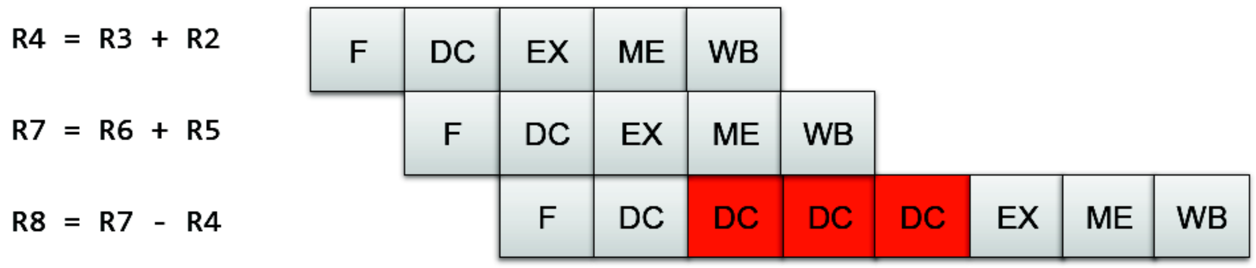
Tutorial 1: Processors and Caching

Question 1: Pipelining

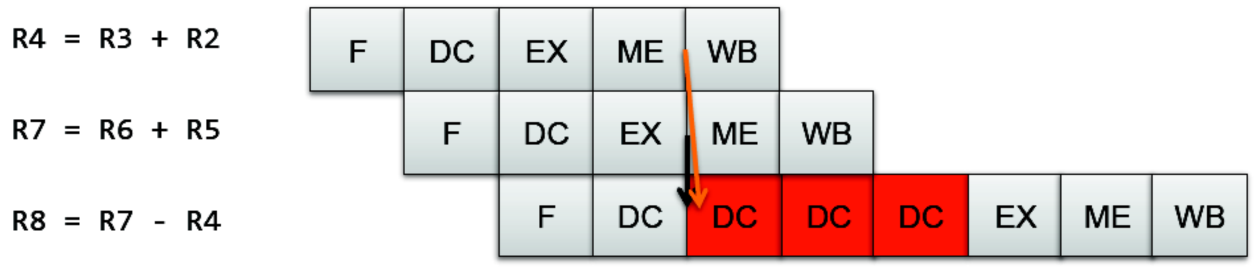
Consider the pipelined execution of the following operations:

```
R4 = R3 + R2

R7 = R6 | R5

R8 = R7 – R4
```

1. How many clock cycles would it take to complete these operations on a canonical RISC machine with no operand forwarding?
2. How many clock cycles would it take if the EX stage output could be forwarded to its input?
3. How many clock cycles would it take if the EX and MEM stage outputs could be forwarded to the EX input?
4. Now assume that the second operand of the OR has to be fetched from L2 Cache with a 10 cycle latency and loaded in to R5 before use. Draw pipelining diagram[1] with four operations in flight, showing the duration of the stall and how long the execution would take overall now

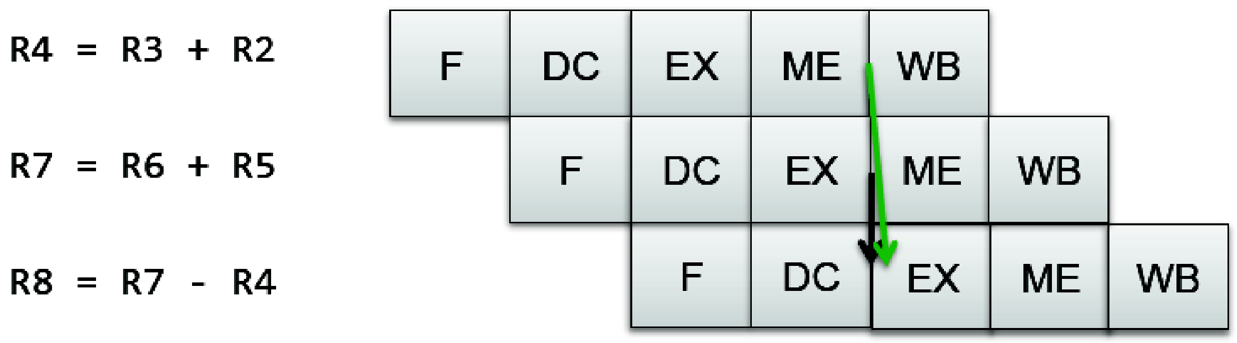


10 cycles with no forwarding (all WB must **complete** before DC can start).



