Лабораторная работа №2	группа 05	2022
Моделирование схем в Verilog	Москаленко Т.Д.	

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 11.0 (стабильная). Далее в этом документе Verilog+SystemVerilog обозначается как Verilog.

Задача: Имеется следующее определение глобальных переменных и функций:

Листинг 1. код на Си

```
#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 команда-

ми).

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

Код запускается на модели

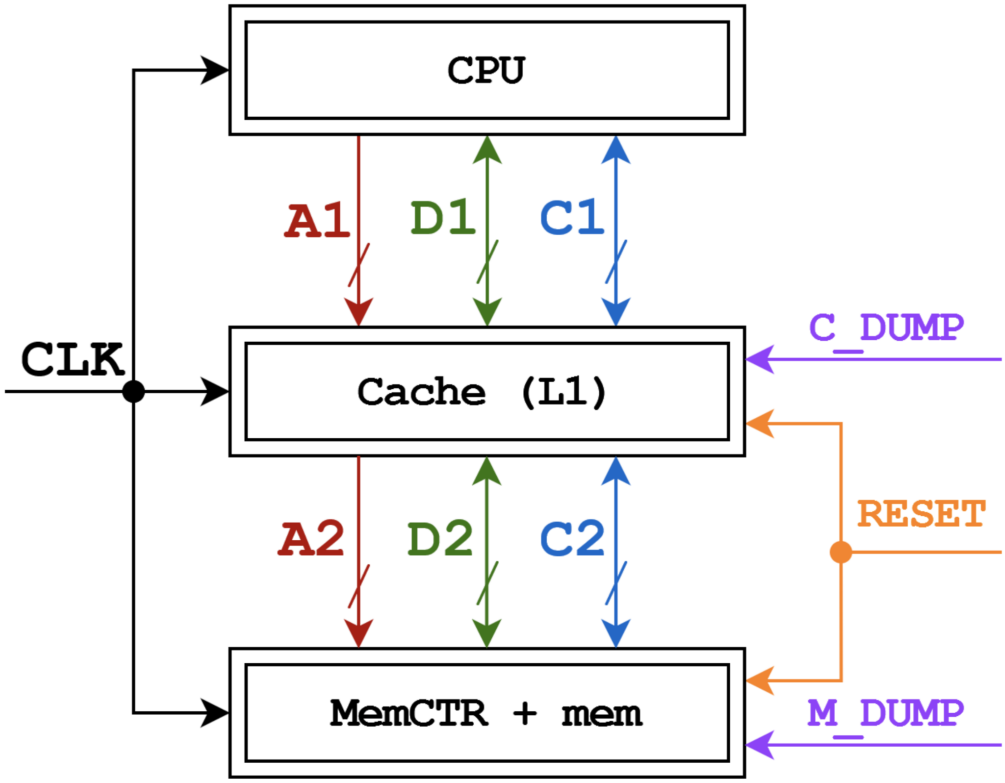

Рис. 1. Система «процессор-кэш-память»

Сигналы:

- `CLK` – синхронизация всей схемы
- `RESET` – сброс в начальное состояние
- `*_DUMP` – сохранение текущего состояния в файл/вывод в консоль для отладки

Модули:

- *CPU* – модель процессора для верификации работы кэша
- *Cache* – одноуровневый кэш. Политика вытеснения — LRU
- *MemCTR* (+ mem) – модель контроллера памяти + модулей памяти для верификации работы кэша

№	$CPU \rightarrow Cache$	$CPU \leftarrow Cache$	$Cache \rightarrow Mem$	$Cache \leftarrow Mem$
0	C1_NOP	C1_NOP	C2_NOP	C2_NOP
1	C1_READ8			C2_RESPONSE
2	C1_READ16		C2_READ_LINE	
3	C1_READ32		C2_WRITE_LINE	
4	C1_INVALIDATE_LINE			
5	C1_WRITE8			
6	C1_WRITE16			
7	C1_WRITE32	C1_RESPONSE		

Таблица 1. Команды

Время отклика:

6 тактов – время, через которое в результате кэш попадания, кэш начинает отвечать.

4 такта – время, через которое в результате кэш промаха, кэш посылает запрос к памяти.

MemCTR – 100 тактов

Рассчитаем параметры системы

```
parameter MEM_SIZE = 1024*512;//размер памяти в байтах
parameter LINE_SIZE = 16; //размер одной кэш линии
parameter LINE_COUNT = 64; //количество кэш линий
parameter SIZE = LINE_SIZE * LINE_COUNT; //суммарный размер всех кэш линий
parameter WAY = 2; //ассофиативность сколько( кэш линий в одном блоке)
parameter SETS_COUNT = LINE_COUNT / WAY; //количество блоков
parameter TAG_SIZE = 10; //размер части адреса, хранящаяся в кэше
parameter SET_SIZE = 5; //замер номера блока
parameter OFFSET_SIZE = 4; //размер номера байта внутри кэшлинии
parameter ADDR_SIZE = TAG_SIZE + SET_SIZE + OFFSET_SIZE; // размер всего
адреса
```

Размер полезных данных в кэше очевидно равен количеству кэш линий умножить на размер одной кэш линии.

$$\texttt{SIZE} = \texttt{LINE\_SIZE} \cdot \texttt{LINE\_COUNT} = 16 \cdot 64 = 1024$$

Количество блоков равно количеству кэш линий разделить на размер одного блока

$$\texttt{SETS\_COUNT} = \frac{\texttt{LINE\_COUNT}}{\texttt{WAY}} = \frac{64}{2} = 32$$

Количество бит, которое уходит на кодировку блока равно двоичному логарифму от количества блоков

$$\texttt{SET\_SIZE} = \log_2(\texttt{SETS\_COUNT}) = \log_2 32 = 5$$

Количество бит, которое уходит на кодировку позиции байта в кэш линии равно двоичному логарифму от количества байтов внутри одной кэш линии.

$$\texttt{OFFSET\_SIZE} = \log_2(\texttt{LINE\_SIZE}) = \log_2 16 = 4$$

Размер всего адреса равен сумме SETS_COUNT, SET_SIZE, OFFSET_SIZE. С другой стороны это двоичный логарифм от размера памяти в байтах.

$$\texttt{ADDR\_SIZE} = \log_2(\texttt{MEM\_SIZE}) = 19$$

$$\texttt{ADDR\_SIZE} = \texttt{TAG\_SIZE} + \texttt{SET\_SIZE} + \texttt{OFFSET\_SIZE} = 19$$

Данные непротиворечивы. Теперь посчитаем размер шин.

$$\texttt{ADDR[1,2]\_BUS\_SIZE} = \max(\texttt{TAG\_SIZE} + \texttt{SET\_SIZE}, \texttt{OFFSET\_SIZE}) = 15$$

$$\texttt{CTR1\_BUS\_SIZE} = \log_2 8 = 3$$

$$\texttt{CTR2\_BUS\_SIZE} = \log_2 4 = 2$$

Аналитическое решение задачи

Рассмотрим решение задачи на языке Python

```python
M = 64
N = 60
K = 32

a = [[0] * K for _ in range(M)]  # array a
b = [[0] * N for _ in range(K)]  # array b
c = [[0] * N for _ in range(M)]  # array c

offset = 0
tagset = 0

for i in range(M):
   for j in range(K):
      a[i][j] = tagset
      offset += 1
      if offset % 16 == 0:
         tagset += 1
         offset = 0

for i in range(K):
   for j in range(N):
      b[i][j] = tagset
      offset += 2
      if offset % 16 == 0:
         tagset += 1
         offset = 0

for i in range(M):
   for j in range(N):
      c[i][j] = tagset
      offset += 4
      if offset % 16 == 0:
         tagset += 1
         offset = 0
```

Смоделируем нашу оперативную память как 3 массива `a`,`b`,`c`. Мысленно разделим память на строчки по 16 байт (размер одной кэш линии). Тогда каждому элементу массива присвоим порядковый номер строчки, целиком в этой памяти (на самом деле это будет `tag` + `set` от адреса соответствующего элемента). Инициализация происходит в трёх соответствующих форах. Заметим, что элементы массивов занимают 1, 2 и 4 байта соответственно, поэтому в форах мы увеличиваем переменную `offset` на разные значения.

