

ЛАБОРАТОРНАЯ РАБОТА №2	М3136	2022
Моделирование схем в Verilog	БАГРИНЦЕВ МИХАИЛ АЛЕКСЕЕВИЧ	

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

Инструментарий: весь код пишется на языке Verilog, компиляция и симуляция – Icarus Verilog version 12.0. Далее в этом документе Verilog+SystemVerilog обозначается как Verilog.

Описание:

Начать написание данной лабораторной работы мне хочется с описания кэша процессора и теоретических аспектов связанных с ним. Для начала следует понять зачем процессору нужен кэш. Зачастую расстояние от процессора (CPU) до ОЗУ (memory) довольно большое, и при каждом обращении процессора к памяти, он вынужден простаивать всё время пока сигнал дойдёт до memory и вернётся. Чтобы сократить время бездействия процессора, нужно расположить хранилище данных как можно ближе. Это и есть идея кэша.

Кэш процессора — это устройство, используемое центральным процессором для уменьшения средней стоимости получения данных из основной памяти.

За долгую историю кэша он претерпел много изменений и оброс разными политиками работы. Давайте поподробнее рассмотрим аспекты работы кэша.

Для начала стоит сказать, что на логическом уровне кэш — это набор кэш-линий. Кэш-линия это последовательность байтов, лежащих рядом в ОЗУ. Когда процессор берёт данные из памяти, он берёт не по одному

байту, а сразу несколько. Сколько именно байт возьмёт кэш зависит от размера его кэш-линии.

valid	dirty	tag	data
1	1	CACHE_TAG_SIZE	CACHE_LINE_SIZE

Рисунок 1 - Устройство кэш-линии

Как вы можете заметить, в каждой кэш-линии присутствуют как минимум 2 служебных бита. В зависимости от политики вытеснения линии их может быть больше, но сейчас остановимся на рисунке 1.

- valid bit - если установлена 1, то данная кэш-линия не устарела и из неё можно отдавать данные процессору.
- dirty bit - если установлена 1, то данная кэш-линия была модифицирована, но изменения не были синхронизированы с основной памятью.
- tag - часть адреса данной кэш-линии в memory.
- data - полезная информация, то есть та, за которой процессор обращается в кэш.

Теперь разберём несколько основных политик, связанных с кэшем.

Политика чтения данных из кэша

Существует два способа обмениваться данными с процессором: Look-Aside и Look-Through. При выборе политики Look-Aside, команды процессора будут посылаться одновременно в Cache и memory Control. Они пойдут искать данную кэш-линию одновременно, и если эта линия лежит в кэше, то Cache посылает запрос memory control на остановку поиска в оперативной памяти. Вторая политика - Look-Through отличается тем, что команда процессора идёт сначала в Cache и только если данной

кэш-линии не было найдено, Cache передаёт запрос memory Control. На практике, если процессор читает данные чаще чем пишет, то эффективней использовать Look-Through. Если же у нас редко случает кэш попадание, то Look-Aside будет выгоднее, потому что к тому моменту, как мы обнаружим кэш-промах, запрос на чтение кэш-линии из memory уже будет послан. Но чаще используется Look-Through, так как отсутствие шины к memory Control из процессора, даёт возможность расположить кэш ближе к процессору и выделить дополнительные провода для передачи данных. По условию работы необходимо было реализовать Look-Through политику.

Политика записи данных в кэш

Также есть политики записи данных в кэш - Write-Through и Write-Back. Первая характерна тем, что при записи чего-либо в кэш, отправляется запрос и на изменение данных в memory (или в кэш более высокого уровня). Политика Write-Back требует хранения бита (dirty bit) для каждой кэш-линии, который будет означать, что изменения не были согласованы с memory. Запись в memory происходит только если у вытесняемой кэш-линии dirty бит равен 1. В современных кэшах применяется Write-Back, так как Write-Through медленнее работает при частых запросах на запись.

По условию работы необходимо было реализовать Write-Back политику.

Ассоциативность

Кэши бывают

1. С прямым отображением - каждая кэш-линия memory соответствует только одной кэш-линии в Cache, но каждой кэш-линии из Cache соответствует несколько кэш-линий в memory.
2. Наборно-ассоциативный кэш - кэш делится на несколько сегментов (ассоциативность = количество сегментов). Такие блоки называют

сетами (set). Теперь каждой кэш-линии в memory соответствует одна кэш-линия в каждом сете. Эффективность такого метода очевидна - мы можем хранить часто используемые кэш-линии в кэше, вытесняя менее используемые (или какими-то более интеллектуальными методами решать, какую кэш-линию заменить при вытеснении).

3. Полностью ассоциативный кэш - любая кэш-линия в memory соответствует любой кэш-линии в Cache. Главный недостаток этого типа кэша - низкая производительность при нахождении кэш-линии, так как нам необходимо перебрать все кэш-линии, чтобы понять есть ли необходимая нам в кэше.

По условию работы необходимо было реализовать наборно-ассоциативный кэш с ассоциативностью = 2.

Давайте здесь прервёмся и поговорим о том, как формируется адрес кэш-линии внутри Cache и memory и как они связаны, если у нас двухассоциативная кэш-память.

tag	set	offset
CACHE_TAG_SIZE	CACHE_SET_SIZE	CACHE_OFFSET_SIZE

Рисунок 2 - Интерпретация адреса кэшем

set - адрес кэш-линии внутри всех сетов. Сама кэш-линия в кэше хранит только tag, потому что set - её индекс, поэтому она его и так знает. Offset - номер байта с которого мы начинаем читать данные внутри кэш-линии. Offset, очевидно, нигде не хранится и появляется он только из адреса, который передаёт процессор. Так как memory тоже разбивается на сеты, то, если представить memory как массив байтов (в моей реализации это именно так), то конкатенация tag, set и offset является адресом первого

байта откуда мы начинаем читать в memory (несмотря на то, что мы не обращаемся в memory за чтением нескольких байт).

Политика вытеснения

Для начала, скажем, что политика вытеснения имеет смысл только для кэша с ассоциативностью больше 1, так как с ассоциативностью 1 вы всегда вытесняете кэш-линию в Cache, которая соответствует этой кэш-линии из memory. Основные политики - LRU(least recently used), LFU (least frequently used) и MFU (most frequently used).

1. LRU - вытесняется кэш-линия к которой дольше всего не было обращений. Нужно хранить список кэш-линий, отсортированный по последнему обращению.
2. LFU - вытесняется кэш-линия к которой реже всего обращаются. Нужно хранить счётчик обращений для каждой кэш-линии.
3. MFU - вытесняется кэш-линия к которой чаще всего обращаются (если мы считаем, что данные повторно использоваться не будут).

По условию работы необходимо было реализовать политику вытеснения LRU.

Существуют ещё политики того какие кэш-линии мы ждём получаем из memory. В этой работе всё просто - memory отправляет ту кэш-линию, которая сейчас необходима кэшу. Но Cache мог попросить, например, не только эту кэш-линию, но и несколько соседних. Подобные политики применяются в настоящих кэшах.

Также стоит сказать, что существует 2 способа передавать данные по шинам данных - little endian и big endian. В little endian мы передаём сначала младшие байты, а потом старшие (по сути в обратном порядке). В big endian мы передаём сначала старшие байты, потом младшие - как при обычном чтении числа - справа налево.

По условию работы необходимо было реализовать порядок little endian.

В нашей работе необходимо было реализовать кэш L1, хотя обычно кэш процессора состоит из кэшей разных уровней и чем больше уровень кэша, тем больше кэш-линий он хранит и тем дольше процессор получает от него ответ. Сейчас большинство кэшей процессоров имеют 3 уровня кэша, где первый уровень представляет из себя Гарвардскую архитектуру, а второй и третий - архитектуру фон Неймана. Для каждого ядра процессора есть свой кэш L1 и L2. Кэш третьего уровня обычно общий. На рисунке 3 изображена структура обычного кэша процессора.

eDRAM Based Cache

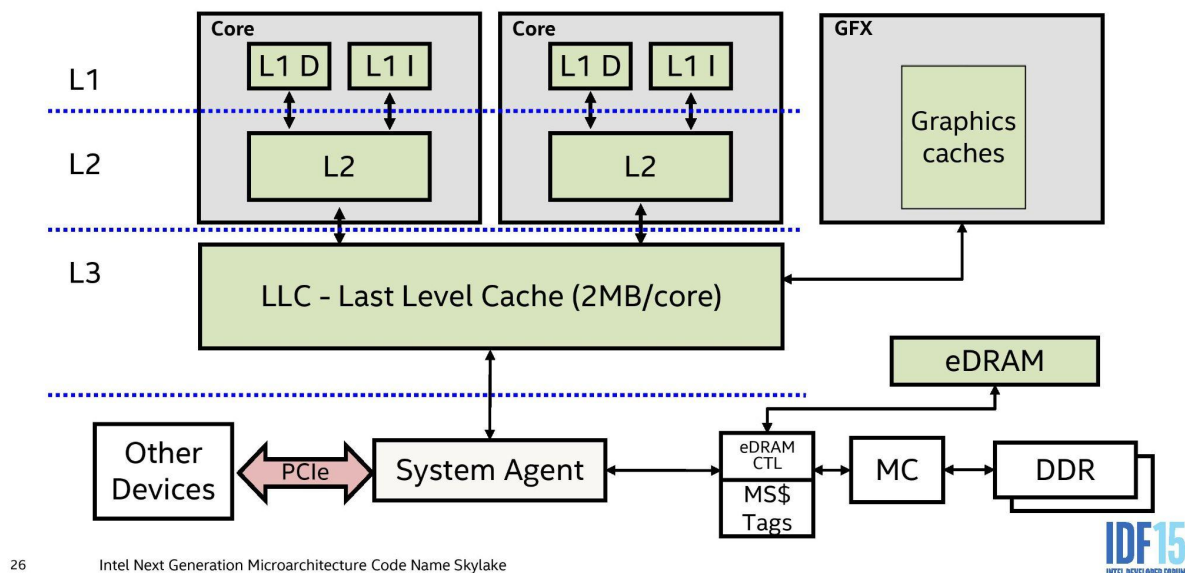


Рисунок 3 – Устройство кэша процессора

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. Как только в описании встречается слово такт, необходимо пояснить, что мы считаем одним тактом. **Тактом** будем называть минимальный период графика синхросигнала. Например, промежуток времени между двумя переходами синхросигнала из 0 в 1.

