Лабораторная работа №2	M3139	2022
Моделирование схем в Verilog	Матвеев Андрей Денисович	

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

Инструментарий и требования к работе: весь код пишется на языке Verilog, компиляция и симуляция – Icarus Verilog 10 и новее (полезные материалы: Verilog.docx). В отчёте нужно указать, какой версией вы пользовались (можно также приложить ссылку на онлайн-платформу). Использовать SystemVerilog допустимо, главное, чтобы код компилился под Icarus 10, 11 или 12. Далее в этом документе Verilog+SystemVerilog обозначается как Verilog.

Описание

Смоделировать систему “процессор-кэш-память” на языке описания Verilog. Решить задачу на подсчёт кэш попаданий с помощью написанной системы.

Вариант 2

Вычисление недостающих параметров системы

Для начала расскажу, как устроен кэш. Кэш делится на блоки, в моём случае блоков получилось 16. В каждом блоке есть кэш линии. Кэш линия имеет следующий вид (рисунок 1):

valid	dirty	tag	data
1	1	CACHE_TAG_SIZE	CACHE_LINE_SIZE

Рисунок 1 – Устройство кэш-линии

Мне были известны следующие параметры:

Размер кэша (CACHE_SIZE) = 1 Кб = 1024 байта

Размер кэш линии (CACHE_LINE_SIZE) = 32 байта

Кол-во бит под хранение индекса набора (блока) (CACHE_SET_SIZE) = 4 бита

Размер памяти (MEM_SIZE) = 1 Мбайт = 2^20 байт

Под хранение индекса блока мы отводим 4 бита, тогда всего таких блоков CACHE_SETS_COUNT = 2^4 = 16. Также, нетрудно найти количество кэш линий, оно равно CACHE_LINE_COUNT = CACHE_SIZE / CACHE_LINE_SIZE = 1024 / 32 = 32. Теперь найдём ассоциативность – количество кэш линий в одном блоке. Очевидно это число равно CACHE_WAY = CACHE_LINE_COUNT / CACHE_SETS_COUNT = 32 / 16 = 2. Теперь сосчитаем количество битов нужное для обращения к памяти. Размер памяти равен MEM_SIZE, тогда нам надо CACHE_ADDR_SIZE = log2(MEM_SIZE) = log2(2^20) = 20 битов. CACHE_OFFSET_SIZE – это число на которое надо сдвинутся в кэш линии, чтобы получить данные, т. к. размер кэша равен 32 байта, тогда CACHE_OFFSET_SIZE = log2(32) = 5. Также, в кэш линии хранится тэг, на него уходит все оставшиеся биты, т. е. CACHE_TAG_SIZE = CACHE_ADDR_SIZE – CACHE_OFFSET_SIZE – CACHE_SET_SIZE = 20 – 5 – 4 = 11.

Теперь посчитаем размер шин. По шине A1 передаётся либо tag + set, либо offset, поэтому размер шины A1 равен ADDR1_BUS_SIZE = max(CACHE_TAG_SIZE + CACHE_SET_SIZE, CACHE_SET_SIZE) = 11 + 4 = 15 бит. По шине A2 передаются адреса без части offset, т.е. ADDR2_BUS_SIZE = CACHE_ADDR_SIZE – CACHE_OFFSET_SIZE = 20 – 5 = 15 бит. По шине C1 передаётся одна из 9-ти команд, поэтому CTR1_BUS_SIZE = log2(9) = 4 бита. По шине C2 передаётся одна из 4-ёх команд, поэтому CTR2_BUS_SIZE = log2(4) = 2 бита.

В итоге получаем:

- **MEM_SIZE** = 2^20 байт
- **CACHE_SIZE** = 1024 байт
- **CACHE_LINE_SIZE** = 32 байта
- **CACHE_LINE_COUNT** = 32
- **CACHE_WAY** = 2
- **CACHE_SETS_COUNT** = 16
- **CACHE_TAG_SIZE** = 11 бит
- **CACHE_SET_SIZE** = 4 бита
- **CACHE_OFFSET_SIZE** = 5 битов
- **CACHE_ADDR_SIZE** = 20 бит
- **ADDR1_BUS_SIZE =** 15 бит
- **ADDR2_BUS_SIZE =** 15 бит
- **CTR1_BUS_SIZE =** 4 бита
- **CTR2_BUS_SIZE =** 2 бита

Аналитическое решение задачи на C++

В этом задании надо было посчитать процент кэш попаданий и количество тактов после исполнения алгоритма перемножения матриц. Что

такое кэш попадание? Когда мы кэшу отправляем запрос на чтение адреса, то сначала кэш в своём внутреннем хранилище ищет данные, и если не находит, то уже запрашивает эти данные у оперативки. Так вот, кэш промах – это когда кэш не находит во внутреннем хранилище данные, иначе происходит кэш попадание.

```
int mem[M * K + K * N * 2 + M * N * 4];
```

Здесь я объявляю массив, моделирующий оперативную память. В нём последовательно хранятся три матрицы a, b, c. Почему я написал K * N * 2, а не K * N, потому что матрица b состоит их 2-ух байтовых чисел, поэтому и занимает в K * N * 2 байт. Такая же ситуация с M * N * 4.

```
int data[SET_COUNT][LINE_COUNT / SET_COUNT][LINE_SIZE];

int line_tag[SET_COUNT][LINE_COUNT / SET_COUNT];

int valid[SET_COUNT][LINE_COUNT / SET_COUNT];

int dirty[SET_COUNT][LINE_COUNT / SET_COUNT];

int time[SET_COUNT][LINE_COUNT / SET_COUNT];
```

data, line_tag, valid, dirty, time – массивы, содержащие в себе соответствующую информацию о кэш линии. Например, в valid[i][j] хранится бит “валидности” для j-ой кэш линии в i-ом блоке. Или же, например, в data[i][j][k] значение, записанное в k-ую позицию j-ой кэш линии i-ого блока.

Основные функции, которые я использую – read_cache_line и write32(). Приведу код функции read_cache_line.

```
int* read_line(int address) {

    int was_address = address;

    full++;

    timer++;

    int offset = address & ((1 << 5) - 1);

    address >>= 5;

    int set = address & ((1 << 4) - 1);

    address >>= 4;

    int tag = address & ((1 << 11) - 1);

    address = was_address;
```

```
    for (int i = 0; i < CACHE_WAY; i++) {
        if (line_tag[set][i] == tag && valid[set][i] == 1) {
            tacts += 6;
            cache_hits++;
            time[set][i] = timer;
            return data[set][i];
        }
    }

    tacts += 4;
    int start = address >> OFFSET << OFFSET;
    int line_index = get_max_time(set);
    if (dirty[set][line_index]) {
        push(set, line_index);
    }
    write_in_line(set, line_index, start, tag);
    return data[set][line_index];
}
```

Каждый адрес интерпретируется кэшем по-особенному (рисунок снизу)

tag	set	offset
CACHE_TAG_SIZE	CACHE_SET_SIZE	CACHE_OFFSET_SIZE

Рисунок 2 – Интерпретация адреса кэшем

В первых 5-ти битах хранится offset – место в линии, в котором хранится нужный байт. В следующих 4-х битах хранится информация о номере блока, в который попал этот адрес. В остальных битах хранится тег. Это то число, которое нам поможет искать кэш линию, в которой хранится наш байт. Дело в том, что каждой кэш линии сопоставляется тег, поэтому, чтобы найти нужную кэш линию, в которой записан наш байт мы должны перебрать все кэш-линии и сравнить теги. Если тег какой-то кэш-линии совпадает с тегом адреса, значит в этой кэш-линии может хранится требуемый байт.

