Лабораторная работа №2	Б10	2022
Моделирование схем в Verilog	Хорохорин Андрей Сергеевич	

Цель работы

Построение кэша и моделирование системы «процессов-кэш память» на языке описания Verilog.

Инструментарий

- Компилятор `Icarus Verilog version 11.0 (stable)`
- SystemVerilog стандарта `IEEE1800-2012`
- Лучший текстовый редактор `Vim` с использованием языкового сервера для SystemVerilog
- gtkwave как средство для просмотра `.vcd` файлов
- LaTeX с расширением `XeLaTeX` для написания отчёта

Задача

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).

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

Вычисление недостающих параметров системы

Вычисление размера областей памяти

Везде далее будет считаться, что байт = 8 бит. Для начала выпишем все параметры кэша и памяти, которые были даны в условии:

- **CACHE_WAY** — 2
- **CACHE_TAG_SIZE** — 10 бит
- **CACHE_LINE_SIZE** — 16 байт
- **CACHE_LINE_COUNT** — 64 линии
- **MEM_SIZE** — 512 Кбайт

Так как всего в нашем распоряжении **CACHE_LINE_COUNT**(64) линии, каждая по **CACHE_LINE_SIZE**(16 байт), то из них легко вычисляется общий объём полезных данных, хранимых нашим кэшом.

$$\textbf{CACHE\_SIZE} = 64 \cdot 16 \text{ байт} = 1024 \text{ байт } = 1 \text{ Кбайт}$$

Исходя из того, что всего необходимо проиндексировать 512 Кбайт памяти:

$$\textbf{CACHE\_ADDR\_SIZE} = \lfloor \log_2(512 \cdot 2^{10}) \rfloor = 19 \text{ бит}$$

Исходя из того, что всего у нас **CACHE_LINE_COUNT**(64) линий и все их нам необходимо разбить в блоки размера **CACHE_WAY**(2), то количество блоков

$$\textbf{CACHE\_SETS\_COUNT} = 64/2 = 32$$

Для индексации внутри каждого блока будет использоваться **CACHE_OFFSET_SIZE**, поэтому его размер зависит непосредственно от размера блока **CACHE_LINE_SIZE**(16 байт)

$$\textbf{CACHE\_OFFSET\_SIZE} = \log_2 16 = 4 \text{ бит}$$

Осталось посчитать размер индекса в наборе кэш-линий **CACHE_SET_SIZE**, он должен индексировать все блоки, которые мы имеем, а их количество нам известно **CACHE_SETS_COUNT**(32).

$$\textbf{CACHE\_SET\_SIZE} = \log_2(32) = 5 \text{ бит}$$

Заметим, что тег адреса, номер блока и сдвиг внутри блока должен однозначно задавать адрес ячейки в оперативной памяти, поэтому можно ещё раз проверить верность расчётов, проверив следующее равенство:

$$\texttt{CACHE\_TAG\_SZ} + \texttt{CACHE\_SET\_SZ} + \texttt{CACHE\_OFFSET\_SZ} = \texttt{CACHE\_ADDR\_SZ}$$
$$10 + 5 + 4 = 19$$

Видим, что всё в порядке — равенство верно.

Подводя итог, ниже перечислим все необходимые параметры системы:

- **MEM_SIZE** — 512 Кбайт
- **CACHE_SIZE** — 1 Кбайт
- **CACHE_LINE_SIZE** — 16 байт
- **CACHE_LINE_COUNT** — 64 линии
- **CACHE_WAY** — 2
- **CACHE_SETS_COUNT** — 32 блока
- **CACHE_TAG_SIZE** — 10 бит
- **CACHE_SET_SIZE** — 5 бит
- **CACHE_OFFSET_SIZE** — 4 бит
- **CACHE_ADDR_SIZE** — 19 бит

Вычисление размерности шин

Начнём с вычисления размера шины адреса между CPU и Cache. В нашей системе обмен данными по этой шине происходит за два такта. За первый необходимо успеть передать данные о теге и о номере блока, а за второй — offset внутри блока. Больше ни для чего эта шина не используется, а значит её размер можно сделать следующим:

$$\texttt{ADDR1\_BUS\_SZ} = \max(\texttt{CACHE\_TAG\_SZ} + \texttt{CACHE\_SET\_SZ}, \texttt{CACHE\_OFFSET\_SZ})$$
$$\texttt{ADDR1\_BUS\_SZ} = \max(10 + 5, 4) = 15 \text{ бит}$$

По шине A2 кэш передаёт интересующий его адрес памяти в контроллер памяти, причём часть адреса offset не передаётся, так как в нашем кэше мы делаем адреса начала кэш линий кратными их размеру. Так как данная шина ни для чего больше не используется, то её размер вычисляется следующем образом:

$$\texttt{ADDR2\_BUS\_SZ} = \texttt{CACHE\_TAG\_SZ} + \texttt{CACHE\_SET\_SZ} = 10 + 5 = 15 \text{ бит}$$

Вычисление размерностей шин команд это тривиальная задача, но для неё нам потребуется вспомнить, какие вообще команды(сигналы) могут передаваться по этим шинам.

Cpu → Cache	**Cpu ← Cache**
0 — **C1_NOP**	0 — **C1_NOP**
1 — **C1_READ8**	7 — **C1_RESPONSE**
2 — **C1_READ16**	
3 — **C1_READ32**	
4 — **C1_INVALIDATE_LINE**	
5 — **C1_WRITE8**	
6 — **C1_WRITE16**	
7 — **C1_WRITE32**	
Cache → Mem	**Cache ← Mem**
0 — **C2_NOP**	0 — **C2_NOP**
2 — **C2_READ_LINE**	1 — **C2_RESPONSE**
3 — **C2_WRITE_LINE**	

Из таблицы хорошо видно, что для передачи необходимых сигналов между Cpu и Cache достаточно 3 бит за такт, а для Cache и Mem — 2 бита за такт. Таким образом, имеем:

CTR1_BUS_SZ = 3 бит
CTR2_BUS_SZ = 2 бит

Аналитическое решение задачи

Попытаемся решить задачу аналитически. Для начала немного более строго сформулировать приведённую в задании систему. Далее будет считаться, что ни процессор ни кэш не обладают ни асинхронностью ни параллельностью. То есть когда процессор отправляет запрос кэшу, он только ждёт ответа и ничего не вычисляет. То же самое относится и к записи. Помимо этого, в ситуации, когда кэш ждёт ответа от памяти, он сначала примет все данные от памяти и только потом начнёт взаимодействие с процессором. То есть никогда не возникает ситуации, что кэш одновременно общается по шине и с процессором и с памятью. В ситуации, когда кэш сообщает памяти новые кэш линии он не ждёт никакого ответа от памяти. Кроме того, в конце выполнения не обязательно все кэш линии будут совпадать с памятью, в подсчёте времени работы не будет закладываться время на то, чтобы полностью синхронизировать кэш и память с точки зрения того, чтобы ни оказалось ни одной кэш линии с параметром dirty=1.

Для того, чтобы упростить себе задачу давайте разделим все такты на два вида: когда процессор просто ждёт ответа кэша и когда процессор что-то вычисляет. Подсчитывать количество попаданий/промахов вручную мне показалось сложным, поэтому был написан следующий код, который симулирует работу кэша по модулю того, что явно не хранит полезные данные и, соответственно, не возвращает их на запрос.

Тип Word используется для указания количества байт, используемых при Read/Write запросах. Класс CacheLine тоже ничего умного не делает, это просто удобный контейнер, представляющий собой кэш-линию, без хранящихся в ней данных.

```
typedef unsigned long long Addr;

enum Word { WORD=8, DWORD=16, QWORD=32 };

struct CacheLine {
    const int size;
    bool valid = 0;
    bool dirty = 0;
    Addr tag = -1;
    int last_call = 0;
    CacheLine& operator = (CacheLine oth) {
        assert(size == oth.size);
        tie(valid, dirty, tag, last_call) = tie(oth.valid, oth.dirty, oth.tag, oth.last_call);
        return *this;
    }
};
```

Ниже описан класс, представляющий собой один блок(set) из кэш-линий. Функции read и write условно эмулируют операцию нашей модели кэша, при этом read ещё возвращает есть ли запрашиваемый адрес в блоке или нет. Наверное, самой интересной функцией является find_LRO, которая ищет кандидата на вытеснение из кэша по следующей логике: если есть свободная линия(valid=0), то вытесняем её, иначе вытесняется линия, к которой условное время последнего запроса было минимальным. За условное время запроса берётся просто количество запросов совершенных до этого. Помимо этого, в классе вычисляется количество вытесняемый кэш линий, которые необходимо записать в память. Эта информация пригодится для работы других классов.

```
struct CacheSet {
private:
```

```
    vector<CacheLine> lines;
    const int line_size;
    int total_mem_pushes = 0;
public:
    CacheSet(int way, int line_size) : line_size(line_size) {
        lines = vector<CacheLine>(way, CacheLine{line_size});
    }

    bool exists(Addr tag) {
        return find_by_tag(tag) != lines.end();
    }

    void read(Addr tag, int call) {
        auto it = find_by_tag(tag);
        if (it != lines.end()) {
            it->last_call = call;
        } else {
            *(find_LRO()) = CacheLine {line_size, 1, 0, tag, call};
        }
    }

    void write(Addr tag, int call) {
        auto it = find_by_tag(tag);
        if (it != lines.end()) {
            it->last_call = call;
            it->dirty = 1;
        } else {
            total_mem_pushes += find_LRO()->dirty;
            *(find_LRO()) = CacheLine {line_size, 1, 1, tag, call};
        }
    }

    int getMemPushes() const {
        return total_mem_pushes;
    }

private:
    vector<CacheLine>::iterator find_by_tag(Addr tag) {
        return find_if(lines.begin(), lines.end(), [&] (const CacheLine& line) {
                return line.valid && line.tag == tag;
            }
        );
    }

    vector<CacheLine>::iterator find_LRO() {
        return min_element(lines.begin(), lines.end(), [] (const CacheLine& a, const CacheLine& b) {
            return tie(a.valid, a.last_call) < tie(b.valid, b.last_call);
        });
    }
};
```