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

Все локальные переменные лежат в регистрах процессора.

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

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

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

Время отклика – расстояние в тактах от первого такта команды до первого такта ответа.

Протокол обмена данными по шине

Команды и ответы на них передаются по шинам за несколько тактов подряд. Но между командой и ответом может быть произвольное кол-во тактов бездействия.

По шине A1 адрес передаётся за 2 такта: в первый такт tag+set, во второй - offset. По шине A2 передаются адреса без части offset за 1 такт.

По шинам D (D1 и D2) в каждый такт передаётся по 16 бит данных, начиная с младших, little endian.

На линиях команд C (C1 и C2) значение держится всё время передачи команды или ответа.

В начальный момент времени (или после Reset) шиной 1 владеет CPU, а шиной 2 - Cache. После подачи команды и до окончания отправки ответа владение шиной переходит к Cache и MemCTR соответственно. Владение шиной означает, какое устройство задает логические уровни на проводах шины.

Вариант: 1.

Параметры для 1-го варианта

Кэш (продолжение)		
Ассоциативность	2 – CACHE_WAY	
Размер тега адреса	10 бит – CACHE_TAG_SIZE	
Размер кэш-линии	16 байта – CACHE_LINE_SIZE	Размер полезных данных.
Кол-во кэш-линий	64 – CACHE_LINE_COUNT	
Память		
Размер памяти	512 Кбайт – MEM_SIZE	

Вычисление недостающих параметров системы.

Шина	Обозначение	Размерность
A1, A2	ADDR1_BUS_SIZE, ADDR2_BUS_SIZE	15 бит
D1, D2	DATA1_BUS_SIZE, DATA2_BUS_SIZE	16 бит
C1	CTR1_BUS_SIZE	3 бита
C2	CTR2_BUS_SIZE	2 бита

Каким образом были вычислены размерности шин? A1 должна за первый такт передавать тег и номер кэш-линии в кэше, за второй номер байта, с которого мы читаем/пишем в кэш-линии. Из соображений оптимизации скорости передачи данных между процессором и кэшем мы хотим минимизировать ширину шины между ними. Обозначим количество бит, из которых состоит тег, номер кэш-линии в кэше и сдвиг внутри кэш-линии через tag, set и offset соответственно. Тогда A1 должна быть $\max(\text{tag}+\text{set}, \text{offset})$. В конфигурации моего варианта:

$$\text{set} = \log_2(64/2) \Rightarrow \text{tag}+\text{set} = 15 \text{ бит},$$

$$\text{offset} = \log_2(\text{CACHE_LINE_SIZE}) = \log_2(16) = 4.$$

Тогда ADDR1_BUS_SIZE = ADDR2_BUS_SIZE = 15.

Размерность C1 и C2 это логарифм от количества команд, которые могут быть переданы по данной шине.

$$\text{CTR1_BUS_SIZE} = \log_2(8) = 3 \text{ бита}.$$

$$\text{CTR2_BUS_SIZE} = \log_2(4) = 2 \text{ бита}.$$

Аналитическое решение задачи

В качестве аналитического решения я приведу программу на языке программирования Java, эмулирующую работу системы. В силу ненадобности, некоторая функциональность исходной системы была либо урезана, либо вовсе удалена. Чтобы считать такты, кэш попадания и кэш-промахи нужно знать только tag и set. То есть ни offset, ни данные нам при решении этой задачи не требуются. По этой причине программа, представляющая решение на Java, работает только с тегом и индексом кэш-линии внутри сета. Раз у нас нет данных, то memory в принципе не имеет смысла, нам достаточно прибавлять такты к счётчику, но саму память реализовывать не обязательно.

Решение на Java состоит из двух классов - CPU и Cache. Для начала посмотрим на CPU.

```
final double[] tacts = new double[1];
final Cache cache = new Cache(tacts);

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

final int aStart = 0;
final int bStart = aStart + M * K;
final int cStart = bStart + 2 * K * N;

int pa = aStart;
int pb = bStart;
int pc = cStart;
int cacheHits = 0;
int cacheMisses = 0;
```

Массив tacts из одной переменной типа double нужен для того, чтобы передавать этот массив другим классам (кэшу) и пользоваться тем, что мы передаём копию ссылки на массив, то есть можем изменять элемент массива за пределами CPU. Тип double нужен для увеличения счётчика на полтакта (подробности в пункте про равенство тактов в реализации на Verilog и на Java). Так как в Java нет указателей из C, то придётся их эмулировать. Для этого были заведены указатели pa, pb, pc по логике указателей из C. Так как это указатели для чисел, выражающихся разным числом байт, шаги в этих указателях отличаются. Шаг в pa равен 1, шаг в pb равен 2, шаг в pc равен 4, поэтому на C прибавление числа N к указателю pb, на Java будет выглядеть как `pb += 2 * N`.

Теперь, давайте рассмотрим код, выполняющий основную задачу.

Для того чтобы понять увеличение счётчика тактов, вам нужно помнить код написанный в условии задачи, а именно код на страницах 6-7 в моём

отчёте. Первый тактом мы начинаем запускаем счётчик. Далее 3 такта тратим на инициализацию двумерных массивов a, b и c и ещё 2 такта на инициализацию указателей pa и pc. В сумме получаем 5 тактов, которые мы и прибавляем в следующей строке. Последующие прибавления тактов в первой строчке после начала - 1 такт за переход на новую итерацию цикла. Остальные прибавления тактов описаны комментариями в коде.

```
double tactsBefore;
tacts[0] = 1;
tacts[0] += 5;
for (int y = 0; y < M; y++) {
    tacts[0] += 1; // new iteration
    for (int x = 0; x < N; x++) {
        tacts[0] += 1; // new iteration
        pb = bStart; tacts[0] += 1;
        tacts[0] += 1; // initialization s
        for (int k = 0; k < K; k++) {
            tacts[0] += 1; // new iteration
            tactsBefore = tacts[0];
            cache.readByte(pa + k);
            if (tacts[0] - tactsBefore > 100) {
                cacheMisses += 1;
            } else {
                cacheHits += 1;
            }
            tactsBefore = tacts[0];
            cache.readTwoBytes(pb + 2 * x);
            if (tacts[0] - tactsBefore > 100) {
                cacheMisses += 1;
            } else {
                cacheHits += 1;
            }
            pb += 2 * N;
            tacts[0] += 2; // two times sum
            tacts[0] += 5; // multiplication
        }
    }
}
```

```

    tactsBefore = tacts[0];

    cache.write(pc + 4 * x);
    if (tacts[0] - tactsBefore > 100) {
        cacheMisses += 1;
    } else {
        cacheHits += 1;
    }
}
pa += K;
pc += 4 * N;
tacts[0] += 7; // two times sum and multiplication
}
tacts[0] += 1; // exit of the function
System.out.println("Cache hits: " + cacheHits);
System.out.println("Cache requests: " + (cacheHits +
cacheMisses));
System.out.println("Percent of cache hits: " + (((double)
cacheHits) / (cacheHits + cacheMisses)) * 100 + "%");
System.out.println("Number of tacts: " + (int) tacts[0]);

```

Как вы можете видеть, я считаю за кэш-промах выполнение запроса к кэшу длиной более чем 100 тактов (время отклика памяти - 100 тактов). Такой способ проще всего в реализации и сохраняет переменные cacheHits и cacheMisses внутри класса CPU.

У кэша написаны методы readByte, readTwoBytes, readFourBytes и write. Причина того, что метод write только один в том, что время работы write не зависит от количества байт, которые мы ему передаём, так как передача адреса уже занимает 2 такта. На самом деле, readByte и readTwoBytes тоже не отличаются по времени, но readFourBytes уже будет работать на 1 такт дольше каждого из них.

Давайте посмотрим на конструктор кэша, чтобы лучше понимать как устроены его поля и методы.

```
public Cache(final double[] tacts) {
    this.tacts = tacts;
    this.tags = new int[CACHE_SETS_COUNT][CACHE_WAY];
    this.valid = new
boolean[CACHE_SETS_COUNT][CACHE_WAY];
    this.dirty = new
boolean[CACHE_SETS_COUNT][CACHE_WAY];
    this.LRU = new int[CACHE_SETS_COUNT];

    for (int i = 0; i < CACHE_SETS_COUNT; i++) {
        for (int j = 0; j < CACHE_WAY; j++) {
            tags[i][j] = 0;
            valid[i][j] = false;
            dirty[i][j] = false;
            LRU[i] = 0;
        }
    }
}
```

Как можно видеть, чтобы обратиться к какому-то свойству кэш-линии нужно указать её индекс в сете как индекс первого массива и номер сета как индекс второго массива. Только у LRU логика немного отличается - LRU по индексу кэш-линии в сете выдаёт индекс сета в котором лежит последняя используемая кэш-линия с таким индексом.

Теперь, посмотрим как кэш обрабатывает запросы на чтение. Например, метод `readTwoBytes` просто вызывает метод `readByte`, так как время работы у них одинаковое.

```
public void readTwoBytes(final int address) {
    readByte(address);
}
```

Пришло время взглянуть на метод `readByte`. За два такта мы получаем адрес, за два отправляем ответ (нужно следующим тактом обнулить значения на проводах) - логично. Получение тега и индекса сета из адреса через побитовые сдвиги тоже все понимают. Дальше строчка с прибавлением двух тактов требует пояснения. По условию задания время отклика при кэш-промахе равно четырём тактам, при кэш попадании - шести тактам, методы `readFromMemory` и `uploadDataToMemory` начинают работать в тот же такт, когда мы их вызываем. Значит эти два прибавляемых такта нужны, чтобы начать отправлять запрос к памяти через 4 такта в случае кэш-промаха. Если посмотреть в каком из `if`-ов мы окажемся в случае кэш попадания, то посчитав сумму $2 + 2 + 2$, мы получим заветные 6 тактов на время отклика.

```
public void readByte(final int address) {
    tacts[0] += 2; // receive data
    final int tag = address >>> (CACHE_SET_SIZE +
    CACHE_OFFSET_SIZE);
    final int set = (address << (32 - CACHE_SET_SIZE -
    CACHE_OFFSET_SIZE)) >>> (32 - CACHE_TAG_SIZE +
    CACHE_SET_SIZE);
    final int index;
    tacts[0] += 2; // wait
    if (tags[set][0] == tag || tags[set][1] == tag) {
        index = (tags[set][0] == tag) ? 0 : 1;
        if (valid[set][index]) {
            tacts[0] += 2; // cache hit
        } else {
            readFromMemory(tag, set, index);
        }
    } else if (!valid[set][0] || !valid[set][1]) {
        index = valid[set][0] ? 1 : 0;
        readFromMemory(tag, set, index);
    } else {
```

```

        index = (LRU[set] == 1) ? 0 : 1;
        if (dirty[set][index] && valid[set][index]) {
            uploadDataToMemory(set, index);
        }
        readFromMemory(tag, set, index);
    }
    tacts[0] += 2; // send response
    LRU[set] = index;
    tags[set][index] = tag;
    valid[set][index] = true;
    return;
}

```

Как мы ищем кэш-линию? Сначала сравниваем теги, если ни один тег не совпадает с переданным, значит нужно будет доставать искомую кэш-линию из memory. Теперь вопрос в том какую линию вытеснить. Сначала ищем не валидную кэш-линию в сетах по заданному индексу. Если все линии валидные, то применяем протокол вытеснения кэш-линий - LRU. Но тут нужно задуматься - мы вытесняем валидную линию, а не может ли так оказаться, что она не сохранена в memory? Да, может. Поэтому мы проверяем dirty бит и сначала записываем её в memory в случае, если она dirty, а только потом читаем в эту кэш-линию новую.