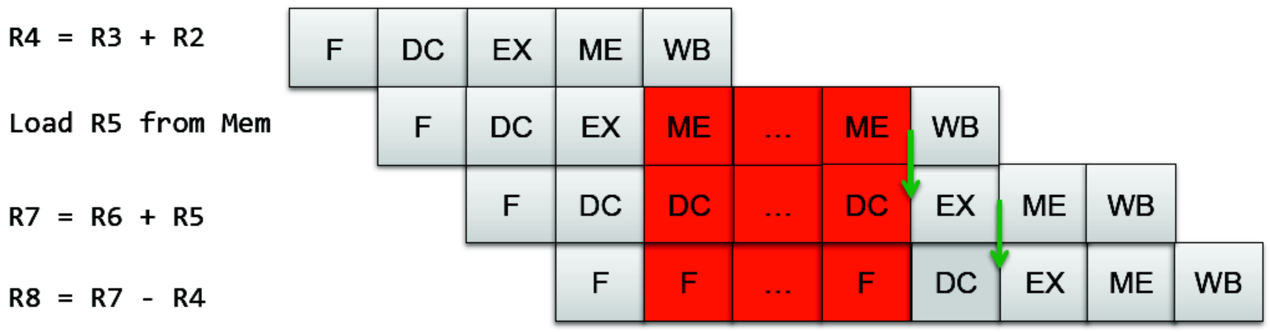
1http://eng.anu.edu.au/courses/ENGN8537/notes/es03-processors.html#/step-22

Forwarding EX only doesn't help as R4 would need to be forwarded from MEM at the same time (red arrow). By the time R4 has completed RB and can be processed normally, R7 is at the MEM stage so cannot be forwarded either.



If MEM and EX can both be forwarded, there are no stalls and the instructions complete in 7 clock cycles.

This seems like a bit of a trick question but raises an important point: Operand Forwarding requires more complexity than one might think. Just having the output of the EX stage forwarded back is insufficient, you have to have a forwarding path back from every downstream unit for full safety! Perhaps not an issue in a 5-stage pipeline, but in a larger pipeline it can lead to significant extra complexity, particularly with respect to timing constraints.



18 cycles with full forwarding. This is not surprising: 7 cycles for the base execution in the previous part, one cycle for the extra instruction, 10 cycles for the memory access.

5.

Question 2: Caching

Referring to the memory access latency benchmark on the next slide (and ignoring the last point at 0 which is a bug in the plotter!), answer the following:

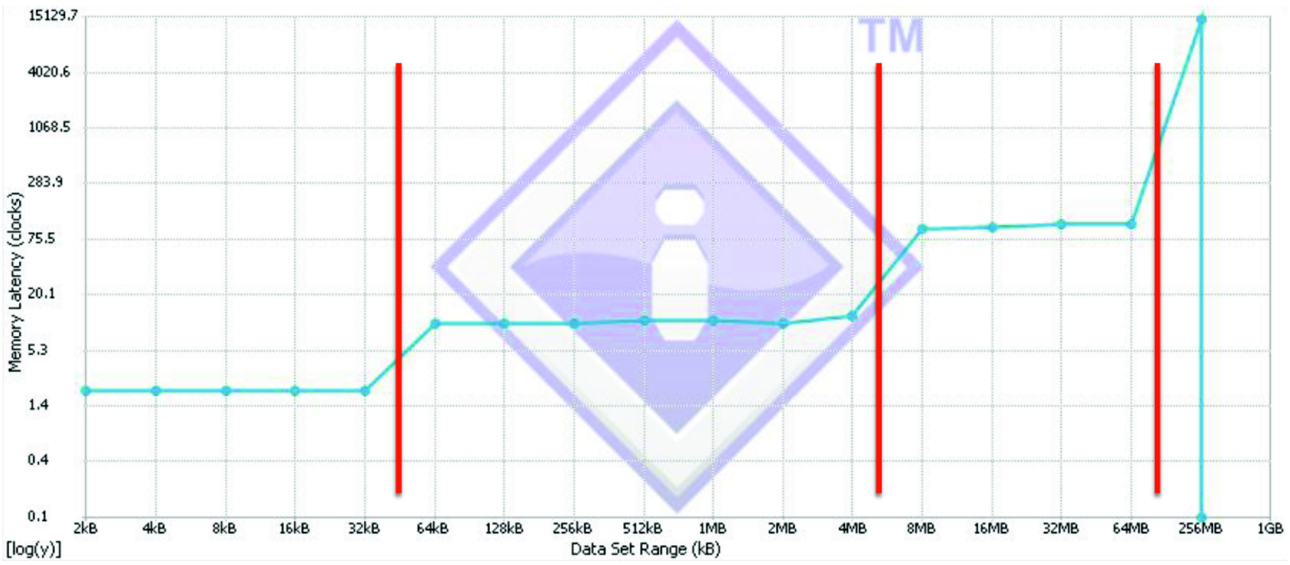
1. How many levels of cache does the processor have and (roughly) how big is each one? How big is Main Memory (RAM)?
2. One of the caches is 6MB, is it most likely to be Direct or Associative?

