1

Лабораторная работа №2	Б05	2022
Моделирование схем в Verilog	Рощина Надежда Романовна	

2 Цель работы

Построение кэша и моделирование системы “процессор-кэш-память” на языке описания Verilog

3 Инструментарий

Моделирование системы – Verilog

Компиляция и симуляция – на локальной машине (macOS) с использованием компилятора и симулятора Icarus Verilog 11 (stable)

4 Формулировка задачи из условия

Имеется следующее определение глобальных переменных и функций:

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

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

Сложение, инициализация переменных и переход на новую итерацию цикла, выход из функции занимают 1 такт. Умножение – 5 тактов. Обращение к памяти вида pc[x] считается за одну команду. Массивы последовательно хранятся в памяти, и первый из них начинается с 0. Все локальные переменные лежат в регистрах процессора. По моделируемой шине происходит только обмен данными (не командами).

Определить:

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

5 Вычисление недостающих параметров системы

Известные нам параметры:

1. $mem\_size = 512\ Kbytes = 2^{19}\ bytes$
2. $cache\_line\_size = 16\ bytes$
3. $cache\_line\_count = 64$
4. $cache\_way = 2$
5. $cache\_tag\_size = 10\ bits$
6. $data1\_bus\_size = 16\ bits$
7. $data2\_bus\_size = 16\ bits$

Поскольку общий размер памяти $mem\_size = 512\ Kbytes = 2^{19}\ bytes$, а минимальный адресуемый учаток памяти – $1\ byte$, общая длина адреса будет равняться $cache\_addr\_size = log_2(2^{19})\ bits = 19\ bits$

Порция кэшируемой памяти $cache\_line\_size = 16\ bytes$, поэтому при интерпретации адреса кэшем смещение $cache\_offset\_size = log_2(16) = 4\ bits$

Соответственно, $cache\_set\_size = cache\_line\_size - cache\_tag\_size - cache\_offset\_size = 19 - 10 - 4\ bits = 5\ bits$

Поскольку размер одного блока $cache\_way = 2$, а $cache\_line\_count = 64$, то $cache\_sets\_count = \frac{cache\_line\_count}{cache\_way} = 32$

Размер полезной памяти кэша $cache\_size = cache\_line\_count \cdot cache\_line\_size = 64 \cdot 16\ bytes = 1\ Kbyte$

По шине $A1$ за один такт передаются $cache\_tag\_size + cache\_set\_size = 15\ bits$ или $cache\_offset\_size = 4\ bits$, поэтому $addr1\_bus\_size = 15\ bits$

По $A2$ за такт передаются $cache\_tag\_size + cache\_set\_size = 15\ bits$, $addr2\_bus\_size = 15\ bits$

$C1$ должна принимать значения команд в диапазоне 0 – 7, поэтому $ctr1\_bus\_size = log_2(8) = 3\ bits$

$C2$ должна принимать значения команд в диапазоне 0 – 3, поэтому $ctr2\_bus\_size = log_2(4) = 2\ bits$

Вычисленные параметры:

1. $cache\_addr\_size = 19\ bits$
2. $cache\_offset\_size = 4\ bits$
3. $cache\_set\_size = 5\ bits$
4. $cache\_sets\_count = 32$
5. $cache\_size = 1\ Kbyte$
6. $addr1\_bus\_size = 15\ bits$
7. $addr2\_bus\_size = 15\ bits$
8. $ctr1\_bus\_size = 3\ bits$
9. $ctr2\_bus\_size = 2\ bits$

6 Аналитическое решение задачи

Сначала определим последовательность адресов памяти, к которым будет обращаться наша программа при исполнении

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

# matrix sizes in the memory
# elements of a take sizeof(int8) = 1 byte
# elements of b - sizeof(int16) = 2 bytes
# elements of c - sizeof(int32) = 4 bytes
size_a = M * K * 1
size_b = K * N * 2
size_c = M * N * 4

# determining the sequence of memory addresses that we access in the function
mem_access_stack = list()

pa = 0    # a data begins right at the 0 address
pc = size_a + size_b    # c data begins right after all a and b data

for y in range(M):

    for x in range(N):

        pb = size_a    # b data begins right after all a data

        for k in range(K):

            # accessing pa[k] = [pa + k]
            mem_access_stack.append(pa + 1 * k)

            # accessing pb[x] = [pb + 2x, pb + 2x + 1]
            for i in range(2):
                mem_access_stack.append(pb + 2 * x + i)

            pb += 2 * N    # moving pb pointer to N sets of 2 bytes

        # accessing pc[x] = [pc + 4x, ..., pc + 4x + 3]
        for i in range(4):
            mem_access_stack.append(pc + 4 * x + i)

    pa += K * 1    # moving pa pointer to K sets of 1 byte
    pc += N * 4    # moving pc pointer to N sets of 4 bytes
```

Теперь будем моделировать работу кэша, не запоминая самих значений, а лишь поддерживая актуальность данных

Заметим, что для оценки числа попаданий нам не нужно разделять запросы на чтение и запись, поскольку и чтение, запись в любом случае подтягивают значение из памяти в кэш, и содержание кэша не зависит от типа обращения

```
# modeling cache system
# we will remember the state of cache line (valid, dirty),
# but won't fetch and store the data itself
```

```
# parsing memory adress
def parse_address(x):
    # x = [b_18 : b_0]
    tag = x >> 9    # [b_18 : b_9]
    index = (x >> 4) % 32    # [b_8 : b_4]
    offset = x % 16    # [b_3 : b_0]
    return tag, index, offset