Теперь можно перейти к самому сложному классу — Cache. Он занимается тем, что принимает запрос к памяти, определяет в каком он блоке выполняет его уже на уровне блока. Но помимо этого он считает интересную нам статистику — количество попаданий в кэш, количество промахов и прогнозируемое общее время работы данных операций на реальном кэше. Ещё он считает количество совершённых обращений к памяти в переменной calls, чтобы обеспечивать каждый запрос корректным условным временем.

Стоит отдельно объяснить почему время считается именно таким образом. В условии задания написано, что, в случае попадания, первый такт ответа кэша происходит через 6 тактов после первого такта запроса. Это значит, что если данные были переданы за один такт, то на седьмой такт процессор продолжит исполнять команды, получив ответ на запрос, который он послал ещё на первом такте. Итого на одном запросе процессор простаивает 7 тактов. Но данные от кэша могут передаваться не один такт, поэтому была сделана внутренняя функция transfer_lag, которая по порции данных определяет сколько она будет передаваться. Так как обе шины данных в нашей

системе имеют одинаковую пропускную способность, эта функция используется для аналогичных расчётов при передаче данных от памяти к кэшу.

В случае, когда на кэш поступает запрос записи всё происходит аналогично чтению за исключением одного момента: передачи данных от кэша к процессору не происходит, передаётся только команда-флаг, что всё отработало, но это всё равно занимает один дополнительный такт.

Помимо всего вышеперечисленного, необходимо помнить, что когда мы вытесняем из кэша линию, которая помечена dirty=1, то необходимо послать дополнительный запрос к памяти, эта поправка вычисляется непосредственно в методе get_time, используя насчитанные значения в каждой из CacheLine.

Также есть «волшебная» функция split_addr, в которой происходит много битовых операций, чтобы просто разбить адрес части на tag, set, offset.

```
struct Cache {
private:
    const int sets_cnt;
    const int way;
    const int line_size;        // byte
    const int data_bus_size;    // bits
    const int mem_size;         // byte

    int total_hits = 0;
    int total_misses = 0;
    int total_time = 0;
    int calls = 0;

    vector<CacheSet> sets;

    struct InnerAddr {
        Addr tag, set, offset;
    };
public:
    Cache(int sets_cnt, int way, int line_size, int data_bus_size, int mem_size) :
        sets_cnt(sets_cnt), way(way), line_size(line_size), data_bus_size(data_bus_size),
    mem_size(mem_size) {
        sets = vector<CacheSet>(sets_cnt, CacheSet(way, line_size));
    }

    void read(Word word, Addr addr) {
        calls++;
        auto [tag, set, offset] = split_addr(addr);
        if (sets[set].exists(tag)) {
            total_hits++;
            total_time += 6; // Cache lag
            total_time += transfer_lag(word) + 1; // Cache -> Cpu
                                                  // 8-9
        } else {
            total_misses++;
            total_time += 4 + 100; // Cache + Mem lag
            total_time += transfer_lag(WORD * line_size) + 1; // Mem -> Cache
            total_time += transfer_lag(word); // Cache -> Cpu
                                              // 114-115
        }
        sets[set].read(tag, calls);
    };

    void write(Word word, Addr addr) {
        calls++;
        auto [tag, set, offset] = split_addr(addr);
        if (sets[set].exists(tag)) {
            total_hits++;
            total_time += 6; // Cache lag
            total_time += 1; // Cache->Cpu response
        } else {
            total_misses++;
            total_time += 4 + 100;                          // Cache + Mem lag
            total_time += transfer_lag(WORD * line_size); // Mem -> Cache
            total_time += 1; // Cache->Cpu response
        }
        sets[set].write(tag, calls);
    };
```

```
    int get_hits() const  {
        return total_hits;
    }

    int get_misses() const {
        return total_misses;
    }

    int get_time() const {
        int total_pushes = accumulate(sets.begin(), sets.end(), 0,
            [] (int acc, const CacheSet& el) {
                return acc + el.getMemPushes();
            }
        );
        return total_time + total_pushes * (100 + 1);
    }

private:
    InnerAddr split_addr(Addr addr) const {
        return {
            (addr & ((mem_size - 1) & ~(sets_cnt * line_size - 1))) / (sets_cnt * line_size),
            (addr & (sets_cnt - 1) * line_size) / line_size ,
            addr & (line_size - 1)
        };
    }

    int transfer_lag(int data) const {
        return (data + data_bus_size - 1) / data_bus_size;
    }

};
```

Теперь давайте направим всё реализованное нами «чудо» на решение задачи. Код ниже не симулирует пример из задачи полностью, а лишь обращается с запросами чтения/записи к тем же адресам, но для анализа времени работы кэша это и требуется. Класс PseudoAllocator просто помогает симулировать последовательное выделение памяти для массивов, возвращая адреса начала каждого из них.

```
struct PseudoAllocator {
    Addr last_allocated_addr = 0;

    Addr allocate(int n, int m, int elem_sz) {
        int res = last_allocated_addr;
        last_allocated_addr += n * m * elem_sz;
        return res;
    }
};

int main() {
    // 512Kb == [0x00000...0x7ffff]
    Cache cache(32, 2, 16, 16, 512 * 1024);
    PseudoAllocator alloc;

    const int M = 64;
    const int N = 60;
    const int K = 32;

    int8_t*  a = (int8_t*) alloc.allocate(M, K, sizeof(int8_t));  // a[M][K];
    int16_t* b = (int16_t*)alloc.allocate(K, N, sizeof(int16_t)); // b[K][N];
    int32_t* c = (int32_t*)alloc.allocate(M, N, sizeof(int32_t)); // c[M][N];

    for (int y = 0; y < M; y++) {
        for (int x = 0; x < N; x++) {
            int16_t* pb = b;
            for (int k = 0; k < K; k++) {
                cache.read(WORD, (Addr) (a + k));
                cache.read(DWORD, (Addr) (pb + x));
                pb += N;
            }
            cache.write(QWORD, (Addr) (c + x));
        }
        a += K;
        c += N;
    }
```

```
    cout << "HITS: " << cache.get_hits() << "\nMISSES: " << cache.get_misses() << "\nTOTAL TIME: " <<
↳   cache.get_time() << endl;
    cout << "RATE: " << 1.0 * (cache.get_hits()) / (cache.get_hits() + cache.get_misses()) << endl;

    return 0;
}
```

Ниже приведён вывод данного кода:

```
HITS: 228080
MISSES: 21520
TOTAL TIME: 4274080
RATE: 0.913782
```

Теперь, когда мы знаем за сколько работает кэш, вычислим время работы всего остального.

- Один такт на выход из функции.
- Инициализация переменных, каждая из которых выполняется за один такт. Она явно выполняется в строчках 9, 10, 13, 14. И неявно в каждом из циклов. Будем считать, что переменные локальные для цикла выделяются заново на каждой итерации внешнего цикла. Тогда суммарное количество инициализаций вычисляется как: $3 + M + M \cdot N \cdot 3 = 11587$
- Подсчитаем количество тактов на сложение, включив в них такты на смену итерации цикла. $M \cdot 3 + M \cdot N + M \cdot N \cdot K \cdot 3 = 372672$
- Количество тактов на умножение: $M \cdot N \cdot K \cdot 5 = 614400$

Итого дополнительно имеем: $1 + 11587 + 372672 + 614400 = 998660$. Тогда итоговое прогнозируемое время работы равно: $998660 + 4274080 = 5272740$

Моделирование на языке Verilog

Структура проекта

Стоит начать с общей структуры модели и с её организации на файлы. Логически есть 3 модуля: процессор, кэш и память, причём процессор в нашем случае не был реализован явно, а лишь в тестового окружения на Verilog, который лишь симулирует работу настоящего процессора для кэша.

Пройдёмся по файлам:

- `parameters.sv` для описания глобальных констант и структур.
- `clock.sv` в котором задаётся модуль `Clock`, который используется для синхронизации схем и оценки времени их работы, так как внутри себя он считает количество тактов.

- `cache.sv` и `mem.sv` задают основные модули данного проекта — `Cache` и `Memory`. Отдельно стоит отметить, что в этих же файлах располагаются удобные интерфейсы доступа к ним: `CacheDriver, MemoryDriver`. Они необходимы для избежания дублирования кода, так как к кэшу и к памяти есть нужда обращаться из разных модулей, а писать больше одного раза код, который описывает протокол общения с кэшом очень непродуктивно.