Если коду метод write не отличается, он скорее для смыслового разграничения, поэтому его мы опустим. Предлагаю перейти к методам работы с памятью, а именно uploadDataToMemory и readFrommemory.

```

private void uploadDataToMemory(final int set, final int
index) {
    tacts[0] += 1;
    tacts[0] += 8;
    tacts[0] += 100 - 8;
    tacts[0] += 2;
    dirty[set][index] = false;
}

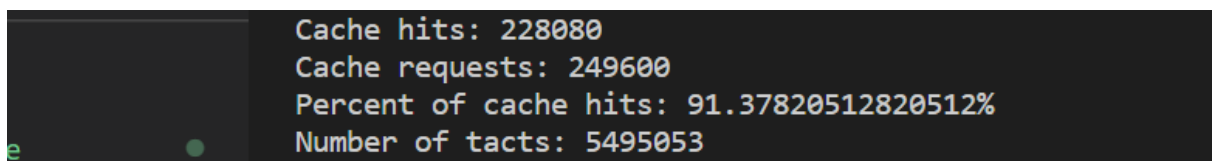
```


Первый такт на принятие данных. Восемь тактов, так как мы выгружаем всю кэш-линию размера 16 байт по 16-ти битному каналу. Далее мы ждём время отклика и за 2 такта посылаем ответ (опять-таки один такт, чтобы вернуть отдать управление проводами, после отправки C2_RESPONSE).

```
private void readFromMemory(final int tag, final int set,
final int index) {
    tacts[0] += 1;
    tacts[0] += 100;
    tacts[0] += 8;
    tacts[0] += 1;
    tags[set][index] = tag;
    valid[set][index] = true;
    dirty[set][index] = false;
    LRU[set] = index;
}
```

В методе readFromMemory мы получаем сигнал за 1 такт, потом 100 тактов ждём время отклика и сразу после этого начинаем загружать кэш-линию на шину. В конце оставляем один такт, чтобы передать управление шиной кэшу.

Пришло время продемонстрировать вывод программы:

A screenshot of a terminal window with a dark background. It displays four lines of text in a light green monospaced font. The first line is 'Cache hits: 228080', the second is 'Cache requests: 249600', the third is 'Percent of cache hits: 91.37820512820512%', and the fourth is 'Number of tacts: 5495053'. On the left side of the terminal, there is a small green dot and a green cursor character 'e' on the line containing 'Number of tacts: 5495053'.

```
Cache hits: 228080
Cache requests: 249600
Percent of cache hits: 91.37820512820512%
Number of tacts: 5495053
```

Рисунок 4 - Результат работы программы на Java

Моделирование заданной системы на Verilog

Вся система состоит из 4 модулей:

- cache - кэш, имеет inout провода D1, C1, D2, C2, input провода A1, clk - для синхронизации, reset - для сброса кэша, C_DUMP - для вывода данных кэша в консоль и output провод A2.
- CPU - процессор, имеет inout провода D1, C1, input провод clk и output провод A1.
- memory - основная память, имеет inout провода D2, C2, input провода A2, clk, reset - для возврата memory в исходное состояние и M_DUMP - для вывода данных памяти в консоль.
- testbench- инициализирующий остальные модули модуль. Он создаёт все провода и подключает их между модулями. Ниже приведён код как инициализации всех модулей в модуле testbench.

```
wire[ADDR1_BUS_SIZE-1:0] A1;  
wire[CTR1_BUS_SIZE-1:0] C1;  
wire[DATA1_BUS_SIZE-1:0] D1;  
wire[ADDR2_BUS_SIZE-1:0] A2;  
wire[CTR2_BUS_SIZE-1:0] C2;  
wire[DATA2_BUS_SIZE-1:0] D2;  
bit clk;  
bit reset;  
bit C_DUMP;  
bit M_DUMP;
```

```
CPU cpu (A1, D1, C1, clk);  
Cache cache (A1, D1, C1, A2, D2, C2, clk, reset, C_DUMP);  
memory memory (A2, D2, C2, clk, reset, M_DUMP);
```

Смена синхросигнала у меня происходит через одну задержку в Verilog, хотя это нигде и не используется.

```
always #1 clk = ~clk;
```

Опишем как работает система в целом и почему в ней не возникает race condition. Сначала объясним, что такое race condition. Race condition (или состояние гонки) - неправильная работа многопоточного приложения или аппаратной схемы, в которой мы опираемся на предположение о том, что какие-то действия происходят одновременно или в определённой последовательности, но в реальности действия могут происходить в разное время и в разной последовательности. Для того чтобы не допустить состояние гонки в своей системе, я задал такой инвариант: записываем в шину по положительному фронту тактовой частоты, а читаем данные с шин по отрицательному фронту тактовой частоты. Тогда данные по всем шинам успеют дойти до того, как мы их начнём читать. Это объясняет причину того, что все действия в CPU происходят в always блоке с чувствительностью к положительному фронту clk, а в cache и memory действия происходят в always блоке с чувствительностью к отрицательному фронту clk. Ведь cache может только отвечать CPU (конечно cache может посылать запросы в memory, но для этого cache ждёт полтакта до положительного фронта и начинает посылать команды по положительному фронту), как и memory может только отвечать cache. У нас есть ещё один инвариант: вызов любой отправляющей данные или запросы task-и осуществляется по положительному фронту синхросигнала, и во время завершения task-и у нас должен быть положительный фронт. Глобальная идея такая: хочешь записать - пиши по положительному фронту, хочешь прочитать - читай по отрицательному фронту. Для каждого модуля доступны две task-и, связанные с clk:

```
task next_clk;  
    begin
```

```

        if (clk == 0) wait(clk != 0);
        else if (clk == 1) wait(clk != 1);
    end
endtask

task next_tact;
    begin
        next_clk();
        next_clk();
    end
endtask

```

Task next_clk просто ждёт смены синхросигнала, то есть бездействует полтакта. Task next_tact уже бездействует весь такт и используется в основном для передачи и получения данных, которые нельзя передать за один такт.

Давайте перейдём к не менее интересному моменту реализации, а именно владение шиной. Для примера рассмотрим код в модуле cache. Для каждой шины, в которую мы можем писать, мы создаём буферы типа logic и через непрерывное присваивание (assign) мы пишем значения буфера с силой strong, если в буфере лежит ненулевое значение, и с силой pull, если в буфере лежит 0. Таким образом мы понимаем как модули с inout шинами будут договариваться - тот, кто владеет шиной, может писать в неё ненулевые значения, а тот, кто не владеет, пишет только нули.

```

logic[ADDR2_BUS_SIZE-1:0] A2_buffer;
logic[DATA1_BUS_SIZE-1:0] D1_buffer;
logic[CTR1_BUS_SIZE-1:0] C1_buffer;
logic[DATA2_BUS_SIZE-1:0] D2_buffer;
logic[CTR2_BUS_SIZE-1:0] C2_buffer;

assign (strong1, pull0) A2 = A2_buffer;
assign (strong1, pull0) D1 = D1_buffer;
assign (strong1, pull0) D2 = D2_buffer;

```

```
assign (strong1, pull0) C1 = C1_buffer;  
assign (strong1, pull0) C2 = C2_buffer;
```

CPU

Теперь мы посмотрим на функционал процессора. У него есть task-и для чтения, записи и инвалидации кэш-линии - всё для удобства пользователя.

Посмотрим как устроена функция (далее task будем называть функцией, несмотря на то, что в Verilog функция это несколько отличающееся от task материя) read16, хотя в процессоре определены и другие функции: read8, read32, write8, write16, write32, invalidate_line.

```
task read16(input [CACHE_TAG_SIZE-1:0] tag,  
[CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0] offset,  
output [16:0] data);  
    begin  
        C1_buffer = 2; A1_buffer = {tag, set};  
        next_tact();  
        A1_buffer = offset;  
        next_tact();  
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;  
        next_clk();  
        while (C1 != 7) begin  
            next_tact();  
        end  
        data = {D1[7:0], D1[15:8]}; C1_buffer = 0; D1_buffer  
= 0; A1_buffer = 0;  
        next_tact(); next_clk();  
    end  
endtask
```

Помните, про инвариант, что все task-и начинают выполняться и заканчивают выполнение на положительном фронте clk? В начале функции мы на положительном фронте, а положительный фронт - это фронт, по которому мы начинаем передавать данные по шинам. Мы передаём

команду чтения по C1 и адрес, получившийся конкатенацией тега и сета, по шине A1. Потом мы ждём такт, после чего мы снова на положительном фронте, но мы знаем, что тот, кому мы передавали данные уже их получил. Поэтому можно отправлять другую часть данных. По нашему протоколу, дальше мы передаём offset по шине адреса. Выждав ещё один такт, мы отправляем нули на все шины, так как мы закончили передавать информацию и нам нужно передать шину кэш. Так как сила нуля меньше, чем любого другого сигнала, кэш без проблем перезапишет ноль. Интересно то, что после записи нулей, мы выжидаем только полтакта (next_clk). Таким образом, мы перешли в режим работы по отрицательному фронту - фронт, который используется для получения данных. Кэш соответственно перейдёт на положительный фронт и отправит ответ по нему. После перехода в отрицательный фронт, мы ждём команды 7 по линии C1, что означает C1_RESPONSE. Как только мы её получаем, начинаем читать данные с шины D1 в формате Little Endian. Потом, мы ждём ещё один такт, чтобы значения на проводах обнулились (их обнуляет кэш) и полтакта, чтобы вернуться в режим работы по положительному фронту.