В первой половине функции я просто достаю информацию из адреса offset, set и tag. Во второй половине я перебираю все кэш-линии в нужном блоке и, как уже говорил выше, сравниваю тег кэш-линии и тег адреса, если они совпадают, значит в этой кэш линии на позиции offset лежит требуемый байт, значит произошло кэш попадание, поэтому мы увеличиваем cache_hits на 1. Также надо не забыть установить время последнего обращения к кэш-линии. Строка time[set][i] = timer; как раз это и делает.

Что же делать, если кэш попадания не произошло? В этом случае придется прочитать адрес из памяти. Вместе с адресом прочитаем все соседние адреса, которые попадут в одну кэш линию с нашим адресом. Куда записать новую кэш линию. На этот счёт есть множество различных эвристик, но в нашем случае используется LRU. LRU (Least Recently Used) нам говорит, что надо выбрать ту кэш-линию, к которой давнее всего не было обращения. Именно для этого я сделал функцию get_max_time, возвращающую кэш линию, к которой дольше всего не было обращения. Снизу реализация.

```
int get_max_time(int set) {
    int min = 0;
    int line_index = -1;
    for (int i = 0; i < CACHE_WAY; i++) {
        if (timer - time[set][i] >= min) {
            min = timer - time[set][i];
            line_index = i;
        }
    }
    return line_index;
}
```

Вернёмся у функции read_line. Что за переменная start? Как я уже выше сказал, если происходит кэш промах, то из оперативки мы читаем не один байт, а всю кэш линию. Тогда start – это самый первый адрес, который попадёт в эту кэш линию. Соответственно start + CACHE_LINE_SIZE – 1 – это последний адрес, который попадёт в кэш-линию.

Что значит следующая строчка:

```
        if (dirty[set][line_index]) {
            push(set, line_index);
        }
```

Бит dirty несёт важную информацию о кэш линии. Dirty означает, что кэш-линия хранит изменённые данные, которые ещё не записаны в память. Кончено, если это бит равен 1, то мы должны записать данные в этой кэш линии в оперативку, за это отвечает функция push.

Функция push:

```
void push(int set, int line) {
    for (int offset = 0; j < CACHE_LINE_SIZE; offset++) {
        int val = data[set][line][offset];
        int address = offset | (set << 5) | (line_tag[set][line] << 9);
        mem[address] = val;
    }
    dirty[set][line] = 0;
    valid[set][line] = 0;
}
```

Вспоминая рисунок 2, мы сразу понимаем, как восстановить реальный адрес в памяти. В данном случае я перебираю offset адреса. Так как set и тег мне известны, то я могу собрать полный адрес байта и записать его в память.

Возвращаемся к функции read_cache_line. Осталось обсудить только последнюю строчку: write_from_ram_to_line(set, line_index, start, tag); Здесь я вызываю функцию write_from_ram_to_line, которая записывает в кэш-линию все адреса в промежутке [start, start + CACHE_LINE_SIZE).

Вторая важная функция – write32.

```
void write32(int address, int value) {
    int was_address = address;
    timer++;
    full++;
    int offset = address & ((1 << 5) - 1);
    address >>= 5;
    int set = address & ((1 << 4) - 1);
    address >>= 4;
```

```
int tag = address & ((1 << 11) - 1);
address = was_address;



for (int line = 0; line < CACHE_WAY; i++) {
    if (line_tag[set][ line] == tag) {
        tacts += 6;
        cnt++;
        data[set][line][offset] = value & ((1 << 8) - 1);
        data[set][line][offset + 1] = (value >> 8) & ((1 << 8) - 1);
        data[set][line][offset + 2] = (value >> 16) & ((1 << 8) - 1);
        data[set][line][offset + 3] = (value >> 24) & ((1 << 8) - 1);
        time[set][line] = timer;
        dirty[set][line] = 1;
        return;
    }
}


tacts += 4;
mem[address] = value;
int line_index = get_max_time(set);
if (dirty[set][line_index])
    push(set, line_index);
int start = address >> OFFSET << OFFSET;
```

```
    write_from_ram_to_line(set, line_index, start, tag);
}
```

Код этой функции чем-то напоминает read_cache_line. В первой части мы достаём offset, tag и set из адреса. Во второй части мы перебираем все кэш линии из нужного блока и сравниваем тэги. Если оказалось так, что тег кэш-линии совпал с тегом адреса, то происходит кэш попадание и я записываю в эту кэш линию 4-байтный value. Так как он занимает 4 байта, то мы должны записать отдельно каждый байт. Также, надо не забыть установить бит dirty, т. к. мы записали изменения только в кэш, а в оперативку не записали.

Что же делать, если произошёл кэш промах? Здесь стратегия точно такая же, как и в read_cache_line. Эвристикой LRU ищем кэш-линию и её замещаем.

Надо ещё сказать как работают функции read8 и read16.

```
int read8(int address) {
    int* line = read_line(address);
    int offset = address & ((1 << OFFSET) - 1);
    return line[offset];
}
int read16(int address) {
    int* line = read_line(address);
    int offset = address & ((1 << OFFSET) - 1);
    return line[offset] | (line[offset + 1] << 8);
}
```

В обоих функциях мы вызываем read_line, она возвращает нам всю кэш-линию, содержащую наш адрес. В случае read8 мы считаем offset нашего адреса и возвращаем значение на позиции offset в кэш-линии. В read16 мы просто берём два последовательных байта и “склеиваем” их.

Ответ на задачу:

Количество кэш попаданий: 213761

Количество обращений к кэшу: 249600

Процент кэш попаданий: 85.6414%

Количество тактов: 6242660

Моделирование заданной системы на Verilog

Модель в verilog’е состоит из двух файлов testbench.sv и design.sv. В design.sv я сделал 3 модуля: CPU, Cache и MemCTR. Все шины объявлены в файле testbench.sv как тип reg, потому что так с ними проще работать. Модули не имеют никаких параметров. Если какому-то модулю надо обратиться к шине x, то просто пишем test.x. Вот как объявлены шины в модуле testbench.sv (рисунок 3)

Рисунок 3 – Объявление шин

Также, для увеличения читабельности кода есть enum, для командных шин C1 и C2 (рисунок 4).

```
typedef enum {C2_NOP,
              C2_READ_LINE,
              C2_WRITE_LINE,
              C2_RESPONSE,
              C1_WRITE32,
              C1_WRITE16,
              C1_WRITE8,
              C1_INVALIDATE_LINE,
              C1_READ32,
              C1_READ16,
              C1_READ8,
              C1_NOP,
              C1_RESPONSE
             } light_1;
```

Рисунок 4 – enum для командных шин C1 и C2

Рассмотрим самый простой модуль CPU.

```
module CPU();
  always @(posedge test.CLK) begin
    case(test.c1)
      C1_RESPONSE: begin
        $display("address = %d, value = %d", test.a1, test.d1);
        test.c1 = C1_NOP;
      end
    endcase
  end
```

Рисунок 5 – модуль CPU

Что делает always? Команды внутри always начнут выполняться **последовательно**, как только условие внутри скобок (в данном случае posedge test.CLK) выполняется. Условие posedge выполняется в тот момент времени, когда test.CLK переходит с нуля на единицу.