- `mem_testbench.sv` и `cache_testbench.sv` хранят реализацию модулей Unit тестов для памяти и кэша соответственно. Помимо корректности, они проверяют ещё и время работы модулей в различных ситуациях, что оказалось чрезвычайно полезным.

Для подключения модулей используется директива **include**, работающая аналогично одноимённой директиве в C и имеющая те же проблемы: а именно множественный include одного и того-же модуля и, как следствие, ошибка компиляции. В Verilog эта проблема решается точно так как, как и в C, при помощи так называемых include guards.

```
`ifndef CPU_GUARD
`define CPU_GUARD
...
`endif
```

Inout шины

Поясним, как реализованы inout шины на Verilog. Для примера возьмём часть кода из файла `mem.sv`, в которой реализована шина адреса `cmd_w`.

```
module Memory
    (
        inout wire[1:0] cmd_w
    );

    bit owner = 0;
    logic[1:0] cmd;

    assign cmd_w = owner ? cmd : 2'bzz;
```

Для каждой шины в модуле заводится bit переменная обозначающая, владеет ли данный модуль шиной в текущий момент времени. В данном примере она названа `owner`. Помимо этого, каждый **inout** выход модуля имеет logic переменную внутри модуля, данном случае это `cmd`. Она необходима для того, чтобы взаимодействовать с шиной `cmd_w` как output, так как напрямую записать значение в неё не получится. Поэтому делается интересный трюк при помощи конструкции
assign `cmd_w = owner ? cmd : 2'bzz` , которая в конце каждого такта присваивает значение `cmd_w` либо `cmd`, если наш модуль должен выставлять значение на

шине, либо высокоимпедансное состояние для того, чтобы на проводках `cmd_w` принимались значения, которые выставляет другая сторона и не появлялось конфликтов сигналов, которые могут реальной схеме даже привести к короткому замыканию и натурально поджарить нашу схему.

Так как описанный выше трюк приходится осуществлять с любой шиной, работающей на ввод-вывод, то эти подробности реализации помещены в модули-драйверы, для удобства моделирования вышестоящих уровней.

Почему размер шин адреса не совпадает с минимальным достаточным?

Отдельно хочу отметить, что размер шин адреса **`ADDR1_BUS_SIZE`** и **`ADDR2_BUS_SIZE`** в моей реализации округлены до 16, для того, чтобы все шины адреса и данных единообразно задавались целым количеством байт, мне показалось это удобным программировать.

В действительности, если бы я писал код заново, лучшим решением было бы хранить все размеры в битах.

Функция skip

Почти в каждом модуле можно видеть следующую функцию, которая необходима для удобного программирования схемы.

```
    task automatic skip(input longint ticks = 1);
        logic enter_clk = clk;
        while (ticks > 0) begin
            wait(clk != enter_clk);
            wait(clk == enter_clk);
            ticks--;
        end
    endtask
```

Стоит отметить, что очень важным является то, что она ждёт не posedge/negedge, а именно такт, то есть данная функция не меняет чётности синхронизации. Это очень важно и много где используется. К сожалению, я не нашёл и не придумал способа не дублировать её код в каждом из модулей, так как если передавать clk по `input logic clk` аргумент, то он не будет обновляться внутри этой функции. Действительно в стандарте SystemVerilog на этот случай есть **`ref logic`** `clk`, что делало ровно то, что нам надо, но эта конструкция не поддерживается в `Icarus Verilog 11`, поэтому от её использования пришлось отказаться.

Про выравнивания структур

Стоит обратить внимание на ключевое слово **`packed`** в объявлениях структур. Выглядит это примерно так:

```
    typedef struct packed {
        logic[cache_tag_size-1:0] tag;
        logic[cache_set_size-1:0] set;
        logic[cache_offset_size-1:0] offset;
    } cacheAddr;
```

Такая конструкция гарантирует, что все данные структуры будут расположены последовательно, без каких-либо пропусков. Это позволяет значительно упростить себе присваивание значений для этой структуры, пользуясь конструкциями, описанными в следующем пункте.

Проблемы при записи в структуру

Из-за особенностей Verilog в массиве структур нельзя изменить отдельное поле, в этом случае при компиляции выдаётся следующая ошибка.
`error: Array index expressions must be constant here.`
Это решается тем, что записывать структуру целиком можно, это выглядит так:

```
    lines[it] <= {1'b1, 1'b1, total_hits + total_misses,
↳   curAddr.tag, buff_b};
```

В коде выше, конструкция с перечислением внутри фигурных скобок называется concatenatio operator, и занимается тем, что конкатенирует всё внутри в один регистр **`logic`**. Интересным моментом является то, как написаны константы `1'b1`, эта запись позволяет задаёт число в виде регистра заданной длины. В противном случае, запись `1` представляла бы собой регистра размера 32 и вся запись в структуру бы «поплыла».

Про блокирующие и неблокирующие присваивания

В коде выше используется нестандартное присваивание, которое называется неблокирующим. Оно применяется не сразу, а только в самом конце текущего такта, что позволяет делать следующие трюки. Вот код, который за такт меняет значение переменных a и b местами.

```
    a <= b;
    b <= a;
```

Его использование предохраняет от состояния «гонки» присвоения, так как гораздо проще спрогнозировать поведение несколько присвоений, которые происходят одновременно, чем нескольких последовательных. Помимо этого, код в котором меньше блокирующих присваиваний будет лучше преобразовываться в реальную схему.

В моей программе, я старался придерживаться следующей логики: переменные, которые используются для внутренней логики модуля и никак не взаимодействуют с другими модулями используют блокирующие присваивание, а значения на шинах и всё, что с ними связано — неблокирующее.

Протоколы взаимодействия модулей

Протокол работы с кэшом

При чтении из кэш у меня реализована следующая схема. На первом и втором такте передачи кэшу передаётся адрес, после чего владение шиной переходит кэшу. Спустя некоторое время, кэш на высокой синхронизации посылает значение **C1_RESPONSE**, одновременно с этим начиная передавать данные.

На схемах нарисован график синхронизации, на верхних значениях которого написаны действия кэша , а на нижних, соответственно, процессора в каждый такт синхронизации. Пунктирной линией обозначена смена владельца шины. В коде это реализовано таким образом, что процессор отрабатывает сразу после `@(posedge clk)`, а кэш наоборот, сразу после `@(negedge clk)`. Исключением из этого правила служит передача доступа от одной стороне другой, в этом случае сторона, которая прекращает владеть шиной, должна сообщить это не в свой такт, но в этот факт она просто снимает с себя владение шиной и ничего больше.

Приношу извинения за картинки от руки, inkscape неожиданно отказался работать должным образом.

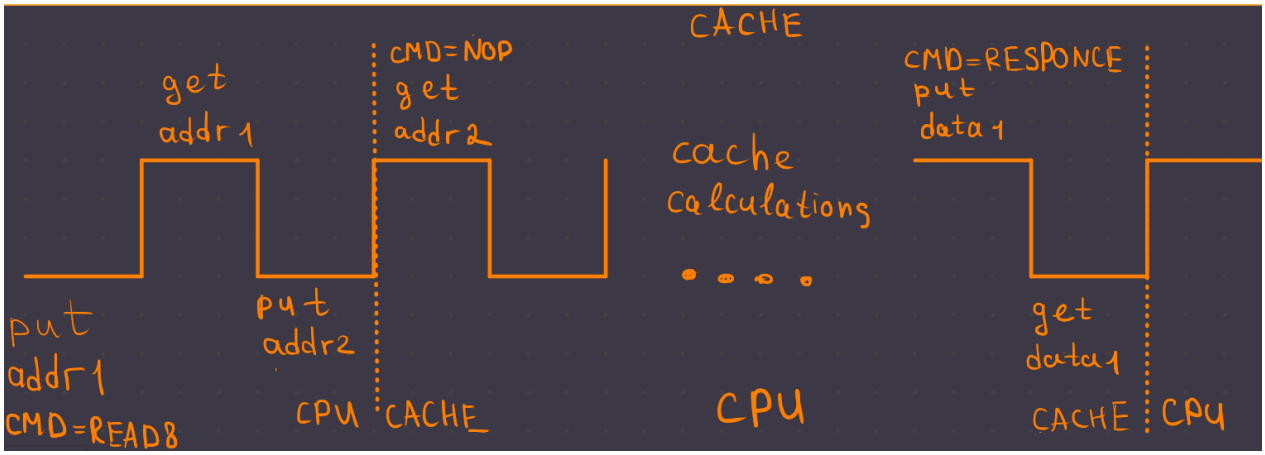

Рис. 1: **C1_READ8**

При запросе записи всё проходит почти аналогично. Обратите внимание, тут данные тоже передаются не сразу, а с задержкой в один такт от адреса. Позже оказалось, что это самое удачное решение, ведь мы тратим лишний такт, но решил всё таки оставить так.