# cache consists of 32 sets of two cache lines
# cache[x][i] - i-th cahce line of x-th set in format [valid, dirty, tag]
cache = [[[False, False, None], [False, False, None]] for i in range(32)]


# returns True if data of xth adress is in cache
# fetches data of xth address to cache and returns False otherwise
import copy
def get_x(x):
    tag, index, offset = parse_address(x)
    for i in range(2):
        if cache[index][i][0] == True and cache[index][i][2] == tag:
            return True
    # if xth adress line is not in cache,
    # we need to fetch it to cache[index][0]
    # and replace cache[index][1] with previous value
    # of cache[index][0], if it was valid
    if cache[index][0][0] == False:
        cache[index][0] = [True, False, tag]
    else:
        cache[index][1] = copy.deepcopy(cache[index][0])
        cache[index][0] = [True, False, tag]
    return False

# any time we access some address in memory, we first search it in cache
# here we model the cache and count the number of hits to the cache
hits = 0
total = len(mem_access_stack)
for x in mem_access_stack:
    if get_x(x):
        hits += 1

print("hits rate: {:.2%}".format(hits/total))
```

Результат работы программы:

```
hits rate: 93.89%
```

Теперь посчитаем общее число тактов. Для этого отдельно посчитаем время, затраченное на работу с памятью, и время, потраченное на итерацию циклов, обновление значений локальных переменных и т.п.

Для оценки времени на работу с памятью опять же промоделируем работу кэша, не запоминая конкретных значений, но будем разделять запросы на чтение и на запись, так как нам нужно учитывать время проталкивания в память вытесненных из кэша измененных данных.

```
# returns time needed to read the data of xth adress line is in cache
def get_time_read_x(x):
    tag, index, offset = parse_address(x)
    # number of bytes we are reading (1 for matrix a, 2 for b, 4 for c)
```

```
    responce_time = 0
    if x < size_a:
        responce_time = 1
    elif x < size_a + size_b:
        responce_time = 2
    else:
        responce_time = 4

    for i in range(2):
        if cache[index][i][0] == True and cache[index][i][2] == tag:
            # if x is in cahce, we immedeatly return the value
            return 6 + responce_time

    # if xth adress line is not in cache,
    # we need to fetch it to cache[index][0]

    # in case cache[index][0] is not valid, we'll write x there
    if cache[index][0][0] == False:
        cache[index][0] = [True, False, tag]
        return 4 + 100 + responce_time
        # 4 for searching in cahce and 100 for fetching from memory

    # otherwise if cache[index][0] is valid,
    # we'll replace cahce[index][1] with it
    else:
        res_time = 4 + 100 + responce_time
        # 4 for searching in cahce and 100 for fetching from memory

        # if cahce[index][1] is valid and dirty,
        # we need to push it to memory
        if cache[index][1][0] == True and cache[index][1][1] == True:
            res_time += 100    # moving cache[index][1] to memoty

        cache[index][1] = copy.deepcopy(cache[index][0])
        cache[index][0] = [True, False, tag]
        return res_time



# returns time needed to read the data of xth adress line is in cache
def get_time_write_x(x):
    tag, index, offset = parse_address(x)
    # number of bytes we are writing (1 for matrix a, 2 for b, 4 for c)
    responce_time = 0
    if x < size_a:
        responce_time = 1
    elif x < size_a + size_b:
        responce_time = 2
    else:
        responce_time = 4
    for i in range(2):
        if cache[index][i][0] == True and cache[index][i][2] == tag:
            cache[index][i][1] = True
            # if x is in cahce,
            # we replace it with new value and mark as dirty
            return 6 + responce_time
```

```
    # otherwise we need to fetch x to cache first, and then replace it,
    # but we shouldn't add 6 clock tics to result
    res_time = get_time_read_x(x)
    get_time_write_x(x)     # marking as dirty
    return res_time + responce_time

res_clk = 0

# adding all time to access the memory/cache
for x in mem_access_stack:
    if x < size_a + size_b:
        # accessing a or b to read data
        res_clk += get_time_read_x(x)
    else:
        # accessing c to write data
        res_clk += get_time_write_x(x)

print("memory access time:", res_clk)

res_clk += 2    # initialize pa, pc
for y in range(M):
    res_clk += 3    # new loop iteration, y += 1 (add & assign)
    for x in range(N):
        res_clk += 1    # new loop iteration
        res_clk += 2    # initialize pb, s
        for k in range(K):
            res_clk += 6
            # multiplication and addition in s += pa[k] * pb[x]
            res_clk += 1    # addition in pb += N
    res_clk += 2    # addition in pa += K, pc += N

res_clk += 1 # exit function

print("total time:", res_clk)
```

Результат работы программы:

```
memory access time: 5455908
total time: 6327911
```

7 Моделирование заданной системы на Verilog

Главные составные элементы нашей схемы – модули памяти (*mem*), кэша (*cache*) и процессора (*cpu*). Они сообщаются между собой с помощью проводов адресов, команд и данных, также используются дополнительные провода сброса значений.

7.1 Шины адресов, данных и команд

Шины $C1, C2, D1$ и $D2$ используются для сообщения между разными модулями и имеют тип *inout*. Владение шиной определяется битом *control*, в каждый момент времени ровно один из двух модулей владеет шиной.

Присвоение значения происходит через регистры, по типу