Так как единственная команда CPU – это C1_RESPONSE, то на неё и будем отвечать. C1_RESPONSE обозначает, что кэш прочитал байт по адресу и его значение лежит на шине D1, тогда мы его просто выводим. Также устанавливаем шину C1 в положение C1_NOP, т. к. никаких команд по этой шине больше не должно быть.

Следующий модуль, который мы рассмотрим – MemCTR.

```
parameter MEM_SIZE = 1 << 20;
reg[7:0] mem[0:MEM_SIZE - 1];

integer passedRead = 0;
integer passedWrite = 0;

integer start, i;
```

Рисунок 7 – Поля модуля MemCTR

MEM_SIZE – это размер (в байтах) кэша, mem – это собственно сама оперативная память, в каждой ячейке которой хранится один байт. Остальные параметры нужны в блоке “always” в качестве локальных, но Verilog запрещает создавать переменные в “always”, поэтому пришлось их создать в самом модуле.

Главный блок always (рисунок 8).

```
always @(posedge test.CLK) begin
  if (test.c2 == C2_WRITE_LINE) begin
    mem[test.a2 + passedWrite] = test.d2 & ((1 << 8) - 1);
    mem[test.a2 + passedWrite + 1] = (test.d2 >> 8) & ((1 << 8) - 1);
    passedWrite += 2;
    if (passedWrite == Cache.CACHE_LINE_SIZE / test.DATA2_BUS_SIZE) begin
      passedWrite = 0;
    end
  end
  else if (test.c2 == C2_READ_LINE) begin
    start = test.a2 << Cache.CACHE_OFFSET_SIZE;
    for (i = start; i < Cache.CACHE_LINE_SIZE + start; i += 2) begin
      test.d2 = mem[i] | (mem[i + 1] << 8);
      test.c2 = C2_RESPONSE;
      @(negedge test.CLK);
    end
  end
end
```

Рисунок 8 – основной блок always модуля MemCTR

В первом блоке always заложена основная функциональность. На каждом такте синхронизации мы смотрим, какая команда сейчас лежит на шине C2, если это C2_WRITE_LINE, то просто записываем значение по адресу, если это C2_READ_LINE, то мы читаем значение по адресу. Почему тогда команда называется WRITE_**LINE**, а пишем мы только байтами? Дело в том, что размер шины D2 очень мал – 2 байта, поэтому для того, чтобы записать всю линию надо вызвать этот блок always 32/2 = 16 раз. Для этого создана локальная переменная passedWrite, которая указывает на следующий байт в кэш-линии, в который надо записать.

C2_READ_LINE работает схожим образом. По шине A2 нам приходит адрес без части offset, поэтому первый байт в этой кэш линии имеет адрес от $A2 * 2^5$, где 5 = CACHE_OFFSET_SIZE. После записи байта на шину D2, мы меняем команду с C2_READ_LINE на C2_RESPONSE, сообщая таким образом, что мы прочитали два байта, и они лежат на шине D2. В следующий такт Cache запишет эти два байта, а на следующей итерации for'а мы продолжим читать байты. @(negedge test.CLK) как раз и нужен для пропуска такта.

Посмотрим на остальные блоки always.

```
always @(posedge test.M_DUMP) begin
  for (i = 0; i < MEM_SIZE; i += 1) begin
    $display("[%d] %d", i, mem[i]);
  end
end

integer SEED = 225526;
always @(posedge test.RESET) begin
  for (i = 0; i < MEM_SIZE; i += 1) begin
    mem[i] = $random(SEED)>>16;
  end
end
```

Рисунок 9 – второстепенные блоки always модуля MemCTR

Во втором блоке always нет ничего интересного, просто выводим все байты памяти.

Третий блок перезапускает память, т. е. заполняет её рандомными байтами.

Следующий модуль самый обширный и самый сложный – Cache. Сначала посмотрим на поля этого модуля (рисунок 6).

```
reg isHit = 0;

parameter CACHE_SIZE = 1024;
parameter CACHE_LINE_SIZE = 32;
parameter CACHE_LINE_COUNT = 32;
parameter CACHE_SET_COUNT = 16;
parameter CACHE_WAY = 2;

parameter CACHE_ADDR_SIZE = 20;
parameter CACHE_SET_SIZE = 4;
parameter CACHE_OFFSET_SIZE = 5;
parameter CACHE_TAG_SIZE = 20 - CACHE_SET_SIZE - CACHE_OFFSET_SIZE;

reg[7:0] data[0:CACHE_SET_COUNT - 1][CACHE_WAY][CACHE_LINE_SIZE];
reg[CACHE_TAG_SIZE - 1:0] lineTag[0:CACHE_SET_COUNT - 1][CACHE_WAY];
reg valid[0:CACHE_SET_COUNT - 1][CACHE_WAY];
reg dirty[0:CACHE_SET_COUNT - 1][CACHE_WAY];
integer lastModify[0:CACHE_SET_COUNT - 1][CACHE_WAY];

integer timer = 0;
integer full = 0;
integer hits = 0;
```

Рисунок 9 – поля модуля Cache

Поля с типом parameter очевидны, их размер мы уже считали. Поля data, lineTag, valid, dirty, lastModify играют такую же роль, как и в модели на C++. Поле timer считает “время”, прошедшее с самого первого обращения к кэшу, оно нужно для алгоритма LRU. Поле full считает количество обращений кэшу. Поле hits считает количество кэш попаданий. У этого модуля есть ещё поля, но они играют роль локальных переменных либо блоков always, либо блоков task.

```
always @(posedge test.C_DUMP) begin
  integer setIndex, lineIndex, offset;
  for (setIndex = 0; setIndex < CACHE_SET_COUNT; setIndex += 1) begin
    $display("Set %d:", setIndex);
    for (lineIndex = 0; lineIndex < CACHE_WAY; lineIndex += 1) begin
      $display("  Cache_line %d:", lineIndex);
      for (offset = 0; offset < CACHE_LINE_SIZE; offset += 1) begin
        $display("    %d", offset);
    end
    end
  end
end

always @(posedge test.RESET) begin
  integer setIndex, lineIndex;
  for (setIndex = 0; setIndex < CACHE_SET_COUNT; setIndex += 1) begin
    for (lineIndex = 0; lineIndex < CACHE_WAY; lineIndex += 1) begin
      valid[setIndex][lineIndex] = 0;
      dirty[setIndex][lineIndex] = 0;
      lastModify[setIndex][lineIndex] = 0;
    end
  end
end
```

Рисунок 10 – второстепенные блоки always модуля Cache

Первый always выводит информацию о кэше в удобном для человека формате. Второй блок always перезапускает кэш, т. е. Просто устанавливает бит valid в ноль, тем самым говоря, что в данной кэш-линии лежит мусор, также изменяя бит dirty и lastModify.