The plot is of steady-state sequential access latency. That is, a loop is run that accesses each memory location in turn, from address 0 up to the length under test. This loop is run several thousand times in succession and latency of each access timed and averaged. This means that any different timing on the first run through can be ignored.

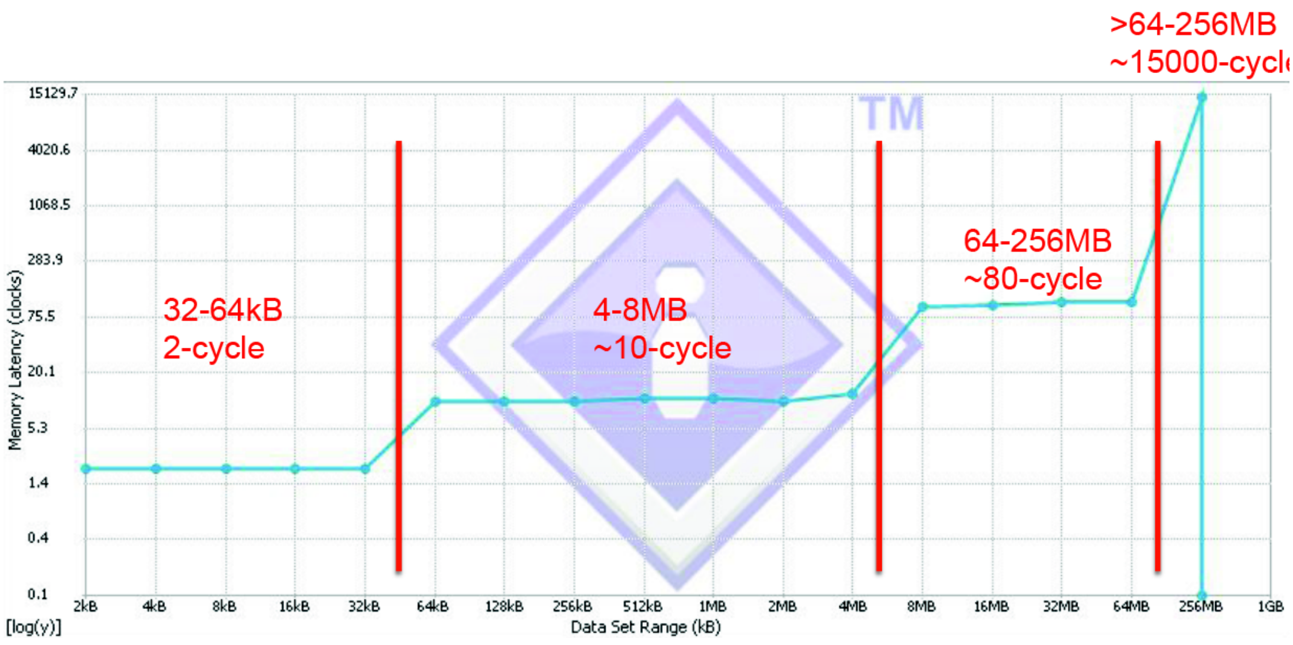
Note that these values are only sampled at the discrete points marked. The lines between points are for visual purposes only.