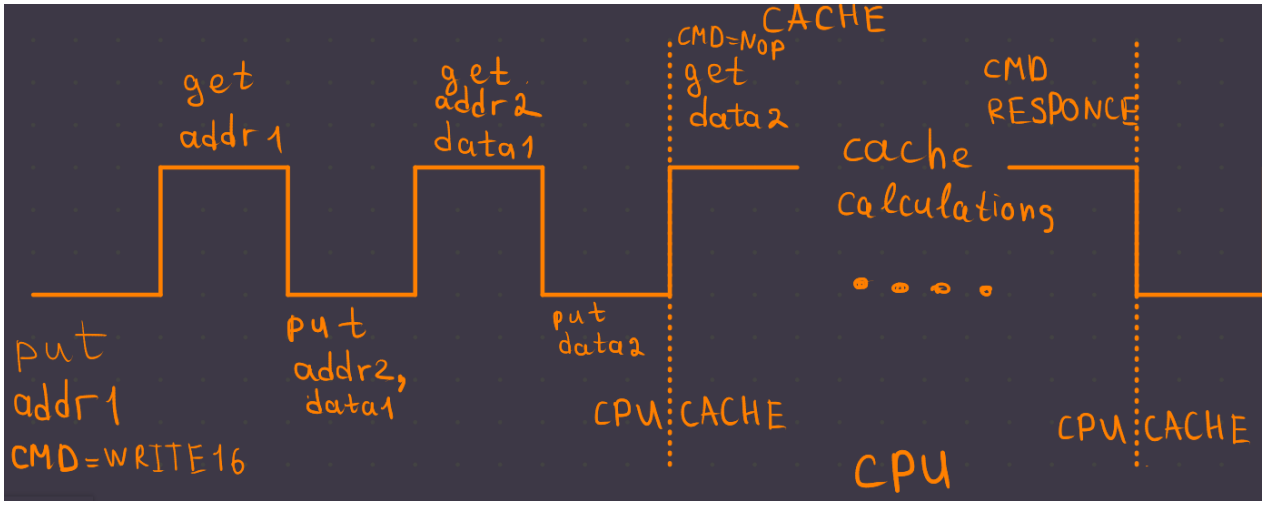

Рис. 2: **C1_WRITE16**

Протокол работы памятью

В ней всё очень похоже на работу кэша, но на всякий случай приведу схемы и для памяти. Сверху обозначены действия памяти, а снизу — кэша.

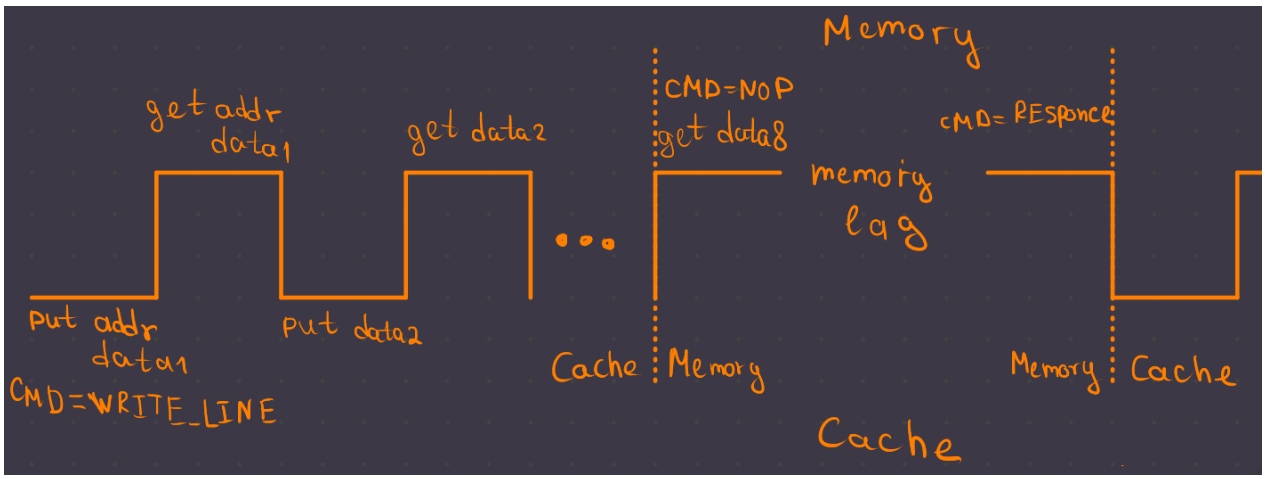

Рис. 3: **C2_WRITELINE**

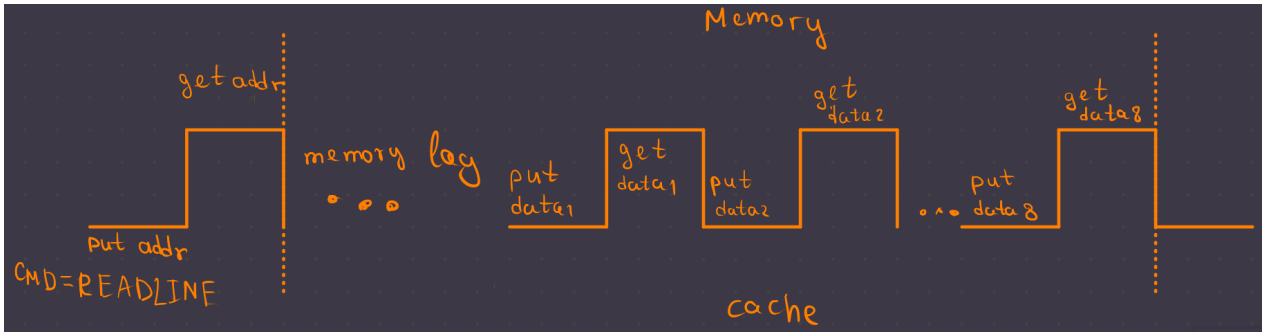

Рис. 4: **C2_READLINE**

Для наглядности также привету код для считывания памяти с обеих сторон.

```
    // Memory
    always @(posedge clk) begin
        if (cmd_w == C2_READ_LINE) begin
            cmd <= C2_NOP;
            owner <= 1;
            skip(mem_feedback_time - 1);
            @(negedge clk)
            cmd <= C2_RESPONSE;
            for (it = addr * cache_line_size; it < addr *
↪   cache_line_size + cache_line_size; it += data2_bus_size) begin
                for (byte_in_bus = 0; byte_in_bus < data2_bus_size;
↪   byte_in_bus += 1) begin
                    data[byte_in_bus * BITS_IN_BYTE +: BITS_IN_BYTE]
↪   <= heap[it + byte_in_bus];
                end
                if (it + data2_bus_size >= addr * cache_line_size +
↪   cache_line_size)
                    @(posedge clk);
                else
                    @(negedge clk);
            end
            owner <= 0;
        end
    end
    // Cache
    task run_read(input logic[cache_set_size + cache_tag_size-1:0]
↪   addr_, output logic[cache_line_size*BITS_IN_BYTE-1:0] data_,
↪   output longint timing);
        @(negedge clk);
        owner <= 1;
        cmd <= C2_READ_LINE;
        addr <= addr_;
        timing = clk_time;
        @(posedge clk);
        owner <= 0;
        wait(cmd_w == C2_RESPONSE);
        @(posedge clk);
        timing = clk_time - timing;
        for (i = 0; i < cache_line_size; i += data2_bus_size) begin
            data_[i * BITS_IN_BYTE +: data2_bus_size * BITS_IN_BYTE]
↪   <= data_w;
            if (i + data2_bus_size >= cache_line_size)
                @(negedge clk);
            else
                @(posedge clk);
        end
        owner <= 1;
        cmd <= C2_NOP;
    endtask
```

Воспроизведение задачи на Verilog

В файле `testbench.sv` задано тестовое окружение для данной задачи, в котором выводится интересная статистика. В неё входит общее количество тактов работы задачи, количество тактов, затраченное на работу кэша, количество попаданий и промахов в кэш, а также время работы после каждой итерации внешнего цикла.

```
    for (y = 0; y < M; y++) begin skip(); // loop
        skip(); // x init;
        ...
        ...
        $display("time: %d %t", y, timing);
        $fflush;
    end

    $display("Finish cpu run");
    $display("Time: %t", timing);
    $display("Cache time: %t", timing - skipped_time);
    $display("Alu time: %t", skipped_time);
    $display("Total hits: %d", total_hits);
    $display("Total misses: %d", total_misses);
    $finish;
```

Результат работы:

```
time: 0 83637
...
time: 62 5191123
time: 63 5272741
Finish cpu run
Time: 5272742
Cache time: 4274080
Alu time: 998662
Total hits: 228080
Total misses: 21520
```

Сравнение полученных результатов

Изначально прогнозированное время работы было меньше реально примерно на 20%. Это произошло главным образом из-за плохой реализации передачи данных по шине.

В коде на Verilog много где была сформирована модель, что принимающая сторона синхронизируется по `posedge clk`, а отдающая по `negedge clk` или наоборот. Из-за этого очередная операция считывания могла быть начата на такт позже из-за того, что на данном такте `negedge clk`, а эта схема синхронизируется по `posedge clk` перед отправкой.

Но описанная выше проблема была исправлена после исправления не оптимальных мест и изменения логики синхронизации. В этом мне очень помогли написанные

тесты. После этих изменений, время, полученное во время симуляции кода на Verilog в точности совпало с прогнозом.

