Лабораторная работа №1	39	2022
Моделирование схем в Verilog	Ляпин Дмитрий Романович	

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

Описание

Вычислить недостающие параметры системы. Оценить эффективность кэша при заданных условиях (процент попаданий) и количество тактов на предложенной программе.

Вариант 3

Параметр	**Пояснение**	**Значение**
CACHE_SIZE	Размер кэша	2 Кбайт
CACHE_LINE_SIZE	Размер кэш-линии	16 байт
CACHE_TAG_SIZE	Кол-во бит под тэг адреса	8 бит
MEM_SIZE	Размер памяти	256 Кбайт

Параметры программы:

$MEM\_SIZE = 2^{18}Б$
$CACHE\_SIZE = 2^{11}Б$
$CACHE\_LINE\_SIZE = 2^{4}Б$
$CACHE\_TAG\_SIZE = 8$

$CACHE\_OFFSET\_SIZE = \log_2 CACHE\_LINE\_SIZE = 4$
$CACHE\_ADDR\_SIZE = \log_2 MEM\_SIZE = 18$
$CACHE\_SET\_SIZE = CACHE\_ADDR\_SIZE - CACHE\_OFFSET\_SIZE - CACHE\_TAG\_SIZE = 18 - 8 - 4 = 6$
$CACHE\_LINE\_COUNT = {}^{CACHE\_SIZE}/_{CACHE\_LINE\_SIZE} = 2^7$
$CACHE\_SETS\_COUNT = 2^{CACHE\_SET\_SIZE} = 2^6$
$CACHE\_WAY = {}^{CACHE\_LINE\_COUNT}/_{CACHE\_SETS\_COUNT} = 2$

Ширины шин: шина Control1 должна пропускать значения 0…7, значит, её размер 3 бита. Шина Data1 – 16 бит за раз. Шина Address1 в первый такт – тэг и сет, во второй – офсет. Её ширина max(8+6, 4) = 14 бит. Шина Control2 – значения 0…3, значит, размер 2 бита. Шина Data2 и Address2 тех же размеров (кроме того, что offset в память передавать нет смысла)

Комментарий:

Offset используется, чтобы определять позицию в линии. Addr – чтобы определять позицию в памяти. Соответственно, и то, и другое – двоичный логарифм соответствующих объёмов в байтах. Addr состоит из Tag, Set и Offset. Размеры Addr, Tag и Offset мы определили. Соответственно, оставшиеся биты определяют Set. Количество блоков – столько, сколько описывается полем Set. То есть, два в степени размер поля Set. Количество же линий – столько, сколько линий длиной ***CACHE_LINE_SIZE*** уместится в кэше размером ***CACHE_SIZE***. Ассоциативность – количество линий в одном блоке.

Аналитическое решение задачи с помощью симуляции на Kotlin

Реализуем модель памяти и кэша на котлине.

Код в приложении:

1) Константы, которыми можно настраивать систему. Единственный параметр из данных, зашитых в структуру – ассоциативность.
2) Объект Memory – оперативная память. Метод check предназначен для сравнения с образцом (при тестировании).
3) Вспомогательные функции – преобразования чисел в массив байтов и обратно, замена подотрезка в массиве и др.
4) Модель кэш-линии CacheLine и кэш-блока CacheSet.
5) Объект Cache – непосредственно кэш.
6) Функция test – тест работоспособности кэша.
7) Функция solve – счёт тактов и попаданий.

Политика замещения – LRU (last recently used), для этой цели в кэш-линии хранится «бит возраста» lastUpd.

Подсчёт попаданий реализован в кэше очевидным образом (hits++ при попадании, misses++ при промахе), а про такты поговорим подробнее.

Строки

```
var pa = 0
var pc = M * K + 2 * K * N // compile-time constant
var pb = M * K // compile-time constant
var s = 0
```

стоят по одной инициализации (все присваиваемые значения здесь – константы времени компиляции)

Каждый цикл `for (i in 0 until X)` – это `X` инкрементов, `X` сравнений вида `i < X` и `X-1` переход на следующую итерацию. Инкремент стоит явно не больше

Строки

```
pa += K
pc += 4 * N
pb += 2 * N
```

– это одно сложение.

Строка

```
s += Cache.read8(pa + k) * Cache.read16(pb + 2 * x)
```

– это одно сложение, одно умножение, два обращения к кэшу. Кроме того, каждое обращение `read8` и `read16` (как будет указано позже) мы будем игнорировать второй такт ответа, но нужно его пропустить, прежде чем направлять следующий запрос – это учтём в счётчике `waits`.

Непосредственно в кэше мы будем считать следующее:

При попадании считаем `hitrs` (hit requests); в случае, если это был read-request, ещё добавляем `resps` (responses). Поскольку нет ни одного запроса на `read32`, всегда будем считать, что время ответа – 1 такт, так как процессору нет смысла ожидать второй «пустой» такт ответа.