Solution part 1

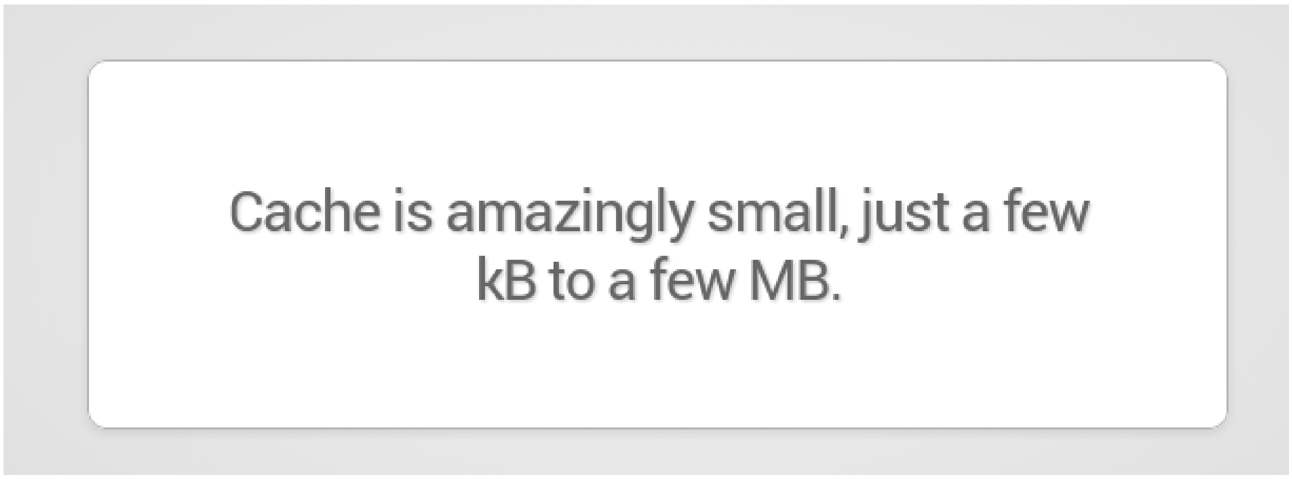
1. Mark distinct regions



2. Label with size, latency



3. Refer to the lecture notes

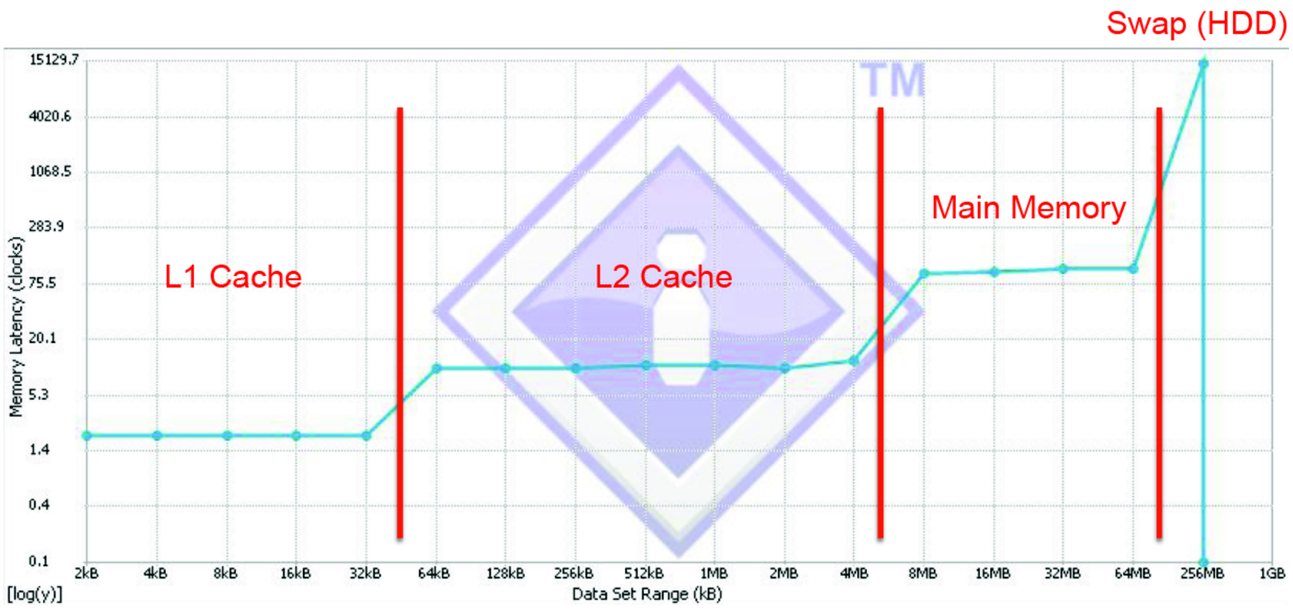


Cache is amazingly small, just a few kB to a few MB.