```
reg control1;
reg[CTR1_BUS_SIZE - 1: 0] cmd1;
assign C1 = control1 ? cmd1 : 2'bzz;
```

при столкновении значений на проводе выигрывает более сильный сигнал, не в высокоимпедансном состоянии, то есть тот, у которого в данный момент контроль.

Адресные шины $A1$ и $A2$ всегда передают адрес сверху вниз, то есть от CPU к *cache* и от *cache* к *mem*; в их использовании не возникает столкновения сигналов разной силы.

7.2 M_DUMP, C_DUMP, RESET

В момент начала работы программы, а также в любой другой момент, когда нужно сбросить всю оперативную память, кэш, или и то и другое, используются специальные входы. При установке $C\_DUMP = 1$ на следующем такте сбрасывается все значение кэша; пи установке $M\_DUMP = 1$ все данные памяти сбрасываются и инициализируются заново, с помощью алгоритма с использованием случайных чисел (описанного в техническом задании).

7.3 Memory

Память представляет собой статический массив из 2^{19} 8-битных регистров. Они адресуются последовательными индексами от 0 до $2^{19} - 1$.

При запросе на чтение или запись нам нужно пройтись по 1-байтным блокам и при очередном такте процессора отправлять на шину $D2$ новую пару байт. И чтение, и запись происходят последовательно, в порядке увеличения абсолютного адреса. 1 такт уходит на чтение команды и адреса, 8 - на последующие чтение/запись, еще 91 мы искусственно ждем, чтобы смоделировать долгий доступ к памяти.

Этот фрагмент кода считывает данные с шины $D2$ и записывает в память; аналогичный внешний цикл используется для чтения из памяти.

```
for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000; offset < 4'b1111; offset++) begin
if (offset[0] == 0) begin
    @(posedge(CLK));
    data[{A2[ADDR2_BUS_SIZE - 1: 0], offset}] = D2[15:8];
end
if (offset[0] == 1) begin
    data[{A2[ADDR2_BUS_SIZE - 1: 0], offset}] = D2[7:0];
end
end
```

7.4 Cache

Это самая сложная часть системы, так как она взаимодействует и с памятью, и с процессором. Исходно шиной команд $C1$ владеет процессор, а шиной $C2$ — кэш. Как только от процессора поступает запрос, владение $C1$ переходит к кэшу и возвращается к процессору только после передачи выполнения всей команды. Если оказалось, что запрос попал мимо кэша, либо ему нужно обновить какие-то данные после записи, то кэш посылает соответсвующую команду в память, и владение $C2$ переходит к *mem*. То есть, в любой момент времени контроль над шинами команд может быть одним из следующих:

$$(1 : cpu, 2 : cache); (1 : cache, 2 : cache); (1 : cache, 2 : mem)$$

7.4.1 cache hit

Пусть в кэш поступил запрос на чтение некоторого участка памяти (tag, set) (предполагаем, что запросов, пересекающих границу кэш-блока в 16 байт, нет). Сначала нужно проверить, есть ли нужные нам данные в кэше: пример соответствующего кода для команды $C1_READ$8б

```
for (reg r = 0; r < CACHE_WAY; r++) begin
// check if line is valid ans has the tag that we need
if (data[CACHE_WAY * set + r][cache_line_len - 1] == 1 &&
    && data[CACHE_WAY * set + r]
        [cache_line_len - 3 : cache_line_len - CACHE_TAG_SIZE - 2] == tag) begin
```

```
// writing data to d1 bus
if (C1 == C1_READ8) begin
    @(posedge CLK)
    data1_0 = data[CACHE_WAY * set][offset +: 8];
end
```

7.4.2 cache miss

Если соответствующего участка данных нет в кэше, то его нужно в кэш поднять. Мы будем фетчить новый участок данных на первую (их двух) линию блока, соответствующего *set*. Однако для этого может потребоваться выгрузить в оперативную память какие-то старые значения из кэша.

Обозначим $line_1$, $line_2$ – две кэш-линии нашего блока.

1. $valid(line_1) == false$ – ничего делать не нужно, так как мы сразу можем загрузить значение из *mem* в $line_1$

2. $valid(line_2) == false || (valid(line_2) == true \&\& dirty(line_2 == false$ – значение второй линии либо невалидно, либо не отличается от уже записанного в оперативной памяти. Значит, ничего выгружать не придется, и достаточно будет скопировать содержимое первой линии во вторую:

$$data[CACHE\_WAY * set + 1] = data[CACHE\_WAY * set]$$

3. $valid(line_2) == false || (valid(line_2) == true \&\& dirty(line_2) == true$ Это самый медленный случай.

Сначала выгрузим вторую линию в память:

```
cmd2 = C2_WRITE_LINE;
address2[ADDR2_BUS_SIZE - 1 : 0] = {data[CACHE_WAY * set + 1][cache_line_len - 3: cache_
@(posedge CLK)
// giving control to mem
control2 = 0;

// sending data to mem
for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000; offset < 4'b1111; offset++) begin
    if (offset[0] == 0) begin
        @(posedge(CLK));
        data2_1 = data[CACHE_WAY * set + 1][CACHE_LINE_SIZE * 8 + offset -: 8];
    end
    if (offset[0] == 1) begin
        data2_0 = data[CACHE_WAY * set + 1][CACHE_LINE_SIZE * 8 + offset -: 8];
    end
end