```python
cache = [[0] * 2 for _ in range(64)]

for i in range(64):
   cache[i][0] = -1
```

Кэш будет состоять из 64 кэш линий, каждая будет содержать 2 числа: `tagset`, `time`. `time` = 1 если кэш линия из двух в блоке использовалась последней, иначе `time` = 0. `tagset` содержит в себе `tagset` кэш линии.

```
misses = 0  # количество промахов
requests = 32 * 64 * 60 * 2 + 64 * 60  # количество запросов всего
clock = 0  # количество тактов
```

`misses` - счётчик количества

`requests` - количество запросов всего. Можно вычислить как удвоенное количество итераций внутреннего цикла плюс количество итераций среднего цикла

$$32 \cdot 64 \cdot 60 \cdot 2 + 64 \cdot 60 = 249600$$

`clock` - счётчик количества тактов.

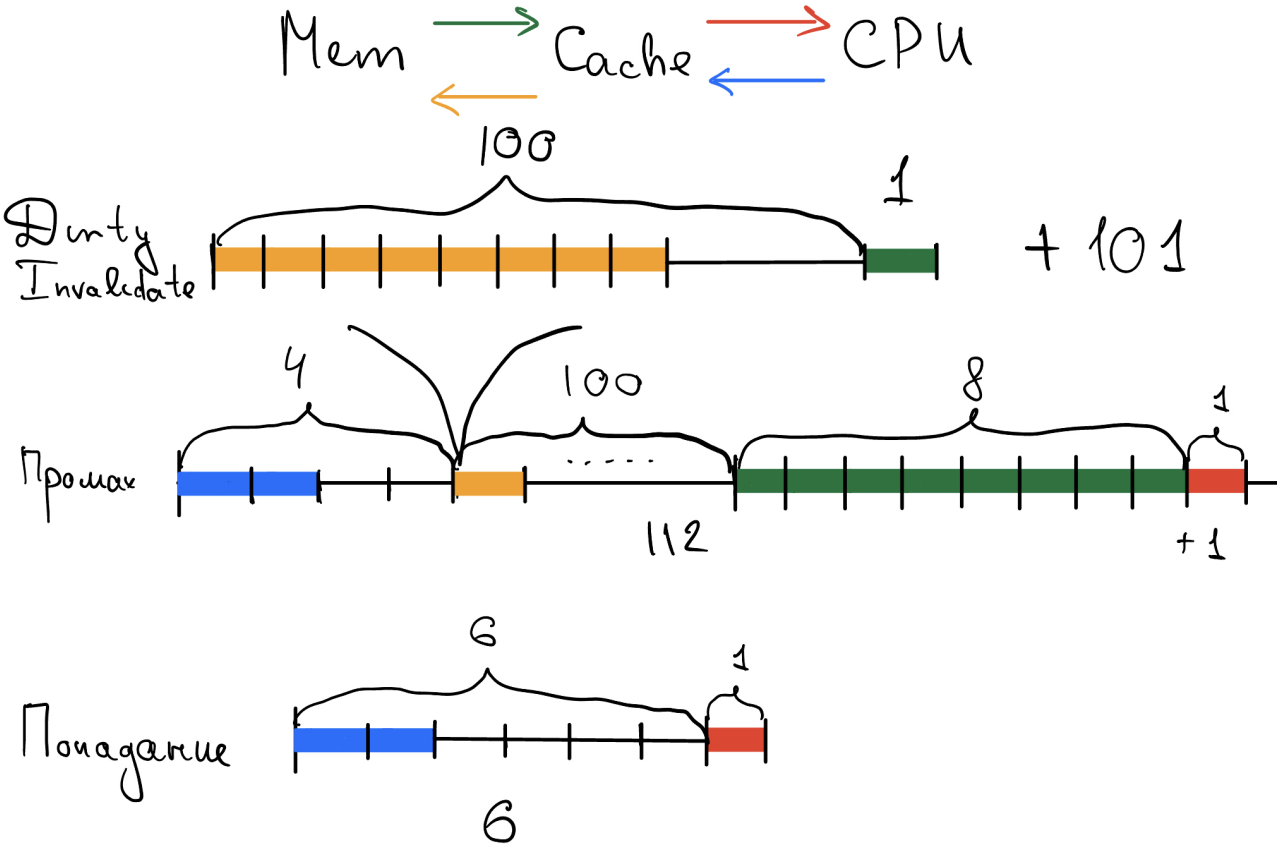

Рис. 2. Объяснение времени работы каждой команды аналитически

```
def f(line_adr: int) -> int:
    global clock
    st = line_adr % 32
    if cache[2 * st][0] == line_adr:  # первая линия нужная
        clock += 6  # время отклика кэша при попадании
        cache[2 * st][1] = 1
        cache[2 * st + 1][1] = 0
        return 0  # попали
```

```
    if cache[2 * st + 1][0] == line_adr:  # вторая линия нужная
        clock += 6  # время отклика кэша при попадании
        cache[2 * st + 1][1] = 1
        cache[2 * st][1] = 0
        return 0  # попали
    if cache[2 * st][1] == 0:  # первая линия использовалась давно
        clock += 112  # 4 + 100 + 8
        # наша линия грязная, когда в ней лежит элемент из c[][]
        if cache[2 * st][0] >= 368:
            clock += 101  # 100 + 1
        cache[2 * st][0] = line_adr
        cache[2 * st][1] = 1
        cache[2 * st + 1][1] = 0
        return 1  # промах
    if cache[2 * st + 1][1] == 0:  # вторая линия использовалась давно
        clock += 112  # 4 + 100 + 8
        # наша линия грязная, когда в ней лежит элемент из c[][]
        if cache[2 * st + 1][0] >= 368:
            clock += 101  # 100 + 1
        cache[2 * st + 1][0] = line_adr
        cache[2 * st + 1][1] = 1
        cache[2 * st][1] = 0
        return 1  # промах
    assert False  # невозможное состояние
```

Функция `f` реализует работу кэша. По заданному `tagset` (в данном случае `line_addr`) она проверяет, есть ли данная часть памяти в кэше.

1. Если есть — у нас кэш попадание, мы прибавляем к `clock` 6 и записываем в соответствующее место кэша `line_addr` (с 4 по 13 строчки). Функция возвращает 0.

2. Если нет — у нас кэш промах. Кэш обращается к памяти через 4 такта после начала запроса, через 100 тактов память ему начинает отвечать и отвечает 8 тактов подряд. Поэтому приавляем 112 к общему количеству тактов. Проверка на то, что у нас линия грязная происходит в строчках 27 и 36. Мы знаем, что грязные линии те и только те, которые содержат в себе элемент массива `c[][]` (только их мы изменяем). $\frac{\texttt{sizeof(a) + sizeof(b)}}{16} = 368$. Мы прибавляем 101 к тактам, потому что 100 тактов проходит с начала запроса к памяти до начала ответа памяти плюс один такт память отвечает. Эти случаи реализованы (с 14 по 31 строчки), функция возвращает 1.

Листинг 2. Реализация самой задачи

```
clock += 1  # initialisation pa
clock += 1  # initialisation pc
clock += 1  # initialisation y
for m in range(M):
```