Hard drive contents can also be cached, this time in normal RAM. This is why if you have lots of programs open your computer it suddenly slows down: You've run out of real RAM and you're now using 'fake RAM' on the hard drive which is much slower.

This is also called 'swap'.

4. Apply



5. Solve

Two levels of Cache

L1 is between 32- and 64kB, can’t be exact

L2 is between 4- and 8MB, this agrees with the 6MB given in the second half of the problem.

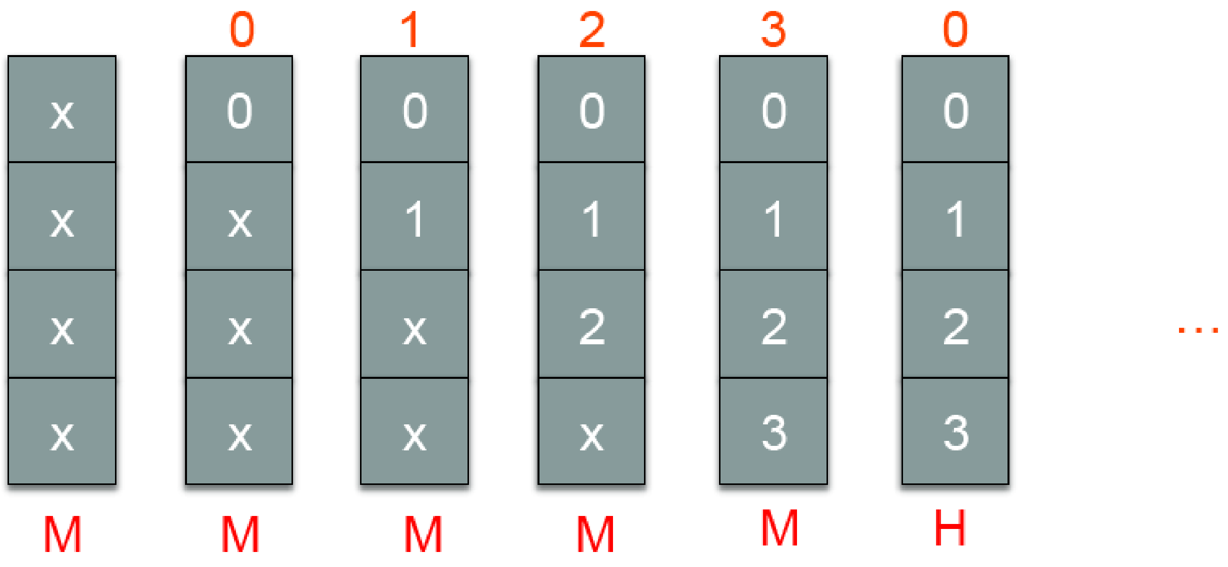
Main memory is between 64- and 256MB.

Solution, Part 2

Examine the misses and hits for a simple cache as you start to access more memory than it can hold.

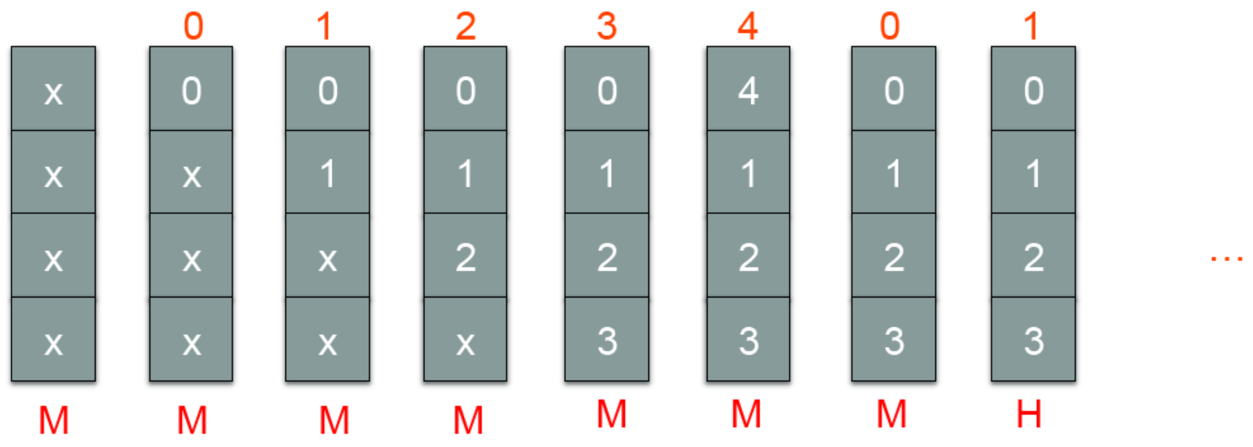
We will start with a four-line direct-mapped cache (line size of one byte or, equivalently, access size of one line).