// regaining control
control2 = 1;
```

А только потом скопируем предыдущее значение первой линии во вторую.

После этого заберем из памяти нужные нам данные на первую линию и ответим на запрос. Мы 1 такт читали команду, еще 2 – адрес, и еще 3 простояли, чтобы прождать ровно 6 тактов перед началом успешного ответа.

Если запрос в кэш не попал, то дополнительно надо простоять еще всего 1 такт.

Также кэш подсчитывает общее число попаданий и промахов при обращении к нему.

7.5 CPU

У процессора как такового нет собственной функциональности. В него лишь встроен код, эмулирующий задачу.

8 Воспроизведение задачи на Verilog

9 Сравнение полученных результатов

:(

10 Листинг кода

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

`timescale 1ns/1ps

module tb #(parameter

    ADDR1_BUS_SIZE = 15,
    DATA1_BUS_SIZE = 16,
    CTR1_BUS_SIZE = 3,

    ADDR2_BUS_SIZE = 15,
    DATA2_BUS_SIZE = 16,
    CTR2_BUS_SIZE = 2,

    _SEED = 225526
);

typedef enum reg[CTR2_BUS_SIZE - 1: 0] { C2_NOP = 2'b00, C2_RESPONSE = 2'b01, C2_READ_LI
typedef enum reg[CTR1_BUS_SIZE - 1: 0] {C1_NOP = 3'b000, C1_READ8  = 3'b001, C1_READ16 =

reg clk = 0;
always #1 clk = ~clk;

always #100 begin
 $display("[%0t] time", $time);
end

wire m_dump;
wire c_dump;
wire reset;

wire [ADDR1_BUS_SIZE - 1 : 0] a1;
wire [DATA1_BUS_SIZE - 1 : 0] d1;
wire [CTR1_BUS_SIZE - 1 : 0] c1;

wire [ADDR2_BUS_SIZE - 1 : 0] a2;
wire [DATA2_BUS_SIZE - 1 : 0] d2;
wire [CTR2_BUS_SIZE - 1 : 0] c2;

mem mem_(.CLK(clk), .M_DUMP(m_dump), .RESET(reset), .A2(a2), .D2(d2), .C2(c2));
cache cache_(.CLK(clk), .C_DUMP(c_dump), .RESET(reset), .A1(a1), .D1(d1), .C1(c1), .A2(a
cpu cpu_(.CLK(clk), .A1(a1), .D1(d1), .C1(c1));

endmodule
```

```
module cpu #(parameter
    ADDR1_BUS_SIZE = 15,  // bits
    DATA1_BUS_SIZE = 16,  // bits
    CTR1_BUS_SIZE = 3,  // bits

    CACHE_ADDR_SIZE = 19,  // bits
    CACHE_TAG_SIZE = 10,  // bits
    CACHE_SET_SIZE = 5,  // bits
    CACHE_OFFSET_SIZE = 4  // bits

) (
    input CLK,

    input [ADDR1_BUS_SIZE - 1 : 0] A1,
    inout [DATA1_BUS_SIZE - 1 : 0] D1,
    inout [CTR1_BUS_SIZE - 1 : 0] C1
);

typedef enum reg[CTR1_BUS_SIZE - 1: 0] {C1_NOP = 3'b000, C1_READ8  = 3'b001, C1_READ16 =

// bit controlling a1/d1/c1 buses
reg control1;

// command that we write to c1 bus
reg[CTR1_BUS_SIZE - 1: 0] cmd1;
assign C1 = control1 ? cmd1 : 2'bzz;

// address that we write to a1
reg [CACHE_ADDR_SIZE - 1:0] address1;

// data that we read/write from/to d1 bus
reg[7:0] data1_0, data1_1;
assign D1[7:0] = control1 ? data1_0 : 8'bzzzzzzzz;
assign D1[15:8] = control1 ? data1_1 : 8'bzzzzzzzz;

// logging & initial values
initial begin
    $monitor("[cpu] [%0t] C1 = %b, cmd1 = %b, control1 = %b", $time, C1, cmd1, control1)
    control1 = 1;
    cmd1 = C1_NOP;
end

integer M, N, K;
reg[18:0] pa, pb, pc;
integer s;
reg[7:0] res1;
reg[15:0] res2;

// run  matrix multicipation
// adding clock tics to model the system we're implementing
initial begin
    @(posedge CLK);
    M = 64;
    @(posedge CLK);
    N = 60;
    @(posedge CLK);
```

```
K = 32;
@(posedge CLK);
pa = 0;
@(posedge CLK);
pc = M * K + K * N;
@(posedge CLK);
for (integer y = 0; y < M; y++) begin
    @(posedge CLK); @(posedge CLK);
    for (integer x = 0; x < N; x++) begin
        @(posedge CLK); @(posedge CLK);
            pb = M * K;
            @(posedge CLK);
            s = 0;
            @(posedge CLK);
            for (integer k = 0; k < K; k++) begin

                @(posedge CLK); @(posedge CLK);

                // accessing pa + k, read8 to res1
                cmd1 = C1_READ8;

                @(posedge CLK);
                // giving control to cache
                control1 = 0;
                address1[15:5] = (pa + k) >> 9;
                address1[4:0] = ((pa + k) >> 4) % 32;

                @(posedge CLK);
                address1[3:0] = (pa + k) % 16;

                @(posedge CLK);
                // reading data from cache
                res1 = data1_0;

                // regaining control
                control1 = 1;

                // accessing pb + 2x, read16 to res2
                cmd1 = C1_READ16;

                @(posedge CLK);
                // giving control to cache
                control1 = 0;
                address1[15:5] = (pb + 2 * x) >> 9;
                address1[4:0] = ((pb + 2 * x) >> 4) % 32;

                @(posedge CLK);
                address1[3:0] = (pb + 2 * x) % 16;

                @(posedge CLK);
                // reading data from cache
                res1 = data1_0;

                // regaining control
                control1 = 1;
                cmd1 = C1_NOP;
```

```
                    s += res1 * res2;
                    @(posedge CLK); @(posedge CLK); @(posedge CLK); @(posedge CLK); @(po
                    pb += N;
                    @(posedge CLK);
                end

            // accessing pc + 4x, write32 from s
            cmd1 = C1_WRITE32_RESPONCE;

            @(posedge CLK);
            // giving control to cache