Листининг кода

Код на C++ для оценки времени работы

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <bitset>
#include <cassert>
#include <fstream>
#include <tuple>

using namespace std;

typedef unsigned long long Addr;

enum Word { WORD=8, DWORD=16, QWORD=32 };

struct CacheLine {
    const int size;
    bool valid = 0;
    bool dirty = 0;
    Addr tag = -1;
    int last_call = 0;
    CacheLine& operator = (CacheLine oth) {
        assert(size == oth.size);
        tie(valid, dirty, tag, last_call) = tie(oth.valid, oth.dirty, oth.tag, oth.last_call);
        return *this;
    }
};

struct CacheSet {
private:
    vector<CacheLine> lines;
    const int line_size;
    int total_mem_pushes = 0;
public:
    CacheSet(int way, int line_size) : line_size(line_size) {
        lines = vector<CacheLine>(way, CacheLine{line_size});
    }

    bool exists(Addr tag) {
        return find_by_tag(tag) != lines.end();
    }

    void read(Addr tag, int call) {
        auto it = find_by_tag(tag);
        if (it != lines.end()) {
            it->last_call = call;
        } else {
            *(find_LRO()) = CacheLine {line_size, 1, 0, tag, call};
        }
    }

    void write(Addr tag, int call) {
        auto it = find_by_tag(tag);
        if (it != lines.end()) {
            it->last_call = call;
            it->dirty = 1;
        } else {
            total_mem_pushes += find_LRO()->dirty;
            *(find_LRO()) = CacheLine {line_size, 1, 1, tag, call};
        }
    }

    int getMemPushes() const {
        return total_mem_pushes;
    }

```

```
private:
    vector<CacheLine>::iterator find_by_tag(Addr tag) {
        return find_if(lines.begin(), lines.end(), [&] (const CacheLine& line) {
                return line.valid && line.tag == tag;
            }
        );
    }

    vector<CacheLine>::iterator find_LRO() {
        return min_element(lines.begin(), lines.end(), [] (const CacheLine& a, const CacheLine& b) {
            return tie(a.valid, a.last_call) < tie(b.valid, b.last_call);
        });
    }
};


struct Cache {
private:
    const int sets_cnt;
    const int way;
    const int line_size;        // byte
    const int data_bus_size;    // bits
    const int mem_size;         // byte

    int total_hits = 0;
    int total_misses = 0;
    int total_time = 0;
    int calls = 0;

    vector<CacheSet> sets;

    struct InnerAddr {
        Addr tag, set, offset;
    };
public:
    Cache(int sets_cnt, int way, int line_size, int data_bus_size, int mem_size) :
        sets_cnt(sets_cnt), way(way), line_size(line_size), data_bus_size(data_bus_size), mem_size(mem_size)
    {
        sets = vector<CacheSet>(sets_cnt, CacheSet(way, line_size));
    }

    void read(Word word, Addr addr) {
        calls++;
        auto [tag, set, offset] = split_addr(addr);
        if (sets[set].exists(tag)) {
            total_hits++;
            total_time += 6; // Cache lag
            total_time += transfer_lag(word) + 1; // Cache -> Cpu
                                                  // 8-9
        } else {
            total_misses++;
            total_time += 4 + 100; // Cache + Mem lag
            total_time += transfer_lag(WORD * line_size) + 1; // Mem -> Cache
            total_time += transfer_lag(word); // Cache -> Cpu
                                              // 114-115
        }
        sets[set].read(tag, calls);
    };

    void write(Word word, Addr addr) {
        calls++;
        auto [tag, set, offset] = split_addr(addr);
        if (sets[set].exists(tag)) {
            total_hits++;
            total_time += 6; // Cache lag
            total_time += 1; // Cache->Cpu response
        } else {
            total_misses++;
            total_time += 4 + 100;                          // Cache + Mem lag
            total_time += transfer_lag(WORD * line_size); // Mem -> Cache
            total_time += 1; // Cache->Cpu response
        }
        sets[set].write(tag, calls);
    };

    int get_hits() const  {
        return total_hits;
    }

    int get_misses() const {
```

```
        return total_misses;
    }

    int get_time() const {
        int total_pushes = accumulate(sets.begin(), sets.end(), 0,
            [] (int acc, const CacheSet& el) {
                return acc + el.getMemPushes();
            }
        );
        return total_time + total_pushes * (100 + 1);
    }

private:
    InnerAddr split_addr(Addr addr) const {
        return {
            (addr & ((mem_size - 1) & ~(sets_cnt * line_size - 1))) / (sets_cnt * line_size),
            (addr & (sets_cnt - 1) * line_size) / line_size ,
            addr & (line_size - 1)
        };
    }

    int transfer_lag(int data) const {
        return (data + data_bus_size - 1) / data_bus_size;
    }

};

struct PseudoAllocator {
    Addr last_allocated_addr = 0;

    Addr allocate(int n, int m, int elem_sz) {
        int res = last_allocated_addr;
        last_allocated_addr += n * m * elem_sz;
        return res;
    }
};

int main() {
    // 512Kb == [0x00000...0x7ffff]
    Cache cache(32, 2, 16, 16, 512 * 1024);
    PseudoAllocator alloc;

    const int M = 64;
    const int N = 60;
    const int K = 32;

    int8_t* const a = (int8_t*) alloc.allocate(M, K, sizeof(int8_t));  // a[M][K];
    int16_t* const b = (int16_t*)alloc.allocate(K, N, sizeof(int16_t)); // b[K][N];
    int32_t* const c = (int32_t*)alloc.allocate(M, N, sizeof(int32_t)); // c[M][N];

    int8_t* pa = a;
    int32_t* pc = c;

    for (int y = 0; y < M; y++) {
        for (int x = 0; x < N; x++) {
            int16_t* pb = b;
            for (int k = 0; k < K; k++) {
                cache.read(WORD, (Addr) (pa + k));
                cache.read(DWORD, (Addr) (pb + x));
                pb += N;
            }
            cache.write(QWORD, (Addr) (pc + x));
        }
        pa += K;
        pc += N;
    }

    cout << "HITS: " << cache.get_hits() << "\nMISSES: " << cache.get_misses() << "\nTOTAL TIME: " <<
↪   cache.get_time() << endl;
    cout << "RATE: " << 1.0 * (cache.get_hits()) / (cache.get_hits() + cache.get_misses()) << endl;

    return 0;
}
```

Listing 1: main.cpp

Код на Verilog для симуляции схемы

```
`ifndef PARAMETERS_GUARD
`define PARAMETERS_GUARD

parameter BITS_IN_BYTE = 8;
parameter INF = 1000000000;

parameter cache_line_size = 16;
parameter cache_way = 2;
parameter cache_tag_size = 10;
parameter cache_set_size = 5;
parameter cache_offset_size = 4;

parameter addr1_bus_size = 2;
parameter data1_bus_size = 2;

parameter data2_bus_size = 2;
parameter addr2_bus_size = 2;

parameter mem_feedback_time = 100;
parameter cache_hit_time = 6;
parameter cache_miss_time = 4 + mem_feedback_time + cache_line_size / data2_bus_size;

parameter _SEED = 225526;

typedef enum logic[1:0] {
   C2_NOP=2'b00,
   C2_RESPONSE=2'b01,
   C2_READ_LINE=2'b10,
   C2_WRITE_LINE=2'b11
} mem_command_codes;

typedef enum logic[2:0] {
    C1_NOP=3'b000,
    C1_READ8=3'b001,
    C1_READ16=3'b010,
    C1_READ32=3'b011,
    C1_INVALIDATE_LINE=3'b100,
    C1_WRITE8=3'b101,
    C1_WRITE16=3'b110,
    C1_WRITE32=3'b111
} cpu_cache_command_codes;

typedef enum logic[2:0] {
    C1_RESPONSE=3'b111
} cache_cpu_command_codes;
`endif
```

Listing 2: parameters.sv