Все остальные функции в CPU работают максимально похоже. Даже task invalidate_line имеет такую же структуру, только она не пердаёт offset, а в остальном одно и то же.

Единственное, что можно посмотреть в CPU, это собственно вызов функций. У функций write первым аргументом передаётся то, что мы хотим записать, а остальными аргументами адрес: tag, set, offset передаются через запятую. У функций read первыми тремя аргументами передаётся адрес, а четвёртым аргументом нужно передать буфер, в который будут считываться данные.

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

```

write32('h12345678, 'b1010101010, 'b00101, 5);
write16('h1389, 'b1010101010, 'b00101, 1);
write8('h09, 'b1010111110, 'b01000, 15);
read32('b1010101010, 'b00101, 5, data32);
read16('b1010101010, 'b00101, 1, data16);
read8('b1010111110, 'b01000, 'b1111, data8);
$display("CPU: t=%0t, data32=%h", $time, data32);
$display("CPU: t=%0t, data16=%h", $time, data16);
$display("CPU: t=%0t, data8=%h", $time, data8);
$finish();
end

```

Вывод работы программы:

```

VCD info: dumpfile dump.vcd opened for output.
CPU: t=531, data32=12345678
CPU: t=531, data16=1389
CPU: t=531, data8=09
./CPU.sv:311: $finish called at 531 (1s)

```

Рисунок 5 - Результат примера вызова функций процессора

Cache

Если описать кэш в двух словах, то он ждёт сигналы от CPU по отрицательному фронту и вызывает разные функции работы с memory, если это необходимо.

```

always @(negedge clk) begin
    if (C1 == 4) begin
    end else if (1 <= C1 && C1 <= 3) begin
    end else if (5 <= C1 && C1 <= 7) begin
    end
end
end

```

Давайте посмотрим, что происходит внутри if, в который мы попадаем если процессор отправляет команду записи данных в кэш. Первым делом, мы считываем данные через конкатенацию (как же это удобно) и сохраняем копию команды, чтобы в конце понять сколько

данных нам нужно записать. Данные с шины D1 записываются в data_buffer_LE - буфер на 32 бита, хранящий данные в порядке Little Endian (хоть это и протокол передачи данных, а не хранения). После ожидания одного такта, если нам передают 32 бита, то считываем остальные 2 байта. В любом случае, считываем offset с шины адреса. Далее нам придётся отправлять данные, команды куда-то, поэтому заранее переходим в режим работы по положительному фронту. Потом ждём 2 такта из-за установленного условием задачи временем отклика. После ожидания вы можете видеть уже знакомую систему ветвлений - знакомую из Java. Поэтому опустим её повторное описание.

```
end else if (5 <= C1 && C1 <= 7) begin
    {tag, set} = A1; C1_copy = C1;
    data_buffer_LE[2*DATA1_BUS_SIZE-1:DATA1_BUS_SIZE] = D1;
    next_tact();
    if (C1 == 7) data_buffer_LE[DATA1_BUS_SIZE-1:0] =
D1;
    offset = A1[0+:CACHE_OFFSET_SIZE];
    next_clk();
    for (int i = 0; i < 2; i++) next_tact(); // waiting
    if (tags[set][0] == tag || tags[set][1] == tag)
begin
    index = (tags[set][0] == tag) ? 0 : 1;
    if (valid[set][index] == 1) begin
        for (int i = 0; i < 2; i++) next_tact();
    end else begin
        read_from_memory(tag, set, index);
    end
    end else if (valid[set][0] == 0 || valid[set][1] ==
0) begin
    index = (valid[set][0] == 0) ? 0 : 1;
    read_from_memory(tag, set, index);
    end else begin
    index = (LRU[set] == 1) ? 0 : 1;
```



```

        if (dirty[set][index] == 1 && valid[set][index] ==
1)
            upload_data_to_memory(set, index);
        read_from_memory(tag, set, index);
    end

```

Посмотрим, что происходит после того, как мы удостоверились, что в данный момент у нас лежит валидная кэш-линия с нужным нам тегом в сете с индексом `index`. Теперь нужно обновить данные об этой кэш-линии. Во-первых, теперь LRU должен указывать на индекс этой кэш-линии в этом сете. Теперь, мы точно знаем, что тег в сете `set` по индексу `index` совпадает с тегом, который мы получили от процессора, значит можно показать это явно. Также мы точно можем сказать, что наша кэш-линия валидная и грязная (`dirty bit = 1`), потому что с невалидными линиями мы не работаем, а грязной она стала, потому что мы перезаписали её часть (точнее перезапишем чуть ниже). Когда мы дошли до `case` блока, стало окончательно понятно, зачем нам нужны были буфер данных (`D1`) и копия команды процессора (`C1`).

```

LRU[set] = index; tags[set][index] = tag;
valid[set][index] = 1; dirty[set][index] = 1;
    case (C1_copy)
        3'b101: begin
            data[set][index][offset] =
data_buffer_LE[23:16];
        end
        3'b110: begin
            data[set][index][offset] =
data_buffer_LE[23:16];
            data[set][index][offset + 1] =
data_buffer_LE[31:24];
        end
    end

```

```

    3'b111: begin
        for (int i = 0; i < 4; i++)
            data[set][index][offset + i] =
data_buffer_LE[i*8+:8];
        end
    endcase
    C1_copy = 0; C1_buffer = 3'b111;
    next_tact();
    C1_buffer = 0;
end

```

После записи данных в кэш, не забываем отправить C1_RESPONSE, выждать такт и выставить 0 на шину команды C1.

Функция чтения работает аналогично, интересней посмотреть на функции общения с memory. Так как функции общения с памятью также очень похожи (речь про read_from_memory и uplod_data_to_memory) покажу только одну из них, а именно read_from_memory.

Функция, опираясь на соглашение, что вызывая её мы находимся на положительном фронте синхронизации, сразу начинает писать в буферы, из которых подаются соответствующие значения на соответствующие провода. Ждём такт и обнуляем значения на проводах, передавая контроль над проводами памяти. Потом ждём пока не появится сигнал C2_RESPONSE.

```

task read_from_memory(input [CACHE_TAG_SIZE-1:0] tag,
[CACHE_SET_SIZE-1:0] set, idx);
    begin
        C2_buffer = 'b10; A2_buffer = {tag, set}; D2_buffer
= 0;
        next_tact();
        C2_buffer = 0; A2_buffer = 0; D2_buffer = 0;
        tags[set][idx] = tag; valid[set][idx] = 1;
        dirty[set][idx] = 0; LRU[set] = idx;
        next_clk();
        while (C2 != 1) begin

```

```

        next_tact();
    end
    for (int i = CACHE_LINE_SIZE - 2; i > -1; i -= 2)
begin
        {data[set][idx][i + 1], data[set][idx][i]} =
D2;
        next_tact();
    end
    next_clk();
end
endtask

```

Когда сигнал C2_RESPONSE появляется, мы начинаем заполнять кэш-линию в кэше данным по 2 байта каждый такт. В конце мы выжидаем полтакта, чтобы вернуться в posedge clk состояние.

Конечно, в кэше есть ещё несколько функций, например, reset_cache и dump_to_terminal, но они тривиальны, поэтому скажу о них не много. Функция reset_cache присваивает всем внутренним переменным значение 0, инвалидирует все кэш-линии и присваивает нули на все позиции в dirty, tags, LRU и data. Функция dump_to_terminal пишет все значения кэш-линий в терминал с помощью встроенных функций write и display.

```

$write("Cache_set=%b, idx=%b, CACHE_LINE: valid(b)=%b,
dirty(b)=%b, LRU(b)=%b, tag(h)=%h, data(h)=",
    i[0+:CACHE_SET_SIZE], j[0+:CACHE_WAY_SIZE],
valid[i][j], dirty[i][j], LRU[i], tags[i][j]
);
for (int k = 0; k < CACHE_LINE_SIZE; k++) begin
    $write("%h", data[i][j][k]);
end
$display();

```

memory

В общих чертах memory похожа на Cache:

```

always @(negedge clk) begin
    if (C2 == 2) begin
    end else if (C2 == 3) begin
    end
end
end

```

Но у memory другой reset. Если в кэше мы просто всем переменным и байтам ставили значение 0, то здесь мы используем функцию \$random():

```

task reset_memory;
begin
    cache_line_addr = 0;
    D2_buffer = 0; C2_buffer = 0;
    for (int i = 0; i < MEM_CACHE_LINE_COUNT; i++)
begin
    for (int j = 0; j < CACHE_LINE_SIZE; j++) begin
        data[i][j] = $random(SEED)>>16;
    end
end
end
endtask

```

Такая инициализация требовалась в условии задачи. Data в memory отличается от data в cache. В memory data это просто массив из кэш-линий.

Так как у memory всего 2 ветвления, то разбор обоих не займёт много времени.

Начнём с разбора кода, который обрабатывает запросы на чтение из memory. Записываем данные во внутренние регистры, потом ждём 100 тактов - время отклика memory. Потом нужно переключиться на режим работы по положительному фронту, чтобы начать отправлять данные - ждём полтакта. После этого начинаем грузить данные в буферы.

```

if (C2 == 2) begin // C2_READ

```

```

    cache_line_addr = A2;
    for (int i = 0; i < 100; i++) next_tact();
    next_clk();
    D2_buffer = {data[cache_line_addr][CACHE_LINE_SIZE -
1], data[cache_line_addr][CACHE_LINE_SIZE - 2]};
    C2_buffer = 1;
    for (int i = CACHE_LINE_SIZE - 4; i > -1; i -= 2)
begin
    next_tact();
    D2_buffer = {data[cache_line_addr][i+1],
data[cache_line_addr][i]};
    end
    next_tact();
    C2_buffer = 0; D2_buffer = 0;
end

```

После выгрузки данных, ждём ещё один такт, чтобы обнулить значения на проводах.

