Notes for Twelfth Lecture (Nov 24)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A Cache Primer

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Q: In hardware terms, what is a cache?

A: A cache is an on-chip storage region, organized as an array, that---unlike the register file---is not under program control. Instead, each cache is controlled by its own FSM controller. In my main narrative, caches are specialized. The I-cache stores instructions and listens to signals from the f-box. The D-cache stores data and listens to signals from the m-box. There exist integrated caches, which store both instructions and data, and listen to both boxes. There also exist multilevel caches. I find single-level D-caches to be pedagogically appropriate for lectures and exams.

Q: What are caches good for?

A: Imagine a processor about to execute a load instruction. If, by chance, there is a copy of the item to be loaded in the D-cache, then it is much less expensive to simply abort the load, and retrieve the item from the D-cache. Think of memory as being at \_great distance\_ from the processor.

Q: What is a cache line?

A: Caches prefer to transfer large amounts of data, for example, 128 bytes, every time they copy anything. Logically, if the quantum of cache data transfer is B bytes, then that alone partitions all of memory into adjacent memory slices, each B bytes in length. A memory slice is called a cache line (or just line), and has a line number. The first line is number 0.

Q: What are the fundamental cache equations?

A: Well, consider one type of cache, the direct-mapped cache. It is an array of frames. Each frame is just large enough to store a single cache line. Let F be the number of frames in a direct-mapped cache.

Q: Is there some relation among the memory address, the line number, and the index of the frame into which the line is to be put?

A: This takes a multipart answer. First, since there are B bytes per line, the line number is given by:

line number = memory address `div` B.

By the way, taking the memory address `mod` B gives the byte offset of the item in the line.

Second, since there are F frames per direct-mapped cache, the array index (or frame number) is given by:

array index = line number `mod` F.

Q: But some bits are left over. What are they good for?

A: Draw a picture of a partitioned memory address.

|--- line number ---|

+--------------------------------------+

|tag value | frame number | byte offset|

+--------------------------------------+

Or

|--- line number ---|

+--------------------------------------+

|tag value | array index | byte offset|

+--------------------------------------+

Performing `div` B shortens the memory address to the line number. Performing `mod` F on the line number allows the frame number to be read. The bits to the left of the frame number allow different lines that map to the same frame to be distinguished.

//Tag value: field that holds values distinguishes different lines that maps to the same frame. (Leftover bits)

Q: What really important thing did you forget to tell us?

A: Well, while computers can perform `mod` and `div` on arbitrary integers, they far prefer both divisors B and F to be powers of two. Say, B = 2^b, and F = 2^f. It is this restriction that allows `mod` and `div` to be calculated by simple shifting and masking, without using an arithmetic functional unit.

Each of the three fields has a defined number of bits, and we can calculate their values with simple bit manipulation. Here, the byte-offset field has 'b' bits, the index field has 'f' bits, and the tag field has whatever number of bits is left over.

Q: Can you show us?

A: Consider a 32-bit address, say, [abcd5678]. Suppose b = 8 and f = 16. Then, by visual inspection, we see:

ab!cd56!78

32bits/4 = 8 => B = 2^8 = 256. But suppose B = 251. In that case, memory address `div` 251 cannot be calculated by simply removing some number of lower-order bits from the memory address.

By the way, b = 7 and f = 15 are fine; the exponents do not need to be multiples of four.

The field lengths in the latter case are:

<10, 15, 7>.

We want capital letters to be powers of 2.

Fundamental Cache Equations

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Every cache is an array of \_elements\_. What an element is, and how a cache-array index is computed from the relevant memory address, differ between two cache families. Both families will be introduced. However, the unit of data transfer between memory and cache, for both families, is the \_line\_. Memory is byte addressed and a line contains some fixed number of bytes, chosen by the cache designer. The cache equations below specify which lines---which are actually memory slices---are to be copied into which cache elements.

Generally true (both families)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0) byte offset = memory address `mod` the number of bytes a line

1) line number = memory address `div` the number of bytes in a line

2) array index = line number `mod` the number of array elements in the cache

3) tag value = line number `div` the number of array elements in the cache

In a direct-mapped cache, the array elements are \_frames\_, but, in an m-way set-associative cache, the array elements are \_sets\_ of frames.

When a line is copied from memory into the cache, it is, according to these equations, placed in one of the cache elements.