First, access four memory addresses.



So in the steady state, after the first loop through, all addresses are in memory and stay in cache, they all 'hit' and the cache runs at full speed.

Now five memory addresses:



Accesses to address '0' and '4' conflict for the first line and so always miss. All other entries hit. Therefore for this, 2 out of 5 memory accesses miss the cache.

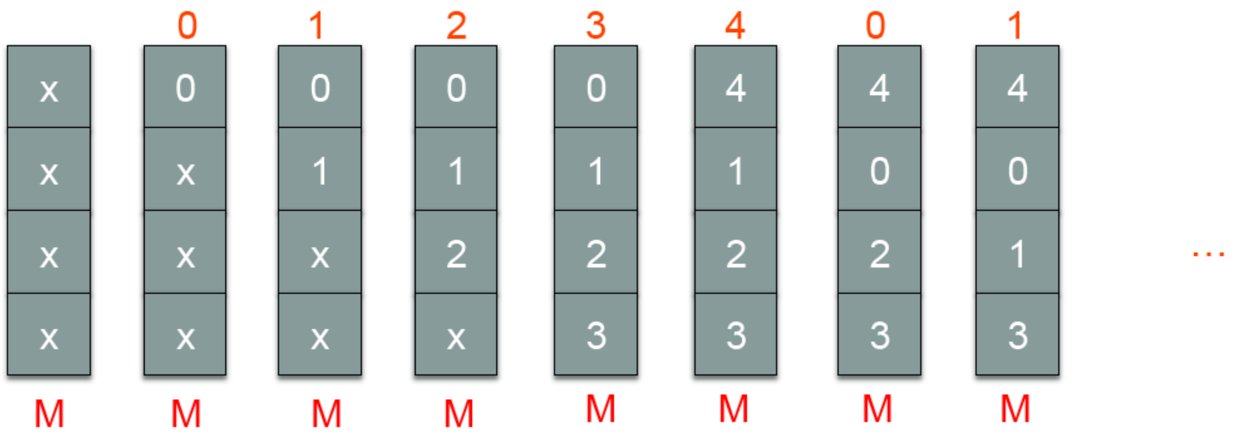
Similarly, with 6 accesses, addresses 0 and 4 conflict for the first line, 1 and 5 conflict for the second line and the other accesses hit. 4 out of 6 miss.

With 7 accesses, 6 out of 7 would miss.

With 8 accesses, all 8 miss.

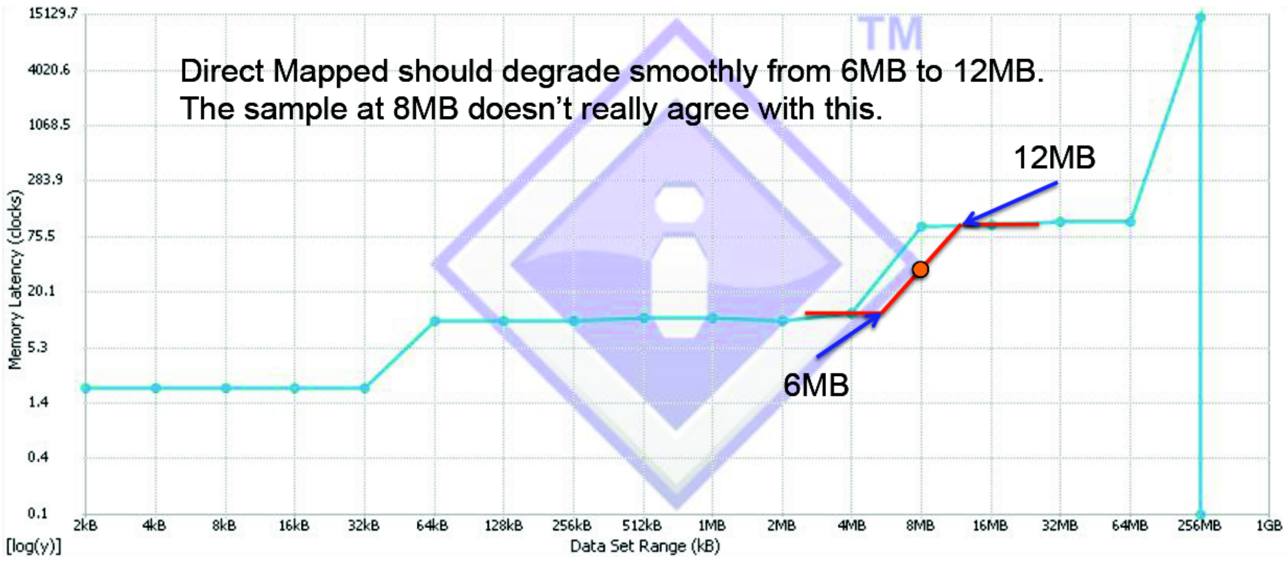
A Direct-Mapped cache runs at full speed for accesses at the cache size, has no benefit at twice the cache size and has a smooth transition in between. It degrades gracefully.