```
`ifndef CLOCK_GUARD
`define CLOCK_GUARD

module Clock(output logic clk, output longint timing);

always #1 clk = ~clk;

always #2 timing++;

initial begin
    timing = 0;
    clk = 0;
end

endmodule

`endif
```

Listing 3: clock.sv

```
`ifndef MEMORY_GUARD
`define MEMORY_GUARD

`include "parameters.sv"
`include "clock.sv"

module Memory
    #(
        parameter mem_size = 512 * 1024
    )
    (
        input wire clk, reset, dump,
        input wire[addr2_bus_size*BITS_IN_BYTE-1:0] addr,
        inout wire[data2_bus_size*BITS_IN_BYTE-1:0] data_w,
        inout wire[1:0] cmd_w
    );

    logic[BITS_IN_BYTE-1:0] heap[0:mem_size-1];
    integer SEED = _SEED;
    integer i = 0;

    bit owner = 0;
    logic[1:0] cmd;
    logic[data2_bus_size*BITS_IN_BYTE-1:0] data;

    assign cmd_w = owner ? cmd : 2'bzz;
    assign data_w = owner ? data : {data2_bus_size*BITS_IN_BYTE{1'bz}};

    always @(posedge reset) begin
        for (i = 0; i < mem_size; i += 1) begin
            heap[i] <= $random(SEED)>>16;
        end
    end

    integer fd;
    always @(posedge dump) begin
        fd = $fopen("mem.dump", "w");
        for (i = 0; i < mem_size; i += 1) begin
            $fdisplay(fd, "[%d] %d", i, heap[i]);
        end
        $display("Memory has dumped to mem.dump");
        $fclose(fd);
    end

    task automatic skip(input longint ticks = 1);
        logic enter_clk = clk;
        while (ticks > 0) begin
            wait(clk != enter_clk);
            wait(clk == enter_clk);
            ticks--;
        end
    endtask

    integer it, byte_in_bus;

    always @(posedge clk) begin
        if (cmd_w == C2_WRITE_LINE) begin
            for (it = addr * cache_line_size; it < addr * cache_line_size + cache_line_size; it += data2_bus_size) begin
                for (byte_in_bus = 0; byte_in_bus < data2_bus_size; byte_in_bus += 1) begin
                    heap[it + byte_in_bus] <= data_w[byte_in_bus * BITS_IN_BYTE +: BITS_IN_BYTE];
                end
                @(posedge clk);
            end
            skip(mem_feedback_time - cache_line_size / data2_bus_size - 1);
            @(negedge clk);
            cmd <= C2_RESPONSE;
            owner <= 1;
            @(posedge clk);
            owner <= 0;
        end
    end

    always @(posedge clk) begin
        if (cmd_w == C2_READ_LINE) begin
            cmd <= C2_NOP;
            owner <= 1;
            skip(mem_feedback_time - 1);
            @(negedge clk)
            cmd <= C2_RESPONSE;
```

```
            for (it = addr * cache_line_size; it < addr * cache_line_size + cache_line_size; it += data2_bus_size) begin
                for (byte_in_bus = 0; byte_in_bus < data2_bus_size; byte_in_bus += 1) begin
                    data[byte_in_bus * BITS_IN_BYTE +: BITS_IN_BYTE] <= heap[it + byte_in_bus];
                end
                if (it + data2_bus_size >= addr * cache_line_size + cache_line_size)
                    @(posedge clk);
                else
                    @(negedge clk);
            end
            owner <= 0;
        end
    end
endmodule

module MemoryDriver
    (
        input logic clk,
        input longint clk_time,
        input logic reset,
        input logic m_dump,
        output logic[addr2_bus_size*BITS_IN_BYTE-1:0] addr_w,
        inout logic[data2_bus_size*BITS_IN_BYTE-1:0] data_w,
        inout logic[1:0] cmd_w
    );

    logic[1:0] cmd = C2_NOP;
    logic[addr2_bus_size*BITS_IN_BYTE-1:0] addr;
    logic[data2_bus_size*BITS_IN_BYTE-1:0] data;
    bit owner = 1;

    assign cmd_w = owner ? cmd : 2'bzz;
    assign addr_w = addr;
    assign data_w = owner ? data : {data2_bus_size*BITS_IN_BYTE{1'bz}};

    integer i;

    task run_read(input logic[cache_set_size + cache_tag_size-1:0] addr_, output logic[cache_line_size*BITS_IN_BYTE-1:0] data_, output longint timing);
        @(negedge clk);
        owner <= 1;
        cmd <= C2_READ_LINE;
        addr <= addr_;
        timing = clk_time;
        @(posedge clk);
        owner <= 0;
        wait(cmd_w == C2_RESPONSE);
        @(posedge clk);
        timing = clk_time - timing;
        for (i = 0; i < cache_line_size; i += data2_bus_size) begin
            data_[i * BITS_IN_BYTE +: data2_bus_size * BITS_IN_BYTE] <= data_w;
            if (i + data2_bus_size >= cache_line_size)
                @(negedge clk);
            else
                @(posedge clk);
        end
        owner <= 1;
        cmd <= C2_NOP;
    endtask

    task run_write(input int addr_, input logic[cache_line_size*BITS_IN_BYTE-1:0] data_, output longint timing);
        @(negedge clk);
        timing = clk_time;
        owner <= 1;
        cmd <= C2_WRITE_LINE;
        addr <= addr_;
        for (i = 0; i < cache_line_size / data2_bus_size; i += 1) begin
            data <= data_[i * data2_bus_size * BITS_IN_BYTE +: data2_bus_size * BITS_IN_BYTE];
            @(negedge clk);
        end
        owner <= 0;
        wait(cmd_w == C2_RESPONSE);
        timing = clk_time - timing;
        cmd <= C2_NOP;
        owner <= 1;
    endtask

endmodule
`endif
```

Listing 4: mem.sv

```
`ifndef CACHE_GUARD
`define CACHE_GUARD

`include "parameters.sv"
`include "mem.sv"
`include "clock.sv"

module Cache (
        input  wire                                  clk, reset, dump,
        input  wire[addr1_bus_size*BITS_IN_BYTE-1:0] addr_cpu_w,
        inout  wire[data1_bus_size*BITS_IN_BYTE-1:0] data_cpu_w,
        inout  wire[2:0]                             cmd_cpu_w,
        output wire[addr2_bus_size*BITS_IN_BYTE-1:0] addr_mem_w,
        inout  wire[data2_bus_size*BITS_IN_BYTE-1:0] data_mem_w,
        inout  wire[1:0]                             cmd_mem_w,
        output integer total_hits = 0,
        output integer total_misses = 0
    );

    localparam cache_sets_count = 1 << cache_set_size;

    bit owner_cpu = 0;
    logic[data1_bus_size*BITS_IN_BYTE-1:0] data_cpu;
    logic[2:0] cmd_cpu;
    assign data_cpu_w = owner_cpu ? data_cpu : {data1_bus_size*BITS_IN_BYTE{1'bz}};
    assign cmd_cpu_w = owner_cpu ? cmd_cpu : 3'bzzz;

    MemoryDriver mem_driver(clk, 64'b0, reset, 1'b0, addr_mem_w, data_mem_w, cmd_mem_w);

    task run_mem_write(input logic[cache_set_size + cache_tag_size-1:0] addr_, input
    logic[cache_line_size*BITS_IN_BYTE-1:0] data_);
        longint timing; // Unusable blackhole
        mem_driver.run_write(addr_, data_, timing);
    endtask

    task run_mem_read(input logic[cache_set_size + cache_tag_size-1:0] addr_, output
    logic[cache_line_size*BITS_IN_BYTE-1:0] data_);
        longint timing; // Unusable blackhole
        mem_driver.run_read(addr_, data_, timing);
    endtask

    typedef struct packed {
        logic valid;
        logic dirty;
        integer last_update;
        logic[cache_tag_size-1:0] tag;
        logic[cache_line_size*BITS_IN_BYTE-1:0] data;
    } cacheLine;

    typedef struct packed {
        logic[cache_tag_size-1:0] tag;
        logic[cache_set_size-1:0] set;
        logic[cache_offset_size-1:0] offset;
    } cacheAddr;

    cacheLine[cache_way * cache_sets_count-1:0] lines;

    function int bytes_cnt_from_cmd(logic[2:0] cmd);
        case (cmd & 3'b011)
            3'b001 : return 1;
            3'b010 : return 2;
            3'b011 : return 4;
        endcase
    endfunction

    task automatic skip(input longint ticks = 1);
        logic enter_clk = clk;
        while (ticks > 0) begin
            wait(clk != enter_clk);
            wait(clk == enter_clk);
            ticks--;
        end
    endtask

    always @(posedge reset) begin
```

```
        for (i = 0; i < cache_way * cache_sets_count; i += 1) begin
            lines[i] <= 0;
        end
    end

    integer action_word, it, i, j;
    logic[cache_line_size * BITS_IN_BYTE-1:0] buff_a, buff_b;
    cacheAddr curAddr;

    always @(posedge clk) begin
        if (owner_cpu == 0 && (cmd_cpu_w == C1_WRITE8 || cmd_cpu_w == C1_WRITE16 || cmd_cpu_w == C1_WRITE32)) begin
            // READ 1-ST
            action_word <= bytes_cnt_from_cmd(cmd_cpu_w);
            {curAddr.tag, curAddr.set} <= addr_cpu_w;
            // READ 2-ND
            @(posedge clk);
            curAddr.offset = addr_cpu_w;
            // Read data
            for (j = 0; j < action_word; j += data1_bus_size) begin
                for (i = 0; i + j < action_word && i < data1_bus_size; i++) begin
                    buff_a[(curAddr.offset + j + i) * BITS_IN_BYTE +: BITS_IN_BYTE] <= data_cpu_w[i * BITS_IN_BYTE +: BITS_IN_BYTE];
                end
                if (j + data1_bus_size >= action_word)
                    @(negedge clk);
                else
                    @(posedge clk);
            end
            owner_cpu <= 1;
            cmd_cpu <= C1_NOP;

            // Searching line
            it = search_by_addr_or_empty(curAddr, lines);
            // Uploading from memory if needed
            if (it == -1 || !lines[it].valid || lines[it].tag != curAddr.tag) begin
                total_misses++;
                skip(action_word != 4);
                run_mem_read({curAddr.tag, curAddr.set}, buff_b);
            end else begin
                total_hits++;
                skip(3 + (action_word != 4));
                buff_b <= lines[it];
            end