При промахе – `misss` (misses) – время для кэша определить, что нужно переадресовать запрос памяти, `maccs` (memory accesses) – здесь учитывается факт, что нужно дождаться ответа. В случае, если запрос был на чтение – ещё нужно учесть ответ (`resps`).

В случае, если произошло вытеснение, нужно учесть время запроса записи в память (`maccs`). Сразу после этого происходит чтение (так как отдельно инвалидации в программе нет), так что не дожидаться ответа мы не можем.

Итак, получаем следующие результаты:

```
122879 jumps, cost: 122879
7682 initializations, cost: 7682
126784 increments, cost: 126784
245888 summations, cost: 245888
126784 comparisons, cost: 126784
122880 multiplications, cost: 614400
230698 cache requests with hit, cost: 1384188
18902 cache requests with miss, cost: 75608
122880 delays between cache read-requests, cost: 122880
245760 cache response delays, cost: 245760
20218 memory accesses, cost: 2021800

summary cost: 4844989
230698 hits, 18902 misses
```

Таким образом, процент попаданий – $\frac{230698}{230698+18902} = 92.427\%$.

Тем не менее, несмотря на кэширование, обращение к памяти остаётся самой дорогой операцией и занимает здесь 41.7% времени. Следующее по стоимости – обращение к кэшу занимает суммарно `1828436` тактов, что составляет 37.7% времени.

могла бы быть описана здесь, если бы удалось написать хоть что-то работающее. Поэтому только прилагается файл с тем, что уж точно работает (то есть почти с ничем).