Теперь к обработке запросов на запись. Сразу считываем адрес и начинаем записывать значения на проводе D2 в data в порядке Little Endian. Так делаем на протяжении 8 тактов (размер кэш-линии 16 байт). Потом ожидаем оставшееся время отклика. Дальше нам нужно отправить ответ, что данные получены, поэтому переключаемся на положительный фронт и отправляем C2_RESPONSE. Ещё через 1 такт присваиваем нули.

```

end else if (C2 == 3) begin // C2_WRITE
    cache_line_addr = A2;
    for (int i = CACHE_LINE_SIZE - 2; i > -1; i -= 2)
begin
    {data[cache_line_addr][i+1],
data[cache_line_addr][i]} = D2;
    next_tact();
    end
    for (int i = 0; i < 100 - (CACHE_LINE_SIZE / 2);
i++) next_tact();
    next_clk();

```

```

C2_buffer = 1;
next_tact();
C2_buffer = 0; D2_buffer = 0;
end

```

Единственная функция memory, которая осталась без внимания - M_DUMP, но она устроена аналогично C_DUMP.

Теперь покажем на временной диаграмме, что мы выполняем условия времени отклика:

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

Давайте, ещё раз запустим код, результат работы которого изображён на рисунке 5. Первое обращение на чтение будет кэш-попаданием. Посмотрим на этот запрос в dump файле.



Рисунок 6 - Временная диаграмма в случае кэш попадания

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

Пусть процессор отправит команду на запись самой первой. Тогда все кэш-линии в кэше находятся в невалидном состоянии и у нас обязательно случится кэш-промах.

конкатенация tag, set и offset. По сути, эти функции просто разбивают абсолютный адрес на tag+set и offset, поэтому не вижу смысла разбирать их подробнее.

```
always @(posedge clk) begin
    for(int i = 0; i < 3; i++) begin
        next_tact(); // init a, b, c arrays
    end
    pa = a_start_idx; next_tact(); // init pa
    pc = c_start_idx; next_tact(); // init pc

    for (int y = 0; y < M; y++) begin

        next_tact(); // new iteration
        //$display("%d out of %d", y, M);

        for (int x = 0; x < N; x++) begin
            next_tact(); // new iteration
            pb = b_start_idx; next_tact(); // init
            s = 0; next_tact(); // init
            for (int k = 0; k < K; k++) begin
                next_tact(); // new iteration

                tacts_cnt_prev = tacts_cnt;
                read8_by_abs_addr(pa + k, data8);

                if (tacts_cnt - tacts_cnt_prev < 100)
                    cache_hit_cnt += 1;
                else
                    cache_miss_cnt += 1;

                tacts_cnt_prev = tacts_cnt;
                read16_by_abs_add(pb + 2*x, data16);

                if (tacts_cnt - tacts_cnt_prev < 100)
                    cache_hit_cnt += 1;
                else
```



```

        cache_miss_cnt += 1;

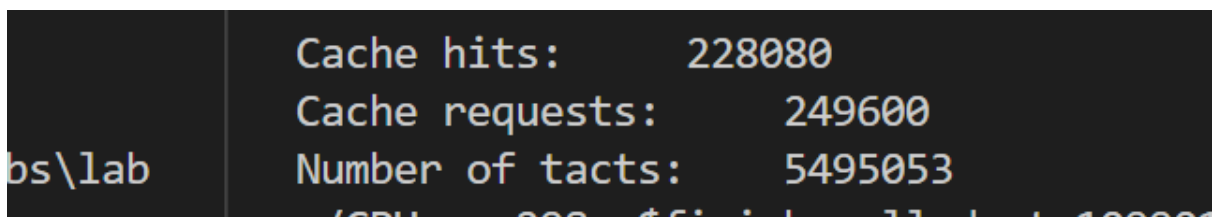
        s += data8 * data16; // sum +1 and
multiplication +5
        pb += 2 * N; // sum +1      => 7 tacts
        for (int i = 0; i < 7; i++) next_tact();
    end
    tacts_cnt_prev = tacts_cnt;
    write32_by_abs_addr(pc + 4 * x, s);

    if (tacts_cnt - tacts_cnt_prev < 100)
        cache_hit_cnt += 1;
    else
        cache_miss_cnt += 1;
    end
    pa += K; // sum +1
    pc += 4 * N; // sum +1, multiplication +5      => 7
tacts
    for (int i = 0; i < 7; i++) next_tact();
    end
    next_tact(); // exit of the function
    $display("Cache hits:%d", cache_hit_cnt);
    $display("Cache requests:%d", (cache_hit_cnt +
cache_miss_cnt));
    $display("Number of tacts:%d", tacts_cnt);
    $finish();
end

```

Раскомментируйте закомментированную строку в коде, чтобы видеть процесс работы программы.

Результат работы программы:



```

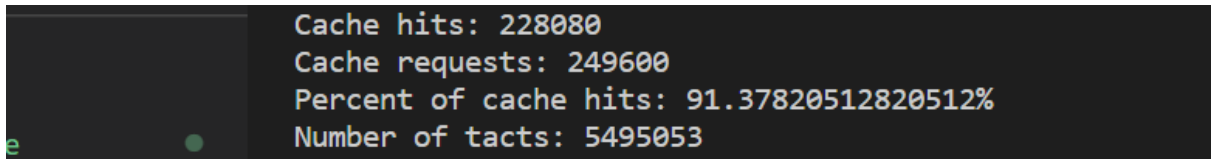
bs\lab      Cache hits:      228080
            Cache requests:  249600
            Number of tacts:  5495053
            (CPU user300: $finish called at 100000

```

Рисунок 9 - Ответ на задачу на языке Verilog

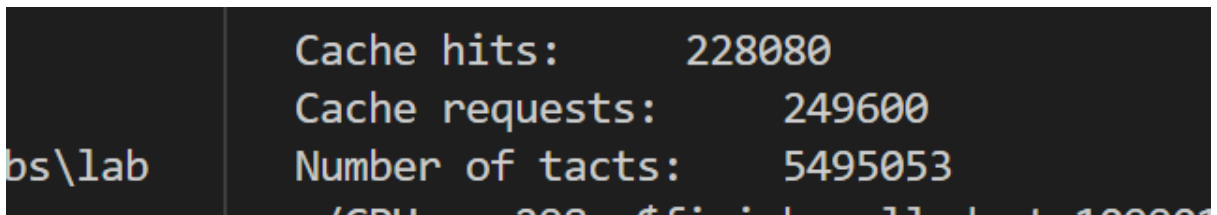
Сравнение полученных результатов

Посмотрим ещё раз на результаты программ.



```
Cache hits: 228080
Cache requests: 249600
Percent of cache hits: 91.37820512820512%
Number of tacts: 5495053
```

Рисунок 10 - Ответ на задачу на языке Java



```
Cache hits:      228080
Cache requests:   249600
Number of tacts:  5495053
```

Рисунок 11 - Ответ на задачу на языке Verilog

Можете наблюдать, что результаты сходятся. Процент кэш попаданий тоже будет одинаковый, так как количество кэш попаданий и количество обращений в кэш совпадает.

Листинг кода

```
module CPU(  
    output wire[ADDR1_BUS_SIZE-1:0] A1,  
    inout wire[DATA1_BUS_SIZE-1:0] D1,  
    inout wire[CTR1_BUS_SIZE-1:0] C1,  
    input clk  
);  
  
    localparam M = 64;  
    localparam N = 60;  
    localparam K = 32;  
  
    localparam a_start_idx = 0;  
    localparam b_start_idx = a_start_idx + M*K;  
    localparam c_start_idx = b_start_idx + 2*K*N;  
  
    int pa;  
    int pb;  
    int pc;  
    int s;  
    int tacts_cnt;  
    int cache_hit_cnt;  
    int cache_miss_cnt;  
    int tacts_cnt_prev;  
  
    logic[31:0] data32;  
    logic[15:0] data16;  
    logic [7:0] data8;  
  
    logic[CTR1_BUS_SIZE-1:0] C1_buffer;  
    logic[DATA1_BUS_SIZE-1:0] D1_buffer;  
    logic[ADDR1_BUS_SIZE-1:0] A1_buffer;  
  
    assign (strong1, pull0) A1 = A1_buffer;  
    assign (strong1, pull0) D1 = D1_buffer;  
    assign (strong1, pull0) C1 = C1_buffer;
```

```

task next_clk;
begin
    if (clk == 0) wait(clk != 0);
    else if (clk == 1) wait(clk != 1);
end
endtask

```

```

task next_tact;
begin
    next_clk();
    next_clk();
end
endtask

```

```

task write32_by_abs_addr(input [CACHE_ADDR_SIZE]
addr, [31:0] data);
begin
    C1_buffer = 7; D1_buffer = {data[7:0],
data[15:8]}; A1_buffer =
addr[CACHE_OFFSET_SIZE+:CACHE_SET_SIZE+CACHE_TAG_SIZE];
    next_tact();
    A1_buffer = addr[0+:CACHE_OFFSET_SIZE];
D1_buffer = {data[23:16], data[31:24]};
    next_tact();
    A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
    next_clk();
    while (C1 != 7) begin
        next_tact();
    end
    next_tact(); next_clk();
end
endtask

```

```

task read32_by_abs_addr(input [CACHE_ADDR_SIZE]
addr, output [31:0] data);
begin

```

```

        C1_buffer = 3; A1_buffer =
addr[CACHE_OFFSET_SIZE+:CACHE_SET_SIZE+CACHE_TAG_SIZE];
        next_tact();
        A1_buffer = addr[0+:CACHE_OFFSET_SIZE];
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_clk();
        end
        data[16:0] = {D1[7:0], D1[15:8]};
        next_tact();
        data[31:16] = {D1[7:0], D1[15:8]}; C1_buffer =
0; D1_buffer = 0; A1_buffer = 0;
        next_tact(); next_clk();
    end
endtask

```

```

    task read8_by_abs_addr(input [CACHE_ADDR_SIZE] addr,
output [7:0] data);
    begin
        C1_buffer = 1; A1_buffer =
addr[CACHE_OFFSET_SIZE+:CACHE_SET_SIZE+CACHE_TAG_SIZE];
        next_tact();
        A1_buffer = addr[0+:CACHE_OFFSET_SIZE];
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        data = D1; C1_buffer = 0; D1_buffer = 0;
A1_buffer = 0;
        next_tact(); next_clk();
    end
endtask;