            // Merging buffers
            for (i = 0; i < action_word; i++) begin
                buff_b[(curAddr.offset + i) * BITS_IN_BYTE +: BITS_IN_BYTE] = buff_a[(curAddr.offset + i) * BITS_IN_BYTE +: BITS_IN_BYTE];
            end

            // Purge if needed
            if (it == -1) begin
                it = find_lru(curAddr.set, lines);
                if (lines[it].dirty) begin
                    $display("PURGE");
                    run_mem_write({lines[it].tag, curAddr.set}, lines[it]);
                end
            end

            lines[it] <= {1'b1, 1'b1, total_hits + total_misses, curAddr.tag, buff_b};

            cmd_cpu <= C1_RESPONSE;
            @(posedge clk);
            owner_cpu <= 0;
        end
    end

    function int search_by_addr_or_empty(
            input cacheAddr addr,
            input cacheLine[cache_way * cache_sets_count-1:0] lines
        );
        it = -1;
        for (i = addr.set * cache_way; i < addr.set * cache_way + cache_way; ++i) begin
            if (lines[i].valid && lines[i].tag == addr.tag)
                it = i;
            if (!lines[i].valid && it == -1)
                it = i;
        end
        return it;
```

```
151     endfunction
152
153     function int find_lru(input logic[cache_set_size-1:0] set, input cacheLine[cache_way *
    ↪   cache_sets_count-1:0] lines);
154         integer it_time;
155         it_time = INF;
156         it = -1;
157         for (i = set * cache_way; i < set * cache_way + cache_way; ++i) begin
158             if (it_time > lines[i].last_update) begin
159                 it = i;
160                 it_time = lines[i].last_update;
161             end
162         end
163         return it;
164     endfunction
165
166     logic[cache_line_size * BITS_IN_BYTE-1:0] buff;
167     always @(posedge clk) begin
168         if (owner_cpu == 0 && (cmd_cpu_w == C1_READ8 || cmd_cpu_w == C1_READ16 || cmd_cpu_w ==
    ↪   C1_READ32)) begin
169             // READ 1-ST
170             action_word <= bytes_cnt_from_cmd(cmd_cpu_w);
171             {curAddr.tag, curAddr.set} <= addr_cpu_w;
172             // READ 2-ND
173             @(posedge clk);
174             curAddr.offset = addr_cpu_w;
175             owner_cpu <= 1;
176             cmd_cpu <= C1_NOP;
177
178             it = search_by_addr_or_empty(curAddr, lines);
179
180             if (it == -1) begin
181                 it = find_lru(curAddr.set, lines);
182             end
183             if (!lines[it].valid || lines[it].tag != curAddr.tag) begin
184                 total_misses++;
185                 skip(2);
186                 run_mem_read({curAddr.tag, curAddr.set}, buff);
187                 lines[it] <= {1'b1, 1'b0, total_hits + total_misses, curAddr.tag, buff};
188             end else begin
189                 total_hits++;
190                 skip(5);
191                 buff = lines[it].data;
192                lines[it] <= {lines[it].valid, lines[it].dirty, total_hits + total_misses, lines[it].tag,
    ↪   buff};
193             end
194
195             cmd_cpu <= C1_RESPONSE;
196             for (j = 0; j < action_word; j += data1_bus_size) begin
197                 for (i = 0; i + j < action_word && i < data1_bus_size; i++) begin
198                     data_cpu[i * BITS_IN_BYTE +: BITS_IN_BYTE] = buff[(curAddr.offset + j + i) *
    ↪   BITS_IN_BYTE +: BITS_IN_BYTE];
199                 end
200                 if (j + data1_bus_size >= action_word)
201                     @(posedge clk); // last iteration
202                 else
203                     @(negedge clk);
204             end
205             owner_cpu <= 0;
206         end
207     end
208
209 endmodule
210
211 module CacheDriver
212     (
213         input logic clk,
214         input longint clk_time,
215         input logic reset,
216         input logic c_dump,
217         output logic[addr1_bus_size*BITS_IN_BYTE-1:0] addr_cpu_w,
218         inout logic[data1_bus_size*BITS_IN_BYTE-1:0] data_cpu_w,
219         inout logic[2:0] cmd_cpu_w
220     );
221
222     bit owner_cpu = 1;
223     logic[2:0] cmd_cpu = C1_NOP;
224     logic[addr1_bus_size*BITS_IN_BYTE-1:0] addr_cpu;
225     logic[data1_bus_size*BITS_IN_BYTE-1:0] data_cpu;
226     assign addr_cpu_w = addr_cpu;
```

```
227     assign data_cpu_w = owner_cpu ? data_cpu : {data1_bus_size*BITS_IN_BYTE{1'bz}};
228     assign cmd_cpu_w = owner_cpu ? cmd_cpu : 3'bzzz;
229
230     task run_read(
231         input logic[cache_tag_size + cache_offset_size + cache_set_size - 1 : 0] addr,
232         input logic[2:0] cmd,
233         output logic[BITS_IN_BYTE*cache_line_size-1:0] data,
234         output longint timing
235     );
236         logic[BITS_IN_BYTE*data1_bus_size-1:0] local_buff;
237         @(negedge clk);
238         timing = clk_time;
239         cmd_cpu <= cmd;
240         addr_cpu <= addr[cache_offset_size +: cache_set_size + cache_tag_size];
241         @(negedge clk);
242         addr_cpu <= addr[0 +: cache_offset_size];
243         @(posedge clk);
244         owner_cpu <= 0;
245         wait(cmd_cpu_w == C1_RESPONSE); // response on posedge!
246         /* @(posedge clk); */
247         timing = clk_time - timing;
248         case (cmd)
249             C1_READ8, C1_READ16 : begin
250                 data = data_cpu_w;
251             end
252             C1_READ32 : begin
253                 local_buff <= data_cpu_w;
254                 @(posedge clk);
255                 data = {data_cpu_w[0 +: data1_bus_size*BITS_IN_BYTE], local_buff[0 +:
    data1_bus_size*BITS_IN_BYTE]};
256             end
257             default : begin
258                 $display("Incorrect run_read cmd: %d", cmd);
259                 $finish;
260             end
261         endcase
262         owner_cpu <= 1;
263         cmd_cpu <= C1_NOP;
264         @(negedge clk);
265     endtask
266
267     task run_write(
268         input logic[cache_tag_size + cache_offset_size + cache_set_size - 1 : 0] addr,
269         input logic[2:0] cmd,
270         input logic[BITS_IN_BYTE*cache_line_size - 1:0] data,
271         output longint timing
272     );
273         /* $monitor("time %t, owner: %b, clk: %b cmd_w: %b addr_w: %b data_w: %b", clk_time, owner_cpu,
    clk, cmd_cpu_w, addr_cpu_w, data_cpu_w); */
274         @(negedge clk);
275         timing = clk_time;
276         cmd_cpu <= cmd;
277         addr_cpu <= addr[cache_offset_size +: cache_set_size + cache_tag_size];
278         @(negedge clk);
279         addr_cpu <= addr[0 +: cache_offset_size];
280         case (cmd)
281             C1_WRITE8, C1_WRITE16 : begin
282                 data_cpu <= data;
283             end
284             C1_WRITE32 : begin
285                 data_cpu <= data[0 +: data1_bus_size * BITS_IN_BYTE];
286                 @(negedge clk);
287                 data_cpu <= data[data1_bus_size*BITS_IN_BYTE +: data1_bus_size*BITS_IN_BYTE];
288             end
289             default : begin
290                 $display("Incorrect run_write cmd: %d", cmd);
291                 $finish;
292             end
293         endcase
294         @(posedge clk);
295         owner_cpu <= 0;
296         @(negedge clk); // ???
297         wait(cmd_cpu_w == C1_RESPONSE);
298         timing = clk_time - timing;
299         owner_cpu <= 1;
300         cmd_cpu <= C1_NOP;
301     endtask
302
303 endmodule
304 `endif
```

Listing 5: cache.sv

Код тестирования в среде Verilog

```
`include "cache.sv"
`include "clock.sv"

module Testbench;

    integer skipped_time = 0;
    task automatic skip(input longint ticks = 1);
        logic enter_clk = clk;
        skipped_time += ticks;
        while (ticks > 0) begin
            wait(clk != enter_clk);
            wait(clk == enter_clk);
            ticks--;
        end
    endtask

    integer total_hits, total_misses;
    longint timing;
    logic clk, reset = 0, c_dump = 0, m_dump = 0;

    wire[addr2_bus_size*BITS_IN_BYTE-1:0] addr_mem_w;
    wire[data2_bus_size*BITS_IN_BYTE-1:0] data_mem_w;
    wire[1:0] cmd_mem_w;
    wire[addr1_bus_size*BITS_IN_BYTE-1:0] addr_cpu_w;
    wire[data1_bus_size*BITS_IN_BYTE-1:0] data_cpu_w;
    wire[2:0] cmd_cpu_w;