```
    clock += 1  # iteration
    clock += 1  # y++
    clock += 1  # initialisation x
    for n in range(N):
        clock += 1  # iteration
        clock += 1  # x++
        clock += 1  # initialisation pb
        clock += 1  # initialisation s
        clock += 1  # initialisation k
        for k in range(K):
            clock += 1  # iteration
            clock += 1  # k++
            misses += f(a[m][k])
            clock += 1  # end of op
            misses += f(b[k][n])
            clock += 1  # end of op
            clock += 5  # mult
            clock += 1  # sum
            clock += 1  # sum
        misses += f(c[m][n])
        clock += 1  # end of op
    clock += 1  # sum
    clock += 1  # sum
clock += 1  # end of func

print("Всего обращений к кэшу:", requests)
print("Всего промахов:", misses)
print("Процент попаданя:", (requests - misses) / requests)
print("Количество тактов:", clock)
```

Каждое прибавление к тактам объяснено в комментариях рядом с ним. `end of op` нужен потому, что `f` учитывает разность значений `clock` между началом выполнения операции и концом. Чтобы посчитать суммарное время работы команды, нужно добавить 1 такт.

Данный код выдаёт такой результат:

```
Всего обращений к кэшу: 249600
Всего промахов: 21520
Процент попаданя: 0.9137820512820513
Количество тактов: 5329302
```

Моделирование заданной системы на Verilog

Модель состоит из четырёх модулей, testbench.sv, memory.sv, cache.sv, cpu.sv.

testbench.sv

Модуль testbench.sv создаёт 3 остальных модуля, инициализирует их начальными значениями, соединяет их проводами. Реализацию можно посмотреть в разделе «Листинг кода».

memory.sv

Модуль memory.sv релизует модель оперативной памяти. Краткое описание: вся память это массив регистров. В модуле есть 2 переменные отсчитывающие задержку в 100 тактов. Каждую положительную синхронизацию мы проверяем получает ли память что-то на вход, при условии, что она ничего не выполняет. в отдельном блоке на отрицательную синхронизацию мы отвечаем кэшу обратно. Ниже подробные объяснения

```
parameter MEM_SIZE = 512*1024;
reg [7:0] memory [0: MEM_SIZE - 1];
```

Вся память это массив регистров.

Листинг 3. dump

```
always @(posedge m_dump) begin
   file = $fopen("mem_dump.txt", "w");
   for (int i = 0; i < MEM_SIZE; i += 16) begin
      for (int j = 0; j < 16; j++) begin
         $fwrite(file, "%b␣", memory[i+j]);
      end
      $fwrite(file,"\n");
   end
   $fclose(file);
end
```

Выше написана реализация dump всей памяти в файл

Листинг 4. reset

```
always @(posedge reset) begin
    delay_read = -1;
    delay_write = -1;
    offset = 0;
    adress_line = 0;
    out = 0;
    for (int i = 0; i < MEM_SIZE - 1; i += 1) begin
        memory[i] = $random(SEED)>>16;
    end
end
```

Так происходит сброс всей памяти в начальное состояние

Листинг 5. `case(C2)`

```
always @(posedge clk) begin
    case (C2)
        `READ_LINE: begin
            if(delay_read == -1) begin
                delay_read = 108;
            end
            adress_line = A2;
        end
        `WRITE_LINE: begin
            if(delay_write == -1) begin
                adress_line = A2;
                delay_write = 102;
                offset = 0;
            end
            memory[A2 * 16 + 2 * offset][7:0] <= D2[15:8];
            memory[A2 * 16 + 2 * offset + 1][7:0] <= D2[7:0];

            delay_write -= 1;
            offset += 1;
        end
    endcase
end
```

Выше написана реализация приёма данных от *cache*. Каждую положительную синхронизацию мы проверяем получаем ли мы что-то от кэша. Если да, то запускаем один из двух сценариев. Переменные `delay_*` изначально равные -1 отвечают за соответствующую задержку (константы подобраны ювелирно, чтобы получалось то, что нужно). Переменная `out` позволяет передавать данные по шине, когда 1 и запрещает, когда 0

```
if (delay_write <= 94 && delay_write > 0) begin
    delay_write -= 1;
end
if(delay_write == 1) begin
    out = 1;
    C2_out = 0;
    offset = 0;
end
if(delay_write == 0) begin
    out = 0;
    delay_write = -1;
end
```

При негативной синхронизации мы отдельно обрабатываем задержку на запись (в конце мы должны один такт возвращать `NOP`, чтобы кэш узнал, что мы закончили записывать линию в память)

```
if (delay_read > 8) begin
```

```
    delay_read -= 1;
end
if (delay_read <= 8 && delay_read >= 1) begin
    if(delay_read == 8) begin
        out = 1;
        C2_out = 1;
        offset = 0;
    end
    D2_out <= (memory[adress_line * 16 + 2 * offset] << 8) +
   memory[adress_line * 16 + 2 * offset + 1];
    offset += 1;
    delay_read -= 1;
end else if(delay_read == 0) begin
    C2_out = 0;
    D2_out = 0;
    offset = 0;
    delay_read = -1;
    out = 0;
end
```

Отдельно обрабатываем чтение данных. Последние 8 тактов возвращаем результат. Реализация выше.

cache.sv

```
reg[7: 0] cache_lines[0: SETS_COUNT - 1][0: WAY - 1][0: LINE_SIZE + 1];
```

Кэш это трёхмерный массив байтов (Номер блока, позиция внутри блока, номер байта внутри одной кэш линии). Первые 2 байта в кэш линии отвечают за служебную информацию. Первый бит это `valid`, второй это `dirty`, третий будет отвечать за `time` (какую из двух кэш линий в блоке использовали последней). с 7 по 16 бит включительно в кэш линии хранится `tag`.

```
reg[TAG_SIZE + SET_SIZE - 1: 0] adress_del_line;
reg[TAG_SIZE + SET_SIZE - 1: 0] adress_line;
reg[OFFSET_SIZE - 1: 0] adress_bit;
reg[31: 0] data;
int cmd = 0;

int set;
always @(adress_line) set = adress_line[4:0];

int tag;
always @(adress_line) tag = adress_line[14:5];

int offset;
always @(adress_bit) offset = 2 + adress_bit;
```

`adress_del_line`, `adress_line`, `adress_bit`, `data` это временные данные которые будут сохранять вход с одной или второй шины.

`adress_del_line` — `tagset` грязной линии, которую мы собираемся удалить

`adress_line` — `tagset` линии, которую мы хотим найти в кэше

`adress_bit` — `offset` линии, которую мы хотим найти в кэше

`data` — информация с шины `D1`

`cmd` — номер команды, которую нам дали на вход `C1`

Следующие 6 строчек преобразуют переменные в читаемые.

```
integer file;
int dirty_inv = 0;
int stage = 0;

int line_num;
int mem_dump = 0;
int wait_delay = 0;
int mimo = 0;
```

`file` — переменная для работы с файлом,

`dirty_inv` — переменная индикатор того, что нам нужно сделать инвалидацию грязной линии,

`stage` — стадия нашей команды, бывает -4, -3, 0, 1, 2, 3, 4, 5, 6. Принцип работы описан ниже

Значение	Этап работы
0	Первый такт считывания данных CPU
1	Второй такт считывания данных CPU
2	Обработка случаев работы в зависимости от того попали мы в кэш или нет
3	Случай промаха и грязной линии, которую мы выкидываем
-3	Ожидание `NOP` от памяти (успешная грязная инвалидация)
4	Случай промаха нам нужно достать линию из памяти
-4	Ожидание `RESPONCE` от памяти; запись получаенной линии в *cache*
5	Первый такт ответа CPU
6	Первый такт ответа CPU (в случае команды READ32)

`line_num` — номер линии внутри одного блока,

`mem_dump` — переменная помогает во время выкачивания информации из памяти. Она отсчитывает шаг (`offset`) в кэш линии,

`wait_delay` — задержка, нужна для того чтобы делать команды строго отведённое время,

`mimo` — счётчик промахов кэша.

