1. Инструментарий

Программа реализована на языке программирования Java (JDK 17), в процессе разработки использовалась операционная система Windows.

2. Результат работы

Программа выполняется без ошибок и выводит требуемые данные в правильном формате.

```
LRU:    hit perc. 99,9976%  time: 3003730
pLRU:   hit perc. 99,9976%  time: 3003730
```

3. Описание работы

Для начала рассчитаем значения тех параметров, которые не даны.

$MEM\_SIZE = 2^{ADDR\_LEN} = 2^{20}$ (B) = 1024 (MB) = 1 (GB)

$ADDR\_LEN = 20$ (Bits)

$CACHE\_WAY = 4$ (Bits)

$CACHE\_TAG\_LEN = 9$ (bits)

$CACHE\_IDX\_LEN = ADDR\_LEN - CACHE\_TAG\_LEN - CACHE\_OFFSET\_LEN = 4$ (bits)

$CACHE\_OFFSET\_LEN = \log_2(CACHE\_LINE\_SIZE) = 7$ (bits)

$CACHE\_SIZE = CACHE\_LINE\_COUNT * CACHE\_LINE\_SIZE = 2^{13}$ (bytes) = 8 (MB)

$CACHE\_LINE\_SIZE = 128$ (bytes)

$CACHE\_LINE\_COUNT = CACHE\_WAY * CACHE\_SETS\_COUNT = 64$

$CACHE\_SETS\_COUNT = 2^{CACHE\_IDX\_LEN} = 16$

$ADDR\_1\_BUS\_LEN = CACHE\_OFFSET\_LEN = 7$

$ADDR\_2\_BUS\_LEN = CACHE\_SET\_LEN + CACHE\_TAG\_LEN$

$DATA\_1\_BUS\_LEN = 16$

$DATA\_2\_BUS\_LEN = 16$

$CTR1\_BUS\_LEN = \log_2(количество\ возможных\ команд) = 3$

$CTR2\_BUS\_LEN = \log_2(количество\ возможных\ команд) = 2$

Теперь перейдём к реализации. В связи с тем, что программа, для которой нужно рассчитать количество Кэш-попаданий реализована на языке *C*, в то время как симулятор был реализован на *Java*, перенести её в том же виде не представляется возможным. Тем не менее, для расчёта требуемых значений вывод результата исходной программы не требуется, поэтому нам не требуется их реализация – достаточно просто вставить их в качестве комментариев в код, чтобы в дальнейшем было удобнее работать с симуляцией. Оставим только переменные, которые будут имитировать указатели, учитывая, что данные хранятся последовательно.

Для того, чтобы в будущем было проще управлять симуляторами с разными политиками замещения, был создан общий интерфейс с единственным методом – запрос к кэшу.

```
package cacher;

public interface Cache {
    Response request(int address, QueryType type, int size);
}
```

Также для реализации кэша нам понадобятся два вспомогательных класса: Response, необходимый для передачи результата работы кэша симулятору: флаг ***hit***, возвращающий *true*, если произошло кэш-попадание, а также число ***time***, хранящее число затраченного на запрос времени, а также ***Line***, хранящий необходимую информацию о кэш-линии: тэг, флаг ***isValid***, содержащий информацию о том, использована кэш-линия или нет, флаг ***isDirty***, который принимает значение *true*, если над этой линий производились операции, а также флаг ***lru***, который требуется для реализации политики вытеснения LRU.

```java
package cacher;

public class Response {
    private final boolean hit;
    private final int time;

    public Response(boolean hit, int time) {
        this.hit = hit;
        this.time = time;
    }

    public boolean isHit() {
        return hit;
    }

    public int getTime() {
        return time;
    }
}
```

Response.java

```java
package cacher;

public class Line {
    private int tag;
    private boolean isValid;
    private boolean isDirty;
    private int lru;

    public Line() {
        tag = 0;
        isValid = false;
        isDirty = false;
        lru = 0;
    }

    public int getTag() {
        return tag;
    }

    public void setTag(int tag) {
        this.tag = tag;
    }

    public boolean isValid() {
        return isValid;
    }

    public void setValid(boolean valid) {
        isValid = valid;
    }

    public boolean isDirty() {
        return isDirty;
    }

    public void setDirty(boolean dirty) {
        isDirty = dirty;
    }

    public int getLru() {
        return lru;
    }
```

```
    public void setLru(int lru) {
        this.lru = lru;
    }

    public void addLru() {
        lru++;
    }
}
```

Line.java

Теперь перейдём непосредственно к реализации класса ***LruCache***, наследующего интерфейс ***Cache***. Рассмотрим его элементы по отдельности.

```
public Response request(int address, QueryType type, int size) {
    int set = (address / Constants.CACHE_LINE_SIZE) % Constants.CACHE_SETS_COUNT;
    int tag = (address / Constants.CACHE_LINE_SIZE) / Constants.CACHE_SETS_COUNT;
    Address temp = new Address(set, tag);
    if (containsAddress(temp, type)) {
        return new Response(
                true,
                calcDTime(size) + Constants.CACHE_HIT_RESPONSE_TIME + Constants.C_CAPACITY
        );
    }
    return new Response(false, add(temp, type, size));
}
```