            control1 = 0;
            address1[15:5] = (pc + 4 * x) >> 9;
            address1[4:0] = ((pc + 4 * x) >> 4) % 32;

            @(posedge CLK);
            address1[3:0] = (pc + 4 * x) % 16;

            // writing data to cache
            @(posedge CLK);
            data1_0 = s % (2 ** 8);
            data1_1 = (s >> 8) % (2 ** 8);
            @(posedge CLK);
            data1_0 = (s >> 16) % (2 ** 8);
            data1_1 = s >> 24;

            // regaining control
            control1 = 1;
            cmd1 = C1_NOP;

        end
        pa += K;
        @(posedge CLK);
        pc += N;
        @(posedge CLK);
    end

end

endmodule

module cache #(parameter

    CACHE_LINE_COUNT = 64,
    CACHE_ADDR_SIZE = 19,   // bits
    CACHE_LINE_SIZE = 16, // bytes
    CACHE_TAG_SIZE = 10,   // bits
    CACHE_SET_SIZE = 5,   // bits
    CACHE_OFFSET_SIZE = 4,   // bits
    CACHE_WAY = 2,
    cache_line_len = 1 + 1 + CACHE_TAG_SIZE + 8 * CACHE_LINE_SIZE,

    ADDR1_BUS_SIZE = 15,   // bits
    DATA1_BUS_SIZE = 16,   // bits
    CTR1_BUS_SIZE = 3,   // bits
```

```
    ADDR2_BUS_SIZE = 15,  // bits
    DATA2_BUS_SIZE = 16,  // bits
    CTR2_BUS_SIZE = 2  // bits
) (
    input CLK,

    input C_DUMP,
    input RESET,

    input [ADDR1_BUS_SIZE - 1 : 0] A1,
    inout [DATA1_BUS_SIZE - 1 : 0] D1,
    inout [CTR1_BUS_SIZE - 1 : 0] C1,

    output [ADDR2_BUS_SIZE - 1 : 0] A2,
    inout [DATA2_BUS_SIZE - 1 : 0] D2,
    inout [CTR2_BUS_SIZE - 1 : 0] C2
);

typedef enum reg[CTR2_BUS_SIZE - 1: 0] { C2_NOP = 2'b00, C2_RESPONSE = 2'b01, C2_READ_LI
typedef enum reg[CTR1_BUS_SIZE - 1: 0] {C1_NOP = 3'b000, C1_READ8  = 3'b001, C1_READ16 =

// bit controlling a1/d1/c1 buses
reg control1;

// command that we write to c1 bus
reg[CTR1_BUS_SIZE - 1 : 0] cmd1;
assign C1 = control1 ? cmd1 : 3'bzzz;

// bit controlling a1/d1/c1 buses
reg control2;

// command that we write to c2 bus
reg [CTR2_BUS_SIZE - 1 : 0] cmd2;
assign C2 = control2 ? cmd2 : 2'bzz;

// address that we write to a2
reg [ADDR2_BUS_SIZE - 1: 0] address2;
assign A2[ADDR2_BUS_SIZE - 1 : 0] = address2;

// address that we read from a1
reg [CACHE_ADDR_SIZE - 1: 0] address1;

// parse address1 and assign parts to tag, set and offset
reg [CACHE_TAG_SIZE - 1 : 0] tag;
assign tag = address1[CACHE_ADDR_SIZE - 1 : CACHE_ADDR_SIZE - CACHE_TAG_SIZE];
reg [CACHE_SET_SIZE - 1 : 0] set;
assign set = address1[CACHE_ADDR_SIZE - CACHE_TAG_SIZE - 1: CACHE_OFFSET_SIZE];
reg [CACHE_OFFSET_SIZE - 1 : 0] offset;
assign offset = address1[CACHE_OFFSET_SIZE - 1: 0];

// actual cache data, format: valid(1 bit)-dirty(1 bit)-tag(10 bits)-data(16 bytes)
reg[cache_line_len - 1 : 0] data [CACHE_LINE_COUNT];

// data that we read/write from/to d1 bus
reg[7:0] data1_0, data1_1;
```

```
assign D1[7:0] = control1 ? data1_0 : 8'bzzzzzzzz;
assign␣D1[15:8]␣=␣control1␣?␣data1_1␣:␣8'bzzzzzzzz;

// data that we read/write from/to d2 bus
reg[7:0] data2_0, data2_1;
assign D2[7:0] = control2 ? data2_0 : 8'bzzzzzzzz;
assign␣D2[15:8]␣=␣control2␣?␣data2_1␣:␣8'bzzzzzzzz;

reg cache_hit;
reg[19:0] hit_counter;
reg[19:0] miss_counter;

// logging & initial values
initial begin
    $monitor("[cache]␣[%0t]␣C1␣=␣%b,␣C2␣=␣%b,␣cmd1␣=␣%b,␣cmd2␣=␣%b,␣control1␣=␣%b,␣contr
    hit_counter = 0;
    miss_counter = 0;
    control1 = 0;
    control2 = 1;
    cmd2 = C2_NOP;
end

// reset system
always @(posedge CLK) begin
    if (RESET) begin
        control1 = 0;
        control2 = 1;
        cmd2 = C2_NOP;
    end
    for (integer i = 0; i < CACHE_LINE_COUNT; i++) begin
        data[i] = 19'b0000000000000000000;
␣␣␣␣end
end

//␣cache␣dump
always␣@(posedge␣CLK)␣begin
␣␣␣␣if␣(C_DUMP)␣begin
␣␣␣␣␣␣␣␣for␣(integer␣i␣=␣0;␣i␣<␣CACHE_LINE_COUNT;␣i++)␣begin
␣␣␣␣␣␣␣␣␣␣␣␣data[i]␣=␣19'b0000000000000000000;
        end
    end
end

// process queries from cpu
always @(posedge CLK) begin
    if (control1 == 0) begin
        case (C1)

            C1_READ8 || C1_READ16 || C1_READ32: begin
                control1 = 1;
                cmd1 = C1_WRITE32_RESPONCE;

                @ (posedge CLK);
                // reading tag + set from a1
                address1[CACHE_ADDR_SIZE - 1: CACHE_OFFSET_SIZE] = A1[ADDR1_BUS_SIZE - 1
```

```
                    @ (posedge CLK);
                    // reading offset from a1
                    address1[CACHE_OFFSET_SIZE - 1: 0] = A1[CACHE_ADDR_SIZE - 1: 0];

                    cache_hit = 0;
                    // data[CACHE_WAY * set + r], 0 <= r < CACHE_WAY, may be the line for ad
                    for (reg r = 0; r < CACHE_WAY; r++) begin
                        // check if line is valid ans has the tag that we need
                        if (data[CACHE_WAY * set + r][cache_line_len - 1] == 1 && data[CACHE_
                            // writing data to d1 bus
                            if (C1 == C1_READ8) begin
                                @(posedge CLK)
                                data1_0 = data[CACHE_WAY * set][offset +: 8];
                            end