```
reg out_mem = 0;
reg out_cpu = 0;

logic [15:0] D1_out = 0;
assign D1 = (out_cpu == 1) ? D1_out : 16'bzzzzzzzzzzzzzzzz;

logic [2:0] C1_out = 0;
assign C1 = (out_cpu == 1) ? C1_out : 3'bzzz;

logic [15:0] D2_out = 0;
assign D2 = (out_mem == 1) ? D2_out : 16'bzzzzzzzzzzzzzzzz;

logic [1:0] C2_out = 0;
assign C2 = (out_mem == 1) ? C2_out : 2'bzz;
```

Код выше позволяет реализовать inout выходы.

`out_mem` — позволяет передавать данные по шине 2, когда 1 и запрещает, когда 0

`out_cpu` — позволяет передавать данные по шине 1, когда 1 и запрещает, когда 0

Листинг 6. `dump`

```
file = $fopen("cache_dump.txt", "w");
for (int j = 0; j < 32; j++) begin
    for (int k = 0; k < 2; k++) begin
        for (int i = 0; i < 2 + LINE_SIZE; i++) begin
            $fwrite(file, "%b␣", cache_lines[j][k][i]);
        end
        $fwrite(file,"\n");
    end
    $fwrite(file,"\n\n");
end
$fclose(file);
```

Дамп такой же как в памяти (Листинг 3)

Листинг 7. `reset`

```
always@(posedge reset) begin
    D1_out = 0;
    D2_out = 0;
    C1_out = 0;
    C2_out = 0;
    out_cpu = 0;
    out_mem = 0;
    dirty_inv = 0;
    cmd = 0;
    stage = 0;
    line_num = 0;
    mem_dump = 0;
    wait_delay = 0;
```

```
    mimo = 0;
    adress_del_line = 0;
    adress_line = 0;
    adress_bit = 0;

    for (int i = 0; i < LINE_COUNT; i++) begin
        for (int j = 0; j < WAY; j++) begin
            for (int k = 0; k < 2 + LINE_SIZE; k++) begin
                cache_lines[i][j][k] = 0;
            end
        end
    end
end
```

Ресет такой же как в памяти (Листинг 4)

Листинг 8. `case(C1)`

```
case (C1)
    `READ8: begin
        case(stage)
            0: begin
                cmd = C1;
                adress_line = A1;
                #1 stage = 1;
            end
            1:begin
                adress_bit = A1[OFFSET_SIZE - 1 : 0];
                #1 stage = 2;
            end
        endcase
    end
    `READ16: begin
        ...
    end
    `READ32: begin
        ...
    end
    `INVALIDATE_LINE: begin
        adress_line = A1;
        if (cache_lines[set][0][0][1:0] << 8 + cache_lines[set][0][1] == tag)
   begin
            line_num = 0;
        end else if (cache_lines[set][1][0][1:0] << 8 +
   cache_lines[set][1][1] == tag) begin
            line_num = 1;
        end else begin
            $display("Такой линии нет в кэше");
        end
        if(cache_lines[set][line_num][0][6:6] == 1) begin
            #1 dirty_inv = 1;
            cache_lines[set][line_num][0][7:7] = 0;
            out_mem = 1;
```

```
            A2 = adress_line;
            C2_out = 3;
        end
    end
    `WRITE8: begin
        ...
    end
    `WRITE16: begin
        ...
    end
    `WRITE32: begin
        case(stage)
            0: begin
                cmd = C1;
                adress_line = A1;
                data[31:16] = D1;
                #1 stage = 1;
            end
            1: begin
                adress_bit = A1[OFFSET_SIZE - 1 : 0];
                data[15:0] = D1;
                #1 stage = 2;
            end
        endcase
    end
endcase
```

`case` реализован внутри блока `always` на позитивное изменение синхронизации.

Изначально `stage` = 0. `stage` 0 и 1 это считывание данных на всех командах кроме `INVALIDATE_LINE` и `NOP`. Реализация выше, происходит на положительную синхронизацию.

`INVALIDATE_LINE` делает инвалидацию линии. Сначала выбирает, какая именно линия нам нужна, затем убирает бит валидности, и делает грязную инвалидацию, если грязный бит — 1.

Листинг 9. `dirty_inv`

```
if(dirty_inv >= 1) begin
    #1 D2_out = (cache_lines[set][line_num][2 * dirty_inv] << 8) +
   cache_lines[set][line_num][2 * dirty_inv + 1];
    if(dirty_inv == 1) begin
        out_mem = 1;
    end
    if(dirty_inv == 8) begin
        #1
        cache_lines[set][line_num][0][6:6] = 0;
        out_mem = 0;
        C2_out = 0;
        dirty_inv = 0;
    end else begin
```

```
13          dirty_inv++;
14      end
15  end
```

Грязная инвалидация реализована внутри блока `always` на позитивное изменение синхронизации.

Отправляет 8 тактов подряд линию из кэша в память. В конце обнуляет `dirty_inv`.

Делаем в `always` блоке `case(stage)` и рассмариваем шаги.

Листинг 10. `stage = 2`

```
1  2: begin
2      if (cache_lines[set][0][0][7:7] == 1 && (cache_lines[set][0][0][1:0] <<
       8) + cache_lines[set][0][1] == tag) begin
3          line_num = 0;
4          stage = 5;
5          wait_delay = 2;
6      end else if (cache_lines[set][1][0][7:7] == 1 &&
       (cache_lines[set][1][0][1:0] << 8) + cache_lines[set][1][1] == tag) begin
7          line_num = 1;
8          stage = 5;
9          wait_delay = 2;
10     end else if (cache_lines[set][0][0][7:7] == 1 &&
       cache_lines[set][1][0][7:7] == 1) begin
11         mimo += 1;
12         line_num = cache_lines[set][0][0][5:5];
13         cache_lines[set][line_num][0][7:7] = 0;
14
15         if (cache_lines[set][line_num][0][6:6] == 1) begin
16             #1 stage = 3;
17         end else begin
18             #1 stage = 4;
19         end
20     end else begin
21         mimo += 1;
22
23         if(cache_lines[set][0][0][7:7] == 0) begin
24             line_num = 0;
25         end else begin
26             line_num = 1;
27         end
28         stage = 4;
29     end
30 end
```

`stage` $= 2$ это изначальный разбор случаев. Первые 2 `if` - проверка на то, что линия лежит в кэше. Мы переходим сразу на стадию 5 (Листинг 15) - вывод ответа на запрос в кэш.

Следующий `if` это случай, когда обе линии валидные. Тогда одну из линий надо выкинуть. Мы говорим, что номер линии, которую мы должны

выкинуть из двух это в точности бит времени линии с индексом 0. Действительно, если он 0, значит нулевую линию мы использовали давно, и нам надо её выкинуть. Иначе он 1 и мы выкидываем первую линию.

Дальше если эта линия грязная, мы переходим на стадию удаления этой линии в память `stage` = 3 (Листинг 11). Иначе эта линия чистая и мы можем перейти сразу же к стадии выгрузки новой линии из памяти `stage` = 4 (Листинг 13);

Листинг 11. `stage = 3`

```
3: begin
    if (wait_delay == 0) begin
        adress_del_line[14:13] = cache_lines[set][line_num][0][1:0];
        adress_del_line[12:5] = cache_lines[set][line_num][1];
        adress_del_line[4:0] = set;
        A2 = adress_del_line;
        C2_out = 3;
        dirty_inv = 1;
        stage = -3;
    end else begin
        wait_delay -= 1;
    end
end
```

`stage` = 3: поисходит выгрузка грязной линии в файл. Мы ждём нужное количество тактов, находящееся в переменной `wait_delay`. Отдаём памяти нужный адрес и делаем `dirty_inv`. stage становится -3 (Листинг 12). Мы начинаем ожидать от памяти команду `NOP`.

Листинг 12. `stage = -3`