Because many lines are mapped to the same element, a tag value is used to identify which line got copied. This is required to be able to look up lines in caches.

This is the punch line. When you understand these equations, you understand how caches work.

----

I will rewrite the last two equations in a family-specific way.

family 1: direct-mapped cache

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) frame index = line number `mod` the number of frames in the cache

3) tag value = line number `div` the number of frames in the cache

family 2: set-associative cache

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) set index = line number `mod` the number of sets in the cache

3) tag value = line number `div` the number of sets

---

I was contemplating giving you some calculational techniques to compute cache quantities.

We all know that, for cache purposes, a memory address (a natural number) must be decomposed into three smaller natural numbers, viz., the tag value, the array index, and the byte offset.

Computers blindly follow the fundamental cache equations, first calculating the line number, and then calculating the index. Humans can skip a step unless they are specifically asked for the line number.

Both humans and computers have an easier time of it if both B = 2^b and F = 2^f are powers of two. I stick to direct-mapped caches here---F is the number of frames in the cache---because set-associative caches don't change the math; they just substitute S = 2^s for F, where S is the number of sets per cache.

Consider the 32-bit memory address [abcd4567]. Let the offset field be 6 bits in length, the index field be 13 bits in length, and the tag field be 13 bits in length.

The offset depends only on [67], so it is easily done by hand.

offset --> 01!10.0111 ==> 0010.0111 = [27].

Hexits 6 7

offset = [27]

But x `mod` y = x - (x/y \* y),

so you may also calculate this as 67 - 40 = 27.

The line number is [abcd45]01. This is mixed binary-hexadecimal notation. You can reduce this by hand with bits, or use an arithmetical approach.

line number = 2af3515

The index is 0101[45]11 = [545]11. Again, bits or arithmetic.

index = 1517

The tag is [abc]1. Again, etc.

tag = 1579

There is a clear trade-off here. Manipulating bits with your pen is "slow and steady wins the race". This requires patience and good handwriting.

But use of arithmetical techniques saves time, and may also be used to \_verify\_ a hand calculation. Arithmetic also converts mixed binary-hexadecimal expressions into pure hexadecimal, which is usually what I require on exams.

And, of course, in the above I have combined hand bit manipulation with arithmetic---for notation conversions.

Replacement Policy

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In a direct-mapped cache, if some line other than you is in your frame when you suffer a cache miss, then there is only one reasonable victim: the guy in your frame. You should flush him from the cache.

In an m-way set-associative cache, when there is a cache miss, the indexed set of 'm' frames may well be full. To make room for a new cache line, a victim line must be selected from the set. The best

scheme is \_least recently used\_ (LRU): Replace that cache line that has been unused for the longest time. If you don't want to pay for the bookkeeping, use \_random\_ replacement. I have heard a suggestion that a

cache twice as large with random replacement has a miss rate equal to the smaller cache with LRU. [This won't be on the test.]

I also wrote:

Write Policy

\_\_\_\_\_\_\_\_\_\_\_\_

Consider a store instruction. If we wrote only into the D-cache and not into the memory, the cached copy would differ from the memory original. We could use \_write through\_ and write the data into both the memory and the D-cache. This typically has poor performance. The alternative to write through is \_write back\_. When a write occurs, the new value is written only to the line in the cache. The modified line is written to the next lower level of the hierarchy, which we will always take to be the main memory, only when it is replaced. Unmodified lines are simply evicted from the cache when they are replaced.

---

This deserves a little explanation. You remember that a memory reference is either a load or a store. In a memory reference, some data item, which has a memory address, is involved. The counter-intuitive point is that, if you don't currently have a copy of that data item, then you can perform \_neither\_ a load \_nor\_ a store.

When you issue a load instruction and the item to be loaded is not present in the cache, you retrieve the relevant line from a lower level, extract the item, and present it to the processor. When you issue a store instruction and the item to be stored into is not present in the cache, you retrieve the relevant line from a lower level, and then overwrite the item in the copied line.

So, the cache response to a cache miss, for either a load or a store, is basically the same. A lot of people miss this symmetry.