```

```

    task read16_by_abs_add(input [CACHE_ADDR_SIZE] addr,
output [15:0] data);
    begin
        C1_buffer = 2; A1_buffer =
addr[CACHE_OFFSET_SIZE+:CACHE_SET_SIZE+CACHE_TAG_SIZE];
        next_tact();
        A1_buffer = addr[0+:CACHE_OFFSET_SIZE];
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        data = {D1[7:0], D1[15:8]}; C1_buffer = 0;
D1_buffer = 0; A1_buffer = 0;
        next_tact(); next_clk();
    end
endtask;

```

```

    task write8(input [7:0] data, [CACHE_TAG_SIZE-1:0]
tag, [CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0]
offset);
    begin
        C1_buffer = 5; D1_buffer = data; A1_buffer =
{tag, set};
        next_tact();
        A1_buffer = offset;
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        next_tact(); next_clk();
    end
endtask

```

```

    task write16(input [15:0] data, [CACHE_TAG_SIZE-1:0]
tag, [CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0]
offset);
    begin
        C1_buffer = 6;
        D1_buffer = {data[7:0], data[15:8]}; A1_buffer
= {tag, set};
        next_tact();
        A1_buffer = offset;
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        next_tact(); next_clk();
    end
endtask

```

```

    task write32(input [31:0] data, [CACHE_TAG_SIZE-1:0]
tag, [CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0]
offset);
    begin
        C1_buffer = 7; D1_buffer = {data[7:0],
data[15:8]}; A1_buffer = {tag, set};
        next_tact();
        A1_buffer = offset; D1_buffer = {data[23:16],
data[31:24]};
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        next_tact(); next_clk();
    end
endtask

```

```

    task read8(input [CACHE_TAG_SIZE-1:0] tag,
[CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0] offset,
output [7:0] data);
    begin
        C1_buffer = 1; A1_buffer = {tag, set};
        next_tact();
        A1_buffer = offset;
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        data = D1; C1_buffer = 0; D1_buffer = 0;
A1_buffer = 0;
        next_tact(); next_clk();
    end
endtask

```

```

    task read16(input [CACHE_TAG_SIZE-1:0] tag,
[CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0] offset,
output [16:0] data);
    begin
        C1_buffer = 2; A1_buffer = {tag, set};
        next_tact();
        A1_buffer = offset;
        next_tact();
        A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
        next_clk();
        while (C1 != 7) begin
            next_tact();
        end
        data = {D1[7:0], D1[15:8]}; C1_buffer = 0;
D1_buffer = 0; A1_buffer = 0;
        next_tact(); next_clk();
    end
end

```


endtask

```
task read32(input [CACHE_TAG_SIZE-1:0] tag,  
[CACHE_SET_SIZE-1:0] set, [CACHE_OFFSET_SIZE-1:0] offset,  
output [31:0] data);
```

```
begin
```

```
    C1_buffer = 3; A1_buffer = {tag, set};
```

```
    next_tact();
```

```
    A1_buffer = offset;
```

```
    next_tact();
```

```
    A1_buffer = 0; D1_buffer = 0; C1_buffer = 0;
```

```
    next_clk();
```

```
    while (C1 != 7) begin
```

```
        next_tact();
```

```
    end
```

```
    data[16:0] = {D1[7:0], D1[15:8]};
```

```
    next_tact();
```

```
    data[31:16] = {D1[7:0], D1[15:8]}; C1_buffer =  
0; D1_buffer = 0; A1_buffer = 0;
```

```
    next_tact(); next_clk();
```

```
end
```

endtask

```
task invalidate_line(input [CACHE_TAG_SIZE-1:0] tag,  
[CACHE_SET_SIZE-1:0] set);
```

```
begin
```

```
    C1_buffer = 4; A1_buffer = {tag, set};
```

```
    next_tact();
```

```
    C1_buffer = 0;
```

```
    next_clk();
```

```
    while (C1 != 7) begin
```

```
        next_tact();
```

```
    end
```

```
    next_tact(); next_clk();
```

```
end
```

endtask

```
initial begin
```

```
A1_buffer = 0;
```

```
C1_buffer = 0;
```

```
D1_buffer = 0;
```

```
end
```

```
initial begin
```

```
pa = a_start_idx; pb = b_start_idx; pc =  
c_start_idx;
```

```
data8 = 0; data16 = 0; data32 = 0;
```

```
cache_hit_cnt = 0; cache_miss_cnt = 0;
```

```
tacts_cnt_prev = 0; tacts_cnt = 0;
```

```
end
```

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

```
tacts_cnt += 1;
```

```
end
```

```
// always @(posedge clk) begin
```

```
//   for(int i = 0; i < 3; i++) begin
```

```
//       next_tact(); // init a, b, c arrays
```

```
//   end
```

```
//   pa = a_start_idx; next_tact(); // init pa
```

```
//   pc = c_start_idx; next_tact(); // init pc
```

```
//   for (int y = 0; y < M; y++) begin
```

```
//       next_tact(); // new iteration
```

```
//       // $display("%d out of %d", y, M);
```

```
//       for (int x = 0; x < N; x++) begin
```

```
//           next_tact(); // new iteration
```

```
//           pb = b_start_idx; next_tact(); // init
```

```
//           s = 0; next_tact(); // init
```

```
//           for (int k = 0; k < K; k++) begin
```

```
//               next_tact(); // new iteration
```

```

//          tacts_cnt_prev = tacts_cnt;
//          read8_by_abs_addr(pa + k, data8);

//          if (tacts_cnt - tacts_cnt_prev <
100)
//              cache_hit_cnt += 1;
//          else
//              cache_miss_cnt += 1;

//          tacts_cnt_prev = tacts_cnt;
//          read16_by_abs_add(pb + 2*x,
data16);

//          if (tacts_cnt - tacts_cnt_prev <
100)
//              cache_hit_cnt += 1;
//          else
//              cache_miss_cnt += 1;

//          s += data8 * data16; // sum +1
and multiplication +5
//          pb += 2 * N; // sum +1      => 7
tacts
//          for (int i = 0; i < 7; i++)
next_tact();
//          end
//          tacts_cnt_prev = tacts_cnt;
//          write32_by_abs_addr(pc + 4 * x, s);

//          if (tacts_cnt - tacts_cnt_prev < 100)
//              cache_hit_cnt += 1;
//          else
//              cache_miss_cnt += 1;
//          end
//          pa += K; // sum +1
//          pc += 4 * N; // sum +1, multiplication +5
=> 7 tacts

```

```

//      for (int i = 0; i < 7; i++) next_tact();
//  end
//  next_tact(); // exit of the function
//  $display("Cache hits:%d", cache_hit_cnt);
//  $display("Cache requests:%d", (cache_hit_cnt +
cache_miss_cnt));
//  $display("Number of tacts:%d", tacts_cnt);
//  $finish();
// end // CacheHits:228080, cacheMisses:21520
tacts:5495053

```

```

always @(posedge clk) begin
write32('h12345678, 'b1010101010, 'b00101, 5);
write16('h1389, 'b1010101010, 'b00101, 1);
write8('h09, 'b1010111110, 'b01000, 15);
read32('b1010101010, 'b00101, 5, data32);
read16('b1010101010, 'b00101, 1, data16);
read8('b1010111110, 'b01000, 'b1111, data8);
$display("CPU: t=%0t, data32=%h", $time, data32);
$display("CPU: t=%0t, data16=%h", $time, data16);
$display("CPU: t=%0t, data8=%h", $time, data8);
$finish();
end
endmodule

```

```

module Cache(
    input wire[ADDR1_BUS_SIZE-1:0] A1,
    inout wire[DATA1_BUS_SIZE-1:0] D1,
    inout wire[CTR1_BUS_SIZE-1:0] C1,
    output wire[ADDR2_BUS_SIZE-1:0] A2,
    inout wire[DATA2_BUS_SIZE-1:0] D2,
    inout wire[CTR2_BUS_SIZE-1:0] C2,
    input wire clk,
    input wire reset,
    input wire C_DUMP
);

    logic[ADDR2_BUS_SIZE-1:0] A2_buffer;

    logic[DATA1_BUS_SIZE-1:0] D1_buffer;
    logic[CTR1_BUS_SIZE-1:0] C1_buffer;

    logic[DATA2_BUS_SIZE-1:0] D2_buffer;
    logic[CTR2_BUS_SIZE-1:0] C2_buffer;

    logic[CACHE_TAG_SIZE-1:0] tag;
    logic[CACHE_SET_SIZE-1:0] set;
    logic[CACHE_OFFSET_SIZE-1:0] offset;
    logic[31:0] data_buffer_LE; // LE - Little Endian
    bit index;
    logic[CTR1_BUS_SIZE-1:0] C1_copy;

```

```

    logic[7:0]
data[CACHE_SETS_COUNT-1:0][CACHE_WAY-1:0][CACHE_LINE_SIZE
-1:0];
    logic valid[CACHE_SETS_COUNT-1:0][CACHE_WAY-1:0];
    logic dirty[CACHE_SETS_COUNT-1:0][CACHE_WAY-1:0];
    bit LRU[CACHE_SETS_COUNT-1:0];
    logic[CACHE_TAG_SIZE-1:0]
tags[CACHE_SETS_COUNT-1:0][CACHE_WAY-1:0];

    assign (strong1, pull0) A2 = A2_buffer;
    assign (strong1, pull0) D1 = D1_buffer;
    assign (strong1, pull0) D2 = D2_buffer;
    assign (strong1, pull0) C1 = C1_buffer;
    assign (strong1, pull0) C2 = C2_buffer;

    task next_clk;
    begin
        if (clk == 0) wait(clk != 0);
        else if (clk == 1) wait(clk != 1);
    end
    endtask

    task next_tact;
    begin
        next_clk();
        next_clk();
    end
    endtask

    task reset_cache;
    begin
        tag = 0; set = 0; offset = 0; data_buffer_LE =
0; C1_copy = 0; index = 0;
        D2_buffer = 0; C2_buffer = 0; A2_buffer = 0;
        C1_buffer = 0; D1_buffer = 0;