```
always @(clk == 0) begin
   if(C2 == `NOP)begin
      if(stage == -3) begin
         stage = 4;
      end
   end
end
```

Как только память отдаёт нам команду `NOP` и `stage` = -3, значит мы дождались ответа от памяти, она записала грязную строку и готова принимать новый запрос.

Листинг 13. `stage = 4`

```
4: begin
    if (wait_delay == 0) begin
        #1 A2 = adress_line;
        C2_out = 2;
        out_mem = 1;
        #2 out_mem = 0;
        A2 = 16'bzzzzzzzzzzzzzzzz;
```

```
        mem_dump = 1;
        stage = -4;
    end else begin
        wait_delay -= 1;
    end
end
```

`stage` = 4, *cache* отправляем запрос в память на считывание линии в свободное место в кэше. *Cache* ждёт нужное количество тактов, находящееся в переменной `wait_delay`, затем делает запрос в память и изменяет `stage` на -4 (Листинг 14).

Листинг 14. `stage = -4`

```
if(C2 == `RESPONSE) begin
   if(mem_dump >= 1) begin
      if(mem_dump == 1) begin
         cache_lines[set][line_num][0][1:0] = tag >> 8;
         cache_lines[set][line_num][1] = tag % 256;
         cache_lines[set][line_num][0][7:7] = 1;
         cache_lines[set][line_num][0][6:6] = 0;
         cache_lines[set][line_num][0][5:5] = 1;
         cache_lines[set][line_num][0][5:5] = 0;
      end
      cache_lines[set][line_num][2 * mem_dump] = D2 >> 8;
      cache_lines[set][line_num][2 * mem_dump + 1] = D2 % 256;
      if(mem_dump == 8) begin
         mem_dump = 0;
         if(stage == -4) begin
            stage = 5;
         end
      end else begin
         mem_dump += 1;
      end
   end
end
```

Данный код на каждую положительную синхронизацию ждёт `RESPONSE` от памяти. Когда память начинает отвечать, мы отсчитваем 8 тактов в переменной `mem_dump` и каждый такт считываем новыую порцию данных. В самый первый такт мы устанавливаем начальные значения. Когда мы выгрузили линию, мы можем начать отвечать *CPU*. Переходим в `stage` = 5 (Листинг 15)

Листинг 15. `stage = 5`

```
5: begin
    if(wait_delay == 0) begin
        cache_lines[set][line_num][0][5:5] = 1;
        cache_lines[set][1 - line_num][0][5:5] = 0;
        case(cmd)
            `READ8: begin
```

```
            ...
        end
        `READ16: begin
            #1 stage = -1;
            C1_out = 7;
            out_cpu = 1;
            D1_out = (cache_lines[set][line_num][offset] << 8) +
    cache_lines[set][line_num][offset + 1];
            #1 out_cpu = 0;
            #1 stage = 0;
        end
        `READ32: begin
            #1 stage = 6;
            out_cpu = 1;
            C1_out = 7;
            D1_out = (cache_lines[set][line_num][offset] << 8) +
    cache_lines[set][line_num][offset + 1];
        end
        `WRITE8: begin
            ...
        end
        `WRITE16: begin
            ...
        end
        `WRITE32: begin
            cache_lines[set][line_num][0][6:6] = 1;
            cache_lines[set][line_num][offset] = data[31:24];
            cache_lines[set][line_num][offset + 1] = data[23:16];
            cache_lines[set][line_num][offset + 2] = data[15:8];
            cache_lines[set][line_num][offset + 3] = data[7:0];
            #1 stage = -1;
            out_cpu = 1;
            C1_out = 0;
            #1 out_cpu = 0;
            #1 stage = 0;
        end
    endcase
  end else begin
    wait_delay -= 1;
  end
end
```

`stage` = 5: *cache* рассматриваем все возможные значение `cmd`. Для каждого из них *cache* отвечает процессору. Если команда `READ32`, *cache* вынужден отвечать 2 такта, поэтому переходит в `stage` = 6. В конце *cache* делает `stage` = 0 и начинает ждать следующей команды от CPU.

Листинг 16. `stage = 6`

```
6: begin
    #1 stage = -1;
    D1_out = cache_lines[set][line_num][offset + 2] << 8 +
```

```
    cache_lines[set][line_num][offset + 3];
    #1 out_cpu = 0;
    #1 stage = 0;
end
```

`stage` = 6: вторая часть команды `READ32`.

```
initial begin
    #11000000
    $display("all␣cpu␣->␣cache␣misses:␣␣%d", mimo);
end
```

В `init` блоке *cache* ждёт пока моделируемая задача точно завершится. Затем выводит суммарное количество промахов.

Воспроизведение задачи на Verilog

cpu.sv

Модуль cpu.sv моделирует саму задачу перемножения матриц.

```
reg out = 0;
logic [15:0] D1_out = 0;
assign D1 = (out == 1) ? D1_out : 16'bzzzzzzzzzzzzzzzz;

logic [2:0] C1_out = 0;
assign C1 = (out == 1) ? C1_out : 3'bzzz;

int clock = 0;
always @(posedge clk) begin
    clock += 1;
end
```

Выше задаются переменные для отправление данных на шину `inout`. Переменная `out` работает как в двух других модулях. Открывает и закрывает шину в зависимости от значения `out`.

```
parameter M = 64;
parameter N = 60;
parameter K = 32;

int pa;
int pb;
int pc;
int resa;
int resb;
int vsego = 0;
int s = 0;
```

Выше определяются параметры, нужные для реализации самой задачи.

```
initial begin
   #2 pa = 0;
   #2 pc = M * K + K * N * 2;
      #2 //initialisation y
         for (int y = 0; y < M; y++) begin
            #2 //iteration
            #2 //y++
            #2 //initialisation x
            for (int x = 0; x < N; x++) begin
               #2 //iteration
               #2 //x++
               #2 pb = M * K;
               #2 s = 0;
               #2 //initialisation k
               for (int k = 0; k < K; k++) begin
                  #2 //iteration
                  #2 // k++
```

```
                    vsego += 1;
                       //read8
                       wait(clk == 0)
                       A1 = (pa + k) >> 4;
                       C1_out = 1;
                       out = 1;
                       #2
                       A1 = (pa + k) % 16;
                       #2
                       out = 0;
                       wait(C1 == 7)
                    resa = D1;
                    #2 // end of op
                    vsego += 1;
                       //read16
                       wait(clk == 0)
                       A1 = (pb + x * 2) >> 4;
                       C1_out = 2;
                       out = 1;
                       #2
                       A1 = (pb + x * 2) % 16;
                       #2
                       out = 0;
                       wait(C1 == 7)
                    resb = D1;
                    #2 // end of op
                    #12 s += resa * resb;
                    #2 pb += N * 2;
                 end
                 vsego += 1;
                    //write32
                    wait(clk == 0)
                    A1 = (pc + x * 4) >> 4;
                    C1_out = 7;
                    D1_out = s >> 16;
                    out = 1;
                    #2 A1 = (pc + x * 4) % 16;
                    D1_out = s % (256*256);
                    #2 // end of op
                    out = 0;
                    wait(C1 == 0);
                 #2; // end of op
              end
           #2 pa += K;
           #2 pc += N * 4;
           $display(y);
        end
     #2 // end of func
     $display("clock: %d", clock);
     $display("all cpu -> cache requests: %d", vsego);
end
```

Выше реализована задача. Около каждой задержки помимо тех, которые происходят внутри отправки, в комментариях написано для чего она нужна. Сама реалицация отличается от реализации на питоне, только отправками команд кэшу. Это делается в два такта.

Вывод программы такой:

```
clock: 5329302
all cpu -> cache requests: 249600
all cpu -> cache misses: 21520
```

Сравнение полученных результатов

Результаты программ в табличке ниже

Результаты	Аналитически	Verilog
Промахи	21520	21520
Запросы в *cache*	249600	249600
Количество тактов	5329302	5329302
Процент попадания	0.913782	0.913782

Результаты сходятся, что означет согласованность модели в verilog с аналитическим решением.

Перед тем как читать некрасивый код, предлагаю посмотреть на красивую панду от Насти (ей обязательно нужно поставить максимальный балл за ЛР2)!

Рис. 3. Красивая панда от Насти

Листинг кода

Листинг 17. acos.py