Следующий блок кода, который мы посмотрим – task'и. Начнём с task'и read_byte1 (рисунок 11), которая пытается по заданному адресу прочитать байт, если он есть в кэше.

```
task read_byte1(integer address);
  timer++;
  isHit = 0;

  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

  for (i = 0; i < CACHE_WAY; i += 1) begin
    if (lineTag[set][i] == tag && valid[set][i] == 1) begin
      isHit = 1;
      lastModify[set][i] = timer;
      test.d1 = data[set][i][offset];
      test.c1 = C1_RESPONSE;
    end
  end
endtask
```

Рисунок 11 – task read_byte1 в модуле Cache

Вспомнив рисунок 2, с помощью битовых операций достанем из адреса его tag, set и offset. Также установим флаг isHit в 0, этим флагом мы будем сообщать было ли кэш попадание или нет.

В цикле for мы перебираем кэш линию, так как их всего две, то много время это не займёт. В цикле проверяем совпадает ли тег адреса и тег кэш-линии, также надо проверить бит valid, т. к. если он равен 0, то в кэш линии лежит мусор. Если же эта проверка прошла, то можно на шину D1 записать значение по заданному адресу, а на шину C1 послать C1_RESPONSE.

Ещё есть похожая task'а: read_byte2. Она делает всё то же самое, но читает два байта, вместо одного.

Следующая task’a самая важная: read_cache_line (рисунок 12).

```
task read_cache_line(integer address);
  timer++;

  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

  getMaxLine(set);

  setResponse = set;
  lineResponse = getMaxLineResult;
  indexResponse = 0;

  lineTag[set][lineResponse] = tag;
  valid[set][lineResponse] = 1;
  dirty[set][lineResponse] = 0;
  lastModify[set][lineResponse] = timer;

  test.c2 = C2_READ_LINE;
  test.a2 = test.a1 >> CACHE_OFFSET_SIZE;
  test.c1 = C1_NOP;
endtask
```

Рисунок 14 – task read_cache_line модуля Cache

Этот блок кода читает кэш-линию из памяти. Для этого мы вычисляем offset, set и tag. Также, надо найти линию, в которую мы запишем кэш-линию, для этого у нас есть task’a getMaxLine, которая находит линию с максимальным временем простоя (LRU). Затем мы инициализируем нашу кэш линию, т. е. устанавливаем биты valid, dirty, и т.д. В конце, мы устанавливаем шину C2 в положение C2_READ_LINE. На шину A2 отправляем часть адреса без offset’a, а на шину C1 отправляем C1_NOP, т. к. пока нам MemCTR не ответила мы ничего не должны делать.

Следующая немаловажная task'а – write_byte (рисунок 15). Из названия понятно, что она пишет один байт.

```
task write_byte(integer address, integer value);
  isHit = 0;
  timer++;

  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

      for (i = 0; i < CACHE_WAY; i += 1) begin
        if (lineTag[set][i] == tag) begin
          isHit = 1;
          data[set][i][offset] = value;
          lastModify[set][i] = timer;
          dirty[set][i] = 1;
          test.c1 = C1_NOP;
        end
      end

      if (isHit == 0) begin
        MemCTR.mem[address] = value;

        getMaxLine(set);
        lineResponse = getMaxLineResult;

        if (dirty[set][lineResponse] == 1) begin
          push(set, lineResponse);
        end

        setResponse = set;
        indexResponse = 0;

        lineTag[set][lineResponse] = tag;
        valid[set][lineResponse] = 1;
        dirty[set][lineResponse] = 1;
        lastModify[set][lineResponse] = timer;

        test.c2 = C2_READ_LINE;
        test.a2 = address >> CACHE_OFFSET_SIZE;
        test.c1 = C1_NOP;
      end
endtask
```

Рисунок 15 – task write_byte в модуле Cache

Изначальные действия аналогичны read_byte1. Мы ищем кэш-линию с таким же тегом, если такая кэш-линия есть, то мы спокойно в неё записываем новое значение. Иначе (isHit = 0) мы должны записать в память по этому адресу новое значение (MemCTR.mem[address] = value), а потом прочитать целиком эту линию и записать её в одну из кэш-линий (в какую именно, определит LRU). Также, в кэш-линии, в которую мы хотим записать данные, могут быть dirty-данные, т. е. значения в этой кэш-линии были записаны в кэш, но не были

записаны в память. Если так произошло, то эти данные надо обязательно записать в оперативку. Для этого мы вызываем task'у push (рисунок 16), т. е. мы проталкиваем значения из кэша в оперативку.

```
  task push(integer whichSet, integer whichLine);
    for (i = 0; i < CACHE_LINE_SIZE; i++) begin
      MemCTR.mem[data[whichSet][whichLine][i] >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE) <<
(CACHE_OFFSET_SIZE + CACHE_SET_SIZE) | (whichSet << CACHE_OFFSET_SIZE) | i] = data[whichSet]
[whichLine][i];
    end
  endtask
```

Рисунок 16 – task push в модуле Cache

Здесь нет ничего сложного, простой битовой арифметикой считаем адрес в памяти и в него записываем нужное значение.

Следующий блок always (рисунок 17) отвечает за общение с MemCTR

```
always @(posedge test.CLK) begin
  case (test.c2)
    C2_RESPONSE: begin

      data[setResponse][lineResponse][indexResponse] = test.d2 & ((1 << 8) - 1);
      data[setResponse][lineResponse][indexResponse + 1] = (test.d2 >> 8) & ((1 << 8) - 1);

      indexResponse += 2;

      if (indexResponse == CACHE_LINE_SIZE) begin
        indexResponse = 0;
        test.c2 = C2_NOP;
      end
    end
  endcase
end
```

Рисунок 17 – блок always для ответа MemCTR

Так как единственная команда, которая нам может прийти это C2_RESPONSE, то мы проверяем пришла ли она или нет. Если она пришла, то это значит, что MemCTR прочитал очередные два байта из памяти и мы должны записать их в кэш линию, для этого хранится локальная переменная indexResponse, индекс следующего значения. Когда мы прочитали всю кэш линию (indexResponse == CACHE_LINE_SIZE), это значит, что больше MemCTR не будет посылать нам команду C2_REPONSE и поэтому шина C2 устанавливается в C2_NOP, т. к. общение с MemCTR закончено.

Следующий блок always отвечает за общение CPU и Cache. Этот блок слишком большой, поэтому я его разобью на две части. Первая часть (рисунок 18) отвечает за команды read и invalidate, а вторая (рисунок 19) – за write.

```
always @(posedge test.CLK) begin
  case(test.c1)
    C1_READ8: begin
      full++;
      read_byte1(test.a1);
      if (isHit == 0) begin
        read_cache_line(test.a1);
        #200 read_byte1(test.a1);
      end
      else begin
        hits++;
      end
    end
    C1_READ16: begin
      full++;
      read_byte2(test.a1);
      if (isHit == 0) begin
        read_cache_line(test.a1);
        #200 read_byte2(test.a1);
      end
      else begin
        hits++;
      end
    end
    C1_READ32: begin
      // No realisation, because d1 is only 16 bit, so No reason to use this Command
    end
    C1_INVALIDATE_LINE: begin
      offset = test.a1 & ((1 << CACHE_OFFSET_SIZE) - 1);
      set = ((test.a1 >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = test.a1 >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

      for (i = 0; i < CACHE_WAY; i += 1) begin
        if (lineTag[set][i][offset] == tag) begin
          valid[set][i] = 0;
        end
      end
    end
```

Рисунок 18 – Первая часть блока always, отвечающего за общение CPU и Cache

Для начала рассмотрим случай, когда нам пришла команда INVALIDATE_LINE, тогда мы просто for-ом пробегаемся по всем линиям в set’е данного адреса и устанавливаем бит valid в 0.