Листинг кода симуляции, Kotlin

```
const val MEMORY_SIZE = 256 * 1024
const val CACHE_SIZE = 2048
const val LINE_SIZE = 16
const val CACHE_TAG_SIZE = 8

val OFFSET_SIZE = LINE_SIZE.countTrailingZeroBits()
val CACHE_ADDR_SIZE = MEMORY_SIZE.countTrailingZeroBits()
val CACHE_SET_SIZE = CACHE_ADDR_SIZE - OFFSET_SIZE - CACHE_TAG_SIZE // 6
val CACHE_SETS_COUNT = 1 shl CACHE_SET_SIZE // 64

fun parseAddr(address: Int) = listOf(
   (address and 0b11111110000000000) shr 10,
   (address and 0b00000001111110000) shr 4,
   (address and 0b00000000000001111),
)

fun intOf(a: Byte, b: Byte, c: Byte, d: Byte) =
   a.toUByte().toInt() shl 24 or (b.toUByte().toInt() shl 16) or
      (c.toUByte().toInt() shl 8) or d.toUByte().toInt()

fun shortOf(c: Byte, d: Byte) = ((c.toInt() shl 8) or d.toInt()).toShort()

fun ByteArray.toInt() = intOf(this[0], this[1], this[2], this[3])
fun ByteArray.toShort() = shortOf(this[0], this[1])

fun Int.toBytes() =
   listOf(this shr 24, this shr 16, this shr 8,
this).map(Int::toByte).toByteArray()

fun Short.toBytes() = listOf(this.toInt() shr 8, this)
                        .map(Number::toByte).toByteArray()

fun Int.toBinary(n: Int) = toString(2).run { "0".repeat(n - length) + this }
fun Byte.toHex() = toUByte().toString(16).run { "0".repeat(2 - length) + this }

operator fun ByteArray.get(rng: IntRange) = slice(rng).toByteArray()
operator fun ByteArray.set(rng: IntRange, byteArray: ByteArray) =
   rng.withIndex().forEach { (j, i) ->
      this[i] = byteArray[j]
   }

object Memory {
   private val data = ByteArray(MEMORY_SIZE)

   fun getLine(line: Int) = data
      .slice((line shl OFFSET_SIZE) until ((line + 1) shl OFFSET_SIZE))
      .toByteArray()

   fun setLine(line: Int, data: ByteArray) {
      val from = line shl OFFSET_SIZE
      for (i in 0 until LINE_SIZE) this.data[from + i] = data[i]
   }
```

```
    fun check(array: ByteArray) = (0 until MEMORY_SIZE).all { data[it] == array[it]
}
}
class CacheLine {
    var lastUpd = false // Uses that way = 2
    var tag: Int = 0
    var valid: Boolean = false
    var dirty: Boolean = false
    var line: ByteArray = ByteArray(LINE_SIZE)

    override fun toString(): String = if (valid) {
        (if (lastUpd) "UPD1" else "upd0") +
            " tag " + tag.toBinary(CACHE_TAG_SIZE) +
            " " + (if (dirty) "drt" else "cln") +
            " " + line.joinToString(":") { it.toHex() }.uppercase(Locale.getDefault())
    } else "INVALID"
}

class CacheSet(val set: Int) {
    private val line1 = CacheLine()
    private val line2 = CacheLine()

    private fun sorted(): Pair<CacheLine, CacheLine> {
        if (!line1.valid && !line2.valid) throw AssertionError()
        return when {
            !line1.valid  -> line2 to line2
            !line2.valid  -> line1 to line1
            line1.lastUpd -> line1 to line2
            else          -> line2 to line1
        }
    }

    fun load(tag: Int, line: ByteArray) {
        if (line1.valid && line2.valid) {
            val sorted = sorted()
            sorted.first.lastUpd = false
            writeToMemory(sorted.second)
            readFromMemory(sorted.second, tag, line)
        } else if (line1.valid) {
            readFromMemory(line2, tag, line)
            line1.lastUpd = false
        } else if (line2.valid) {
            readFromMemory(line1, tag, line)
            line2.lastUpd = false
        } else {
            readFromMemory(line1, tag, line)
            line2.lastUpd = false
        }
    }

    private fun writeToMemory(line: CacheLine) {
        if (!line.dirty) return
        maccs++
        Memory.setLine(line.tag shl CACHE_SET_SIZE or set, line.line)
        line.dirty = false
    }
```

```
    fun invalidate(tag: Int) {
        if (line1.valid && line1.tag == tag) {
            writeToMemory(line1)
            line1.valid = false
            line2.lastUpd = false
        }
        if (line2.valid && line2.tag == tag) {
            writeToMemory(line2)
            line2.valid = false
            line1.lastUpd = false
        }
    }

    private fun readFromMemory(cacheline: CacheLine, tag: Int, line: ByteArray) {
        cacheline.tag = tag
        cacheline.line = line
        cacheline.dirty = false
        cacheline.valid = true
        cacheline.lastUpd = true
    }

    fun hits(tag: Int) = (line1.valid && line1.tag == tag) ||
        (line2.valid && line2.tag == tag)

    fun lineWith(tag: Int) =
        if (line1.valid && line1.tag == tag) {
            line1.lastUpd = true
            line2.lastUpd = false
            line1
        } else if (line2.valid && line2.tag == tag) {
            line2.lastUpd = true
            line1.lastUpd = false
            line2
        } else throw java.lang.AssertionError()

    fun invalidateAll() {
        writeToMemory(line1)
        line1.valid = false
        line2.lastUpd = false
        writeToMemory(line2)
        line2.valid = false
        line1.lastUpd = false
    }
}

object Cache {
    val data = List(CACHE_SETS_COUNT) { CacheSet(it) }

    fun read8(address: Int): Byte {
        val (tag, set, off) = parseAddr(address)
        val line = accessLine(set, tag, false)
        return line[off]
    }

    fun read16(address: Int): Short {
        val (tag, set, off) = parseAddr(address)
        val line = accessLine(set, tag, false)
```

```
    return line[off..(off + 1)].toShort()
}

fun read32(address: Int): Int {
    val (tag, set, off) = parseAddr(address)
    val line = accessLine(set, tag, false)
    return line[off..(off + 3)].toInt()
}

fun write8(address: Int, b: Byte) {
    val (tag, set, off) = parseAddr(address)
    val line = accessLine(set, tag, true)