```
M = 64
N = 60
K = 32

a = [[0] * K for _ in range(M)]  # array a
b = [[0] * N for _ in range(K)]  # array b
c = [[0] * N for _ in range(M)]  # array c

offset = 0
tagset = 0

for i in range(M):
   for j in range(K):
      a[i][j] = tagset
      offset += 1
      if offset % 16 == 0:
         tagset += 1
         offset = 0

for i in range(K):
   for j in range(N):
      b[i][j] = tagset
      offset += 2
      if offset % 16 == 0:
         tagset += 1
         offset = 0

for i in range(M):
   for j in range(N):
      c[i][j] = tagset
      offset += 4
      if offset % 16 == 0:
         tagset += 1
         offset = 0

cache = [[0] * 2 for _ in range(64)]

for i in range(64):
   cache[i][0] = -1

misses = 0  # количество промахов
requests = 32 * 64 * 60 * 2 + 64 * 60  # количество запросов всего
clock = 0  # количество тактов


def f(line_adr: int) -> int:
   global clock
   st = line_adr % 32
```

```
    if cache[2 * st][0] == line_adr:  # первая линия нужная
        clock += 6  # время отклика кэша при попадании
        cache[2 * st][1] = 1
        cache[2 * st + 1][1] = 0
        return 0  # попали
    if cache[2 * st + 1][0] == line_adr:  # вторая линия нужная
        clock += 6  # время отклика кэша при попадании
        cache[2 * st + 1][1] = 1
        cache[2 * st][1] = 0
        return 0  # попали
    if cache[2 * st][1] == 0:  # первая линия использовалась давно
        clock += 112  # 4 + 100 + 8
        # наша линия грязная, когда в ней лежит элемент из c[][]
        if cache[2 * st][0] >= 368:
            clock += 101  # 100 + 1
        cache[2 * st][0] = line_adr
        cache[2 * st][1] = 1
        cache[2 * st + 1][1] = 0
        return 1  # промах
    if cache[2 * st + 1][1] == 0:  # вторая линия использовалась давно
        clock += 112  # 4 + 100 + 8
        # наша линия грязная, когда в ней лежит элемент из c[][]
        if cache[2 * st + 1][0] >= 368:
            clock += 101  # 100 + 1
        cache[2 * st + 1][0] = line_adr
        cache[2 * st + 1][1] = 1
        cache[2 * st][1] = 0
        return 1  # промах
    assert False  # невозможное состояние


clock += 1  # initialisation pa
clock += 1  # initialisation pc
clock += 1  # initialisation y
    for m in range(M):
    clock += 1  # iteration
    clock += 1  # y++
    clock += 1  # initialisation x
        for n in range(N):
        clock += 1  # iteration
        clock += 1  # x++
        clock += 1  # initialisation pb
        clock += 1  # initialisation s
        clock += 1  # initialisation k
            for k in range(K):
            clock += 1  # iteration
            clock += 1  # k++
            misses += f(a[m][k])
            clock += 1  # end of op
            misses += f(b[k][n])
            clock += 1  # end of op
            clock += 5  # mult
```

```
            clock += 2  # sum
        misses += f(c[m][n])
        clock += 1  # end of op
    clock += 2  # sum
clock += 1  # end of func

print("Всего обращений к кэшу:", requests)
print("Всего промахов:", misses)
print("Процент попаданя:", (requests - misses) / requests)
print("Количество тактов:", clock)
```

Листинг 18. testbench.sv

```
`include "memory.sv"
`include "cache.sv"
`include "cpu.sv"

module cache_tb;
    wire [15:0] D2;
    wire [15:0] D1;
    wire [1:0] C2;
    wire [2:0] C1;
    reg [14:0] A2;
    reg [14:0] A1;
    reg clk = 0;
    reg reset;
    reg m_dump;
    reg c_dump;

    memory _memory(D2, C2, A2, clk, reset, m_dump);
    cache _cache(D1, C1, A1, D2, C2, A2, clk, reset, c_dump);
    cpu _cpu(D1, C1, A1, clk);
    initial begin
        // $monitor("%0t: A1 = %b, C1 = %b, D1 = %b\n\n\tA2 = %b, C2 = %b, D2
   = %b\n\n", $time, A1, C1, D1, A2, C2, D2);
        reset = 1;
        #1 reset = 0;
        #11000000
        $finish;
    end
    always #1 clk = ~clk;
endmodule
```

Листинг 19. memory.sv

```
`define NOP 0
`define READ_LINE 2
`define WRITE_LINE 3

module memory(
    inout wire [15:0] D2,
    inout wire [1:0] C2,
```

```
    input [14:0] A2,
    input clk,
    input reset,
    input m_dump
    );
    parameter MEM_SIZE = 512*1024;
    parameter _SEED = 225526;
    integer SEED = _SEED;

    reg [7:0] memory [0: MEM_SIZE - 1];

    int delay_read;
    int delay_write;
    int offset = 0;
    int adress_line;
    int file;

    reg out = 0;
    logic [15:0] D2_out = 0;
    assign D2 = (out == 1) ? D2_out : 16'bzzzzzzzzzzzzzzzz;

    logic [1:0] C2_out = 0;
    assign C2 = (out == 1) ? C2_out : 2'bzz;

    always@(posedge m_dump) begin
        file = $fopen("mem_dump.txt", "w");
        for (int i = 0; i < MEM_SIZE; i += 16) begin
            for (int j = 0; j < 16; j++) begin
                $fwrite(file, "%b␣", memory[i+j]);
            end
            $fwrite(file,"\n");
        end
        $fclose(file);
    end

    always@(posedge reset) begin
        delay_read = -1;
        delay_write = -1;
        offset = 0;
        adress_line = 0;
        out = 0;
        for (int i = 0; i < MEM_SIZE - 1; i += 1) begin
            memory[i] = $random(SEED)>>16;
        end
    end

    always @(posedge clk) begin
        case (C2)
            `READ_LINE: begin
                if(delay_read == -1) begin
                    delay_read = 108;
                end
```

```
60              adress_line = A2;
61          end
62          `WRITE_LINE: begin
63              if(delay_write == -1) begin
64                  adress_line = A2;
65                  delay_write = 102;
66                  offset = 0;
67              end
68              memory[A2 * 16 + 2 * offset][7:0] <= D2[15:8];
69              memory[A2 * 16 + 2 * offset + 1][7:0] <= D2[7:0];
70
71              delay_write -= 1;
72              offset += 1;
73          end
74      endcase
75  end
76
77
78  always @(negedge clk) begin
79      if (delay_write <= 94 && delay_write > 0) begin
80          delay_write -= 1;
81      end
82      if(delay_write == 1) begin
83          out = 1;
84          C2_out = 0;
85          offset = 0;
86      end
87      if(delay_write == 0) begin
88          out = 0;
89          delay_write = -1;
90      end
91
92      if (delay_read > 8) begin
93          delay_read -= 1;
94      end
95      if (delay_read <= 8 && delay_read >= 1) begin
96          if(delay_read == 8) begin
97              out = 1;
98              C2_out = 1;
99              offset = 0;
100         end
101         D2_out <= (memory[adress_line * 16 + 2 * offset] << 8) +
    memory[adress_line * 16 + 2 * offset + 1];
102         offset += 1;
103         delay_read -= 1;
104     end else if(delay_read == 0) begin
105         C2_out = 0;
106         D2_out = 0;
107         offset = 0;
108         delay_read = -1;
109         out = 0;
110     end
```