Сразу заметим, что C1_READ32 реализовывать бессмысленно, т. к. размер шины 2 байта и на неё просто не поместится 4-х байтное значение. Реализации C1_READ8 и C1_READ16 практически одинаковы, поэтому рассмотрим C1_READ8. Сначала мы увеличиваем переменную full, и пытаемя прочитать байт из кэша. Если произошёл кэш промах, т. е. isHit == 0, то нам придется прочитать кэш-линию из памяти, для этого мы вызываем task read_cache_line, ждём 200 тактов, т. к. обращение к памяти работает очень долго. И заново читаем байт опять из кэша.

Теперь рассмотрим вторую часть этого always блока (рисунок 19)

```
    C1_WRITE8: begin
      full++;
      write_byte(test.a1, test.d1);
      if (isHit == 1) begin
        hits++;
      end
    end
    C1_WRITE16: begin
      full++;
      write_byte(test.a1, test.d1 & ((1 << 8) - 1));
      if (isHit == 1) begin
        hits++;
      end
      write_byte(test.a1 + 1, (test.d1 >> 8) & ((1 << 8) - 1));
    end
    C1_WRITE32: begin
      // No realisation, because d1 is only 16 bit, so No reason to use this Command
    end
  endcase
```

Рисунок 19 – Вторая часть always блока модуля Cache, отвечающая за write

Опять же, реализовать C1_WRITE32 мы не можем, т. к. нам это не позволяет сделать размер шины. Рассмотрим реализацию C1_WRITE8. Здесь нет ничего необычного, мы просто увеличиваем счётчик обращений к кэшу (full++), а потом вызываем write_byte. И если произошло кэш попадание, то увеличиваем соответствующий счётчик (hits++). C1_WRITE16 работает похожим образом, но пишет отдельно каждый байт.

Теперь рассмотрим модуль test, в котором расположен сам алгоритм перемножения матриц. Начинается модуль с объявления шин, которые расположены в глобальном доступе (рисунок 20)

```
parameter ADDR1_BUS_SIZE = 15;
parameter ADDR2_BUS_SIZE = 15;
parameter DATA1_BUS_SIZE = 16;
parameter DATA2_BUS_SIZE = 16;
parameter CTR1_BUS_SIZE = 4;
parameter CTR2_BUS_SIZE = 2;

reg[DATA1_BUS_SIZE - 1:0] d1, d2;
reg[ADDR1_BUS_SIZE - 1:0] a1, a2;
reg[CTR1_BUS_SIZE - 1:0] c1;
reg[CTR2_BUS_SIZE - 1:0] c2;
reg C_DUMP, M_DUMP, RESET, CLK = 0;

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

Рисунок 20 – поля модуля test

В первой части объявляются размеры шин, во второй части объявляются сами шины, которые мы использовали в остальных модулях. В третьем блоке просто объявляются размеры матрицы.

Также был добавлен простой блок always, который каждый такт переключает синхронизацию (рисунок 21)

```
always #1 begin
  CLK = ~CLK;
end
```

Рисунок 21 – always, переключающий CLK

Также было сделано несколько task'ов, чтобы читать и писать байты (Рисунок 22)

```
task read8(integer address);
  c1 = C1_READ8;
  a1 = address;
  #300 read8Result = d1;
endtask

task read16(integer address);
  c1 = C1_READ16;
  a1 = address;
  #300 read16Result = d1;
endtask

task write32(integer address, integer value);
  c1 = C1_WRITE16;
  a1 = address;
  d1 = value & ((1 << 16) - 1);
  #300;
  c1 = C1_WRITE16;
  a1 += 2;
  d1 = (value >> 16) & ((1 << 16) - 1);
  #300;
  Cache.full--;
  Cache.hits--;
  Cache.tacts -= 6;
endtask
```

Рисунок 22 – task'и в модуле test

read8 принимает адрес и записывает в переменную read8Result результат. Принцип работы достаточно простой, 300 тактов ждём, потому что мог быть кэш промах. Аналогично работает read16. Отдельного внимания заслуживает write32. Размер шины D1 только 2 байта, поэтому придётся в два захода писать значение. Но по факту это должно происходить в одно обращение к кэшу,

поэтому надо вычесть одно обращение к кэшу и одно кэш попадание, а также 6 тактов.

Теперь перейдём к блоку initial (рисунок 23)

```
  initial begin
    c1 = C1_NOP;
    c2 = C2_NOP;
    #1 RESET = 0;
    #1 RESET = 1;

    #100;
    for (ii = 0; ii < M; ii += 1) begin
      for (jj = 0; jj < K; jj += 1) begin
        $write("[%d, %d]", MemCTR.mem[ii * K + jj], ii * K + jj);
      end
      $write("\n");
    end
    $display("-----------");

    #100;
    for (ii = 0; ii < K; ii += 1) begin
      for (jj = 0; jj < N * 2; jj += 2) begin
        res = MemCTR.mem[M * K + ii * N * 2 + jj] | (MemCTR.mem[M * K + ii * N *
2 + jj + 1] << 8);
        $write("[%d, %d]",res, M * K + ii * N * 2 + jj);
      end
      $write("\n");
    end
    $display("-----------");
```

Рисунок 23 – начало блока initial в модуле test

Изначально проставим на все командные шины NOP’ы, чтобы они случайно ничего не наделали.

#1 RESET = 0;

#1 RESET = 1;

Эти две строки задействуют условие @(posedge test.RESET) и кэш с памятью перезагрузятся (память заполнится рандомными числами).

В следующем (рисунок 24) куске кода заранее считаются такты, кроме read и write

```
Cache.tacts += 7; // For initializating variables
Cache.tacts += M * N * (K - 1) + M * (N - 1) + (M - 1); // for jump instructions
Cache.tacts += M * N * K * 5; // for multiplication read8Result * read16Result
Cache.tacts += M * N * K; // for pb += N * 2
Cache.tacts += M * N * K * 2; // for pa + k and pb + x * 2
Cache.tacts += M * N * K + M * N + M; // for k += 1, x += 1 and y += 1
Cache.tacts += M * 2; // for pa += K and pc += N * 4
```

Рисунок 24 – Подсчёт тактов, кроме команд write и read

Наконец, мы дошли до самого перемножения матриц (рисунок 25).

```
pa = 0;
pc = M * K + K * N * 2;
for (y = 0; y < M; y += 1) begin
  for (x = 0; x < N; x += 1) begin
    pb = M * K;
    s = 0;
    for (k = 0; k < K; k += 1) begin
      read8(pa + k);
      read16(pb + x * 2);
      s += read8Result * read16Result;
      pb += N * 2;
    end
    write32(pc + x * 4, s);
  end
  pa += K;
  pc += N * 4;
end
```

Рисунок 25 – перемножение матриц в модуле test

read8 читает значение по адресу и записывает его в переменную read8Result. Аналогично работает read16. write32 пишет значение 4-байтное в адрес.

В самом конце (рисунок 26) мы выводим результат: количество кэш-попаданий, количество обращений к кэшу, отношение этих двух величин и количество тактов.