                            if (C1 == C1_READ16) begin
                                @(posedge CLK)
                                data1_0 = data[CACHE_WAY * set][offset +: 8];
                                data1_1 = data[CACHE_WAY * set][offset + 8 +: 8];
                            end
                            if (C1 == C1_READ32) begin
                                @(posedge CLK)
                                data1_0 = data[CACHE_WAY * set][offset +: 8];
                                data1_1 = data[CACHE_WAY * set][offset + 8 +: 8];
                                @(posedge CLK)
                                data1_0 = data[CACHE_WAY * set][offset + 16 +: 8];
                                data1_1 = data[CACHE_WAY * set][offset + 24 +: 8];
                            end
                            cache_hit = 1;
                        end
                    end

                    if (!cache_hit) begin

                        // we will fetch needed line from memory to 1st line in this cache b

                        // 1st line in this block is invalid - don't have to do anything
                        if (data[CACHE_WAY * set][cache_line_len - 1] == 0);
                        // 2nd line in this block is invalid or valid but not dirty - copy 1
                        if (data[CACHE_WAY * set + 1][cache_line_len - 1] == 0 || data[CACHE_
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣data[CACHE_WAY␣*␣set␣+␣1]␣=␣data[CACHE_WAY␣*␣set];
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣1st␣and␣2nd␣are␣valid;␣2nd␣is␣dirty␣—␣write␣2nd␣to␣mem,␣then␣copy
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(data[CACHE_WAY␣*␣set␣+␣1][cache_line_len␣-␣1]␣==␣0␣&&␣data[CACHE_

                            cmd2 = C2_WRITE_LINE;
                            address2[ADDR2_BUS_SIZE - 1 : 0] = {data[CACHE_WAY * set + 1][ca
                            @(posedge CLK)
                            // giving control to mem
                            control2 = 0;

                            // sending data to mem
                            for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000;␣offset␣<␣4'
                                if (offset[0] == 0) begin
                                    @(posedge(CLK));
                                    data2_1 = data[CACHE_WAY * set + 1][CACHE_LINE_SIZE * 8 -
                                end
```

```
                    if (offset[0] == 1) begin
                        data2_0 = data[CACHE_WAY * set + 1][CACHE_LINE_SIZE * 8 -
                    end
                end

                // regaining control
                control2 = 1;

                // now we can safely write 1st line data to 2nd
                data[CACHE_WAY * set + 1] = data[CACHE_WAY * set];

            end

            // now we are fetching needed data to 1st line of our cache block

            cmd2 = C2_READ_LINE;
            address2[ADDR2_BUS_SIZE - 1 : 0] = {tag, set};
            @ (posedge CLK);
            // giving control to mem
            control2 = 0;

            // this line becomes valid and not dirty
            data[CACHE_WAY * set][cache_line_len - 1 : cache_line_len - 2] = 2'b

        //recieving data from mem
        for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000; offset < 4'b111
            if (offset[0] == 0) begin
                @(posedge(CLK));
                data[CACHE_WAY * set][CACHE_LINE_SIZE * 8 + offset -: 8] = d
            end
            if (offset[0] == 1) begin
                data[CACHE_WAY * set][CACHE_LINE_SIZE * 8 + offset -: 8] = d
            end
        end

        //regaining control
        control2 = 1;

        //now needed data is in 1st line of cache
        //writing data to d1 bus
        if (C1 == C1_READ8) begin
            @(posedge CLK)
            data1_0 = data[CACHE_WAY * set][offset +: 8];
        end
        if (C1 == C1_READ16) begin
            @(posedge CLK)
            data1_0 = data[CACHE_WAY * set][offset +: 8];
            data1_1 = data[CACHE_WAY * set][offset + 8 +: 8];
        end
        if (C1 == C1_READ32) begin
            @(posedge CLK)
            data1_0 = data[CACHE_WAY * set][offset +: 8];
            data1_1 = data[CACHE_WAY * set][offset + 8 +: 8];
            @(posedge CLK)
            data1_0 = data[CACHE_WAY * set][offset + 16 +: 8];
            data1_1 = data[CACHE_WAY * set][offset + 24 +: 8];
```

```
                    end

                end

                if (cache_hit) begin
                    hit_counter++;
                end
                if (!cache_hit) begin
                    miss_counter++;
                end

                // giving control back to cpu
                control1 = 0;

            end

            C1_INVALIDATE_LINE: begin
                control1 = 1;
                cmd1 = C1_WRITE32_RESPONCE;

                @ (posedge CLK);
                // reading tag + set from a1
                address1[CACHE_ADDR_SIZE - 1:CACHE_OFFSET_SIZE] = A1[ADDR1_BUS_SIZE - 1