Q: What is Little's Law all about? (strictly speaking, not a cache topic)

A: Latency, Bandwidth, and Concurrency (theory of pipelines)

1) In any system that transports items from input to output without creating or destroying them,

\*\* latency times bandwidth equals concurrency \*\* [c = b \* t]

2) Queueing theory calls this result Little's Law:

pipeline

+-----------------+

--> | --> --> --> | -->

bandwidth = 2 | concurrency = 6 | bandwidth = 2

--> | --> --> --> | -->

+-----------------+

latency = 3

3) Import: There are transparent and opaque pipelines. In the figure, we probably assume that the (limiting) hardware bandwidth is two operations per cycle. The user of the pipeline must continually supply new operations, ideally, two new operations per cycle. We can, equivalently, speak of operation-insertion bandwidth, or of the concurrency that results. This is just different language for describing the same thing.

---

Again, some explanation is required. We have studied the 'fdxmw' instruction-execution pipeline. We promised to supply one new instruction in every cycle. The pipeline was willing to accept that much input bandwidth. If we keep our promise, the pipeline delivers a speedup of 5. And a full 'fdxmw' pipeline has a concurrency of five instructions. c = b \* t : 5 = 1 \* 5

However, if we tried to push in two new instructions per cycle, the pipeline would strenuously object (we couldn't do it!). If we pushed in one new instruction every two cycles, both the output bandwidth and the pipeline concurrency would be cut in half.

We don't need queueing theory to understand the 'fdxmw' pipeline. But there are many other pipelines, such as the roundtrip pipeline from processor to memory, that are very important and that explain why RISC CPUs are so bad at tolerating network/memory latency, and consequently depend so critically on

their caches.

Cache Notation

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I change my notation more often than I change my shirt. But good notation makes calculation easy.

Consider a computer with 28-bit memory addresses, and a direct-mapped cache.

Let the offset width be 6 bits, the index width 16 bits, and the tag width 6 bits.

Pictorily, +--------------------------------------------------+

| tag = 6 bits | index = 16 bits | offset = 6 bits |

+--------------------------------------------------+

16 bits trivializes the calculation, but I'm only showing you notation.

Specifically, I am reminding you of mixed binary-hexadecimal notation.

Let the memory address be aebcddf. With the hex table in front of us, we count over 6 bits from the right and the left to get:

[a]11!10[bcd]11!01[f]

Now, if you're a computer, you calculate:

line number = [2baf37]

and then

frame number = af37. [This calculation is trivial for a human.]

frame number is a multiple of 4.

But a human could go:

tag = [a]11 = [2b]

index = 10[bcd]11 = [2bcd]11 = [af37]

offset = 01[f] = [1f]

for a final partition of:

2b!af37!1f

End of sales pitch for this notation.

Mixed bin/hex notation (some calculations)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

It always useful to have the following nearby:

Hex table:

0 0000 4 0100 8 1000 c 1100

1 0001 5 0101 9 1001 d 1101

2 0010 6 0110 a 1010 e 1110

3 0011 7 0111 b 1011 f 1111

Sometimes we need to calculate the tag value, the index value, and the offset value, from large memory addresses represented in hexadecimal. Often, it is easier to combine "finger" calculations with only occasional use of a pocket calculator. Of course, this is a matter of personal taste.

Consider the address:

abcdabcdabcdabcdabcdabcd (24 \* 4 = 96-bit memory address)

We break this apart once we have a partition of 96. Some partitions are ridiculously easy. Consider <20, 56, 20>. By inspection, the three values are:

abcda ! bcdabcdabcdabc ! dabcd (24 \* 4 = 96-bit memory address)

But what about <22, 52, 22>? In mixed bin/hex notation, this is:

[abcda]10 ! 11[cdabcdabcdab]11 ! 00[dabcd]

We want to recover a pure hexadecimal representation. (6 hexits > 22 bits)

[abcda]10 = [2af36a] 00[dabcd] = [0dabcd] (or [dabcd])

11[cdabcdabcdab]11 = [f36af36af36af] (13 hexits = 52 bits)

The last calculation was a bit of a stretch (I actually guessed it!).

More down to Earth:

01[dabcd] = [1dabcd]

10[dabcd] = [2dabcd]

11[dabcd] = [3dabcd]

As long as there are not too many hexits in [...], it's not too hard. Let's do three more:

101[dabcd] = [5dabcd] (6 hexits > 23 bits)

[dabcd]110 = [6d5e6e] (6 hexits > 23 bits)

11[dabcd]111 = [3dabcd]111 = [1ed5e6f] (7 hexits > 25 bits)