```
    temp = Cache.hits;
    #100 $display("%d %d %f %d", Cache.hits, Cache.full, temp / Cache.full,
Cache.tacts);
    $finish;
```

Рисунок 26 – Вывод окончательной информации

Воспроизведение задачи на Verilog.

Кэш попаданий: 213761

Обращений к кэшу: 249600

Процент попаданий: 0.856414

Количество тактов: 6242660

Сравнение полученных результатов.

Результаты совпали и это неудивительно, т. к. все обращения к кэшу являются детерминированными, это следует из эвристики поиска кэш-линии для замещения. В данном случае мы использовали LRU, которая работает детерминировано.

Листинг кода

Симмуляция на C++:

```
#include <iostream>

#define M 64
#define N 60
#define K 32
#define int8 char
#define int16 short
#define int unsigned int

const int OFFSET = 5;
const int LINE_COUNT = 1 << 5;
const int LINE_SIZE = 1 << 5;
const int TAG_SIZE = 11;
const int SET_COUNT = 1 << 4;
int mem[M * K + K * N * 2 + M * N * 4];
int data[SET_COUNT][LINE_COUNT / SET_COUNT][LINE_SIZE];
int line_tag[SET_COUNT][LINE_COUNT / SET_COUNT];
int valid[SET_COUNT][LINE_COUNT / SET_COUNT];
int dirty[SET_COUNT][LINE_COUNT / SET_COUNT];
int time[SET_COUNT][LINE_COUNT / SET_COUNT];
```

```
int timer = 0;
int full = 0;
int cnt = 0;
int tacts = 0;

void push(int set, int i) {
    for (int j = 0; j < LINE_SIZE; j++) {
        int val = data[set][i][j];
        int address = j | (set << 5) | (line_tag[set][i] << 9);
        mem[address] = val;
    }
    dirty[set][i] = 0;
    valid[set][i] = 0;
}

void write_from_ram_to_line(int set, int line_idx, int start, int tag) {
    tacts += 100;
    line_tag[set][line_idx] = tag;
    time[set][line_idx] = timer;
    for (int j = start; j < start + LINE_SIZE; j++) {
        data[set][line_idx][j % LINE_SIZE] = mem[j];
    }
    dirty[set][line_idx] = 1;
    valid[set][line_idx] = 1;
```

```
}

int get_max_time(int set) {
    int max = 0;
    int line_index = -1;
    for (int i = 0; i < LINE_COUNT / SET_COUNT; i++) {
        if (timer - time[set][i] >= max) {
            max = timer - time[set][i];
            line_index = i;
        }
    }
    return line_index;
}

int* read_line(int address) {
    int was_address = address;
    full++;
    timer++;
    int offset = address & ((1 << 5) - 1);
    address >>= 5;
    int set = address & ((1 << 4) - 1);
    address >>= 4;
    int tag = address & ((1 << 11) - 1);
    address = was_address;
```

```
    for (int i = 0; i < LINE_COUNT / SET_COUNT; i++) {
        if (line_tag[set][i] == tag && valid[set][i] == 1) {
            tacts += 6;
            cnt++;
            time[set][i] = timer;
            return data[set][i];
        }
    }
    tacts += 4;
    int start = address >> OFFSET << OFFSET;
    int line_index = get_max_time(set);
    if (dirty[set][line_index]) {
        push(set, line_index);
    }
    write_from_ram_to_line(set, line_index, start, tag);
    return data[set][line_index];
}

int read8(int address) {
    int* line = read_line(address);
    int offset = address & ((1 << OFFSET) - 1);
    return line[offset];
}
```

```
int read16(int address) {
    int* line = read_line(address);
    int offset = address & ((1 << OFFSET) - 1);
    return line[offset] | (line[offset + 1] << 8);
}

void write32(int address, int value) {
    int was_address = address;
    timer++;
    full++;
    int offset = address & ((1 << 5) - 1);
    address >>= 5;
    int set = address & ((1 << 4) - 1);
    address >>= 4;
    int tag = address & ((1 << 11) - 1);
    address = was_address;

    for (int i = 0; i < LINE_COUNT / SET_COUNT; i++) {
        if (line_tag[set][i] == tag) {
            tacts += 6;
            cnt++;
            data[set][i][offset] = value & ((1 << OFFSET) - 1);
            data[set][i][offset + 1] = (value >> 8) & ((1 << OFFSET) - 1);
            data[set][i][offset + 2] = (value >> 16) & ((1 << OFFSET) - 1);
```

```
            data[set][i][offset + 3] = (value >> 24) & ((1 << OFFSET) - 1);
            time[set][i] = timer;
            dirty[set][i] = 1;
            return;
        }
    }

    tacts += 4;
    mem[address] = value;
    int line_index = get_max_time(set);
    if (dirty[set][line_index])
        push(set, line_index);
    int start = address >> OFFSET << OFFSET;
    write_from_ram_to_line(set, line_index, start, tag);
}

void mmul()
{
    for (int i = 0; i < SET_COUNT; i++) {
        for (int j = 0; j < LINE_COUNT / SET_COUNT; j++) {
            valid[i][j] = 0;
            dirty[i][j] = 0;
            for (int k = 0; k < LINE_SIZE; k++) {
                data[i][j][k] = -1;
```

```
        }
    }
}

tacts += 7; // For initializating variables
tacts += M * N * (K - 1) + M * (N - 1) + (M - 1); // for jump instructions
tacts += M * N * K * 5; // for multiplication read8Result * read16Result
tacts += M * N * K; // for pb += N * 2
tacts += M * N * K * 2; // for pa + k and pb + x * 2
tacts += M * N * K + M * N + M; // for k += 1, x += 1 and y += 1
tacts += M * 2; // for pa += K and pc += N * 4

int pa = 0;
int pc = M * K + K * N * 2;
for (int y = 0; y < M; y++)
{
    for (int x = 0; x < N; x++)
    {
        int pb = M * K;
        int s = 0;
        for (int k = 0; k < K; k++)
        {
            int left = read8(pa + k);
            int right = read16(pb + x * 2);
```

```
            s += left * right;
            pb += N * 2;
        }
        write32(pc + x * 4, s);
    }
    pa += K;
    pc += N * 4;
  }

  printf("%d %d %.10f\n%d\n", cnt, full, (double)cnt/full, tacts);
}



signed main() {
  srand(23);
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < K; j++) {
      mem[i * K + j] = rand() % 5;
      printf("%d ", mem[i * K + j]);
      fflush(stdout);
    }
    printf("\n");
  }
  printf("------\n");
```

```
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < N * 2; j += 2) {
            mem[M * K + i * N * 2 + j] = rand() % 5;
            mem[M * K + i * N * 2 + j + 1] = 0;
            printf("%d ", mem[M * K + i * N * 2 + j + 1] << 8 | mem[M * K + i * N * 2
+ j]);
            fflush(stdout);
        }
        printf("\n");
    }
    printf("------\n");
    mmul();
}
```

Программа на verilog’е (в одном файле testbench.sv)