                @ (posedge CLK);
                // reading offset from a1
                address1[CACHE_OFFSET_SIZE - 1:0] = A1[CACHE_ADDR_SIZE - 1:0];

                // data[CACHE_WAY * set + r], 0 <= r < CACHE_WAY, may be the line for ad
                for (reg r = 0; r < CACHE_WAY; r++) begin
                    // check if line has the tag that we need
                    if (data[CACHE_WAY * set + r][cache_line_len - 3:cache_line_len - 
                        // set validness bit to 0
                        data[CACHE_WAY * set + r][cache_line_len - 1] = 0;
                        // ...
                    end
                end

                control1 = 0;
            end

            C1_WRITE8 || C1_WRITE16 || C1_WRITE32_RESPONCE: begin
                control1 = 1;
                cmd1 = C1_WRITE32_RESPONCE;

                @ (posedge CLK);
                // reading tag + set from a1
                address1[CACHE_ADDR_SIZE - 1:CACHE_OFFSET_SIZE] = A1[ADDR1_BUS_SIZE - 1

                @ (posedge CLK);
                // reading offset from a1
                address1[CACHE_OFFSET_SIZE - 1:0] = A1[CACHE_ADDR_SIZE - 1:0];

                cache_hit = 0;
                // data[CACHE_WAY * set + r], 0 <= r < CACHE_WAY, may be the line for ad
                for (reg r = 0; r < CACHE_WAY; r++) begin
```

```
                    // check if line is valid ans has the tag that we need
                    if (data[CACHE_WAY * set + r][cache_line_len - 1] == 1 && data[CACHE_
                        // marking line as dirty
                        data[CACHE_WAY * set + r][cache_line_len - 1] = 1;
                        // writing data to cache
                        if (C1 == C1_WRITE8) begin
                            @ (posedge CLK)
                            data[CACHE_WAY * set + r][offset +: 8] = data1_0;
                        end
                        if (C1 == C1_WRITE16) begin
                            @ (posedge CLK)
                            data[CACHE_WAY * set + r][offset +: 16] = {data1_1, data1_0}
                        end
                        if (C1 == C1_WRITE32_RESPONCE) begin
                            @ (posedge CLK)
                            data[CACHE_WAY * set + r][offset +: 16] = {data1_1, data1_0}
                            @(posedge CLK)
                            data[CACHE_WAY * set + r][offset + 16 +: 16] = {data1_1, data
                        end
                        cache_hit = 1;
                    end
                end

                if (!cache_hit) begin

                    // we will fetch needed line from memory to 1st line in this cache b

                    // 1st line in this block is invalid - don't have to do anything
                    if (data[CACHE_WAY * set][cache_line_len - 1] == 0);
                    // 2nd line in this block is invalid or valid but not dirty - copy 1
                    if (data[CACHE_WAY * set + 1][cache_line_len - 1] == 0 || data[CACHE_
                        data[CACHE_WAY * set + 1] = data[CACHE_WAY * set];
                    end
                    // 1st and 2nd are valid; 2nd is dirty - write 2nd to mem, then copy
                    if (data[CACHE_WAY * set + 1][cache_line_len - 1] == 0 && data[CACHE_

                        cmd2 = C2_WRITE_LINE;
                        address2[ADDR2_BUS_SIZE - 1 : 0] = {data[CACHE_WAY * set + 1][ca
                        @(posedge CLK)
                        // giving control to mem
                        control2 = 0;

                        // sending data to mem
                        for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000; offset < 4'
                            if (offset[0] == 0) begin
                                @(posedge(CLK));
                                data2_1 = data[CACHE_WAY * set + 1][CACHE_LINE_SIZE * 8 -
                            end
                            if (offset[0] == 1) begin
                                data2_0 = data[CACHE_WAY * set + 1][CACHE_LINE_SIZE * 8 -
                            end
                        end

                        // regaining control
                        control2 = 1;
```

```
                        // now we can safely write 1st line data to 2nd
                        data[CACHE_WAY * set + 1] = data[CACHE_WAY * set];

                    end

                    // now we are fetching needed data to 1st line of our cache block

                    cmd2 = C2_READ_LINE;
                    address2[ADDR2_BUS_SIZE - 1 : 0] = {tag, set};
                    @ (posedge CLK);
                    // giving control to mem
                    control2 = 0;

                    // this line becomes valid and not dirty
                    data[CACHE_WAY * set][cache_line_len - 1 : cache_line_len - 2] = 2'b

                //recieving data from mem
                for (reg [CACHE_OFFSET_SIZE-1:0] offset = 4'b0000; offset < 4'b111