```

```

        for(int i = 0; i < CACHE_SETS_COUNT; i++) begin
            for(int j = 0; j < CACHE_WAY; j++) begin
                valid[i][j] = 0;
                dirty[i][j] = 0;
                LRU[i] = 0;
                tags[i][j] = 0;
                for (int k = 0; k < CACHE_LINE_SIZE;
k++) begin
                    data[i][j][k] = 0;
                end
            end
        end
    end
endtask

task dump_to_terminal;
begin
    $display("----CACHE_DUMP----");
    for(int i = 0; i < CACHE_SETS_COUNT; i++) begin
        for(int j = 0; j < CACHE_WAY; j++) begin
            $write("Cache_set=%b, idx=%b,
CACHE_LINE: valid(b)=%b, dirty(b)=%b, LRU(b)=%b,
tag(h)=%h, data(h)=",
                i[0+:CACHE_SET_SIZE],
j[0+:CACHE_WAY_SIZE], valid[i][j], dirty[i][j], LRU[i],
tags[i][j]
            );
            for (int k = 0; k < CACHE_LINE_SIZE;
k++) begin
                $write("%h", data[i][j][k]);
            end
            $display();
        end
    end
end
endtask

```

```

    task read_from_memory(input [CACHE_TAG_SIZE-1:0]
tag, [CACHE_SET_SIZE-1:0] set, idx);
    begin
        C2_buffer = 'b10; A2_buffer = {tag, set};
D2_buffer = 0;
        next_tact();
        C2_buffer = 0; A2_buffer = 0; D2_buffer = 0;
        tags[set][idx] = tag;
        valid[set][idx] = 1;
        dirty[set][idx] = 0;
        LRU[set] = idx;
        next_clk();
        while (C2 != 1) begin
            next_tact();
        end
        for (int i = CACHE_LINE_SIZE - 2; i > -1; i -=
2) begin
            {data[set][idx][i + 1], data[set][idx][i]}
= D2;
            next_tact();
        end
        next_clk();
        // log_read_from_memory(set, idx);
    end
endtask

```

```

    task log_read_from_memory(input [CACHE_SET_SIZE-1:0]
set, idx);
    begin
        $write("read_from_memory: \t");
        for (int i = 0; i < CACHE_LINE_SIZE; i++)
$write("%h", data[set][idx][i]);
        $write("\tttag: %b, set: %b, idx: %b",
tags[set][idx], set, idx);
        $display("");
    end
endtask

```



```

    task upload_data_to_memory(input
[CACHE_SET_SIZE-1:0] set, idx);
    begin
        D2_buffer = {data[set][idx][CACHE_LINE_SIZE -
1], data[set][idx][CACHE_LINE_SIZE - 2]};
        A2_buffer = {tags[set][idx], set};
        C2_buffer = 3;
        next_tact();
        for (int i = CACHE_LINE_SIZE - 4; i >= 0; i -=
2) begin
            D2_buffer = {data[set][idx][i + 1],
data[set][idx][i]};
            next_tact();
            A2_buffer = 0;
        end
        C2_buffer = 0; D2_buffer = 0;
        dirty[set][idx] = 0;
        next_clk();
        while (C2 != 1) begin
            next_tact();
        end
        next_tact(); next_clk();
        // log_upload_data_to_memory(set, idx);
    end
endtask

```

```

    task log_upload_data_to_memory(input
[CACHE_SET_SIZE-1:0] set, idx);
    begin
        $write("upload_data_to_memory: \t");
        for (int i = 0; i < CACHE_LINE_SIZE; i++)
$write("%h", data[set][idx][i]);
        $display("");
    end
endtask

```

```

initial begin
reset_cache();
end

always @(posedge C_DUMP) begin
dump_to_terminal();
end

always @(posedge reset) begin
reset_cache();
end

always @(negedge clk) begin
if (C1 == 4) begin //
=====C1_INVALIDATE_LINE=====
=====
        {tag, set} = A1;
        if (tags[set][0] == tag || tags[set][1] == tag)
begin
            index = (tags[set][0] == set) ? 0 : 1;
            valid[set][index] = 0;
        end
        while (C1 != 0) begin
            next_tact();
        end
        C1_buffer = 3'b111;
        next_tact();
        C1_buffer = 0;
    end else if (1 <= C1 && C1 <= 3) begin //
=====C1_READING=====
=====
        {tag, set} = A1; C1_copy = C1;
        next_tact();
        offset = A1[0+:CACHE_OFFSET_SIZE];
        next_clk();
        for (int i = 0; i < 2; i++) next_tact(); //
waiting

```

```

        if (tags[set][0] == tag || tags[set][1] == tag)
begin
        index = (tags[set][0] == tag) ? 0 : 1;
        if (valid[set][index] == 1) begin
            for (int i = 0; i < 2; i++)
next_tact();
        end else begin
            read_from_memory(tag, set, index);
        end
        end else if (valid[set][0] == 0 ||
valid[set][1] == 0) begin
        index = (valid[set][0] == 0) ? 0 : 1;
        read_from_memory(tag, set, index);
        end else begin
            index = (LRU[set] == 1) ? 0 : 1;
            if (dirty[set][index] == 1 &&
valid[set][index] == 1)
                upload_data_to_memory(set, index);
            read_from_memory(tag, set, index);
        end
        LRU[set] = index; tags[set][index] = tag;
valid[set][index] = 1;
        C1_buffer = 3'b111;
        case (C1_copy)
            3'b001: begin
                D1_buffer = data[set][index][offset];
            end
            3'b010: begin
                D1_buffer = {data[set][index][offset +
1], data[set][index][offset]};
            end
            3'b011: begin
                D1_buffer = {data[set][index][offset +
3], data[set][index][offset + 2]};
                next_tact();
                D1_buffer = {data[set][index][offset +
1], data[set][index][offset]};

```

```

        end
    endcase
    C1_copy = 0;
    next_tact();
    C1_buffer = 0; D1_buffer = 0;

    end else if (5 <= C1 && C1 <= 7) begin //
=====C1_WRITE=====
=====
        {tag, set} = A1; C1_copy = C1;
        data_buffer_LE[2*DATA1_BUS_SIZE-1:DATA1_BUS_SIZE] = D1;
        next_tact();

        if (C1 == 7) data_buffer_LE[DATA1_BUS_SIZE-1:0]
= D1;

        offset = A1[0+:CACHE_OFFSET_SIZE];
        next_clk();

        for (int i = 0; i < 2; i++) next_tact(); //
waiting

        if (tags[set][0] == tag || tags[set][1] == tag)
begin
            index = (tags[set][0] == tag) ? 0 : 1;
            if (valid[set][index] == 1) begin
                for (int i = 0; i < 2; i++)
next_tact();
            end else begin
                read_from_memory(tag, set, index);
            end
        end else if (valid[set][0] == 0 ||
valid[set][1] == 0) begin
            index = (valid[set][0] == 0) ? 0 : 1;
            read_from_memory(tag, set, index);
        end else begin
            index = (LRU[set] == 1) ? 0 : 1;

```

```

        if (dirty[set][index] == 1 &&
valid[set][index] == 1)
            upload_data_to_memory(set, index);
            read_from_memory(tag, set, index);
        end

        LRU[set] = index; tags[set][index] = tag;
valid[set][index] = 1; dirty[set][index] = 1;

        case (C1_copy)
            3'b101: begin
                data[set][index][offset] =
data_buffer_LE[23:16];
            end
            3'b110: begin
                data[set][index][offset] =
data_buffer_LE[23:16];
                data[set][index][offset + 1] =
data_buffer_LE[31:24];
            end
            3'b111: begin
                for (int i = 0; i < 4; i++)
                    data[set][index][offset + i] =
data_buffer_LE[i*8+:8];
            end
        endcase

        C1_copy = 0; C1_buffer = 3'b111;
        next_tact();
        C1_buffer = 0;

    end
end
endmodule

```

```

module Memory(
    input wire[ADDR2_BUS_SIZE - 1:0] A2,
    inout wire[DATA2_BUS_SIZE - 1:0] D2,
    inout wire[CTR2_BUS_SIZE - 1:0] C2,
    input wire clk,
    input wire reset,
    input wire M_DUMP
);

    integer SEED = _SEED;

    logic[7:0]
data[MEM_CACHE_LINE_COUNT-1:0][CACHE_LINE_SIZE-1:0];

    logic[DATA2_BUS_SIZE-1:0] D2_buffer;
    logic[CTR2_BUS_SIZE-1:0] C2_buffer;

    logic[CACHE_ADDR_SIZE+CACHE_OFFSET_SIZE-1:0]
cache_line_addr;

    assign (strong1, pull0) D2 = D2_buffer;
    assign (strong1, pull0) C2 = C2_buffer;

    task next_clk;
    begin
        if (clk == 0) wait(clk != 0);
    end

```

```

        else if (clk == 1) wait(clk != 1);
    end
endtask

task next_tact;
begin
    next_clk();
    next_clk();
end
endtask

task reset_memory;
begin
    cache_line_addr = 0;
    D2_buffer = 0; C2_buffer = 0;
    for (int i = 0; i < MEM_CACHE_LINE_COUNT; i++)
begin
        for (int j = 0; j < CACHE_LINE_SIZE; j++)
begin
            data[i][j] = $random(SEED)>>16;
        end
    end
end
endtask

task dump_to_terminal;
begin
    $display("----MEMORY_DUMP----");
    for(int i = 0; i < MEM_CACHE_LINE_COUNT; i++)
begin
        $write("%b: ",
i[0+:CACHE_TAG_SIZE+CACHE_SET_SIZE]);
        for (int j = 0; j < CACHE_LINE_SIZE; j++)
begin
            $write("%h", data[i][j]);
        end
    end
end
endtask

```

```

        $display();
    end
end
endtask

initial begin
    reset_memory();
end

always @(posedge reset) begin
    reset_memory();
end

always @(posedge M_DUMP) begin
    dump_to_terminal();
end

always @(negedge clk) begin

    if (C2 == 2) begin //
        =====C2_READ=====
        =====

        cache_line_addr = A2;

        for (int i = 0; i < 100; i++) next_tact();

        next_clk();
        D2_buffer =
{data[cache_line_addr][CACHE_LINE_SIZE - 1],
data[cache_line_addr][CACHE_LINE_SIZE - 2]};
        C2_buffer = 1;
        for (int i = CACHE_LINE_SIZE - 4; i > -1; i -=
2) begin
            next_tact();
            D2_buffer = {data[cache_line_addr][i+1],