```
    end
endmodule
```

Листинг 20. cache.sv

```
`define NOP 0
`define READ8 1
`define READ16 2
`define READ32 3
`define INVALIDATE_LINE 4
`define WRITE8 5
`define WRITE16 6
`define WRITE32 7
`define RESPONSE 1

module cache(
    inout wire [15:0] D1,
    inout wire [2:0] C1,
    input [14:0] A1,
    inout wire [15:0] D2,
    inout wire [1:0] C2,
    output reg[14:0] A2,
    input clk,
    input reset,
    input c_dump
);
    parameter LINE_SIZE = 16;
    parameter LINE_COUNT = 64;
    parameter SIZE = LINE_SIZE * LINE_COUNT;
    parameter WAY = 2;
    parameter SETS_COUNT = LINE_COUNT / WAY;
    parameter TAG_SIZE = 10;
    parameter SET_SIZE = 5;
    parameter OFFSET_SIZE = 4;
    parameter ADDR_SIZE = TAG_SIZE + SET_SIZE + OFFSET_SIZE;

    reg[7: 0] cache_lines[0: SETS_COUNT - 1][0: WAY - 1][0: LINE_SIZE + 1];

    reg[TAG_SIZE + SET_SIZE - 1: 0] adress_del_line;
    reg[TAG_SIZE + SET_SIZE - 1: 0] adress_line;
    reg[OFFSET_SIZE - 1: 0] adress_bit;
    reg[31: 0] data;

    integer file;
    int dirty_inv = 0;
    int cmd = 0;
    int stage = 0;

    int line_num;
    int mem_dump = 0;
    int wait_delay = 0;
    int mimo = 0;
```

```
48
49      reg out_mem = 0;
50      reg out_cpu = 0;
51
52      logic [15:0] D1_out = 0;
53      assign D1 = (out_cpu == 1) ? D1_out : 16'bzzzzzzzzzzzzzzzz;
54
55      logic [2:0] C1_out = 0;
56      assign C1 = (out_cpu == 1) ? C1_out : 3'bzzz;
57
58      logic [15:0] D2_out = 0;
59      assign D2 = (out_mem == 1) ? D2_out : 16'bzzzzzzzzzzzzzzzz;
60
61      logic [1:0] C2_out = 0;
62      assign C2 = (out_mem == 1) ? C2_out : 2'bzz;
63
64
65      int set;
66      always @(adress_line) set = adress_line[4:0];
67
68      int tag;
69      always @(adress_line) tag = adress_line[14:5];
70
71      int offset;
72      always @(adress_bit) offset = 2 + adress_bit;
73
74      always@(posedge c_dump) begin
75          file = $fopen("cache_dump.txt", "w");
76          for (int j = 0; j < 32; j++) begin
77              for (int k = 0; k < 2; k++) begin
78                  for (int i = 0; i < 2 + LINE_SIZE; i++) begin
79                      $fwrite(file, "%b␣", cache_lines[j][k][i]);
80                  end
81                  $fwrite(file,"\n");
82              end
83              $fwrite(file,"\n\n");
84          end
85          $fclose(file);
86      end
87
88      always@(posedge reset) begin
89          D1_out = 0;
90          D2_out = 0;
91          C1_out = 0;
92          C2_out = 0;
93          out_cpu = 0;
94          out_mem = 0;
95          dirty_inv = 0;
96          cmd = 0;
97          stage = 0;
98          line_num = 0;
99          mem_dump = 0;
```

```
        wait_delay = 0;
        mimo = 0;
        adress_del_line = 0;
        adress_line = 0;
        adress_bit = 0;

        for (int i = 0; i < LINE_COUNT; i++) begin
            for (int j = 0; j < WAY; j++) begin
                for (int k = 0; k < 2 + LINE_SIZE; k++) begin
                    cache_lines[i][j][k] = 0;
                end
            end
        end
    end

    always @(posedge clk) begin
        if(C2 == `RESPONSE) begin
            if(mem_dump >= 1) begin
                if(mem_dump == 1) begin
                    cache_lines[set][line_num][0][1:0] = tag >> 8;
                    cache_lines[set][line_num][1] = tag % 256;
                    cache_lines[set][line_num][0][7:7] = 1;
                    cache_lines[set][line_num][0][6:6] = 0;
                    cache_lines[set][line_num][0][5:5] = 1;
                    cache_lines[set][line_num][0][5:5] = 0;
                end
                cache_lines[set][line_num][2 * mem_dump] = D2 >> 8;
                cache_lines[set][line_num][2 * mem_dump + 1] = D2 % 256;
                if(mem_dump == 8) begin
                    mem_dump = 0;
                    if(stage == -4) begin
                        stage = 5;
                    end
                end else begin
                    mem_dump += 1;
                end
            end
        end

        case(stage)
            2: begin
                if (cache_lines[set][0][0][7:7] == 1 &&
    (cache_lines[set][0][0][1:0] << 8) + cache_lines[set][0][1] == tag) begin
                    line_num = 0;
                    stage = 5;
                    wait_delay = 2;
                end else if (cache_lines[set][1][0][7:7] == 1 &&
    (cache_lines[set][1][0][1:0] << 8) + cache_lines[set][1][1] == tag) begin
                    line_num = 1;
                    stage = 5;
                    wait_delay = 2;
                end else if (cache_lines[set][0][0][7:7] == 1 &&
```

```
    cache_lines[set][1][0][7:7] == 1) begin
150                     mimo += 1;
151                     line_num = cache_lines[set][0][0][5:5];
152                     cache_lines[set][line_num][0][7:7] = 0;
153
154                     if (cache_lines[set][line_num][0][6:6] == 1) begin
155                         #1 stage = 3;
156                     end else begin
157                         #1 stage = 4;
158                     end
159                 end else begin
160                     mimo += 1;
161
162                     if(cache_lines[set][0][0][7:7] == 0) begin
163                         line_num = 0;
164                     end else begin
165                         line_num = 1;
166                     end
167                     stage = 4;
168                 end
169             end
170             3: begin
171                 if (wait_delay == 0) begin
172                     adress_del_line[14:13] =
    cache_lines[set][line_num][0][1:0];
173                     adress_del_line[12:5] = cache_lines[set][line_num][1];
174                     adress_del_line[4:0] = set;
175                     A2 = adress_del_line;
176                     C2_out = 3;
177                     dirty_inv = 1;
178                     stage = -3;
179                 end else begin
180                     wait_delay -= 1;
181                 end
182             end
183             4: begin
184                 // $display($time);
185                 if (wait_delay == 0) begin
186                     #1 A2 = adress_line;
187                     C2_out = 2;
188                     out_mem = 1;
189                     #2 out_mem = 0;
190                     A2 = 16'bzzzzzzzzzzzzzzzz;
191                     mem_dump = 1;
192                     stage = -4;
193                 end else begin
194                     wait_delay -= 1;
195                 end
196             end
197             5: begin
198                 if(wait_delay == 0) begin
199                     cache_lines[set][line_num][0][5:5] = 1;
```