                    if (offset[0] == 0) begin
                        @(posedge(CLK));
                        data[CACHE_WAY*set][CACHE_LINE_SIZE*8+offset-:8] = d
                    end
                    if (offset[0] == 1) begin
                        data[CACHE_WAY*set][CACHE_LINE_SIZE*8+offset-:8] = d
                    end
                end

                //regaining control
                control2 = 1;

                //now  needed data is in 1st line if out cache block
                //marking line as dirty
                data[CACHE_WAY*set][cache_line_len-1] = 1;
                //writing data to cache
                if (C1 == C1_WRITE8) begin
                    @ (posedge CLK)
                    data[CACHE_WAY*set][offset+:8] = data1_0;
                end
                if (C1 == C1_WRITE16) begin
                    @ (posedge CLK)
                    data[CACHE_WAY*set][offset+:16] = {data1_1, data1_0};
                end
                if (C1 == C1_WRITE32_RESPONCE) begin
                    @ (posedge CLK)
                    data[CACHE_WAY*set][offset+:16] = {data1_1, data1_0};
                    @(posedge CLK)
                    data[CACHE_WAY*set][offset+16+:16] = {data1_1, data1_0};
                end

            end

            if (cache_hit) begin
                hit_counter++;
            end
            if (!cache_hit) begin
                miss_counter++;
```

```
                end

                // giving control back to cpu
                control1 = 0;

            end

        endcase
    end
end

endmodule

module mem #(parameter

    MEM_SIZE = 512 * 1024,  // bytes

    CACHE_OFFSET_SIZE = 4,

    ADDR2_BUS_SIZE = 15,  // bits
    DATA2_BUS_SIZE = 16,  // bits
    CTR2_BUS_SIZE = 2,  // bits

    _SEED = 225526
)(
    input CLK,
    input M_DUMP,
    input RESET,
    input [ADDR2_BUS_SIZE - 1 : 0] A2,
    inout [DATA2_BUS_SIZE - 1 : 0] D2,
    inout [CTR2_BUS_SIZE - 1 : 0] C2
);

typedef enum reg [CTR2_BUS_SIZE - 1: 0] { C2_NOP = 2'b00, C2_RESPONSE = 2'b01, C2_READ_LI

// bit controlling a2/d2/c2 buses
reg control2;

// command that we write to c2 bus
reg [CTR2_BUS_SIZE - 1 : 0] cmd2;
assign C2 = control2 ? cmd2 : 2'bzz;

// data that we read/write from/to d2 bus
reg [7:0] data2_0, data2_1;
assign D2[7:0] = control2 ? data2_0 : 8'bzzzzzzzz;
assign D2[15:8] = control2 ? data2_1 : 8'bzzzzzzzz;

// actual memory data
reg [7:0] data [MEM_SIZE];

integer SEED = _SEED;

// logging & initial values
initial begin
    $monitor("[mem] [%0t] C2 = %b, cmd2 = %b, control2 = %b", $time, C2, cmd2, control2)
    control2 = 0;
```

```
    for (integer i = 0; i < MEM_SIZE; i++) begin
        data[i] = $random(SEED)>>16;
    end
end

// reset system
always @(posedge CLK) begin
    if (RESET) begin
        control2 = 0;
        for (integer i = 0; i < MEM_SIZE; i++) begin
            data[i] = $random(SEED)>>16;
        end
    end
end

// memory dump
always @(posedge CLK) begin
    if (M_DUMP) begin
        for (integer i = 0; i < MEM_SIZE; i++) begin
            data[i] = $random(SEED)>>16;
        end
    end
end


// process queries from cache
always @(posedge CLK) begin
    if (control2 == 0) begin
        case (C2)
            // write data of 1 cache line (16 bytes)
            // d2 bus size is 16 bit (2 bytes)
            // A2 stores adress tag + set
            C2_WRITE_LINE: begin
                control2 = 1;
                cmd2 = C2_RESPONSE;

                for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000; offset < 4'b1111; o
                    if (offset[0] == 0) begin
                        @(posedge(CLK));
                        data[{A2[ADDR2_BUS_SIZE - 1: 0], offset}] = D2[15:8];
                    end
                    if (offset[0] == 1) begin
                        data[{A2[ADDR2_BUS_SIZE - 1: 0], offset}] = D2[7:0];
                    end
                end

                // mem operation should be 100 clk ticks
                for (int i = 0; i < 100 - 9; ++i) @(posedge CLK);
                cmd2 = C2_NOP;
                control2 = 0;
                end
            C2_READ_LINE: begin
                control2 = 1;
                cmd2 = C2_RESPONSE;
                // d2 = [data2_1 data2_0]
```

```
                    for (reg [CACHE_OFFSET_SIZE - 1:0] offset = 4'b0000; offset < 4'b1111; o
                        if (offset[0] == 0) begin
                            @(posedge(CLK));
                            data2_1 = data[{A2[ADDR2_BUS_SIZE - 1: 0], offset}];
                        end
                        if (offset[0] == 1) begin
                            data2_0 = data[{A2[ADDR2_BUS_SIZE - 1: 0], offset}];
                        end
                    end

                    // mem operation should be 100 clk ticks
                    for (int i = 0; i < 100 - 9; ++i) @(posedge CLK);
                    cmd2 = C2_NOP;
                    control2 = 0;
                    end
            endcase
        end
    end

endmodule
```