

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 участок 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 address
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 address 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 address 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)

```

```

response_time = 0
if x < size_a:
    response_time = 1
elif x < size_a + size_b:
    response_time = 2
else:
    response_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 + response_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 + response_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 + response_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)
    response_time = 0
    if x < size_a:
        response_time = 1
    elif x < size_a + size_b:
        response_time = 2
    else:
        response_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 + response_time

```

```

    # otherwise we need to fetch x to cache first, and then replace it,
    # but we shouldn't add 6 clock ticks 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; при 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 пройти 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 соответствующую 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_READ86*

```
for (reg r = 0; r < CACHE_WAY; r++) begin
  // check if line is valid and 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
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.