    Clock cloker(clk, timing);
    Memory mem(clk, reset, m_dump, addr_mem_w, data_mem_w, cmd_mem_w);
    Cache cache(clk, reset, c_dump, addr_cpu_w, data_cpu_w, cmd_cpu_w, addr_mem_w, data_mem_w,
    cmd_mem_w, total_hits, total_misses);
    CacheDriver cpu(clk, timing, reset, c_dump, addr_cpu_w, data_cpu_w, cmd_cpu_w);

    logic[BITS_IN_BYTE-1:0] wbuff;
    logic[2 * BITS_IN_BYTE-1:0] dbuff;
    logic[4 * BITS_IN_BYTE-1:0] qbuff;

    localparam M = 64;
    localparam N = 60;
    localparam K = 32;
    integer a_addr = 0;
    integer b_addr = a_addr + M * K;
    integer c_addr = b_addr + K * N * 2;

    integer y, x, k;
    integer pa, pb, pc;
    integer s;
    integer blackhole;

    initial begin
        reset <= 1;
        skip();
        reset <= 0;
        skip();

        pa = a_addr; skip(); // init pa
        pc = c_addr; skip(); // init pc

        skip(); // y init;
        for (y = 0; y < M; y++) begin skip(); // loop
            skip(); // x init;
            for (x = 0; x < N; x++) begin skip(); // loop
                pb = b_addr; skip(); // init pb
                s = 0; skip(); // init s
                skip(); // k init;
                for (k = 0; k < K; k++) begin skip(); // loop
                    // s += pa[k] * pb[x] begin
                    cpu.run_read(pa + k, C1_READ8, wbuff, blackhole);
                    cpu.run_read(pb + x * 2, C1_READ16, dbuff, blackhole);
```

```
                    qbuff = wbuff * dbuff; skip(5); // (*)
                    s = s + qbuff; skip(); // (+)
                    // s += pa[k] * pb[x] end
                    pb = pb + N * 2; skip(); // (+)
                end
                cpu.run_write(pc + x * 4, C1_WRITE32, s, blackhole);
            end
            pa = pa + K; skip(); // (+)
            pc = pc + N * 4; skip(); // (+)
            $display("time: %d %t", y, timing);
            $fflush;
        end
        skip(); // function exit

        $display("Finish cpu run");
        $display("Time: %t", timing);
        $display("Cache time: %t", timing - skipped_time);
        $display("Alu time: %t", skipped_time);
        $display("Total hits: %d", total_hits);
        $display("Total misses: %d", total_misses);
        $finish;

        /* Finish cpu run */
        /* Time:              5272742 */
        /* Cache time         4274080 */
        /* Alu time            998662 */
        /* Total hits:         228080 */
        /* Total misses:        21520 */

    end
endmodule
```

Listing 6: testbench.sv

```
`include "mem.sv"

module MemoryTestbench;

    logic reset = 0, m_dump = 0, clk;
    wire[addr2_bus_size*BITS_IN_BYTE-1:0] addr_w;
    wire[data2_bus_size*BITS_IN_BYTE-1:0] data_w;
    wire[1:0] cmd_w;
    longint clk_time;

    Clock clock(clk, clk_time);
    Memory #(64) mem(clk, reset, m_dump, addr_w, data_w, cmd_w);
    MemoryDriver driver(clk, clk_time, reset, m_dump, addr_w, data_w, cmd_w);

    integer i;

    logic[cache_line_size * BITS_IN_BYTE - 1 : 0] buff_a, buff_b;
    longint timing = 100;
    integer test_addr = 5;
    logic[cache_line_size * BITS_IN_BYTE - 1 : 0] test_payload = {cache_line_size{$random()}};

    always @(timing)
        if (timing != mem_feedback_time)
            $display("Memory timing error! Expected: %d, Real %d", mem_feedback_time, timing);

    initial begin
        reset <= 1;
        #1;
        reset <= 0;
        #1;

        begin : TEST_ZERO_ADDR
            driver.run_read(0, buff_a, timing);
            driver.run_write(0, test_payload, timing);
            driver.run_read(0, buff_b, timing);
            if (buff_b != test_payload)
                $display("Memory correctness unit test failed, real: %b expected: %b", buff_b,
↳   test_payload);
        end

        begin : CHECK_CUSTOM_ADDR
            driver.run_write(test_addr, buff_a, timing);
            driver.run_read(test_addr, buff_b, timing);
            if (buff_a != buff_b)
```

```
                $display("Memory correctness unit test failed, real: %b expected: %b", buff_b, buff_a);
        end

        $display("Finish memory testing");
        $finish;
    end
endmodule
```

Listing 7: mem_testbench.sv

```
`include "cache.sv"

module CacheTestbench;

    logic                                   clk, reset=0, c_dump=0, m_dump=0;
    wire[addr1_bus_size*BITS_IN_BYTE-1:0] addr_cpu_w;
    wire[data1_bus_size*BITS_IN_BYTE-1:0] data_cpu_w;
    wire[2:0]                               cmd_cpu_w ;
    wire[addr2_bus_size*BITS_IN_BYTE-1:0] addr_mem_w;
    wire[data2_bus_size*BITS_IN_BYTE-1:0] data_mem_w;
    wire[1:0]                               cmd_mem_w ;
    integer total_hits, total_misses;

    longint clk_time;
    Clock cloker(clk, clk_time);
    Memory mem(clk, reset, m_dump, addr_mem_w, data_mem_w, cmd_mem_w);
    Cache cache(clk, reset, c_dump, addr_cpu_w, data_cpu_w, cmd_cpu_w, addr_mem_w, data_mem_w,
    cmd_mem_w, total_hits, total_misses);
    CacheDriver driver(clk, clk_time, reset, c_dump, addr_cpu_w, data_cpu_w, cmd_cpu_w);

    logic[cache_line_size * BITS_IN_BYTE - 1:0] buff;
    logic[cache_tag_size + cache_offset_size + cache_set_size - 1 : 0] test_addr = 19'b1111010101001001001;
    logic[4 * BITS_IN_BYTE:0] test_payload = $random();
    integer i;
    longint timing;

    task resetCache();
        reset <= 1;
        @(posedge clk);
        reset <= 0;
        @(negedge clk);
    endtask

    initial begin

        begin : TEST_CACHE_HITS
            resetCache();
            driver.run_read(0, C1_READ8, buff, timing);
            if (timing != cache_miss_time)
                $display("Cache read-miss timing error! Expected: %d, Real %d", cache_miss_time, timing);
            for (i = 1; i < 32; ++i) begin
                driver.run_read(i, C1_READ8, buff, timing);
                if (i != 16 && timing != cache_hit_time) // i = 16 -- cache miss
                    $display("Cache read-hit timing error! Expected: %d, Real %d", cache_hit_time, timing);
            end
            if (total_hits != 30)
                $display("Wrong cache hits! Expected: 30, Found: %d", total_hits);
        end

        begin : TEST_SINGLE_READ_WRITE_32
            resetCache();
            driver.run_write(test_addr, C1_WRITE32, test_payload, timing);
            if (timing != cache_miss_time)
                $display("Cache write-miss timing error! Expected: %d, Real %d", cache_miss_time, timing);
            driver.run_read(test_addr, C1_READ32, buff, timing);
            if (buff[0 +: BITS_IN_BYTE * 4] != test_payload[0 +: BITS_IN_BYTE * 4]) begin
                $display("Cache correctness qword unit test failed, real: %b expected: %b",
                    buff[0 +: BITS_IN_BYTE * 4], test_payload[0 +: BITS_IN_BYTE * 4]);
            end
            driver.run_write(test_addr, C1_WRITE32, test_payload, timing);
            if (timing != cache_hit_time)
                $display("Cache write-hit timing error! Expected: %d, Real %d", cache_hit_time, timing);
        end

        begin : TEST_SINGLE_READ_WRITE_16
            resetCache();
            driver.run_write(test_addr, C1_WRITE16, test_payload, timing);
            if (timing != cache_miss_time)
```

```
68                  $display("Cache write-miss timing error! Expected: %d, Real %d", cache_miss_time, timing);
69              driver.run_read(test_addr, C1_READ16, buff, timing);
70              if (buff[0 +: BITS_IN_BYTE * 2] != test_payload[0 +: BITS_IN_BYTE * 2]) begin
71                  $display("Cache correctness dword unit test failed, real: %b expected: %b",
72                      buff[0 +: BITS_IN_BYTE * 2], test_payload[0 +: BITS_IN_BYTE * 2]);
73              end
74              driver.run_write(test_addr, C1_WRITE16, test_payload, timing);
75              if (timing != cache_hit_time)
76                  $display("Cache write-hit timing error! Expected: %d, Real %d", cache_hit_time, timing);
77          end
78
79          begin : TEST_SINGLE_READ_WRITE_8
80              resetCache();
81              driver.run_write(test_addr, C1_WRITE8, test_payload, timing);
82              driver.run_read(test_addr, C1_READ8, buff, timing);
83              if (buff[0 +: BITS_IN_BYTE] != test_payload[0 +: BITS_IN_BYTE]) begin
84                  $display("Cache correctness word unit test failed, real: %b expected: %b",
85                      buff[0 +: BITS_IN_BYTE], test_payload[0 +: BITS_IN_BYTE]);
86              end
87              driver.run_write(test_addr, C1_WRITE8, test_payload, timing);
88              if (timing != cache_hit_time)
89                  $display("Cache write-hit timing error! Expected: %d, Real %d", cache_hit_time, timing);
90          end
91
92          $display("Finish cache testing");
93          $finish;
94      end
95
96  endmodule
```

Listing 8: cache_testbench.sv