;//смена синхронизации
            if (cacheTact < 2 * 100) {
                cacheHit += 1;
            }
            cacheCount += 1;
            tact += cacheTact;//pb[x] чтение из памяти

            tact += (5 * 2);//pa[k] * pb[x] умножение
            tact += (1 * 2);//s + pa[k] * pb[x]; сложение
            tact += (1 * 2);//s = s + pa[k] * pb[x]; инициализация

            pb += N * 2;
            tact += (1 * 2);//pb + N; сложение
            tact += (1 * 2);//pb = pb + N; инициализация

            tact += (1 * 2);//переход на новую итерацию цикла
        }

        cacheTact = cache.changeCache(pc + x * 4, 7);
        tact++;//смена синхронизации
        if (cacheTact < 2 * 100) {
            cacheHit += 1;
        }
        tact += cacheTact;//pc[x] = s; запись в память
        cacheCount += 1;

        tact += (1 * 2);//переход на новую итерацию цикла
    }
    pa += K;
    tact += (1 * 2);//pa + K сложение
    tact += (1 * 2);//pa = pa + K инициализация

    pc += N * 4;
    tact += (1 * 2);//pc + N сложение
    tact += (1 * 2);//pc = pc + N инициализация

    tact += (1 * 2);//переход на новую итерацию цикла
}
tact += (1 * 2);//выход из функции

System.out.println("Time: " + tact / 2);
```

```
        System.out.println("Percentage of hits: " + cacheHit + " / " +
(cacheCount) + " = " + cacheHit * 100 / cacheCount + "%");
    }
}
```