    line[off] = b
}

fun write16(address: Int, s: Short) {
    val (tag, set, off) = parseAddr(address)
    val line = accessLine(set, tag, true)
    line[off..(off + 1)] = s.toBytes()
}

fun write32(address: Int, i: Int) {
    val (tag, set, off) = parseAddr(address)
    val line = accessLine(set, tag, true)
    line[off..(off + 3)] = i.toBytes()
}

private fun accessLine(set: Int, tag: Int, pollute: Boolean): ByteArray {
    val block = data[set]
    if (block.hits(tag)) {
        hits++
        if (!pollute) { //Read request
            hitrs++; resps++
        } else { //Write request
            hitrs++ //No response wait needed
        }
    } else {
        misss++
        maccs++
        if (!pollute) resps++
        misses++
        block.load(tag, getBytes(tag, set))
    }
    val line = block.lineWith(tag)
    if (pollute) line.dirty = true
    return line.line
}

private fun getBytes(tag: Int, set: Int): ByteArray {
    return Memory.getLine(tag shl CACHE_SET_SIZE or set)
}

fun invalidate(address: Int) {
    val (tag, set, off) = parseAddr(address)
    data[set].invalidate(tag)
}
```

```
    fun invalidateAll() = data.forEach(CacheSet::invalidateAll)
}
fun test() {
    val local = ByteArray(MEMORY_SIZE)
    val rnd = Random()
    for (i in 0 until 1000000000) {
        when (rnd.nextInt(8)) {
            0 -> Unit // NOP - ignore
            1 -> { //READ8
                val addr = rnd.nextInt(MEMORY_SIZE)
                val read = Cache.read8(addr)
                if (local[addr] != read) println("Expected byte ${local[addr]}, got
$read")
            }

            2 -> { //READ16
                val addr = rnd.nextInt(MEMORY_SIZE / 2) * 2
                val read = Cache.read16(addr)
                val actual = local[addr..(addr + 1)].toShort()
                if (actual != read) println("Expected short ${actual}, got $read")
            }

            3 -> { //READ32
                val addr = rnd.nextInt(MEMORY_SIZE / 4) * 4
                val read = Cache.read32(addr)
                val actual = local[addr..(addr + 3)].toInt()
                if (actual != read) println("Expected int ${actual}, got $read")
            }

            4 -> { //INVALIDATE
                val addr = rnd.nextInt(MEMORY_SIZE)
                Cache.invalidate(addr)
            }

            5 -> { //WRITE8
                val addr = rnd.nextInt(MEMORY_SIZE)
                val value = rnd.nextInt().toByte()
                Cache.write8(addr, value)
                local[addr] = value
            }

            6 -> { //WRITE16
                val addr = rnd.nextInt(MEMORY_SIZE / 2) * 2
                val value = rnd.nextInt().toShort()
                Cache.write16(addr, value)
                local[addr..(addr + 1)] = value.toBytes()
            }

            7 -> { //WRITE32
                val addr = rnd.nextInt(MEMORY_SIZE / 4) * 4
                val value = rnd.nextInt()
                Cache.write32(addr, value)
                local[addr..(addr + 3)] = value.toBytes()
            }
        }
    }
    Cache.invalidateAll()
```

```
   println(Memory.check(local))
}
var hits = 0
var misses = 0

const val NEXT_ITERATION = 1
const val INITIALIZATION = 1
const val SUMMATION = 1
const val INCREMENT = 1
const val COMPARISON = 1
const val MULTIPLICATION = 5
const val HIT_REQUEST_DELAY = 6
const val MISS_REQUEST_REDIRECT_DELAY = 4
const val RESPONSE_TIME = 1 // Never send read32 request
const val CONNECTION_WAIT = 1 // Never send read32 request
const val MEM_CTR = 100
const val M = 64
const val N = 60
const val K = 32 //array creation outside the function

var iters = 0
var inits = 0
var incrs = 0
var summs = 0
var comps = 0
var mults = 0
var hitrs = 0
var misss = 0
var resps = 0
var waits = 0
var maccs = 0

fun countFor(iterations: Int) {
   iters += iterations - 1
   incrs += iterations
   comps += iterations
}

fun main() = count()

fun count() {
   var pa = 0
   inits++
   var pc = M * K + 2 * K * N // compile-time constant
   inits++
   countFor(M)
   for (y in 0 until M) {
      countFor(N)
      for (x in 0 until N) {
         var pb = M * K // compile-time constant
         inits++
         var s = 0
         inits++
         countFor(K)
         for (k in 0 until K) {
            s += Cache.read8(pa + k) * Cache.read16(pb + 2 * x)
            summs++; mults++; waits++//Tactes for memory access are count in Cache
```

```
            pb += 2 * N
            summs++
         }
         Cache.write32(pc + 4 * x, s)
      }
      pa += K
      summs++
      pc += 4 * N
      summs++
   }
   println(
      """
$iters jumps, cost: ${iters * NEXT_ITERATION}
$inits initializations, cost: ${inits * INITIALIZATION}
$incrs increments, cost: ${incrs * INCREMENT}
$summs summations, cost: ${summs * SUMMATION}
$comps comparisons, cost: ${comps * COMPARISON}
$mults multiplications, cost: ${mults * MULTIPLICATION}
$hitrs cache requests with hit, cost: ${hitrs * HIT_REQUEST_DELAY}
$misss cache requests with miss, cost: ${misss * MISS_REQUEST_REDIRECT_DELAY}
$waits delays between cache read-requests, cost: ${waits * CONNECTION_WAIT}
$resps cache response delays, cost: ${resps * RESPONSE_TIME}
$maccs memory accesses, cost: ${maccs * MEM_CTR}

summary cost: ${
         iters * NEXT_ITERATION +
            inits * INITIALIZATION +
            summs * SUMMATION +
            comps * COMPARISON +
            mults * MULTIPLICATION +
            hitrs * HIT_REQUEST_DELAY +
            misss * MISS_REQUEST_REDIRECT_DELAY +
            resps * RESPONSE_TIME +
            maccs * MEM_CTR
      }
   """.trimIndent()
   )
   println("$hits hits, $misses misses")
}
```