```
200                     cache_lines[set][1 - line_num][0][5:5] = 0;
201                     case(cmd)
202                         `READ8: begin
203                             #1 stage = -1;
204                             C1_out = 7;
205                             out_cpu = 1;
206                             D1_out = cache_lines[set][line_num][offset];
207                             #1 out_cpu = 0;
208                             #1 stage = 0;
209                         end
210                         `READ16: begin
211                             #1 stage = -1;
212                             C1_out = 7;
213                             out_cpu = 1;
214                             D1_out = (cache_lines[set][line_num][offset] <<
    8) + cache_lines[set][line_num][offset + 1];
215                             #1 out_cpu = 0;
216                             #1 stage = 0;
217                         end
218                         `READ32: begin
219                             #1 stage = 6;
220                             out_cpu = 1;
221                             C1_out = 7;
222                             D1_out = (cache_lines[set][line_num][offset] <<
    8) + cache_lines[set][line_num][offset + 1];
223                         end
224                         `WRITE8: begin
225                             cache_lines[set][line_num][0][6:6] = 1;
226                             cache_lines[set][line_num][offset] = data[7:0];
227                             #1 stage = 0;
228                             out_cpu = 1;
229                             C1_out = 0;
230                             #2 out_cpu = 0;
231                         end
232                         `WRITE16: begin
233                             cache_lines[set][line_num][0][6:6] = 1;
234                             cache_lines[set][line_num][offset] = data[15:8];
235                             cache_lines[set][line_num][offset + 1] =
    data[7:0];
236                             #1 stage = 0;
237                             out_cpu = 1;
238                             C1_out = 0;
239                             #2 out_cpu = 0;
240                         end
241                         `WRITE32: begin
242                             cache_lines[set][line_num][0][6:6] = 1;
243                             cache_lines[set][line_num][offset] = data[31:24];
244                             cache_lines[set][line_num][offset + 1] =
    data[23:16];
245                             cache_lines[set][line_num][offset + 2] =
    data[15:8];
246                             cache_lines[set][line_num][offset + 3] =
```

```
data[7:0];
                            #1 stage = -1;
                            out_cpu = 1;
                            C1_out = 0;
                            #1 out_cpu = 0;
                            #1 stage = 0;
                        end
                    endcase
                end else begin
                    wait_delay -= 1;
                end
            end
            6: begin
                #1 stage = -1;
                D1_out = cache_lines[set][line_num][offset + 2] << 8 +
cache_lines[set][line_num][offset + 3];
                #1 out_cpu = 0;
                #1 stage = 0;
            end
        endcase

        if(dirty_inv >= 1) begin
            #1 D2_out = (cache_lines[set][line_num][2 * dirty_inv] << 8) +
cache_lines[set][line_num][2 * dirty_inv + 1];
            if(dirty_inv == 1) begin
                out_mem = 1;
            end
            if(dirty_inv == 8) begin
                #1
                out_mem = 0;
                C2_out = 0;
                dirty_inv = 0;
            end else begin
                dirty_inv++;
            end
        end

        case (C1)
            `READ8: begin
                case(stage)
                    0: begin
                        cmd = 1;
                        adress_line = A1;
                        #1 stage = 1;
                    end
                    1:begin
                        adress_bit = A1[OFFSET_SIZE - 1 : 0];
                        #1 stage = 2;
                    end
                endcase
            end
            `READ16: begin
```

```
296                 case(stage)
297                     0: begin
298                         cmd = 2;
299                         adress_line = A1;
300                         #1 stage = 1;
301                     end
302                     1:begin
303                         adress_bit = A1[OFFSET_SIZE - 1 : 0];
304                         #1 stage = 2;
305                     end
306                 endcase
307             end
308             `READ32: begin
309                 case(stage)
310                     0: begin
311                         cmd = C1;
312                         adress_line = A1;
313                         #1 stage = 1;
314                     end
315                     1:begin
316                         adress_bit = A1[OFFSET_SIZE - 1 : 0];
317                         #1 stage = 2;
318                     end
319                 endcase
320             end
321             `INVALIDATE_LINE: begin
322                 adress_line = A1;
323                 if (cache_lines[set][0][0][1:0] << 8 + cache_lines[set][0][1]
    == tag) begin
324                     line_num = 0;
325                 end else if (cache_lines[set][1][0][1:0] << 8 +
    cache_lines[set][1][1] == tag) begin
326                     line_num = 1;
327                 end else begin
328                     $display("Такой линии нет в кэше");
329                 end
330                 if(cache_lines[set][line_num][0][6:6] == 1) begin
331                     #1 dirty_inv = 1;
332                     cache_lines[set][line_num][0][7:7] = 0;
333                     out_mem = 1;
334                     A2 = adress_line;
335                     C2_out = 3;
336                 end
337             end
338             `WRITE8: begin
339                 case(stage)
340                     0: begin
341                         cmd = C1;
342                         adress_line = A1;
343                         data[7:0] = D1;
344                         #1 stage = 1;
345                     end
```

```
                    1:begin
                        adress_bit = A1[OFFSET_SIZE - 1 : 0];
                        #1 stage = 2;
                    end
                endcase
            end
            `WRITE16: begin
                case(stage)
                    0: begin
                        cmd = C1;
                        adress_line = A1;
                        data[15:0] = D1;
                        #1 stage = 1;
                    end
                    1:begin
                        adress_bit = A1[OFFSET_SIZE - 1 : 0];
                        #1 stage = 2;
                    end
                endcase
            end
            `WRITE32: begin
                case(stage)
                    0: begin
                        cmd = C1;
                        adress_line = A1;
                        data[31:16] = D1;
                        #1 stage = 1;
                    end
                    1: begin
                        adress_bit = A1[OFFSET_SIZE - 1 : 0];
                        data[15:0] = D1;
                        #1 stage = 2;
                    end
                endcase
            end
        endcase
    end

    always @(clk == 0) begin
        if(C2 == `NOP)begin
            if(stage == -3) begin
                stage = 4;
            end
        end
    end

    initial begin
        #11000000
        $display("all cpu -> cache misses:  %d", mimo);
    end
endmodule
```

Листинг 21. cpu.sv

```
module cpu(
    inout wire [15:0] D1,
    inout wire [2:0] C1,
    output reg[14:0] A1,
    input clk
    );
    parameter M = 64;
    parameter N = 60;
    parameter K = 32;

    reg out = 0;
    int pa;
    int pb;
    int pc;
    int resa;
    int resb;
    int vsego = 0;
    int clock = 0;
    int s = 0;

    logic [15:0] D1_out = 0;
    assign D1 = (out == 1) ? D1_out : 16'bzzzzzzzzzzzzzzzz;

    logic [2:0] C1_out = 0;
    assign C1 = (out == 1) ? C1_out : 3'bzzz;

    always @(posedge clk) begin
        clock += 1;
    end

    initial begin
        #2 pa = 0;
        #2 pc = M * K + K * N * 2;
        #2 //initialisation y
        for (int y = 0; y < M; y++) begin
            #2 //iteration
            #2 //y++
            #2 //initialisation x
            for (int x = 0; x < N; x++) begin
                #2 //iteration
                #2 //x++
                #2 pb = M * K;
                #2 s = 0;
                #2 //initialisation k
                for (int k = 0; k < K; k++) begin
                    #2 //iteration
                    #2 // k++
                    vsego += 1;
                        //read8
                        wait(clk == 0)
                        A1 = (pa + k) >> 4;
```

```
52                          C1_out = 1;
53                          out = 1;
54                          #2
55                          A1 = (pa + k) % 16;
56                          #2
57                          out = 0;
58                          wait(C1 == 7)
59                      resa = D1;
60                      #2
61                      vsego += 1;
62                          //read16
63                          wait(clk == 0)
64                          A1 = (pb + x * 2) >> 4;
65                          C1_out = 2;
66                          out = 1;
67                          #2
68                          A1 = (pb + x * 2) % 16;
69                          #2
70                          out = 0;
71                          wait(C1 == 7)
72                      resb = D1;
73                      #2
74                      #12 s += resa * resb;
75                      #2 pb += N * 2;
76                  end
77                  vsego += 1;
78                      //write32
79                      wait(clk == 0)
80                      A1 = (pc + x * 4) >> 4;
81                      C1_out = 7;
82                      D1_out = s >> 16;
83                      out = 1;
84                      #2
85                      A1 = (pc + x * 4) % 16;
86                      D1_out = s % (256*256);
87                      #2
88                      out = 0;
89                      wait(C1 == 0);
90                  #2;
91              end
92              #2 pa += K;
93              #2 pc += N * 4;
94              $display(y);
95          end
96          #2
97          $display("clock:␣%d", clock);
98          $display("all␣cpu␣->␣cache␣requests:␣%d", vsego);
99      end
100
101 endmodule
```