```



```

data[cache_line_addr][i]];
    end
    next_tact();
    C2_buffer = 0; D2_buffer = 0;

    end else if (C2 == 3) begin //
=====C2_WRITE=====
=====

        cache_line_addr = A2;

        for (int i = CACHE_LINE_SIZE - 2; i > -1; i -=
2) begin
            {data[cache_line_addr][i+1],
data[cache_line_addr][i]} = D2;
            next_tact();
        end

        for (int i = 0; i < 100 - (CACHE_LINE_SIZE /
2); i++) next_tact();
        next_clk();
        C2_buffer = 1;
        next_tact();
        C2_buffer = 0; D2_buffer = 0;
    end
end

endmodule

`include "memory.sv"
`include "cache.sv"
`include "CPU.sv"

// bytes:
parameter MEM_SIZE = (1 << 19);
parameter CACHE_SIZE = (64 * 16);

```

```
parameter CACHE_LINE_SIZE = (16);
```

```
// integers:
```

```
parameter _SEED = (225526);
```

```
parameter MEM_CACHE_LINE_COUNT = MEM_SIZE /  
CACHE_LINE_SIZE;
```

```
parameter CACHE_LINE_COUNT = (64);
```

```
parameter CACHE_WAY = (2);
```

```
parameter CACHE_SETS_COUNT = CACHE_LINE_COUNT /  
CACHE_WAY;
```

```
// bits:
```

```
parameter CACHE_TAG_SIZE = (10);
```

```
parameter CACHE_SET_SIZE = (5);
```

```
parameter CACHE_OFFSET_SIZE = (4);
```

```
parameter CACHE_ADDR_SIZE = (19);
```

```
parameter CACHE_WAY_SIZE = (1);
```

```
parameter ADDR1_BUS_SIZE = (CACHE_TAG_SIZE +  
CACHE_SET_SIZE);
```

```
parameter DATA1_BUS_SIZE = (16);
```

```
parameter CTR1_BUS_SIZE = (3);
```

```
parameter ADDR2_BUS_SIZE = ADDR1_BUS_SIZE;
```

```
parameter DATA2_BUS_SIZE = (16);
```

```
parameter CTR2_BUS_SIZE = (2);
```

```
module testbench();
```

```
    wire[ADDR1_BUS_SIZE-1:0] A1;
```

```
    wire[CTR1_BUS_SIZE-1:0] C1;
```

```
    wire[DATA1_BUS_SIZE-1:0] D1;
```

```
    wire[ADDR2_BUS_SIZE-1:0] A2;
```

```
    wire[CTR2_BUS_SIZE-1:0] C2;
```

```
    wire[DATA2_BUS_SIZE-1:0] D2;
```

```
    bit clk;
```

```
    bit reset;
```

```
    bit C_DUMP;
```

```
    bit M_DUMP;
```

```

    CPU cpu (A1, D1, C1, clk);
    Cache cache (A1, D1, C1, A2, D2, C2, clk, reset,
C_DUMP);
    Memory memory (A2, D2, C2, clk, reset, M_DUMP);

    task next_clk;
    begin
        if (clk == 0) wait(clk != 0);
        else if (clk == 1) wait(clk != 1);
    end
    endtask

    always #1 clk = ~clk;

    initial begin
        clk = 0;
        reset = 0;
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, testbench);
    end

    // initial begin
    //     next_clk(); next_clk(); next_clk(); next_clk();
    //     M_DUMP = 1;
    // end

endmodule

```

```

package java_cache;

public class CPU {
    public static void main(String[] args) {
        final long[] tacts = new long[1];
        final Cache cache = new Cache(tacts);

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

        final int aStart = 0;
        final int bStart = aStart + M * K;
        final int cStart = bStart + 2 * K * N;

        int pa = aStart;
        int pb = bStart;
        int pc = cStart;
        int cacheHits = 0;
        int cacheMisses = 0;

        double tactsBefore;

        tacts[0] = 1;
        tacts[0] += 5;
        for (int y = 0; y < M; y++) {
            tacts[0] += 1; // new iteration
            for (int x = 0; x < N; x++) {
                tacts[0] += 1; // new iteration
                pb = bStart; tacts[0] += 1;
                tacts[0] += 1; // initialization s
                for (int k = 0; k < K; k++) {
                    tacts[0] += 1; // new iteration
                    tactsBefore = tacts[0];
                    cache.readByte(pa + k);
                    if (tacts[0] - tactsBefore > 100) {
                        cacheMisses += 1;
                    }
                }
            }
        }
    }
}

```

```

        } else {
            cacheHits += 1;
        }
        tactsBefore = tacts[0];
        cache.readTwoBytes(pb + 2 * x);
        if (tacts[0] - tactsBefore > 100) {
            cacheMisses += 1;
        } else {
            cacheHits += 1;
        }
        pb += 2 * N;
        tacts[0] += 2; // two times sum
        tacts[0] += 5; // multiplication
    }
    tactsBefore = tacts[0];

    cache.write(pc + 4 * x);
    if (tacts[0] - tactsBefore > 100) {
        cacheMisses += 1;
    } else {
        cacheHits += 1;
    }
}
pa += K;
pc += 4 * N;
tacts[0] += 7; // two times sum and
multiplication
}
tacts[0] += 1; // exit of the function
System.out.println("Cache hits: " + cacheHits);
System.out.println("Cache requests: " + (cacheHits +
cacheMisses));
System.out.println("Percent of cache hits: " +
(((double) cacheHits) / (cacheHits + cacheMisses)) * 100
+ "%");
System.out.println("Number of tacts: " + (int)
tacts[0]);

```

```
    }  
}
```

```
package java_cache;
```

```
public class Cache {  
    public static final int CACHE_SETS_COUNT = 32;  
    public static final int CACHE_WAY = 2;  
    public static final int CACHE_TAG_SIZE = 10;  
    public static final int CACHE_SET_SIZE = 5;  
    public static final int CACHE_OFFSET_SIZE = 4;  
  
    private final long[] tacts; // array with length = 1  
    private final int tags[][];  
    private final boolean valid[][];  
    private final boolean dirty[][];  
    private final int LRU[];  
  
    public Cache(final long[] tacts) {  
        this.tacts = tacts;  
  
        this.tags = new int[CACHE_SETS_COUNT][CACHE_WAY];  
        this.valid = new  
boolean[CACHE_SETS_COUNT][CACHE_WAY];  
        this.dirty = new  
boolean[CACHE_SETS_COUNT][CACHE_WAY];  
        this.LRU = new int[CACHE_SETS_COUNT];  
  
        for (int i = 0; i < CACHE_SETS_COUNT; i++) {  
            for (int j = 0; j < CACHE_WAY; j++) {  
                tags[i][j] = 0;  
                valid[i][j] = false;  
                dirty[i][j] = false;  
            }  
        }  
    }  
}
```

```

        LRU[i] = 0;
    }
}
}

public void readByte(final int address) {
    tacts[0] += 2;
    final int tag = address >>> (CACHE_SET_SIZE +
    CACHE_OFFSET_SIZE);
    final int set = (address << (32 - CACHE_SET_SIZE -
    CACHE_OFFSET_SIZE)) >>> (32 - CACHE_TAG_SIZE +
    CACHE_SET_SIZE);
    final int index;
    tacts[0] += 2;
    if (tags[set][0] == tag || tags[set][1] == tag) {
        index = (tags[set][0] == tag) ? 0 : 1;
        if (valid[set][index]) {
            tacts[0] += 2;
        } else {
            readFromMemory(tag, set, index);
        }
    } else if (!valid[set][0] || !valid[set][1]) {
        index = valid[set][0] ? 1 : 0;
        readFromMemory(tag, set, index);
    } else {
        index = (LRU[set] == 1) ? 0 : 1;
        if (dirty[set][index] && valid[set][index]) {
            uploadDataToMemory(set, index);
        }
        readFromMemory(tag, set, index);
    }
    // sending data to the CPU in one tact
    tacts[0] += 2;
    LRU[set] = index;
    tags[set][index] = tag;
    valid[set][index] = true;
    return;
}

```

```
}
```

```
public void readTwoBytes(final int address) {  
    readByte(address);  
}
```

```
public void readFourBytes(final int address) {  
    readTwoBytes(address);  
    tacts[0] += 1;  
}
```

```
public void write(final int address) {  
    tacts[0] += 2;  
    final int tag = address >>> (CACHE_SET_SIZE +  
    CACHE_OFFSET_SIZE);  
    final int set = (address << (32 - CACHE_SET_SIZE -  
    CACHE_OFFSET_SIZE)) >>> (32 - CACHE_TAG_SIZE +  
    CACHE_SET_SIZE);  
    // System.out.println("W " +  
    Integer.toBinaryString(tag) + " " +  
    Integer.toBinaryString(set));  
    final int index;  
    tacts[0] += 2;  
    if (tags[set][0] == tag || tags[set][1] == tag) {  
        index = (tags[set][0] == tag) ? 0 : 1;  
        if (valid[set][index]) {  
            tacts[0] += 2;  
        } else {  
            readFromMemory(tag, set, index);  
        }  
    } else if (!valid[set][0] || !valid[set][1]) {  
        index = valid[set][0] ? 1 : 0;  
        readFromMemory(tag, set, index);  
    } else {  
        index = (LRU[set] == 1) ? 0 : 1;  
        if (dirty[set][index] && valid[set][index]) {  
            uploadDataToMemory(set, index);  
        }  
    }  
}
```



```

        }
        readFromMemory(tag, set, index);
    }
    // sending response to the CPU
    tacts[0] += 2;
    LRU[set] = index;
    dirty[set][index] = true;
    tags[set][index] = tag;
    valid[set][index] = true;
}

private void uploadDataToMemory(final int set, final
int index) {
    tacts[0] += 1;
    tacts[0] += 8; // sending data to memory
    tacts[0] += 100 - 8; // delay
    // sending response on the C2 wire
    tacts[0] += 2;
    dirty[set][index] = false;
}

private void readFromMemory(final int tag, final int
set, final int index) {
    tacts[0] += 1;
    tacts[0] += 100; // delay
    tacts[0] += 7; // sending data to the cache
    tacts[0] += 2;
    tags[set][index] = tag;
    valid[set][index] = true;
    dirty[set][index] = false;
    LRU[set] = index;
}
}

```