```
typedef enum {C2_NOP,
              C2_READ_LINE,
              C2_WRITE_LINE,
              C2_RESPONSE,
             C1_WRITE32,
              C1_WRITE16,
              C1_WRITE8,
              C1_INVALIDATE_LINE,
              C1_READ32,
              C1_READ16,
              C1_READ8,
              C1_NOP,
              C1_RESPONSE
             } light_1;

module test();
  parameter ADDR1_BUS_SIZE = 15;
  parameter ADDR2_BUS_SIZE = 15;
  parameter DATA1_BUS_SIZE = 16;
  parameter DATA2_BUS_SIZE = 16;
  parameter CTR1_BUS_SIZE = 4;
```

```
  parameter CTR2_BUS_SIZE = 2;

  reg[DATA1_BUS_SIZE - 1:0] d1, d2;
  reg[ADDR1_BUS_SIZE - 1:0] a1, a2;
  reg[CTR1_BUS_SIZE - 1:0] c1;
  reg[CTR2_BUS_SIZE - 1:0] c2;
  reg[7:0] read8Result;
  reg[15:0] read16Result;
  reg[31:0] read32Result;
reg C_DUMP, M_DUMP, RESET, CLK = 0;
integer x, y, k, pa, pc, pb, s;
  parameter M = 64;
  parameter N = 60;
  parameter K = 32;

  integer forDebug = 0;
  real temp;

  always #1 begin
    CLK = ~CLK;
  end

  task read8(integer address);
    c1 = C1_READ8;
```

```
  a1 = address;
  #300 read8Result = d1;
endtask

task read16(integer address);
  c1 = C1_READ16;
  a1 = address;
  #300 read16Result = d1;
endtask

task read32(integer address);
  read8(address);
  read32Result = read8Result;
  read8(address + 1);
  read32Result |= read8Result << 8;
  read8(address + 2);
  read32Result |= read8Result << 16;
  read8(address + 3);
  read32Result |= read8Result << 24;
endtask

task write8(integer address, integer value);
  c1 = C1_WRITE8;
  a1 = address;
```

```
  d1 = value;
  #300;
endtask

task write32(integer address, integer value);
  c1 = C1_WRITE16;
  a1 = address;
  d1 = value & ((1 << 16) - 1);
  #300;
  c1 = C1_WRITE16;
  a1 += 2;
  d1 = (value >> 16) & ((1 << 16) - 1);
  #300;
  Cache.full--;
  Cache.hits--;
  Cache.tacts -= 6;
endtask

integer ii, jj, res;

initial begin
  c1 = C1_NOP;
  c2 = C2_NOP;
  #1 RESET = 0;
```

```
    #1 RESET = 1;


    #100;
    for (ii = 0; ii < M; ii += 1) begin
      for (jj = 0; jj < K; jj += 1) begin
        $write("[%d, %d]", MemCTR.mem[ii * K + jj], ii * K + jj);
      end
      $write("\n");
    end
    $display("-----------");


    #100;
    for (ii = 0; ii < K; ii += 1) begin
      for (jj = 0; jj < N * 2; jj += 2) begin
        res = MemCTR.mem[M * K + ii * N * 2 + jj] | (MemCTR.mem[M * K + ii *
N * 2 + jj + 1] << 8);
        $write("[%d, %d]",res, M * K + ii * N * 2 + jj);
      end
      $write("\n");
    end
    $display("-----------");


    Cache.tacts += 7; // For initializating variables
    Cache.tacts += M * N * (K - 1) + M * (N - 1) + (M - 1); // for jump instructions
    Cache.tacts += M * N * K * 5; // for multiplication read8Result * read16Result
```

```
Cache.tacts += M * N * K; // for pb += N * 2
Cache.tacts += M * N * K * 2; // for pa + k and pb + x * 2
Cache.tacts += M * N * K + M * N + M; // for k += 1, x += 1 and y += 1
Cache.tacts += M * 2; // for pa += K and pc += N * 4

pa = 0;
pc = M * K + K * N * 2;
for (y = 0; y < M; y += 1) begin
  for (x = 0; x < N; x += 1) begin
    pb = M * K;
    s = 0;
    for (k = 0; k < K; k += 1) begin
      read8(pa + k);
      read16(pb + x * 2);
      s += read8Result * read16Result;
      pb += N * 2;
    end
    write32(pc + x * 4, s);
  end
  pa += K;
  pc += N * 4;
end

temp = Cache.hits;
```

```
    #100 $display("%d %d %f %d", Cache.hits, Cache.full, temp / Cache.full, Cache.tacts);
    $finish;
  end



endmodule




module CPU();
  always @(posedge test.CLK) begin
    case(test.c1)
      C1_RESPONSE: begin
        //$display("address = %d, value = %d", test.a1, test.d1);
        test.c1 = C1_NOP;
      end
    endcase
  end


endmodule


module Cache();
  reg isHit = 0;
```

```
  parameter CACHE_SIZE = 1024;

  parameter CACHE_LINE_SIZE = 32;

  parameter CACHE_LINE_COUNT = 32;

  parameter CACHE_SET_COUNT = 16;

  parameter CACHE_WAY = 2;


  parameter CACHE_ADDR_SIZE = 20;

  parameter CACHE_SET_SIZE = 4;

  parameter CACHE_OFFSET_SIZE = 5;

  parameter CACHE_TAG_SIZE = 20 - CACHE_SET_SIZE - CACHE_OFFSET_SIZE;


  reg[7:0] data[0:CACHE_SET_COUNT - 1][CACHE_WAY][CACHE_LINE_SIZE];

  reg[CACHE_TAG_SIZE - 1:0] lineTag[0:CACHE_SET_COUNT - 1][CACHE_WAY];

  reg valid[0:CACHE_SET_COUNT - 1][CACHE_WAY];

  reg dirty[0:CACHE_SET_COUNT - 1][CACHE_WAY];

  integer lastModify[0:CACHE_SET_COUNT - 1][CACHE_WAY];


  integer timer = 0;

  integer full = 0;

  integer hits = 0;

  integer tacts = 0;


  integer getMaxLineResult;
```

```
  integer setResponse, lineResponse, indexResponse;


  integer i, lineIndex;
  reg[CACHE_TAG_SIZE - 1:0] tag;
  reg[CACHE_SET_SIZE - 1:0] set;
  reg[CACHE_OFFSET_SIZE - 1:0] offset;
  reg flag;


  task push(integer whichSet, integer whichLine);
    for (i = 0; i < CACHE_LINE_SIZE; i++) begin
      MemCTR.mem[data[whichSet][whichLine][i] >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE) << (CACHE_OFFSET_SIZE + CACHE_SET_SIZE) | (whichSet << CACHE_OFFSET_SIZE) | i] = data[whichSet][whichLine][i];
    end
  endtask


  integer min;
  task getMaxLine(integer set);
    min = 0;
    for (i = 0; i < CACHE_WAY; i += 1) begin
      if (timer - lastModify[set][i] >= min) begin
        min = timer - lastModify[set][i];
        getMaxLineResult = i;
      end
```

```
  end
endtask

task read_byte1(integer address);
  timer++;
  isHit = 0;