Метод ***request*** обрабатывает запросы к кэшу и возвращает ***Response*** с результатами работы. В случае, если в кэше есть линия с указанным тэгом, то просто вернём, что кэш-попадание успешно и (метод ***containsAdress*** возвращает *true*) и затраченное время. В противном случае, сделаем запрос ***add*** для добавления элемента в кэш. Рассмотрим этот метод

```
public int add(Address address, QueryType type, int size) {
    int time = 2 * calcDTime(size) + Constants.CACHE_MISS_RESPONSE_TIME +
Constants.MEMORY_RESPONSE_TIME;
    for (int ch = 0; ch < Constants.CACHE_WAY; ch++) {
        if (!lines[address.getSet()][ch].isValid()) {
            write(address, ch, type);
            return time;
        }
    }
    int toOverwrite = findReplace(address);
    write(address, toOverwrite, type);
    if (lines[address.getSet()][toOverwrite].isDirty()) {
        time += Constants.MEMORY_RESPONSE_TIME +
                calcDTime(size) +
                Constants.CACHE_HIT_RESPONSE_TIME +
                Constants.C_CAPACITY;
    }
    return time;
}
```

Для начала метод пытается найти незанятую кэш-линию. Если он находит такую, то записывает в неё. В противном случае, вызывает метод ***findReplace***, чтобы выбрать линию, которая будет перезаписана, после чего перезаписывает в неё. Рассмотрим все задействованные вспомогательные методы.

Метод ***update*** обновляет флаги *lru*, за исключением того, который был перезаписан/записан.

```
public void update(Address address, int skip) {
    for (int ch = 0; ch < Constants.CACHE_WAY; ch++) {
        if (ch != skip && lines[address.getSet()][ch].isValid()) {
            lines[address.getSet()][ch].addLru();
        }
    }
}
```

Метод ***write*** непосредственно записывает в кэш-линию.

```
public void write(Address address, int ch, QueryType type) {
    lines[address.getSet()][ch].setLru(0);
    lines[address.getSet()][ch].setValid(true);
    lines[address.getSet()][ch].setTag(address.getTag());
    lines[address.getSet()][ch].setDirty(type == QueryType.w);
    update(address, ch);
}
```

Метод ***findReplace*** выбирает кэш-линию, которая будет перезаписана, в соответствие с политикой вытеснения.

```
public int findReplace(Address address) {
    int toOverwrite = 0;
    for (int ch = 1; ch < Constants.CACHE_WAY; ch++) {
        if (lines[address.getSet()][ch].getLru() >
lines[address.getSet()][toOverwrite].getLru()) {
            toOverwrite = ch;
        }
    }
    return toOverwrite;
}
```

Теперь рассмотрим реализацию класса ***PLruCache***, наследующего LruCache. Поскольку отличается только политика вытеснения, переопределить нужно только те методы, где она влияет на реализацию – ***findReplace*** и ***update***.

```
public void update(Address address, int skip) {
    lines[address.getSet()][skip].setLru(1);

    int count = 0;
    for (int ch = 0; ch < Constants.CACHE_WAY; ch++) {
        count += lines[address.getSet()][ch].getLru();
    }

    if (count == Constants.CACHE_WAY) {
        for (int ch = 0; ch < Constants.CACHE_WAY; ch++) {
            lines[address.getSet()][ch].setLru(0);
        }
    }
}
```

Теперь мы выбираем самый левый, у которого флаг *lru* равен *0*.

```
public int findReplace(Address address) {
    int res = 0;
    for (int ch = 0; ch < Constants.CACHE_WAY; ch++) {
        if (lines[address.getSet()][ch].getLru() == 0) {
            res = ch;
            break;
        }
    }
    return res;
}
```

Для того, чтобы симулировать работу (обрабатывать кэш-запросы и считать время в тактах), был реализован класс Simulator и SimulationList, чтобы удобнее взаимодействовать с несколькими симуляциями. Большого интереса данные классы не представляют, поэтому не будем останавливаться на их реализации.

Теперь перейдём к ***Main***'у, в котором симулируется работа программы. От массивов нам потребуется только их адрес – посчитаем его, в соответствии с размерностями и количеством занимаемых байт. Тогда чтобы обратиться по индексу нужно просто прибавить его к этому адресу. Теперь добавим все операции, влияющие на такты и кэш. В конце добавим форматированный вывод для получения ответа.

```java
import cacher.*;

import java.util.List;

public class Main {
    private static final int M = 64;
    private static final int N = 60;
    private static final int K = 32;

    public static void main(String[] args) {
        SimulationList simulationList = new SimulationList(List.of(
           new Simulator(new LruCache()),
           new Simulator(new PLruCache())
        ));

        int pa = 64; //8 bit
        simulationList.initializeVariable();
        int pc = 64 + M * K + 2 * K * N; //32 bit
        simulationList.initializeVariable();

        simulationList.initializeVariable(); // cycle begins
        for (int y = 0; y < M; y++) {
            simulationList.initializeVariable(); // cycle begins
            for (int x = 0; x < N; x++) {
                int pb = 64 + M * K; //16 bit
                simulationList.initializeVariable();
                //int s = 0;
                simulationList.initializeVariable();

                simulationList.initializeVariable(); // cycle begins
                for (int k = 0; k < K; k++) {
                    //s += pa[k] * pb[x];
                    simulationList.cacheRequest(pa + k, QueryType.r, 8);
                    simulationList.cacheRequest(pb + x * 2, QueryType.r, 16);
                    simulationList.addVariable();
                    simulationList.multiplyVariable();
                    //pb += N;
                    simulationList.addVariable();

                    simulationList.addVariable(); // next iteration
                }
                //pc[x] = s;
                simulationList.initializeVariable();
                simulationList.cacheRequest(pc + 4 * x, QueryType.w, 32);

                simulationList.addVariable(); //next iteration
            }
            //pa += K;
            simulationList.addVariable();
            //pc += N;
            simulationList.addVariable();

            simulationList.addVariable(); //next iteration
        }

        System.out.printf(
                "LRU:\thit perc. %3.4f%%\ttime: %d\npLRU:\thit perc. %3.4f%%\ttime: %d\n",
                simulationList.getHitRate(0),
                simulationList.getTime(0),
                simulationList.getHitRate(1),
                simulationList.getTime(1)
        );
    }
}
```