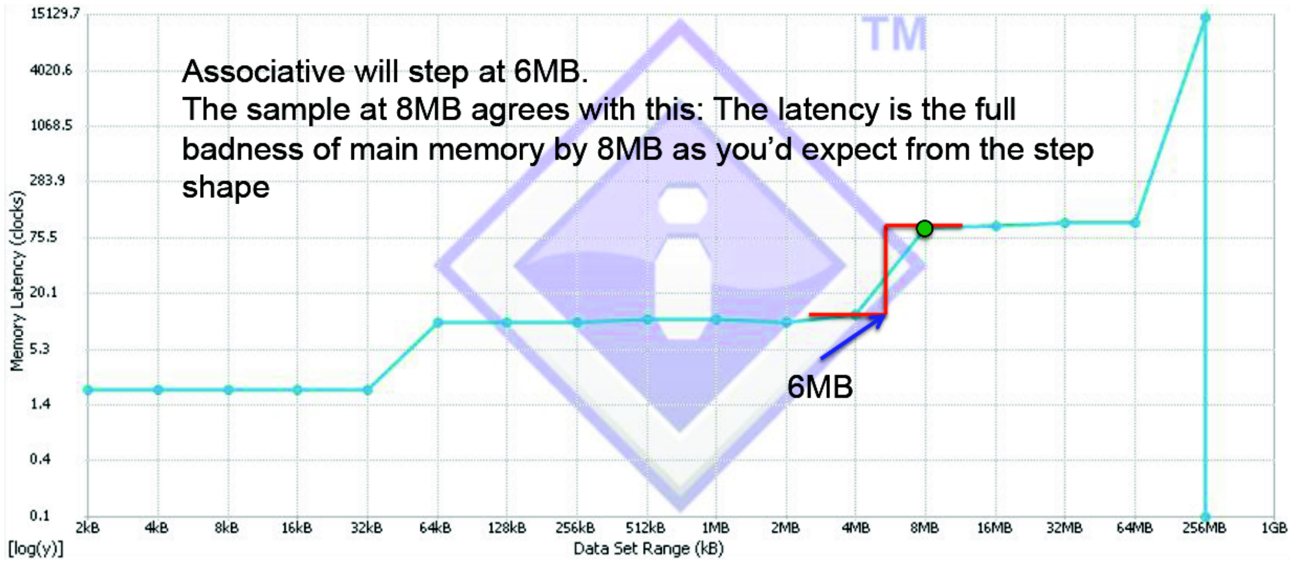
In an Associative cache, 4 accesses will similarly run at full speed. Let's look at five accesses.



With this linear access pattern, the oldest entry is the one that is going to be evicted but that is precisely the one we need next! As such, a single access over the cache size and the cache immediately becomes useless (always misses).

Let's see how these compare to the observed data





If the cache were direct-mapped, we'd expect than an 8-MB access would feel some benefit of the cache and run run at the full, slow, main-memory speed.

This is not what we observe, we observe that the L2 cache has no effect on 8-MB accesses and, given the problem states that it's a 6-MB cache, this is only true of an associative cache.

The cache is associative.