  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

  for (i = 0; i < CACHE_WAY; i += 1) begin
    if (lineTag[set][i] == tag && valid[set][i] == 1) begin
      isHit = 1;
      lastModify[set][i] = timer;
      test.d1 = data[set][i][offset];
      test.c1 = C1_RESPONSE;
     end
  end
endtask

task read_byte2(integer address);
  timer++;
  isHit = 0;
```

```
  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

  for (i = 0; i < CACHE_WAY; i += 1) begin
    if (lineTag[set][i] == tag && valid[set][i] == 1) begin
      isHit = 1;
      lastModify[set][i] = timer;
      test.d1 = data[set][i][offset] | (data[set][i][offset + 1] << 8);
      test.c1 = C1_RESPONSE;
    end
  end
endtask

task read_cache_line(integer address);
  timer++;

  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

  getMaxLine(set);
```

```
  setResponse = set;
  lineResponse = getMaxLineResult;
  indexResponse = 0;

  lineTag[set][lineResponse] = tag;
  valid[set][lineResponse] = 1;
  dirty[set][lineResponse] = 0;
  lastModify[set][lineResponse] = timer;

  test.c2 = C2_READ_LINE;
  test.a2 = test.a1 >> CACHE_OFFSET_SIZE;
  test.c1 = C1_NOP;
endtask

task write_byte(integer address, integer value);
  isHit = 0;
  timer++;

  offset = address & ((1 << CACHE_OFFSET_SIZE) - 1);
  set = ((address >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
  tag = address >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

      for (i = 0; i < CACHE_WAY; i += 1) begin
        if (lineTag[set][i] == tag) begin
```

```
    isHit = 1;
    data[set][i][offset] = value;
    lastModify[set][i] = timer;
    dirty[set][i] = 1;
    test.c1 = C1_NOP;
  end
end

if (isHit == 0) begin
  MemCTR.mem[address] = value;

  getMaxLine(set);
  lineResponse = getMaxLineResult;

  if (dirty[set][lineResponse] == 1) begin
    push(set, lineResponse);
  end

  setResponse = set;
  indexResponse = 0;

  lineTag[set][lineResponse] = tag;
  valid[set][lineResponse] = 1;
  dirty[set][lineResponse] = 1;
```

```
        lastModify[set][lineResponse] = timer;


        test.c2 = C2_READ_LINE;
        test.a2 = address >> CACHE_OFFSET_SIZE;
        test.c1 = C1_NOP;
      end
  endtask


  always @(posedge test.CLK) begin
    case (test.c2)
      C2_RESPONSE: begin


      data[setResponse][lineResponse][indexResponse] = test.d2 & ((1 << 8) - 1);
      data[setResponse][lineResponse][indexResponse + 1] = (test.d2 >> 8) & ((1 <<
8) - 1);


      indexResponse += 2;


      if (indexResponse == CACHE_LINE_SIZE) begin
        indexResponse = 0;
        test.c2 = C2_NOP;
      end
    end
    endcase
  end
```

```
always @(posedge test.CLK) begin
  case(test.c1)
    C1_READ8: begin
      full++;
      read_byte1(test.a1);
      if (isHit == 0) begin
        read_cache_line(test.a1);
        #200 read_byte1(test.a1);
        tacts += 104;
      end
      else begin
        hits++;
        tacts += 6;
      end
    end
    C1_READ16: begin
      full++;
      read_byte2(test.a1);
      if (isHit == 0) begin
        read_cache_line(test.a1);
        #200 read_byte2(test.a1);
        tacts += 104;
      end
```

```
      else begin
        hits++;
        tacts += 6;
      end
    end
    C1_READ32: begin
      // No realisation, because d1 is only 16 bit, so No reason to use this Command
    end
    C1_INVALIDATE_LINE: begin
      offset = test.a1 & ((1 << CACHE_OFFSET_SIZE) - 1);
      set = ((test.a1 >> CACHE_OFFSET_SIZE) & ((1 << CACHE_SET_SIZE) - 1));
   tag = test.a1 >> (CACHE_OFFSET_SIZE + CACHE_SET_SIZE);

      for (i = 0; i < CACHE_WAY; i += 1) begin
        if (lineTag[set][i][offset] == tag) begin
          valid[set][i] = 0;
        end
      end
    end
    C1_WRITE8: begin
      full++;
      write_byte(test.a1, test.d1);
      if (isHit == 1) begin
        tacts += 6;
```

```
        hits++;
      end
      else begin
        tacts += 104;
      end
    end
    C1_WRITE16: begin
      full++;
      write_byte(test.a1, test.d1 & ((1 << 8) - 1));
      if (isHit == 1) begin
        hits++;
        tacts += 6;
      end
      else begin
        tacts += 104;
      end
      write_byte(test.a1 + 1, (test.d1 >> 8) & ((1 << 8) - 1));
    end
    C1_WRITE32: begin
      // No realisation, because d1 is only 16 bit, so No reason to use this Command
    end
  endcase
end
```

```
  always @(posedge test.C_DUMP) begin
    integer setIndex, lineIndex, offset;
    for (setIndex = 0; setIndex < CACHE_SET_COUNT; setIndex += 1) begin
      $display("Set %d:", setIndex);
      for (lineIndex = 0; lineIndex < CACHE_WAY; lineIndex += 1) begin
        $display("  Cache_line %d:", lineIndex);
        for (offset = 0; offset < CACHE_LINE_SIZE; offset += 1) begin
          $display("    %d", offset);
      end
      end
    end
  end

  always @(posedge test.RESET) begin
    integer setIndex, lineIndex;
    for (setIndex = 0; setIndex < CACHE_SET_COUNT; setIndex += 1) begin
      for (lineIndex = 0; lineIndex < CACHE_WAY; lineIndex += 1) begin
        valid[setIndex][lineIndex] = 0;
        dirty[setIndex][lineIndex] = 0;
        lastModify[setIndex][lineIndex] = 0;
      end
    end
  end
endmodule
```

```
module MemCTR();
  parameter MEM_SIZE = 1 << 20;
  reg[7:0] mem[0:MEM_SIZE - 1];

  integer passedRead = 0;
  integer passedWrite = 0;

  integer start, i;

  always @(posedge test.CLK) begin
    if (test.c2 == C2_WRITE_LINE) begin
      mem[test.a2 + passedWrite] = test.d2 & ((1 << 8) - 1);
      mem[test.a2 + passedWrite + 1] = (test.d2 >> 8) & ((1 << 8) - 1);
      passedWrite += 2;
      if (passedWrite == Cache.CACHE_LINE_SIZE / test.DATA2_BUS_SIZE)
begin
        passedWrite = 0;
      end
    end
    else if (test.c2 == C2_READ_LINE) begin
      start = test.a2 << Cache.CACHE_OFFSET_SIZE;
      for (i = start; i < Cache.CACHE_LINE_SIZE + start; i += 2) begin
        test.d2 = mem[i] | (mem[i + 1] << 8);
        test.c2 = C2_RESPONSE;
```

```
      @(negedge test.CLK);
    end
  end
end

always @(posedge test.M_DUMP) begin
  for (i = 0; i < MEM_SIZE; i += 1) begin
    $display("[%d] %d", i, mem[i]);
  end
end

integer SEED = 225526;
always @(posedge test.RESET) begin
  for (i = 0; i < MEM_SIZE; i += 1) begin
    mem[i] = $random(SEED)>>16;
  end
end

endmodule
```